arXiv:2301.02194v1 [cs.PL] 5 Jan 2023

作为归纳族视图的内置类型

Guillaume Allais

2023年1月6日

摘要

依赖类型语言中最先进的优化遍可以帮助消除富含不变量的数据结构和程序中常见的冗余信息。这些自动化过程不会显著改变数据的 *structure*，尽管可能存在效率更高的表示方式。

借助定量类型论，我们展示了如何定义一种包含丰富不变量的类型检查时数据结构，将高效的运行时表示与运行时无关不变量打包在一起。随后，编译器可以在编译期间积极地擦除所有此类不变量。

与其他方法不同，最终表示的复杂度完全可预测；我们不要求两种表示具有相同的结构，同时又能无缝地编程，仿佛使用的是高层结构。

1 引言

依赖类型语言使用户能够借助归纳族 [Dyb94]，精确描述其论述领域。程序员可以将关键不变量直接融入定义，从而细化函数的输入和输出。受约束的输入使他们在模式匹配时只需考虑相关情形，而经过细化的输出则保证客户端代码可以安全地依赖这些不变量始终得到维护。这种编程风格称为“构造即正确”。

然而，如果宿主语言以朴素方式编译归纳族，依赖归纳族可能会带来不可忽视的运行时开销。即使是依赖类型语言最先进的优化遍也无法创造奇迹：如果源代码效率低下，可执行程序也不会高效。

例如，先进的编译器能够成功地将长度索引列表编译为普通列表，从而使空间复杂度从列表长度的二次方降至线性。然而，面对一个长度在静态上已知小于 64 的布尔值列表时，它却无法将其打包进单个机器字，因此耗费了线性空间，而常数空间本就足够。

在第2节中，我们将探讨一个优化示例，借此展现当前技术水平在消除使用归纳族可能带来的运行时开销方面的优势与局限。

在第 3 节中，我们将简要介绍定量类型论。这种富有表现力的语言赋予程序员同时拥有强不变量和可靠、高效运行时表示的能力。

在第4节中，我们将考察一个归纳族，它在 TypOS 项目 [AAM+22] 中以对性能要求极高的方式使用，而其编译效果受到第2节所强调的限制。我们目前采用的做法并不理想：在 Agda 中试验时依赖安全、方便的归纳族，然后在实际的 Haskell 实现中将其替换为不安全但效率高得多的表示。

最后，在第5节中，我们将研究以 Idris 2 实现的高效且蕴含丰富不变式的解决方案。我们还将展示，借助智能构造器和视图，我们几乎可以重获使用归纳族编程的所有便利。

2 一个优化示例

归纳族的朴素¨编译方式效率低下，最典型的例子大概是向量类型（Vect）和有限数类型（Fin）。它们之间的关系可通过查找函数来展示。让我们研究这个例子，看看在此情形下，连续的优化遍如何消除使用索引族而非普通数据所带来的开销。

向量是长度索引列表。类型 Vect 以所存储值的类型为参数，并由一个自然数索引，该自然数对应于向量的长度。更具体地说，它的 Nil 构造器会构造一个大小为 Z（即零）的空向量，而它的 (::) （读作“构造子”）构造器会将一个类型为 a 的值（头部）与一个大小为 n 的子向量（尾部）组合起来，构造一个大小为（Sn）（即 n 的后继）的向量。

```
数据Vect : 自然数 -> 类型 -> 类型 其中Nil :
VectZ a(::): a -> Vect n a -> Vect (S n) a
```

大小 n 并未在 (::)的类型中显式绑定。在 Idris 2 中，这意味着它会以前缀方式自动泛化，这让人联想到 ML 系列语言对自由类型变量的处理。因此，它是该构造器的一个隐式参数。鉴于 自然数 是一元表示的 自然数 类型，对（¨）的朴素运行时表示Vect n a的大小会与 n呈二次关系。即使采用完美共享的更聪明表示，开销仍相当可观，正如 Brady、McBride 和 McKinna 所观察到的 [BMM03]。

有限数是指已知严格小于给定自然数的数。类型 Fin 以该上界为索引。它的 Z 构造器表示 0 ，且任何非零上界都可以作为其上界；它的 S 构造器接收一个以上界 n 限定的数，并返回它的后继，其上界为 (1 + n)。朴¨素编译在这里也会导致运行时表示遭遇二次膨胀。

```
数据Fin : 自然数 -> 类型 其中Z :
Fin (Sn)S : Finn -> Fin (S n)
```

这引出了 查找函数的定义。给定一个大小为 n 的向量，以及一个由同一个 n 界定的有限数 k，我们可以定义一个全函数，用于查找向量中位置 k 处存储的值。它必定会返回一个值。请注意，在模式匹配子句中，我们无需考虑空向量的情况，因为 Fin 各个构造器的所有返回类型都要求索引非零；而向量和有限数描述的是同一个 n，因此空向量会自动意味着存在一个类型为 (Fin0) 的值，这显然是不可能的。

```
查找: Vect n a -> Fin n -> a查找(x :: _)
Z   = x 查找 (_ :: xs) (S k)   = 查找 xs k
```

多亏了索引族，我们得以定义一个绝不可能失败的函数，也能够只讨论合理的模式匹配子句。这似乎是以效率为代价的，但幸运的是，关于擦除的研究早已十分深入，可以自动检测冗余数据 [BMM03]，或检测那些在运行时不会用到的数据 [Tej20]。

2.1 优化 Vect、Fin和 查找

采用 Brady、McBride 和 McKinna 的风格进行分析 [BMM03] 可以通过观察到向量所索引的自然数完全由向量的主干决定，从而解决上文强调的二次膨胀问题。具体来说，尾部的长度无需作为构造器的一部分存储：它可以重构为整个向量长度的前驱。因此，运行时只需用一个自然数和一个列表组成的数对，就足以表示一个向量。同样，一个有界数也可以用两个自然数组成的数对表示。综合以上各点，并记住向量和有限数共享同一个n，查找可以编译成一个接收两个自然数和一个列表的函数。在 Idris 2 中，我们会将优化后的 查找写成如下形式（我们使用 **partial**关键字，因为转换后的版本在该类型下并非全函数）。

```
partial
查找 : (n : 自然数) -> 列表 a -> 自然数 -> a 查找 (S n)
(x :: _) Z   = x 查找 (S n) (_ :: xs) (S k)   = 查找 n xs k
```

我们可以看到，在第二个子句中，递归调用作用于列表（以前的向量）的尾部，因此 lookup所对应的、表示向量大小的第一个参数减小了 1。尽管不变式不再显式表示，但它仍得以维持。

Teji˘s˘c´ak 风格的分析 [Tej20] 还可以进一步发现，查找函数从未使用上界的值，因此可以将其完全丢弃。由此，针对向量的查找函数会被编译为作用于列表的、看似部分化的对应函数。

```
部分
查找: 列表 a -> 自然数 -> a查找(x :: _)
Z  = x查找(_ :: xs) (S k)  = 查找 xs k
```

尽管在我们看来，这是一个相当有说服力的例子，展示了如何消除归纳族带来的表面复杂性，但这种方法有两个缺点。

首先，它依赖于编译器能够并且会自动执行这些优化这一事实。但类型系统并未阻止用户无意间使用他们以为会被擦除的值，从而导致 Teji˘s˘c´ak 式优化无法生效。在对性能要求严苛的场景中，用户或许更希望明确表达自己的意图，并由编译器确保其意图得到保留，以换取可预测且有保证的优化。

其次，这种方法本质上仅限于保留类型整体结构的变换：运行时数据结构虽然更简单，但仍然非常相似。我们不能指望做到太多。到目前为止，期望通过改变表示方式，例如用平衡二叉树替代列表，以获得对数级查找而非线性查找，仍不现实。

2.2 NoMagic 方案

即使我们能够通过充分擦除，在运行时获得更紧凑的归纳族表示，也不能保证运行效率。正如 Coq 手册 [CDT22] 提醒用户的那样，将一元自然数提取为高效的机器表示时，提取并不会神奇地优化掉用户定义的二次乘法算法。为了实用起见，Coq、Agda 和 Idris 2 都采用了特设规则，用目标语言中渐近效率更高的对应函数，替换实现方便但效率低下的数值函数。

然而，这种方法无法扩展：如果我们愿意将可信核心扩展为一个高质量的无界整数库，就不希望仅仅出于效率考虑，用一个截然不同且不可信的实现替换我们的代码——这些代码之所以能被证明正确，恰恰是因为依赖了复杂的不变量。

在本文中，我们使用 Idris 2 [Bra21] 中实现的定量类型论 [McB16、Atk18]，在基于归纳族的不变式丰富但效率低下的表示，与使用底层原语实现的安全性较低但效率高的实现之间架起桥梁。归纳族使我们能够将 [Wad87、MM04] 在底层高效表示中编码的运行时相关信息视为信息丰富的编译时数据结构。此外，数量注释保证了这一附加信息

将在编译过程中被擦除。

3 Idris 2 的一些关键特性

Idris 2 实现了定量类型论，这是一种经过扩展的 Martin-L¨of 类型论，其中使用数量半环对值的使用方式进行分类。在一个类型中，每个绑定符都用其参数必须遵守的数量进行注释。

3.1 数量

一个值可以是运行时无关的、线性的或不受限的。

运行时无关的值（**0** 数量）绝不可能影响控制流，因为它们会在编译期间被完全擦除。这迫使语言对这些值上的模式匹配施加严格限制。典型例子包括 a 在（列表a）中这样的类型参数，或像（n 在（Vect n a）中这样的自然数指标。这些值保证会在编译时被擦除。相较于 Teji˘s˘c´ak 风格的分析，其优势在于，用户可以声明某个参数应当与运行时无关，而语言会坚持要求证明它确实如此。

线性值（**1** 数量）必须恰好使用一次。典型例子包括 Idris 2 用于实现 % World令牌的 IO 单子 `（类似 Haskell），以及不能丢弃、必须先显式关闭文件的文件句柄。在运行时，这些值可以被破坏性地更新。本文不会使用线性特性。

最后，不受限值（即没有数量标注的值）可以流入任意位置、被复制或丢弃。它们是函数式编程中常见的不可变值。

同时运用运行时无关性和不受限数量的最基础例子，是恒等函数。

```
id :  {0 a : 类型}-> (x : a) -> aidx
 = x
```

它的类型以一个使用花括号的绑定符开头。这表示它引入了一个隐式变量，调用时用户无需提供该变量，合一会重建它。引入的变量名为a ，类型为 类型。它带有 **0** 数量标注，这表示该参数在运行时无关，因此会在编译期间被擦除。

第二个绑定符使用圆括号。它引入了一个显式变量，变量名为 x ，类型是刚刚绑定的类型 a 。它没有数量标注，这表示它是一个不受限变量。

最后，返回类型是之前绑定的 类型。这正如预期，是一个从 到 的多态函数。它通过一个子句实现：在左侧绑定 ，然后立即在右侧返回它。

如果我们尝试将 的绑定符标注为 `0` 数量，使其在运行时无关，那么 Idris 2 理应拒绝这个定义。下面的 **失败**代码块展示了部分错误消息，其中抱怨无法在右侧以不受限数量使用。

```
失败 “x 在此上下文中不可访问。”id :  {0 a :
类型}-> (0 x : a) -> aidx  = x
```

3.2 证明搜索

在 Idris 2 中，Haskell 风格的特设多态 [WB89] 被一种更通用的证明搜索机制所取代。该机制不采用类型类、实例和约束这些备受推崇的概念，而是可以将任何依赖函数类型的定义域标记为**自动**。这表示编译器会将相应参数视为隐式参数，并且不会仅通过合一来重建它，而是会通过证明搜索来完成。搜索算法会使用适当的用户声明提示，以及作用域内的局部变量。

默认情况下，数据类型的构造器总会被添加到提示数据库中。因此，以下声明会将索引族引入作用域`So` ，它包含关于给定布尔值为 `True` 的证明，以及唯一的构造器`Oh` ，该构造器会自动作为提示添加。

```
数据So : 布尔型 -> 类型 其中Oh:
So True
```

因此，例如，我们可以定义一个记录类型，说明`n` 是偶数意味着什么：将它的一半与一个证明一同存储，而该证明既是运行时无关的，又由证明搜索填充。由于（`2 * 3 ==6`）计算结果为`True`，Idris 2 能够使用`Oh` 提示自动填充定义中缺失的证明。

```
记录Even (n : 自然数) where构造器
MkEvenhalf: 自然数{auto0 prf : So
(2 * half  == n)}
```

```
偶数6: 偶数 6偶数6 = 构造偶
数 { 一半 = 3 }
```

我们将在第5.3节中同时使用`So` 和**auto**机制。

3.3 应用：`Vect`，如列表

我们可以利用定量类型论的特性来实现 `Vect` ，并确保它在运行时擦除为一个列表，且不受编译器所实现的优化遍影响。与第2节所述的优化遍相比，这种方法的优势在于用户可以控制

运行时表示，而无需依赖编译器部署这些优化。

核心思想是：通过定义一个将编码为列表以及其长度等于预期自然数索引的证明打包在一起的记录，使“向量是长度索引列表”这一口号成为现实。该证明被标记为运行时无关，以确保编译后只剩下列表。

记录Vect (n : 自然数) (a : 类型)
where构造器 MkVect编码: 列表 a0 有效 :
长度编码 === n

智能构造器 现在我们已经定义了向量，可以通过定义智能构造器来恢复向量通常的构建模块，也就是说，定义函数 Nil 和 (::)，用它们替代归纳族的数据构造器。

```
Nil : Vect Z a
Nil = MkVect [] Refl
```

智能构造器 Nil返回一个空向量。毫不意外，它被编码为空列表（[]）。由于（长度 []）静态计算为 Z，因此可以通过自反性证明编码是有效的。

```
(::): a -> Vect n a -> Vect (S n) ax :: MkVect xs eq
 = MkVect(x :: xs) (cong S eq)
```

使用 (::) 我们可以将一个头部和一个长度为 n 的尾部组合起来，得到一个长度为 (S n) 的向量。其编码通过将头部构造到尾部的编码前面得到；证明该编码有效的部分（cong S eq）利用了命题相等具有同余性这一事实，以及 (length(x :: xs)) 可计算为 (S (lengthxs)。

视图 现在我们已经知道如何构造向量，接下来演示如何使用视图将它们拆开。

类型 T 的视图，按照 Wadler [Wad87]，的定义，并经 McBride 和 McKinna [MM04]，改进，是一个以 T 为指标的归纳族 V，并配有一个全函数，将 t 中的每个元素映射到类型 (Vt) 的一个值。这个简单的构件提供了一种强大且可由用户扩展的模式匹配泛化形式。模式以归纳方式定义，可以是模式变量、强制项（即由另一个模式产生的约束所确定的任意表达式），也可以是完全应用于子模式的数据构造子。相比之下，归纳族构造器的返回指标可以是任意表达式。

在我们关心的情形中，视图让我们能够模拟“匹配”，判断构造待拆解向量时用的是哪一个智能构造器：Nil还是 (::)。

```
数据视图: Vect n a -> 类型 其中Nil : ViewNil(::):
(x : a) -> (xs : Vect n a) -> View (x :: xs)
```

归纳族 View以一个向量为索引，包含两个构造器，分别对应两个智能构造器。我们利用 Idris 2 的重载能力，将每个 View 构造器命名为其所对应的智能构造器。通过对类型为 (View xs) 的值进行模式匹配，我们就能将 xs 拆解为其组成部分，并观察到它等于 Nil，或取出它的头部和尾部。

```
视图: (xs : Vect n a) -> View xs视图 (MkVect [] Refl)  =
Nil视图 (MkVect(x :: xs) Refl)  = x :: MkVect xs Refl
```

函数 view 表明，只要检查编码列表，就总能判断使用了哪个构造器。如果它为空，那么向量就是用 Nil 智能构造器构造的。否则，我们便得到了编码的头部和尾部；（对尾部进行一些重新包装后）它们实际上就是通过智能构造器组合在一起的头部和尾部。

3.3.1 应用：**映射**

接下来，我们可以使用这些构造来实现向量上的函数 map ，而无需显式操作编码。完全语法糖化版本的 map 如下：

```
映射 : (a -> b) -> Vectn a -> Vect n
b映射 f xs@_ 通过 (视图xs)_ | [] = []
_ | hd :: tl = f hd :: 映射 f tl
```

在左侧，视图使我们能够无缝地对输入向量进行模式匹配。使用 **with**关键字，我们局部修改了函数定义，使其接收一个额外参数，即中间计算（视图 xs）的结果。相应地，我们有两个子句对这个额外参数进行匹配；符号 | 将原始左侧（此处用 _ 省略，因为它与父子句中的内容完全相同）和额外的模式分隔开来。这个模式可以是[] 或（hd :: tl），相应地，我们得知 xs 要么是 [] ，要么是（hd :: tl）。

在右侧，智能构造器使我们能够构造输出向量。将函数映射到空向量上会得到空向量，而映射到 cons 节点上则会得到一个 cons 节点，其头部和尾部都已相应修改。

这个 map 的语法糖版本等价于下面这个更显式的版本：

```
映射 : (a -> b) -> Vect n a -> Vect n b 映射 f xs
with (view xs) 映射 f .([]) | [] = []映射 f .(hd:: tl) |
hd :: tl = f hd :: 映射 f tl
```

在父子句中，我们显式地绑定了 `xs`，而不是仅仅通过写作（`xs@`）为它引入一个别名，因此我们需要明确说明这个模式在两个 with 子句中分别如何细化。

在 with 子句中，我们明确重复了父子句左侧经过细化后的形式。具体来说，我们使用了点模式，强调`xs` 现在完全由（匹配结果）确定，这是通过匹配 (`view`xs) 的结果实现的。

我们已经看到，通过匹配调用 (`view xs`) 的结果，我们可以像在`xs`上进行匹配一样进行匹配，仿佛`Vect` 是一个归纳类型。这就是视图的强大之处。

3.3.2 应用：查找

类型 (`Fin n`) 同样可以表示为一个自然数，以及一个证明该数小于 `n` 的运行时无关证明。这里省略这些定义，感兴趣的读者可以尝试自行实现，也可以查看随附的代码。

结合这些定义，我们可以定义一个类似于第2节中所定义的 查找函数。

```
查找: Vect n a -> Fin n -> a 查找xs@_ k@_
使用 (视图 xs) | (视图 k)_ | hd :: _ | Z =
hd_ | _ :: tl | S k' = 查找 tl k'
```

我们看起来是在两种不同的类型（视图 （`Vect` 和 `Fin` 分别）上使用视图，但实际上这两处分别指向不同的函数：Idris 2 允许函数重载，并会执行类型引导消歧。

出于教学目的，这个经过语法简化的 查找版本也可以展开为一个更明确的版本，以展示视图的强大之处。

```
查找: Vect n a -> Fin n -> a 查找xs k 使用 (视图 xs) | (视图 k) 查
找 .(hd :: tl) .(Z) | hd :: tl | Z = hd 查找 .(hd :: tl) .(S k') | hd ::
tl | S k' = 查找 tl k'
```

这个定义的主要优点是，仅根据其类型，我们就能确定该函数在运行时处理的必定是一个列表和一个自然数。这种高效的运行时表示并不依赖于最先进的优化遍会被部署这一假设。

我们已经了解了 Idris 2 的一些强大功能，以及如何借助这些功能，让用户能够控制他们所操作的归纳族的运行时表示。这个简单示例只让我们得以复现那些采用最先进优化遍的编译器已经能够实现的性能。在接下来的章节中，我们将看到如何

运用同样的核心理念，将归纳族编译成截然不同的运行时表示，同时保持良好的高级使用体验。

4 变薄，两种实现方式

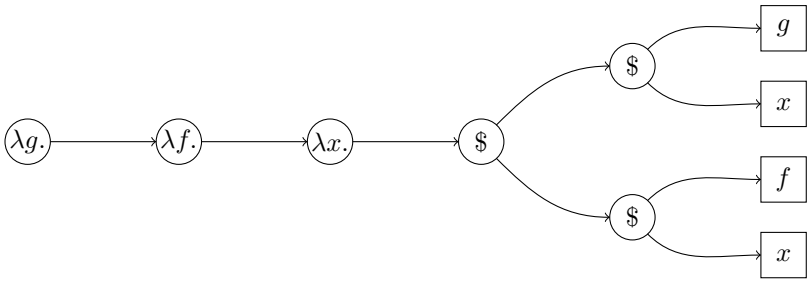
在持续开发 TypOS [AAM+22]（一种用于定义并发类型检查器和精化器的领域特定语言）的过程中，我们遇到了编译归纳族的一项重大限制。该项目的核心是定义能够操作带绑定的通用语法概念的执行体。内部而言，这种带绑定语法的项基于 co-de Bruijn 表示（我们将在下文解释这种编码），而该表示高度依赖变薄。从源作用域到目标作用域的变薄（也称为保序嵌入 [Cha09]）是将较小作用域保序单射到较大作用域中的一种方式。它们通常使用归纳族表示。变薄在 co-de Bruijn 表示中无处不在，因此其运行时表示对性能至关重要。

我们先提醒读者注意：在命名表示、de Bruijn 表示和 co-de Bruijn 表示中，抽象语法树的结构分别是什么样的。随后，我们将讨论变薄的两种表示：一种是安全且便捷的归纳族表示，另一种则是不安全但高效的编码，即使用一对任意精度整数表示。

4.1 命名、德布鲁因与 co-de Bruijn 语法

本节将以 S 组合子（$\lambda g.\lambda f.\lambda x.gx(fx)$）为贯穿示例，并使用语法树表示项，其中构造器节点用圆形表示，变量节点用方形表示。为了描绘 S 组合子，我们只需要 λ-抽象节点和应用节点（以 $ 表示）。构造器的参数成为树中的子节点。树从左到右排列，构造器的参数则从上到下显示。

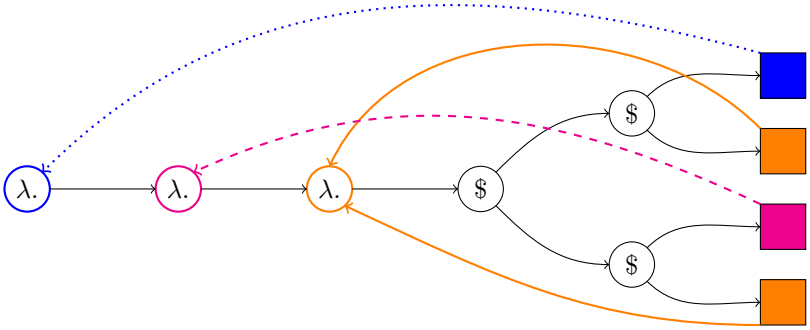
具名语法 第一种表示方式使用显式名称。每个绑定符都有一个对应的名称，每个变量节点都携带一个名称。变量指向最近的、恰好使用相同名称的外层绑定符。



要检查两个项在结构上是否等价（α-等价），可能需要重命名绑定名称。为了进行简单且廉价的 α-等价性检查，我们可以改用无名表示。

德布鲁因语法 基于德布鲁因指标的抽象语法树 [dB72]用自然数替代名称，该自然数表示变量与其绑定位置之间相隔的绑定符数量。S组合子现在写作（$\lambda\lambda\lambda 2\,0\,(1\;0)$）。

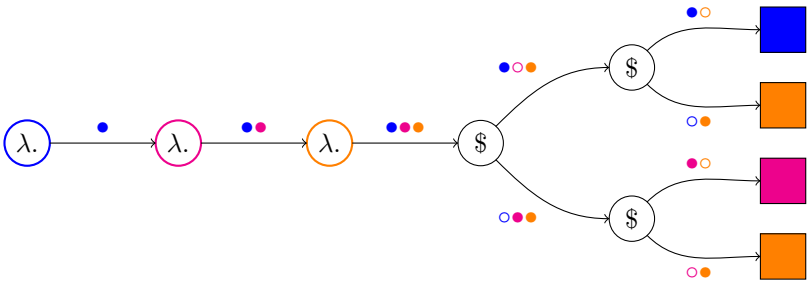
从下面的图示可以看出， λ抽象不再带有名称，变量则直接指向引入它们的绑定符。我们把方框留空了，但实际中，各种颜色的箭头会用自然数表示。例如，洋红色虚线箭头对应 1，因为要先忽略一个 λ抽象（也就是橙色箭头），沿着树向根节点前进，然后才能到达相应的洋红色绑定符。



要检查一个子项是否未提及给定的一组变量（即增厚测试；它与稀疏化相反，后者会用未使用的变量扩展当前作用域），需要遍历整个项。为了让增厚测试简单且开销低，我们可以确保每个子项都准确知道自己的支撑集是什么，以及它如何嵌入其父项的支撑集中。

余德布鲁因语法 在余德布鲁因表示中 [McB18]，每个子项都会恰好选出在该项作用域内的变量，因此，变量构造子最终指向到达该处时作用域内唯一剩下的变量。这种表示确保我们准确知道给定项当前的作用域。

在下面的图示中，我们将稀疏化表示为由 full（•）或空（○）圆盘组成的列表，具体取决于对应变量是被保留还是被丢弃。例如， ○•• 所表示的稀疏化会舍弃蓝色变量，并保留洋红色和橙色变量。



可以看到，在这种表示中，树中的每个节点都会为每个子项存储一个稀疏化。除非我们能高效地表示稀疏化，否则这种做法将难以行得通。

4.2 co-de Bruijn 的性能挑战

采用 co-de Bruijn 方法时，任意上下文中的项都表示为一对：一个 co-de Bruijn 语法中的项，以及一个从其支撑集到更大作用域的变薄。精确掌握每个项的支撑集后，我们便能更高效地执行变薄、替换、合一或公共子表达式消除等操作。

对项进行变薄时，我们不再需要遍历它。实际上，将项嵌入更大的上下文并不会改变其支撑集，因此我们只需复合两个变薄，同时保留该项不变。

替换可以跳过那些不会发生变化的子项。实际上，现在它可以轻松检测出替换的定义域是否与子项的支撑集不相交。

合一时，如果要用一个看似处于更大上下文中的项来求解在给定上下文中声明的元变量，就需要执行稠化测试。我们同样无需遍历该项即可完成此测试，只需检查外部变薄是否可以稠化。

公共子表达式消除要求我们识别可能存在于不同上下文中的 α 等价项。使用 de Bruijn 表示时，它们在句法上可能有所不同：由自然数 v 表示的变量在 Γ 中会表示为（$1+v$），而在 Γ 中则表示为（σ），在 $2+v$ 中则表示为（Γ）。co-de Bruijn 表示通过舍弃所有不属于支撑集的变量，保证我们能够再次使用句法相等来检测 α 等价。这种编码也被 Maziarz、Ellis、Lawrence、Fitzgibbon 和 Peyton-Jones 在其《模 α 等价哈希》工作 [MEL+21] 中使用，只是他们并未意识到这一点。

基于所有这些原因，正如我们之前提到的，我们在 TypOS [AAM+22] 的实现中选择了 co-de Bruijn 表示。因此，对于性能而言，拥有一种紧凑的变薄表示至关重要。

4.2.1 TypOS 中的稀疏化

我们先在 Agda 中仔细处理实现中较为棘手的部分，再将最终代码移植到 Haskell。这个过程凸显出一个显著的差距：一方面是使用强类型的变薄归纳表示所做的实验，另一方面则是其在 Haskell 中效率更高但不安全的编码。

Agda 基于 Agda 的实验采用归纳族，明确表达关键不变量，有助于跟踪复杂约束，并在类型检查时发现设计缺陷。指标保证我们始终会对

变薄进行适当处理，在添加或删除绑定变量时皆如此。在 Idris 2 中，变薄的归纳族表示形式可写为：

data Thinning : (sx, sy : SnocList a) -> 类型 where Done : Thinning [<] [<]保留 : Thinning sx sy -> (0 x : a) -> Thinning (sx :< x) (sy :< x) 丢弃 : Thinning sx sy -> (0 x : a) -> Thinning sx (sy :< x)

该 稀疏化族由两个作用域索引（以 snoc 列表表示，即从右侧扩展的列表，正如推理规则中的上下文）： sx 表示较紧的作用域， sy 表示较宽的作用域。 Done 构造器对应于从空作用域到其自身的稀疏化（[<] 是 Idris 2 中空 snoc 列表的语法糖），而 Keep和 Drop 分别通过保留或丢弃最局部的变量来扩展给定的稀疏化（:< 是“snoc”构造子，是一种反向的“cons”）。这个“名称”（x 类型为 a）标记为数量 0 ，以确保它在编译时被擦除（参见第 3 节）。

在编译过程中，Idris 2 会在归纳族的指标被强制时将其擦除（这里的“强制”采用 Brady、McBride 和 McKinna [BMM03] 的定义），并丢弃标记为运行时无关的构造器参数。最终得到的归纳类型将是以下简单数据类型。

数据 稀疏 化 = Done | 保留 稀疏化 | 丢弃 稀 疏化

因此，在运行时，这种表示本质上是一个由布尔值构成的链表（Done 对应 Nil，而 Keep 和 Drop 分别对应（True ::）和（False::）。

Haskell Haskell 实现利用了这一观察结果，并将这个布尔值列表紧凑编码为一对整数。一个整数表示列表的长度 n ，另一个整数的 n 个最低有效位则将该列表编码为位模式，其中 1 表示 Keep，而 0 表示 Drop。

稀疏化的基本操作通过显式操作单个位来实现。它没有进行索引，因此所有不变式的跟踪都必须手动完成。这导致了许多难以诊断的错误。

4.2.2 Idris 2 中的稀疏化

Idris 2 是一种自举语言，其核心数据类型目前基于良好作用域的 de Bruijn 表示。通过按作用域对项进行精确索引，彻底消除了一整类曾困扰 Idris 1 合一机制的错误。

如果我们要将核心语言改为使用 co-de Bruijn 表示，就会希望也应该能够兼得两者之长：一种安全且高效的表示！

值得庆幸的是，Idris 2 实现了定量类型理论（QTT），让我们能够充分控制哪些内容在运行时相关，哪些内容会在编译过程中被擦除。这应该能让我们坚持采用类似归纳族的高级接口，同时确保除

一对整数外的所有内容都在编译时被擦除。我们将利用第 3 节介绍的 QTT 关键特性，做到两全其美。

5 一种高效的富不变量表示

我们可以结合第4.2节中介绍的两种方法，定义一个以源作用域（sx）和目标作用域（sy）为参数的记录，分别对应变薄两端，就像定义归纳族时那样。该记录打包了两个数和一个运行时无关证明。

首先，我们有一个称为 bigEnd 的自然数，对应于变薄大端（sy）的大小。这里使用（一元）自然数没有问题，因为我们知道 Idris 2 会将其编译为无界整数。

其次，我们有一个称为 encoding 的整数，对应于用位向量表示的变薄；该位向量说明了每个变量是保留还是丢弃。我们只关心该整数的 bigEnd 个最低有效位，并假设其余位都为 0。

第三，我们有一个运行时无关证明 invariant，证明 encoding 确实是一个有效编码，其大小为 bigEnd，表示从 sx 到 sy 的变薄。我们将在第5.3节进一步探讨关系 Invariant 的定义。

```
记录 Th {a : 类型} (sx, sy : SnocLista) 其中构
造器 MkTh大端: 自然数编码: 整数0 不变式: 不变
式 大端 编码 sx sy
```

这个定义足够恰当的第一个迹象，是我们能够构造任意有效的稀疏化。我们通过引入一些函数来证明这一点，这些函数充当归纳族数据构造子的智能构造器类比。

5.1 Th 的智能构造器Th

第一个也是最简单的函数是 done，它将一对 0 (大端的大小和空编码)与一个证明打包在一起，证明该编码恰当地表示从空作用域到其自身的稀疏化。在这个例子中，证明就是 Done构造器。

```
done : Th [<] [<]done = MkTh { 大端 = 0, 编码 = 0, 不变式
= Done }
```

为了实现 保留和 丢弃，我们需要执行位级操作。Idris 2 的 Bits 接口让这些操作变得简单，它提供了将位模式向左或向右移位的函数（shiftl、shiftr）、在指定位置设置或清除位的函数（setBit、clearBit），以及执行按位逻辑运算的函数，例如析取（.|.）或合取（.&.）等。

无论是 保留还是 丢弃，我们都需要在编码中扩展一个额外的位。为此，我们引入 构造子 函数，它接受一个位 b 和一个现有编码 bs，并返回新的编码 $bs{\cdot}b$。

```
构造子 : 布尔型 -> 整数-> 整数构造子 b bs  = 令 bs
0 = bs `左移`1 于是假如 b 则 (bs0 `setBit`0) 否则 bs0
```

无论新位的值是什么，我们都会先将编码左移，为它腾出空间；这样就得到 bs0 其中包含位模式$bs{\cdot}0$。如果该位为 真，那么我们还需要将位置 0 的位设为 1，从而得到 $bs{\cdot}1$。否则，如果该位为 假，我们可以直接返回 $bs{\cdot}0$左移所得的编码。该函数的正确性由两个引理保证：在执行 cons 后测试索引 0 处的位，结果就是 cons 的位；将 cons 后的编码右移，则会回到未扩展的编码。

```
testBit0Cons: (b : 布尔型) -> (bs : 整数) ->
testBit(构造子 b bs)  0 === b
consShiftR: (b : 布尔型) -> (bs : 整数) ->(构
造子 b bs) `shiftR` 1 === bs
```

这个保留智能构造器表明，给定从sx 到sy 的稀疏化以及一个运行时无关变量x ，我们可以计算从扩展后的源作用域（sx :< x）到目标作用域（sy :< x）的稀疏化，其中 x 被保留。

```
保留 : Th sx sy -> (0 x : a) -> Th (sx :< x) (sy :< x) 保留 th x  = MkTh{
大端 = S (th .大端) , 编码 = 构造子 True (th .编码) , 不变式 =let 0 b  =
eqToSo $ testBit0Cons True (th .编码) in Keep (rewrite consShiftR
True (th .编码) in th.invariant)x}
```

外层作用域增加了一个变量，因此我们将大端递增。编码通过构造子布尔值True 记录了这个新变量被保留这一事实。最后，我们使用上面展示的两个引理，让 Idris 2 确信不变式仍然成立。

类似地，丢弃函数表明，我们可以计算一种稀疏化，从目标作用域中移除变量 x ，该变量是刚刚添加的。

```
drop : Th sx sy -> (0 x : a) -> Th sx (sy :< x) drop th x = MkTh{ 大端 =
S (th .大端) , 编码 = 构造子 False (th .编码) , 不变式 =let 0 prf = testBit0构
造子 False (th .编码) 0 nb = eqToSo $ cong not prf in 丢弃 (rewrite
consShiftR False (th .encoding) in th .invariant)x}
```

我们再次递增大端，使用构造子记录该变量正被丢弃，并使用保证其正确性的引理，让 Idris 2 确信不变式得以维持。

我们已经可以使用这些智能构造器来实现生成变薄的函数。我们以which为例。它是一个类似筛选器的函数，返回一个依赖对，其中包含满足布尔谓词的元素，以及一个证明，表明存在一个变薄嵌入，可将这些元素映射回输入的SnocList。

```
which : (a -> 布尔型) -> (sy : SnocList a) ->(sx :
SnocLista ** Th sx sy) which p [<] = ([<]**
done) which p (sy :< y) =let (sx ** th) =
which p sy in if p y then (sx :< y ** 保留 th
y) else (sx ** 丢弃 th y)
```

如果输入的 SnocList 为空，那么输出也应为空，而done会构造一个从[<]到自身的变薄。如果它非空，我们就可以对 SnocList 的尾部进行递归调用，然后根据谓词对头部的判断结果，选择保留或丢弃它。

现在，我们已经拥有了这些智能构造器，可以无缝地构造变薄。为了恢复归纳族的全部表达能力，我们还需要能够分解这些变薄。接下来我们就来解决这个问题。

5.2 对 Th 进行模式匹配Th

这个视图族是一个以变薄为索引的和类型。它为每个智能构造器都关联了一个数据构造子，并存储该构造器的参数。

```
data View: Th sx sy -> 类型 where Done : Viewdone Keep :
(th : Th sx sy) -> (0 x : a) -> View (keep th x) Drop : (th : Th
sx sy) -> (0 x : a) -> View (drop th x)
```

随附的视图函数证明，任意稀疏化都属于这三种情况之一。

```
视图 : (th : Th sx sy) -> Viewth
```

我们将完整展示视图的实现，但省略它所调用的技术性辅助引理。有兴趣的读者可以在随附材料中找到这些引理。不过，我们将在第 5.5 节检查视图擦除后编译生成的代码，以确认这些辅助定义不会带来额外的运行时开销。

我们首先对稀疏化的大端进行模式匹配。如果它是0，那么我们就知道该稀疏化必定是空稀疏化。借助名为isDone的反演引理，我们可以得到许多等式证明：编码bs 必须是0，源作用域和目标作用域sx 和sy 必须是空 snoc 列表，而不变式的证明prf 必须具有特定的形式。根据这些等式进行重写会充分改变目标类型，使类型检查器最终能够看出该稀疏化是用智能构造器done构造的，因此我们可以使用视图的Done构造器。

```
视图(MkTh0 bs prf)  =let 0 eqs
 = isDoneprf inrewritebsIsZero
eqs inrewrite fstIndexIsLineqs
inrewrite sndIndexIsLineqs
inrewrite invariantIsDoneeqs
inDone
```

如果稀疏化非空，我们就需要检查编码的第 0 位，以确定它是保留还是丢弃最局部的变量。这通过调用choose函数来完成。该函数接受一个布尔值b，并返回类型为 (Either(So b) (So (not b)) 的值；也就是说，我们不仅检查这个布尔值，还会用第 3 节中引入的So 族，在证明中记录得到的是哪个值。

```
视图 (MkTh (S i) bs prf)  = 分情况 choose(testBit bs Z) of
```

如果该位已设置，那么我们就知道这个变量被保留了。因此，我们可以调用一个反演引理，它会再次为我们提供许多等式，我们可以立即利用它们来重塑目标的类型。最终，这使我们能够构造一个子稀疏化，并使用视图的 Keep构造器。

左 所以 =>
令 0 eqs = isKeep prf so 为……，然后在其中用
fstIndexIsSnoc eqs 重写，再用 sndIndexIsSnoc eqs 重写，再用 invariantIsKeep eqs 重写，再用
isKeepInteger bs so 重写；令 th : Th eqs.fstIndexTail eqs.sndIndexTail th = MkTh i (bs ‘shiftR‘ 1)
eqs.subInvariantin cast $ Keep th eqs.keptHead

如果没有设置该位，我们就能得知该稀疏化是使用

丢弃构造的。我们可以再次使用反演引理来重排目标，最后调用该视图的 丢弃 构造器。

```
右 所以不是 =>
  令 0 等式 = 是丢弃证明 所以不是 在重写 第二索引是
  Snoc等式 在重写 不变式是丢弃等式 在重写 是丢弃整数
  bs 所以不是 在令th : Th sx 等式.sndIndexTailth =
  MkTh i (bs `shiftR` 1) 等式.subInvariant在强制转换
  $ 丢弃th 等式.保留头部
```

我们可以直接使用这个函数来实现拆解稀疏化的模式匹配函数。例如，我们可以定义 保留，这个函数用于统计 保留 在构造输入稀疏化时使用的智能构造器数量，并返回一个证明，表明这恰好等于源作用域 sx的长度。

```
保留 : Th sx sy -> (n : 自然数 ** 长度 sx === n) 保
留 th = 按 viewth 模式匹配 Done => (0 ** Refl)
Keep th x => 令 (n ** eq) = 保留 th in (S n **
cong S eq) Drop th x => 保留th
```

我们通过对输入稀疏化调用 视图函数来进行处理，它会立即告诉我们只需考虑三种情况。由于分支中精化后的类型表明 Done情况很容易处理，因为 sx 和sy 都是空 SnocList [<]，其长度显然是 0。在 Keep分支中，我们得知 sx 的形状为 (_ :< x)，因此必须返回递归调用结果的后继。最后，在 Drop 情况下，sx保持不变，因此简单地递归调用即可。请注意，该函数被正确判定为全函数，因为目标作用域 sy 确实在每一次递归调用中都以结构方式变小。它在运行时无关，但编译器仍可以成功地将其用作终止度量。

5.3 不变式关系

我们已经展示了面向用户的 Th，并声称可以定义智能构造器 done、keep和丢弃，以及一个视图函数。一旦展示 不变式 的实际定义，这一点就显而易见了。

5.3.1 不变式的定义

该关系维持记录字段 大端(a 自然数) 与 编码(an 整数) 以及索引作用域 sx 和 sy之间的不变式。其定义可以兼顾易用性与运行时效率，因为我们在静态阶段就知道所有

该不变式证明都会在编译期间被擦除。

data 不变式 : (i : 自然数) -> (bs : 整数) ->(sx, sy : SnocLista) -> 类型 where Done : 不变式 Z 0 [<] [<]Keep : 不变式 i (bs ‘ shiftR‘1) sx sy -> (0 x : a) ->{auto 0 b : So (testBit bs Z)} ->不变式 (S i) bs (sx :< x) (sy :< x) Drop : 不变式 i (bs ‘shiftR‘1) sx sy -> (0 x : a) ->{auto 0 nb : So (not (testBit bs Z))} ->不变式 (S i) bs sx (sy :< x)

一如既往，Done 构造器最为简单。它表示大小为 Z 且编码为位模式 0 的稀疏化为空稀疏化。

Keep 构造器保证，大小为（S i）且编码为bs的稀疏化表示从（sx :< x）到（sy :< x）的单射，前提是 Z 处的bs 位已置位，并且位模式的其余部分（通过对 bs右移得到）是从 i 到 sx 到 sy的有效稀疏化。

丢弃构造器的结构与此相同，只不过它要求位置 Z 处不设置该位。

我们可以轻松地利用这个关系来证明，一些基本编码是有用稀疏化的有效表示。

5.3.2 不变式证明示例

例如，我们总能定义一个从空作用域到任意作用域 sy的稀疏化。

```
none : (sy : SnocLista) -> Th [<]
synonesy = MkTh(长度 sy) 0 (none sy)
```

此稀疏化的编码为 0 ，因为每个变量都被丢弃，其大端是外层作用域 sy的长度。此编码有效性的证明由下面的 none 引理给出。我们再次利用 Idris 2 的重载机制，为不同类型中作用相似的函数赋予相同的名称。

```
none : (sy : SnocList a) -> 不变式(长度 sy) 0 [<] synone
[<] = Donenone (sy :< y) = 丢弃 (nonesy) y
```

证明通过对外层作用域 sy进行归纳来完成。如果它为空，我们只需使用空稀疏化的构造器。否则，可以将丢弃 应用于归纳假设。由于 (testBit0 Z) 求值为 False，因此 Idris 2 的证明搜索（参见第3.2节）可以自动构造 nb 证明，而 (0 ‘shiftR‘1) 求值为 0，这意味着归纳假设的类型恰好符合要求。

恒等变薄的定义稍微复杂一些。对于大小为 n 的作用域，我们需要生成一个由 n 个 1 组成的位模式。我们分两步定义它。首先，cofull定义一个位模式：先是 k 个 0，后接无限多个 1；方法是将仅由 1 组成的位模式向左移动 k 位。然后，对 cofull 取补码，得到 cofull。

```
cofull: 自然数 -> 整数cofull n  =
oneBits `左移` n
```

```
full : 自然数-> 整数full n  = 补
码(cofull n)
```

接着，我们可以通过配对（ n full n）来定义大小为 n）作为编码和 n 作为大端。

```
ones : (sx : SnocLista) -> Th sx sx
ones sx  = 令 n : 自然数; n  = 长度sx 于 MkTh n (full n) (ones sx)
```

大部分工作依然集中在同名引理中，该引理证明了这种编码是有效的。

```
ones : (sx : SnocList a) ->令 n  = length sx in Invariant n
(full n) sx sx ones [<] = Done ones (sx :< x)  =令 0 nb  =
eqToSo (testBitFull(S (length sx)) Z) in Keep (rewrite
shiftRFull (length sx) in ones sx) x
```

这个证明再次通过对作用域进行归纳法来完成。如果作用域为空，那么又可以使用空稀疏化的构造子。在非空的情况下，我们首先借助一个辅助引理（此处未展示）来构造证明 nb ，以说明对于一个非零Z 位置上的位，full 整数的值已知为 True。接下来，我们需要使用另一个引理，对提及（full(length sx)）的归纳假设进行类型转换，使其能用于需要一个关于（full (length (sx :< x)) `shiftR` 1）证明的位置。

5.3.3 不变式关系的性质

这个关系具有许多方便的性质。

首先，它是证明无关的：对于同一组 i、bs、sx 和 sy相关的任意两个证明，都可以证明它们相等。因此，Th 值相等，等价于 大端 和 编码值相等。特别是，判断给定的稀疏化是空稀疏化还是恒等变薄，开销很低。

其次，它可以被逆向还原 [CT95] 只需知道两个比特：自然数是否为空，以及编码中位置 Z 处的位值。这使我们能够通过这两项检查，再将视图函数的不变式证明逆向还原，从而获得余下部分的稀疏化编码有效的证明。我们将在第 5.5 节看到，这样就能为视图生成高效的运行时代码。

5.4 选择你自己的抽象层级

既能访问高层的视图，又能访问内部的不变式关系，这意味着程序员可以选择自己想要采用的抽象层次。他们可能需要显式操作比特，以实现性能关键路径上使用的关键运算；但对于不那么重要的操作，或在证明运行时无关的性质时，也可以停留在最高抽象层。

在上一节中，我们在定义 无（使用常量位模式 0）和 全 1 时，看到了这些位操作的简单示例：通过位移和补码，构造出一个由 1 组成、后接 0 的初始段。

其他自然的示例还包括两个共享同一较宽作用域的变薄之间的 交 和 并 。例如，可以将并视为一个函数：对第一个变薄作归纳法，对第二个变薄作分类讨论；只要任一输入产生 保留构造器，就输出该构造器。或者，我们也可以注意到，并的位模式恰好是两个输入各自位模式的析取，并改为证明一个关于 不变式关系的引理。可以将其可视化如下。在每一列中，只要任一输入是 •，交就是一个



并尤其重要，因为将一个项的“打开”视图转换为其 co-de Bruijn 对应形式时就会用到它。正如前面提到的，任意作用域中的 co-de Bruijn 项表示为一对内容：一个由其精确支撑集索引的项，以及一个将该支撑集嵌入回较宽作用域的变薄嵌入。使用这种表示时，能够访问一种“打开”的视图会很方便：它将外部变薄推入内部，从而显露项的顶层构造子，以便进行分类讨论。

下图展示了使用视图的“展开”应用节点（菱形“$”节点，两个子项都位于外部作用域中）与其 co-de Bruijn 形式（圆形“$”节点，带有一个用于选取项支撑集的外部稀疏化）之间的对应关系。



co-de Bruijn 项的外部稀疏化，正是通过计算“展开”应用中函数和参数各自的外部稀疏化的并得到的。

这些显式位操作会在编译过程中保留，从而生成效率更高的代码。

5.5 编译后的代码

以下代码块展示了编译 `view` 函数时生成的 JavaScript 代码。我们选择使用 JavaScript 后端，而不是例如 ChezScheme 后端，因为它生成的代码相当易读。我们修改了后端，使其也能写入注释，提醒读者所定义函数的类型，以及自然数标签对应的数据构造器。这些改动现已在 Idris 2 当前的开发版本中向所有人开放。

我们唯一进行的手动修改，是内联与 **case** 块对应的函数、重命名变量和属性名使其更易读、引入 $tail 定义以缩短代码行，以及对布局进行少许调整。

```
/* Thin.Smart.view: (th : Th sx sy) -> View th */ function Thin_
Smart_view($th) { switch($th.bigEnd) { case 0n: return {h: 0 /*
Done */}; default: { const $predBE = ($th.bigEnd-1n); const $test
 = choose(notEq(($th.encoding&1n), 0n)); switch($test.tag) {
case 0: /* 左 */ { const $tail = $th.encoding>>1n; return { tag: 1 /*
保留 */ , val: {bigEnd: $predBE, encoding: $tail}}; } case 1: /* 右 */
{ const $tail = $th.encoding>>1n; return { tag: 2 /* 丢弃 */ , val:
{bigEnd: $predBE, encoding: $tail}}; } }}}}
```

读者可以看到，编译过程抹去了所有指标，以及用于证明将高效运行时表示与高级规范联系起来的不变式得以保持的证明。稀疏化在运行时表示为一个 JavaScript 对象，具有两个属性，分别对应于 `Th` 的运行时相关字段： 大端和 编码。两者都存储一个 JavaScript大整数（一个对应于 自然数，另一个对应于 整数）。例如，稀疏化 [01101]在运行时会表示为 `{ bigEnd: 5n, encoding: 13n }`。

该视图分两步进行。首先，如果 大端为 `0n` ，那么我们就知道稀疏化为空，可以立即返回 `Done`构造器。否则，我们知道稀疏化非空，因此可以通过将非零的 大端减一，计算其尾部的大端（`$predBE`）。然后，我们可以检查位置 `0` 上的位，以决定返回 `Keep`还是 丢弃 构造器。具体来说，我们使用位掩码将其他所有位清零（`$th.bigEnd&1n`），并检查结果是否为零。如果不等于 0，我们就输出

保留，并通过将原始编码移位以丢弃第 0 位，计算稀疏化的 `$tail` 。否则，我们输出丢弃，并计算相同的尾部。

对这个视图进行 [01101]稀疏化后，我们会得到 (保留 [0110])，也就是说`{ tag: 1, val: { 大端: 4n, 编码: 6n } }`。

得益于 Idris 2 对定量类型论的实现，我们成功构造出一种高级表示，它可以像经典归纳族一样通过智能构造器和视图进行操作，同时又丝毫不放弃对其运行时表示的控制。

剩下的问题，例如我们构造视图的构造器后立即又将其拆开，从而造成不必要的分配，可以通过重用 Wadler 的分析（ [Wad87] 的第 12 节）来解决。

6 结论

我们已经看到，归纳族通过强不变量为程序员提供了排查错误的方法。遗憾的是，尽管现有的优化遍会擦除冗余或运行时无关的数据，这些归纳族仍可能妨碍生成高性能代码。这种矛盾促使我们利用定量类型论来设计一个兼具两全之美的库：既拥有归纳族的强不变量和易用性，又具备显式位操作的运行时性能。

6.1 相关工作

由于历史和易用性方面的原因，Coq 中的惯用代码往往以用一种与 OCaml 非常接近的语言子集编写的程序为中心，然后在运行时无关的 Prop 片段中证明这些程序的性质。当未精化输入迫使用户考虑本应不可能出现的情况时，这可能导致笨拙的编码。常见的应对策略包括放宽类型以引入少量部分性，例如返回一个选项类型，或增加一个输入参数作为默认返回值。这种做法完全忽视了类型驱动开发的要义。在交互式编辑过程中，尽可能多地获取信息会带来很大帮助。这些信息不仅能确保我们始终维持关键不变量，从而使无效状态无法表示，极大地帮助我们正确编写定义；也让程序员能够使用类型驱动工具和自动化。值得庆幸的是，Equations [Soz10、SM19]can 等库可以帮助用户编写更多依赖类型程序，同时替用户处理 Coq 中所需的复杂编码。采用类似于我们的方法、但使用 Prop 而非零数量的基于视图的方法，应该也是可行的。我们预计，以这种方式在 Coq 中编码的视图会有更糟糕的计算行为，因为 Equations 会使用复杂的 elaboration 过程，将依赖模式匹配编码为 Gallina。不过，Coq 确实受益于良好的自动化支撑集，可用于展开引理、反演原理，以及通过等式进行重写

而这些等式或许能够弥补编码所带来的不便。

关于擦除的先前工作 [Tej20] 有一个优点：能够对代码进行全自动分析。其主要不便之处在于，用户无法明确声明某段数据应当是运行时无关的，因此可能会无意中使用它，从而阻止其被擦除。定量类型论允许我们用户明确选择哪些内容与运行时相关、哪些无关，而数量检查器会确保我们言行一致。这应能确保生成的程序具有可预测性高得多的复杂度。

Brady、McKinna 和 Hammond 在电路设计的背景下探索过一个有些相关的想法 [BMH07]。在他们的验证工作中，他们用一个一元自然数所表示的含义，为一种高效表示（将自然数表示为比特列表）建立索引。所有操作都构造即正确，这一点可以从他们使用对应的一元操作作为类型级规范看出。最终，他们的算法仍然处理归纳族，而不是直接处理二进制数。在他们的设定中，这样做合情合理，因为他们要构造电路，因此会显式操作承载比特的导线。相比之下，在我们的示例中，我们真正想要的是直接处理实际的（无界）整数，而不是由比特组成的链表。

6.2 局限性与未来工作

总体而言，我们发现这项使用 Idris 2 开展的案例研究非常令人鼓舞。Idris 2 是一种基于定量类型论的先进语言。该语言的实现仍处于实验阶段（例如，参见附录 B，其中列出了一些我们发现的错误），但这些问题都不是固有的局限性。我们希望能进一步推进这项工作，解决以下局限性，并探索更高级的用例。

6.2.1 局限性

遗憾的是，从命题上来说，只有 (视图 (保留 thx)) 归约为 (Keep th x) 才是真的（done/Done和 drop/丢弃) 的情况也类似）。这意味着，用户在证明通过对调用视图函数的结果进行模式匹配而定义的函数的性质时，可能需要手动应用这些引理。如果能够像 Agda 中那样，将用户证明的等式扩展到 Idris 2 的归约规则中（Cockx、Tabareau 和 Winterhalter 对此进行了形式化研究），这一麻烦便可消除 [CTW21]。

在本文的案例研究中，我们能够设计出核心不变式关系，将不变量显式化，并以可证明地具有证明无关性的方式实现。这一点在 Idris 2 当前实现的类型论下未必总是可能。为命题添加一种证明无关的排序（例如参见 Altenkirch、McBride 和 Swierstra 的工作 [AMS07]）或许可以一劳永逸地解决这一问题。

好在，Idris 2 标准库为我们提供了一个完善的纯函数接口，让我们能够显式地操作整数的比特。不过，这些内置操

作没有附带任何内置性质。因此，我们不得不公设一组（最小的）公理（见附录 A），并自行证明许多有用的推论。对于其他底层操作，支撑集则更少，例如读取只读数组或操作指针。

我们还发现，使用运行时无关性（`0` 数量）有时会让人颇感挫折。目前，只有当编译器能够明确判定某个运行时无关值必然是通过该族的某个构造子构造出来时，才能在运行时相关的上下文中对其进行模式匹配。遗憾的是，在非平凡的情况下，这一点往往只能证明，而非显而易见。因此，我们不得不费尽周折来满足数量检查器，最终还要定义复杂的反演引理来绕过这些限制。结合改进类型检查器和利用先前自动化反演的思路进行元编程，或许可以解决这一问题 [CT95, McB96, Mon10]。

6.2.2 未来工作

我们计划进一步探索配备高级接口的内存映射表示。

我们已经取得实验结果，证明可以使用只读数组作为二叉搜索树的运行时表示。搜索可以实现为一个正确性已获证明的高级判定过程，它看起来是在递归地遍历这棵“树”。然而在运行时，这实际上会像经典的二分法搜索一样，在数组上执行。

更广泛地说，我们预计，LoCal [VKR+19] 中借助编译器的特定支撑集完成的、针对序列化数据的许多工作，都可以原样移植到基于 QTT 的编程语言中。事实上，QTT 的类型系统足够强大，可以完全通过库代码来追踪这些不变量。

短期内，我们希望设计一种小型嵌入式领域专用语言，让用户能够更轻松地构建和拆解以本文所述方式高效表示的积与和。阶段化有助于确保使用这种嵌入式领域专用语言不会产生任何运行时开销。我们计划为 Idris 2 添加由类型保证的阶段化，从而使其真正成为我们项目的理想宿主语言。

我们的长期计划是超越只读数据，研究如何利用分离逻辑证明命令式程序的正确性，并探究能否将这类事后推理更多地融入类型中，从而实现一种高层次、正确性由构造保证的编程风格，同时保持相同的运行时行为。

致谢 我们感谢 Conor McBride 与我们讨论 TypOS 中所用不安全编码的细节，也感谢 James McKinna、Fredrik Nordvall Forsberg、Ohad Kammar 和 Jacques Carette 对本文早期版本提出宝贵的意见和建议。

引用

[AAM+22]Guillaume Allais、Malin Altenm¨uller、Conor McBride、Georgi Nakov、Fredrik Nordvall Forsberg 和 Craig Roy。TypOS：用于类型检查执行体的操作系统。载于第 28 届《证明与程序的类型》国际会议（TYPES 2022），2022年6月20日至25日，南特，法国，2022年。[AMS07] Thorsten Altenkirch、Conor McBride 和 Wouter Swierstra。观测等式，现在！载于 Aaron Stump 和 Hongwei Xi 编，《ACM“编程语言与程序验证相遇”研讨会论文集》，PLPV 2007，弗赖堡，德国，2007年10月5日，第57–68页。ACM，2007年。[Atk18] Robert Atkey。定量类型论的语法与语义。载于 Anuj Dawar 和 Erich Gr¨adel 编，《第33届 ACM/IEEE计算机科学逻辑研讨会论文集》，LICS 2018，牛津，英国，2018年7月9日至12日，第56–65页。ACM，2018年。[BMH07] Edwin C. Brady、James McKinna 和 Kevin Hammond。构造正确电路：利用依赖类型验证硬件规范的函数式方面。载于 Marco T. Moraz´an 编，《第八届函数式编程趋势研讨会论文集》，TFP 2007，纽约市，纽约州，美国，2007年4月2日至4日，《函数式编程趋势》第8卷，第159–176页。Intellect，2007年。[BMM03] Edwin C. Brady、Conor McBride 和 James McKinna。归纳族无需存储其指标。载于 Stefano Berardi、Mario Coppo 和 Ferruccio Damiani 编，《证明与程序的类型》国际研讨会，TYPES 2003，都灵，意大利，2003年4月30日至5月4日，修订精选论文，《计算机科学讲义》第3085卷，第115–129页。Springer，2003年。[Bra21] Edwin C. Brady。Idris 2：实践中的定量类型论。载于 Anders Møller 和 Manu Sridharan 编，第35届欧洲面向对象编程会议，ECOOP 2021，2021年7月11日至17日，奥胡斯，丹麦（线上会议），LIPIcs 第194卷，第9:1–9:26页。Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，2021年。[CA20] Jesper Cockx 和 Andreas Abel。展开依赖（余）模式匹配：不遗漏任何模式。《函数式编程杂志》，30:e2，2020年。[CDT 22] Coq开发团队。Coq证明助手参考手册，8.15.2版，2022年5月。[Cha09] James Maitland Chapman。类型检查与正规化。博士论文，诺丁汉大学，2009年7月。

[CT95] Cristina Cornes 和 Delphine Terrasse。自动化求逆 Coq 中的归纳谓词。载于 Stefano Berardi 和 Mario Coppo（编），《证明与程序的类型》，TYPES'95 国际研讨会，意大利都灵，1995年6月5日至8日，论文选集，《计算机科学讲义》第1158卷，第85–104页。Springer，1995年。[CTW21] Jesper Cockx、Nicolas Tabareau 和 Th´eo Winterhalter。驯服 rew：一种带有计算假设的类型论。《ACM程序设计语言会议录》，5(POPL):1–29，2021年。[dB72] Nicolaas Govert de Bruijn。带无名占位符的 λ 演算记法：用于公式自动处理的工具，并应用于丘奇-罗瑟定理。《数学探究》（会议录），75(5):381–392，1972年。[Dyb94] Peter Dybjer。归纳族。《计算机形式化方面》，6(4):440–465，1994年。[McB96] Conor McBride。求逆 LEGO 中归纳定义的关系。载于 Eduardo Gim´enez 和 Christine Paulin-Mohring（编），《证明与程序的类型》，TYPES'96 国际研讨会，法国奥苏瓦，1996年12月15日至19日，论文选集，《计算机科学讲义》第1512卷，第236–253页。Springer，1996年。[McB16] Conor McBride。我一无所有。载于 Sam Lindley、Conor McBride、Philip W. Trinder 和 Donald Sannella（编），《改变世界的成功案例集——献给Philip Wadler的论文集：值此其60岁生日之际》，《计算机科学讲义》第9600卷，第207–233页。Springer，2016年。[McB18] Conor McBride。每个人都得有个归处。载于 Robert Atkey 和 Sam Lindley（编），《第七届数学结构化函数式编程研讨会论文集》，MSFP@FSCD 2018，英国牛津，2018年7月8日，《EPTCS》第275卷，第53–69页，2018年。[MEL+21] Krzysztof Maziarz、Tom Ellis、Alan Lawrence、Andrew W. Fitzgib- bon 和 Simon Peyton Jones。模 α 等价哈希。载于 Stephen N. Freund 和 Eran Yahav（编），PLDI '21：第42届 ACM SIGPLAN 编程语言设计与实现国际会议，线上会议，加拿大，2021年6月20日至25日，第960–973页。ACM，2021年。[MM04] Conor McBride 和 James McKinna。从左侧看。《函数式编程杂志》，14(1):69–111，2004年。[Mon10] Jean-Fran¸cois Monin。证明技巧：小型求逆。载于 Yves Bertot（编），第二届Coq研讨会，英国爱丁堡，2010年7月。Yves Bertot。

[SM19] Matthieu Sozeau 和 Cyprien Mangin。Equations 重装上阵：高级依赖类型函数式编程与 Coq 证明。《ACM程序设计语言会议录》，3(ICFP):86:1–86:29，2019年。[Soz10] Matthieu Sozeau。Equations：依赖模式匹配编译器。载于 Matt Kaufmann 和 Lawrence C. Paulson（编），《交互式定理证明》，第一届国际会议，ITP 2010，英国爱丁堡，2010年7月11日至14日。论文集，《计算机科学讲义》第6172卷，第419–434页。Springer，2010年。[Tej20] Mat´uˇs Teji ˇs ˇc´ak。带模式匹配和擦除推断的依赖类型演算。《ACM程序设计语言会议录》，4(ICFP):91:1–91:29，2020年。[VKR+19] Michael Vollmer、Chaitanya Koparkar、Mike Rainey、Laith Sakka、Milind Kulkarni 和 Ryan R. Newton。LoCal：一种用于处理序列化数据的程序语言。载于 Kathryn S. McKinley 和 Kathleen Fisher（编），第40届 ACM SIGPLAN 编程语言设计与实现会议论文集，PLDI 2019，美国亚利桑那州菲尼克斯，2019年6月22日至26日，第48–62页。ACM，2019年。[Wad87] Philip Wadler。视图：让模式匹配与数据抽象共存的方法。载于第十四届 ACM 编程语言原理研讨会会议记录，德国慕尼黑，1987年1月21日至23日，第307–313页。ACM Press，1987年。[WB89] Philip Wadler 和 Stephen Blott。如何让特设多态不那么特设。载于第十六届 ACM 编程语言原理研讨会会议记录，美国得克萨斯州奥斯汀，1989年1月11日至13日，第60–76页。ACM Press，1989年。

关于 **比特**接口的公设引理

用比特来推理整数通常更方便。我们将按位相等定义为根据`testBit`进行逐点相等。

```
(~~~) : 整数-> 整数 -> 类型bs  ~~~ cs  = (i : 自然数) -
> testBitbs i  === testBit cs i
```

我们的第一个公设是一种外延性原理，指出按位相等蕴含命题相等。

```
外延地:  {bs, cs : 整数} -> bs  ~~~ cs -> bs  === cs
```

一旦与说明各种原语在位级上的行为的公理结合起来，这就为我们提供了强大的推理原则。这就是为什么其余几乎所有公理都用 testBit调用来表示。

A.1 逻辑运算

我们的第一组公理将整数上的逻辑运算与其布尔对应项联系起来。这实质上是在说明，这些运算是按位进行的。

```
testBitAnd: (bs, cs : 整数) -> (i : 自然数) ->testBit(bs .&.
cs) i  === (testBit bs i && testBit cs i)
testBitOr: (bs, cs : 整数) -> (i : 自然数) ->testBit(bs .|.
cs) i  === (testBit bs i  || testBit cs i)

testBitComplement: (bs : 整数) -> (i : 自然数) ->
testBit (补码bs) i  === not (testBit bs i)
```

结合上文提到的外延性原理，我们已经可以证明，例如，二元运算符满足交换律和结合律、德摩根定律成立，或者合取对析取满足分配律。

A.2 位移

第二组公理描述了左移和右移对位模式的作用。

大小为 k 的右移会丢弃 k 个最低有效位；因此，测试右移后整数的第 i 位，等价于测试原整数的第（k + i）位。

```
testBitShiftR: (bs : 整数) -> (k : 自然数) ->(i : 自然数) ->
testBit (bs `shiftR`k) i  === testBit bs (k + i)
```

左移会添加 k 个初始化为 0 的新最低有效位；因此，测试左移后整数的第 i 位时，如果 i 严格小于 k，则返回 False；否则返回原整数位置（i - k）处的位。

为简洁起见，我们通过一方面考虑非零左移的效果，另一方面考虑左移 0 个比特是恒等函数这一事实，来陈述这些结果，而不提及“严格小于”关系。

```
testBit0左移: (bs : 整数) -> (k : 自然数) ->
testBit (bs `左移`S k) Z  === False
testBitSShiftL: (bs : 整数) -> (k : 自然数) -> (i : 自然数) ->testBit
(bs `左移`S k) (S i)  === testBit (bs `左移` k) i
```

```
shiftL0 : (bs : 整数)-> (bs `左移` 0)  === bs
```

A.3 位测试

最后一组公理规定了某一位被置位时会发生什么。

测试未被置位的其他位，相当于在原始整数上测试该位。

```
testSetBitOther: (bs : 整数) -> (i, j : 自然数) -> 非 (i  === j) ->
testBit (setBitbs i) j  === testBit bs j
```

最后，我们有一个公理说明整数 (位 i)（即 2^i）非零。

```
bitNonZero: (i : 自然数) -> (位 i  == 0)  === False
```

B Idris 2 的当前局限

这项由 Jacques Carette 提出的挑战，凸显了 Idris 2 当前的一些局限。

B.1 问题陈述

目标是使用 Vect类型（定义于第 3.3 节），并定义一个视图来逆转向量追加。这是依赖类型编程中的经典练习；有趣的问题是，我们能否用这种编码同样无缝地实现该函数。

向量追加可以通过对第一个向量进行归纳轻松定义。

```
(++): Vect m a -> Vect n a -> Vect (m  + n)
axs@_ ++ ys with (视图 xs)_ | [] = ys_ | hd ::
tl = hd :: (tl  ++ ys)
```

如果第一个向量为空，我们可以直接返回第二个向量。如果它以 cons 构造子开头，我们可以返回头部，并通过递归调用计算尾部。

有了这个定义，我们就可以声明一种视图类型，并仿照其处理列表的弱类型对应形式，将其称为SplitAt。它表明，长度为xs 的向量p 可以在m 处分割，前提是p 恰好为（m + n），且xs恰好为（pref ++ suff），其中pref 和suff 的长度分别为m和n。

```
数据 SplitAt: (m : 自然数) -> (xs : Vect p a) -> 类型 其中
MkSplitAt: (pref : Vect m a) -> (suff : Vect n a) ->SplitAtm (pref
 ++ suff)
```

难点在于定义一个函数，用来证明大小为（m + n）的向量可以在m处分割。

B.2 失败的尝试

证明必然要对 m 使用归纳法，然后对输入向量进行分类讨论，并在非零情况下进行递归调用。

我们的第一次失败尝试成功地分割了自然数，并调用向量上的视图xs 将其拆开，但随后在对 splitAt 进行递归调用时失败。

失败 “在此上下文中 tl 不可访问”

```
splitAt : (m : 自然数) -> (xs : Vect (m + n) a) -> SplitAtm xs 分割
At 0 xs = MkSplitAt [] xs 分割At (S m) xs@_ with (视图 xs)
_ | hd :: tl@_ with (分割At m tl)_ | res = ?a
```

这揭示了 Idris 2 在处理 @ 模式与数量之间相互作用时存在的问题：编译器随意决定让别名 tl 在运行时无关，随后却抱怨当我们要执行递归调用（tl 时，splitAtm tl）无法访问它！

为了绕过这一限制，我们决定放弃 @ 模式，亲自编写完全显式的子句，并使用点模式标记被强制的表达式。

失败 “无法匹配 ?postpone [作用域中没有局部变量]（用户点模式）”

```
splitAt : (m : 自然数) -> (xs : Vect (m + n) a) -> SplitAt m xs
splitAt 0 xs = MkSplitAt [] xs splitAt (S m) xs@_ 使用视图
xs)_ | hd :: tl 使用 (splitAt m tl) splitAt (S m) .(hd :: (pref ++
suff))| hd :: .(pref ++ suff)| MkSplitAtpref suff = ?a
```

左侧现在可以通过类型检查，但案例树构建器却因一个令人费解的错误而失败。这暴露了 Idris 2 在将模式匹配函数 elaboration 为案例树时的实现存在一个 bug。构建案例树时，它没有忽略点表达式（这些表达式是强制的，因此它们提及的变量必定已在另一个模式中绑定），反而试图用它们来驱动分类策略。这是一个已有深入研究的问题，参考 Cockx 和 Abel 的工作 [CA20]应该就能修复。

B.3 绕过 Idris 2 的局限

由此，我们得到了当前的解决方案。有些矛盾的是，为了绕过 Idris 2 中的这些 bug，我们反而找到了一个原则性更强的方案：将模式匹配步骤中用于调整递归调用返回结果的操作，抽象到一个辅助函数中，其类型清楚地说明了其中的处理过程。

对于 xs 的一个 m 分割，我们可以通过将 x 构造到前缀上，轻松计算出 x :: xs 的一个 (S m) 分割。

```
(::) : (x : a) -> SplitAt m xs -> SplitAt (S m) (x :: xs)
x :: MkSplitAt pref@(MkVect _ Refl) suff
  = MkSplitAt (x :: pref) suff
```

在这个辅助函数中，`xs` 显然是运行时无关的，因此案例分割器不会尝试检查它，从而生成正确的案例树。我们必须进一步对 `pref` 进行匹配（具体来说，要令相等性证明为 `Refl`），以便在类型层级上进行恰到好处的计算，让类型检查器确认各部分确实吻合。若这里使用一种证明无关的命题相等类型，就能有所帮助。

我们可以将这些部分组合起来，最终得到 `splitAt` 视图。

splitAt: (m : 自然数) -> (xs : Vect (m + n) a) -> SplitAt m xs
splitAt0 xs = MkSplitAt [] xs splitAt(S m) xs@_ with (视图 xs)_ | hd :: tl = hd :: splitAt m tl

我们确实想再次强调，这些限制并非该方法本身固有的局限，只是当前 Idris 2 语言实验性实现中的缺陷，而且这些问题能够也应该得到解决。