arXiv:2301.12852v1 [cs.PL] 30 Jan 2023

类型理论作为语言工作台

Jan de Muijnck-Hughes[1],Guillaume Allais[2],和 Edwin Brady[2]

[1]格拉斯哥大学[2]圣安德鲁斯大学

2023年1月31日

摘要

语言工作台为语言设计者提供了一个表达力丰富的环境，让他们能够创建自己的领域特定语言（领域特定语言）。同样，关于机械化元理论的研究表明，依赖类型语言提供了表达力丰富的环境，可用于形式化和研究领域特定语言及其元理论性质。但是，我们能否声称依赖类型语言也符合语言工作台的定义？我们的回答是肯定的！

我们开发了一种名为 V´elo 的示例领域特定语言，它不仅展示了如何运用依赖类型技术来实现和操作中间表示（IR），也展示了依赖类型语言如何成为出色的语言工作台。V´elo 是一种简单的、经过验证的语言，支持类型正确的空洞，并配备了完整的编译器流水线：解析器、细化器、交互式解释器、求值器和编译器遍。具体而言，我们介绍了带有类型正确空洞支撑集的中间表示设计方案，说明如何在带类型环境中实现公共子表达式消除（CSE），以及 V´elo 的机械化类型健全性证明如何成为求值器的来源。

1 引言

语言工作台（例如 Spoofax） [28], 为语言设计者提供了一个富有表现力的环境，用于设计、实现和部署领域特定语言（DSL） [16]。概括而言，语言工作台 [15]是一种支持以下功能的工具：描述语言的记法——即如何向用户呈现语言的具体语法；实现语言的语义——即如何实现语言的行为；以及通过编辑器与用户交互。除这些核心标准外，各种语言工作台还支持语言验证、测试和组合。

与此同时， [7, 1]机械化元理论研究计划已促使编程语言理论领域开发出大量工具和技术。尤其是，诸如

Idris 2 [9], Agda [21], 和 Coq [26] 已被广泛用于形式化领域特定语言，并研究其元理论性质。依赖类型允许类型依赖于值——也就是说，类型是一等公民——并提供了一个富有表现力的环境，让我们能够推理并编写程序。使用依赖类型语言开展的工作，范围从研究特定的核心演算 [4, 24, 10]到构建通用推理框架 [25, 3]。然而，这些机械化软件验证项目通常止步于构建前端，未能让用户运行经过验证的语言实现。如果我们的语言实现通过了类型检查，那我们不妨也把它们发布出去！Idris 2 通过成为自身的实现语言，成功证明这并非无法避免的结局 [9]。但现在，我们能否声称依赖类型语言也称得上语言工作台?

V´elo[1] 是一种极简的函数式语言，我们用 Idris 2 实现了它，以展示如何运用依赖类型技术来实现和操作中间表示（中间表示）。本文介绍 V´elo，但最重要的是，旨在说明依赖类型语言能够成为优秀的语言工作台。我们探讨了语言工作台挑战 [15]所强调的、构成语言工作台的核心标准以及一些可选扩展。尽管依赖类型语言尚未满足所有可选标准，但我们相信，通过额外的工程工作（例如利用现有成果 Quickchick [18]），便能满足更多可选标准。

语言工作台（例如 Spoofax）的另一项关键原则，是创建语言的便捷性。同样地，我们开发了一系列可复用模块，封装了许多语言共有的功能，从而减少了在 Idris 2 中创建嵌入式领域特定语言（嵌入式领域特定语言）时所需的样板代码 。

尽管我们努力让本文中的依赖类型编程介绍更易于理解，但有兴趣的读者还可以参考更多入门资料 [29, 8]。

2 Velo 简介

V´elo 背后的设计刻意保持简单直接：它是在简单类型 λ 演算（STLC）的基础上扩展了 let 绑定、布尔值及其合取，以及自然数及其加法。为了支持交互式编辑，V´elo 还支持类型正确的空洞。下面展示了一个 V´elo 程序示例，其中包含一个被多次使用的空洞，以及一段交互式解释器会话记录，其中列出了当前的空洞集合。

[1]我们提供了可复现的制品及其源代码作为补充材料。

```
letb = false in let double= (fun
x: nat => (add x x)) in let x =
(double ?hole) in (double ?hole)
```

```
Velo> : 空洞 b：布
尔值 double：自然
数 -> 自然数 ---------- ?
空洞：自然数
```

Velo 的轻量级语言设计′帮助我们更好地展示如何将依赖类型语言用作语言工作台 [17]。无论语言的复杂程度如何，Velo′仍然是一门完整的语言，具备标准的编译器流水线和交互式解释器。领域特定语言捕获语言的具体语法，而解析器则将领域特定语言实例转换为原始的未检查项。双向类型检查将具体语法中的类型注解降至最低，并有助于将原始的无类型项更好地细化为一组类型良好的中间表示：`Holey` 用于支撑作用域良好的带类型空洞；以及 `Terms` 作为捕获语言抽象语法的核心表示。我们在第 3 节介绍中间表示设计中有趣的方面。此外，细化还会执行标准的语法去糖，例如将 let 绑定转换为函数应用，从而减小核心表示的规模。基于核心表示，我们还使用余 De Bruijn 索引提供作用域良好的公共子表达式消除（CSE）（第 4 节），并提供经验证的求值器，将项归约为值（第 5 节）。

3 语言设计

我们首先详细说明在 Idris 2 中实现 V′elo 静态语义的关键设计考量。我们选择为 V′elo 提供外部具体语法（领域特定语言，DSL），供用户编写程序。借助依赖类型语言，我们还可以直接在宿主语言 [6] 中，将抽象语法及其静态语义表示为内在作用域明确且带类型的嵌入式领域特定语言（EDSL）。也就是说，我们以特定方式设计数据结构，使其只能表示作用域和类型均正确的项；相应地，作用域检查和类型检查阶段也就能保证已拒绝无效的用户输入。为使阐述简洁，我们聚焦于该语言的核心子集。感兴趣的读者可在随附材料中找到完整定义。

类型通常通过其上下文无关文法来引入。左侧展示了该文法，它为用户提供了两种基本类型（Nat 和 Bool）以及一个用于构造函数类型的类型构造子（· → ·）。右侧则给出了它们在 Idris 2 中作为归纳类型的内部表示。[2]

$$
\begin{array}{rcl}
t : \mathrm{TYPE} & ::= & \mathrm{Nat} \\
& | & \mathrm{Bool} \\
& | & t \to t
\end{array}
$$

```
data Ty = TyNat
        | TyBool
        | TyArr Ty Ty
```

[2]本文中的 Idris 2 代码片段均使用语义高亮器自动渲染。关键字以**粗体**排版，类型以蓝色显示，数据构造子以红色显示，函数定义以绿色显示，已绑定变量以紫色显示，注释以灰色显示。

上下文也可以用类似的方式通过上下文无关文法来定义：上下文可以是空的（ϵ），也可以是已有的上下文（Γ）在右侧用逗号扩展后，添加一个新的类型赋值（$x : t$）。在 Idris 2 中，我们采用无名表示，因此使用由类型构成的 `SnocList` 来表示这些上下文（也就是从右侧增长的列表）。请注意，Idris 2 编译器会自动支持列表和后缀列表的语法糖：`[1,2,3]`表示从 `1` 递增到 `3` 的列表，而 `[<1,2,3]` 则是对应的后缀列表，按递减顺序排列。特别是，`[<]` 表示空后缀列表，也称为 `Lin`。

$$\Gamma : \text{CONTEXT} ::= \epsilon \mid \Gamma, x : t$$

```
数据 SnocList a =线性
  | (:<) (SnocList a) a
```

类型判断 由关系给出，并在 Idris 2 中使用归纳族编码；归纳族是归纳类型的一种推广形式 [14]。每条规则都会成为该族的一个构造子，因此每个证明 $\Gamma \vdash t : a$ 都会对应一个类型为 (Term Γ a) 的项 t 。左侧展示了两个判断：上下文成员关系和类型判断；右侧则给出了相应的归纳族声明。

$$\Gamma \ni x : a$$
$$\Gamma \vdash t : a$$

```
数据 Elem : (gamma :SnocList ty) -> (a: ty) -> 类型
数据 Term : (gamma :SnocList Ty) -> (a: Ty) -> 类型
```

我们将在下一节给出 `Elem` 的定义，这里则重点讨论 `Term`。最基本的类型规则是公理，它们没有前提，并映射到不带参数的构造子。我们使用 Idris 2 注释（`--`）来格式化构造子的类型，使其看起来像相应的推理规则。这里展示了说明 0 是自然数的规则，以及它对应的 `Zero` 构造子。

$$\frac{}{\Gamma \vdash \text{Zero} : \text{NAT}}$$

```
零: ------------------

Term gamma 自然数类型
```

接下来是只有一个前提、且该前提不是关系本身的子推导的类型规则。它们对应于只有一个参数的构造子。这里展示变量的类型规则：给定一个证明，表明上下文中的某处存在一个类型为 a 的变量，我们就可以在该上下文中构造一个类型为 a 的项。

$$\frac{\Gamma \ni x : a}{\Gamma \vdash x : a}$$

```
Var :  Elem 伽马 a ->
       ----------
       Term 伽马 a
```

接下来，我们来看只有一个前提的类型规则，该前提是一个子推导。这些规则映射为只有一个参数的构造子，而该参数属于表示此子推导的归纳族。这里展示后继的类型规则：如果在给定的上下文中有一个自然数，那么它的后继也是

同一上下文中的一个自然数。

$$\frac{\Gamma \vdash n : \text{NAT}}{\Gamma \vdash (\text{Inc}\, n) : \text{NAT}}$$

```
递增 :  Term gamma 自然数类型->
        ----------
        Term gamma自然数类型
```

类似地，具有两个前提的规则会被转换为具有两个参数的构造子，每个参数对应一个子推导。这里我们给出应用节点的类型规则：如果函数具有函数类型，并且参数的类型与函数的定义域相匹配，那么应用的类型就对应于函数的陪域。请注意，上下文 Γ 在整个规则中保持不变，因此处处都使用相同的 `gamma` 。

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash t : a}{\Gamma \vdash f\ \$\ t : b}$$

```
App :  Term gamma (TyArr a b) ->
       Term gamma a ->
       ----------------------
       Term gamma b
```

最后，我们有一条前提的上下文已扩展的规则：通过提供一个类型为 b 的项（该项定义在扩展后的上下文中，其中新增了一个类型为 a 的变量），即可构造一个类型为 $(a \to b)$ 的函数。

$$\frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash (\lambda(x) \cdot t) : a \to b}$$

```
Func :  Term (gamma :< a) b ->
        ------------------------
        Term gamma (TyArr a b)
```

借助这种内禀类型表示，我们可以轻松表示完整的类型推导。下面的示例[3] 展示了`Plus2` 的内部表示，其中推导证明$(\lambda(x) \cdot (\text{Inc}\, (\text{Inc}\, x)))$ 可以被赋予类型 $(\text{NAT} \to \text{NAT})$。

$$\frac{\dfrac{\vdots}{\epsilon, x : \text{NAT} \vdash (\text{Inc}\, (\text{Inc}\, x)) : \text{NAT}}}{\epsilon \vdash (\lambda(x) \cdot (\text{Inc}\, (\text{Inc}\, x))) : \text{NAT} \to \text{NAT}}$$

```
Plus2 : Term  [<](自然数类型到自然数类
型)Plus2 = Func(Inc (Inc (Var Here)))
```

在我们的编译器中将 `Term` 用作中间表示后，我们便让整类无效程序都无法表示：不可能构造出作用域不合法或类型不合法的项。事实上，尝试编写作用域不合法或类型不合法的程序会导致静态错误，如下方的 **失败**代码块所示。[4] 在第一个示例中，我们尝试在空上下文中引用一个变量。Idris 2 正确地指出这是不可能的。

[3] `Here` 将在第 3.1 节中定义为 `Elem` 族的构造子。[4] Idris 2 仅在检查失败块的内容得到与给定字符串匹配的错误时，才接受这些失败块。

失败 "不匹配：?gamma：< 自然数类型 和 [<]。"

```
哎呀 : Term [<] 自然数
类型哎呀 =变量 此处
```

在第二个示例中，我们尝试将恒等函数的类型指定为从Nat 到 Bool的函数。这会被静态地判定为不合逻辑并拒绝： 自然数类型 和 布尔类型 是不同的构造子！

失败 "不匹配：布尔类型和自然数类型。"

```
哎呀 : Term [<] (TyArr TyNat TyBool)
哎呀 =Func (Var Here)
```

借助这类内在类型化嵌入式领域特定语言，我们可以通过静态方式确保 elaborator 会检查解析用户输入后得到的原始项是否作用域正确且类型正确。使用这些富含不变量的中间表示编写编译器遍（模型到模型转换）和求值引擎（模型到宿主语言转换），还能确保每一步都遵循该语言的静态语义。事实上，我们将在第 5 节中说明，如何使用嵌入式领域特定语言既验证静态语义，又描述动态语义。

对于配备了更高级类型系统、无法轻易通过静态方式强制执行这些保证的语言，我们可以使用作用域正确的核心语言，而非类型正确的核心语言，来保留其中一些保证。这正是 Idris 2 所采用的方法；它已经帮助消除了一整类错误，这类错误会在尝试用定义于不同上下文中的项求解元变量时产生 [9]。

3.1 高效的 De Bruijn 表示

实现良好作用域项的一种常见策略是使用带类型的*De Bruijn*索引 [13]，，它们可以轻松地实现为一个归纳族 [14]，用于指示变量在类型级上下文中的绑定位置。

具体来说，我们以一个上下文（再次表示为由种类组成的 SnocList ）以及它所表示的变量种类为索引，构造 Elem 族。

$\Gamma \ni x : a$ **数据** `Elem : (gamma :SnocList ty) -> (a: ty) -> 类型`

然后，我们将每条上下文成员关系推理规则与一个构造子相匹配。该这里的构造子表示，所关注的变量是作用域中距离当前位置最近的变量（注意左侧（x： a）的非线性出现形式，以及右侧相应的类型）。

$$\frac{}{\Gamma, x : a \ni x : a}$$

```
此处 : ----------------------
Elem(伽马 : < 类型) 类型
```

该那里构造子跳过距离当前位置最近的变量，以便在上下文更深处查找所关注的变量。

$$\frac{\Gamma \ni x : a}{\Gamma, y : b \ni x : a}$$

```
这里：   Elem 伽马类型->
        ----------------
        Elem (伽马: <  _) ty
```

虽然这是一个有效的定义，但遗憾的是，这种方法无法扩展到大型上下文：每个 `Elem` 证明的开销都与其表示的 De Bruijn 索引大小成线性关系。为了提高这种表示的运行时效率，我们改用自然数来建模 De Bruijn 索引，而 Idris 2 会将其编译为 GMP 风格无界整数。此外，我们还需要定义一个`AtIndex` 族，确保我们使用的所有自然数都对应有效的索引。我们特意复用了 `Elem` 这些名称，因为这里 和 那里构造子所起的作用完全相同。

```
data AtIndex : (ty : kind) -> (ctxt : SnocList kind) ->(idx : 自然数) ->
Type where Here : AtIndex ty (ctxt :< ty) 0 There : AtIndex ty ctxt
idx -> AtIndex ty (ctxt :< _) (1 + idx)
```

接着，我们将变量定义为一个自然数与一项擦除的（如绑定位置上`0` 注释所示）`prf`证明，该证明表明给定的自然数确实是有效的 De Bruijn 索引。

```
数据 IsVar : (ctxt: SnocList kind) -> (ty : kind) -> Type 其中V: (idx :
自然数) -> (0 prf : AtIndex ty ctxt idx) -> IsVar ctxt ty
```

得益于 Idris 2 中实现的定量类型理论 [19, 5]，编译器知道可以安全地擦除这些与运行时无关的证明。现在我们兼得两全：对 De Bruijn 指标进行良好作用域约束的实现，并且能够高效编译。

正如 De Bruijn 索引表示的朴素定义并不适合实际实现一样，第 3 节中介绍的归纳族 `Term` 也不便于使用。接下来，我们将了解它的一个局限，以及我们在 V´elo 中如何解决了这个问题。

3.2 紧凑常量折叠

《软件基础》的《编程语言基础》以一道常量折叠转换练习 [23, 章 1]开篇。我们从一种小型表达式语言出发（其中包含自然数、变量、加法、减法和乘法），运用半环性质来化简表达式。由于源语言的结构方式，化简遍历的定义包含大量重复代码：所有二元运算都是独立的构造子，必须先对它们的子项进行结构化化简，才能判断是否适用某条规则。正确性证明也同样有大量重复

因为它必须遵循所要化简的函数的调用图结构。这里唯一值得庆幸的是，Coq 的战术语言允许用户编写适用于许多相似目标的脚本，从而避免在源文件中重复代码。

在 V´elo 中，我们以代数方式构造核心语言的表示，从而无需重复这种操作。所有内置运算符（从内置类型上的原始操作到函数应用本身）都由单个 `Call` 构造子表示，它接受一个操作和一个由类型索引列表组成的子项。

```
data Term : (ctxt : SnocList Ty) -> Ty -> Type where
Var : IsVar ctxt ty -> Term ctxt ty Fun : Term (ctxt :<
a)b -> Term ctxt (TyArr a b) Call : {tys : _} ->
(operator : Prim tys ty) -> (operands : Terms ctxt tys) ->
Term ctxt ty
```

这里 `Terms` 是将`Term` 逐点提升到类型列表上的结果。实际使用中，我们采用通用的`All` 量词，但从本质上说，这与下面给出的专用版本等价：

```
数据Terms : (上下文:SnocList Ty) -> List Ty -> 类型其中Nil: Terms 上下文
Nil(::) :Term 上下文 类型 -> Terms 上下文 类型列表-> Terms 上下文(类型:: 类型
列表)
```

现在可以将原始操作枚举在单个数据类型中`Prim`其中列出了原始操作的参数及其对应的返回类型。

```
数据Prim : (参数: List Ty) -> (返回值 : Ty) -> 类型 其中
Zero: Prim  [] TyNatInc : Prim [TyNat] TyNatApp : Prim [
TyArr dom cod, dom] cod
```

使用`Prim`，现在可以通过统一处理`Call` 节点子项上的递归调用来实现结构操作，然后再根据运算符进行分派，以判断是否可以进行额外的简化。同样，所有正确性证明中的重复部分都被提取到同一处，在那里可以调用归纳假设。

3.3 类型正确的空洞

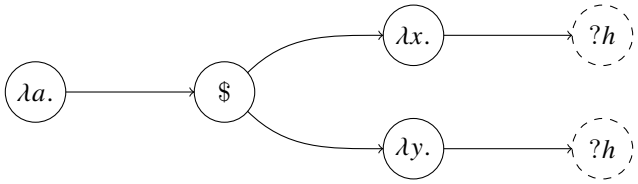
空洞是一种特殊的占位符，程序员可以用它表示程序中尚未编写的部分。在带类型语言中，每个空洞都会根据其所在的上下文被赋予一个类型。

类型驱动开发 [22] 是一种实践，用户通过与 编译器展开对话，以交互方式构建程序。我们可以通过为「空洞」及其操作提供特殊的语言支撑，部分实现类型驱动编程。这些操作包括检查、

细化、用空洞进行计算，以及用类型适当的项实例化空洞。我们认为，类型驱动开发最基本的语言支撑至少应包括以下能力：(1) 检查空洞的类型及其所在的局部上下文；(2) 用类型适当的项实例化空洞；(3) 安全地求值仍包含空洞的程序。V´elo 提供了这三种能力。

Idris 2 在遇到空洞时会对其进行细化：将其转化为没有关联定义的全局声明。由于这一设计选择，用户无法在不同位置显式提及同一个空洞，以表明这些尚未编写的项应当彼此相同。用户可以通过名称引用空洞的解，但该空洞只位于一个特定位置，Idris 2 正是根据这个位置推断其上下文。

然而，在 V´elo 中，我们允许在任意多个不同的局部上下文中任意多次提及空洞。在下面的例子中，空洞 ?h 出现在两个不同的上下文中：$\epsilon,\ a,\ x$ 以及 $\epsilon,\ a,\ y$。



因此，只有当某个项恰好属于这两个上下文的共同前缀（$\epsilon,\ a$）时，它才能填入该空洞。事实上，在 $\epsilon,\ a,\ y$ 中引用 x 是没有意义的；对于 y，反过来也一样。

我们的细化器分两步进行。首先，自底向上遍历会记录遇到的空洞；对于包含多个子项的节点，则通过计算恰当的局部上下文限制，协调彼此冲突的空洞出现位置。这个过程会生成一个列表，其中包含空洞、它们的类型和局部上下文，同时还会生成一个 `Holey` 项，其中包含不变量，以确保这些收集到的空洞确实能适配该项。其次，自顶向下遍历会生成一个核心 `Term`，其索引由`Meta` 列表给出（这是一种简单的记录类型，包含空洞的名称、其所属的上下文以及其类型）。每个空洞出现位置最终都会被赋予一个变薄，将元变量的实际上下文嵌入其出现位置所在的上下文中。第 4 节将讨论变薄及其在 V´elo 中的用途。

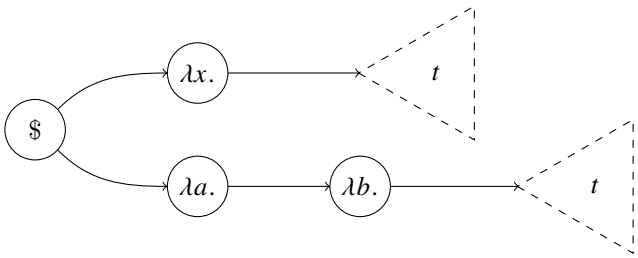
尽管这些中间表示是 V´elo 专属的，但这项技术和不变量具有通用性，任何想在自己的函数式领域特定语言中实现作用域良好的空洞的人都可以复用。

4 编译器遍

既然我们的核心语言在构造时就已确保作用域良好，那么我们也必须证明编译器遍能够保持作用域。这并非一项新要求，只是将过去非正式执行的约束具体化了。更重要的是，我们通过编译器遍展示了，模型到同一模型

转换我们的嵌入式领域特定语言是可行的，而且所需的基础设施并非 V´elo 专属。

公共子表达式消除的目的是识别语法树中多次出现的子项，并对它们进行抽象，以避免在运行时进行不必要的重复计算。例如，在下面的例子中，我们希望在应用节点（记作 $）之前进行 let 绑定 t，这样 t 就能由两个子树共享。



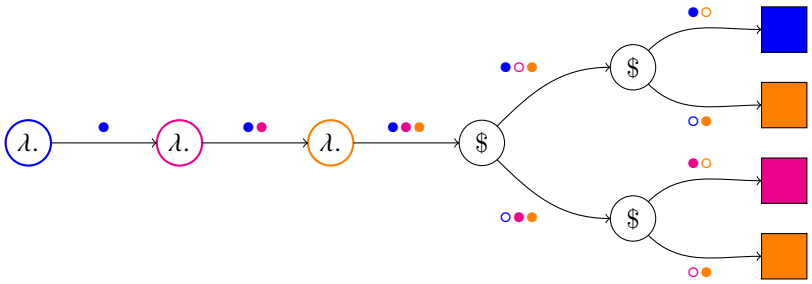
如上例所示，公共子表达式消除面临的挑战之一是，感兴趣的项可能深埋在各自独立的上下文中。在我们的内在作用域表示中，t 在作用域 $\Gamma, x:\sigma$ 中的项，不一定在语法上与位于 $\Gamma, a:\tau, b:\nu$ 中的副本相等。实际上，例如，在 Γ 中绑定的变量 ν 在 $\Gamma, x:\sigma$ 中会表示为 De Bruijn 索引（$1+\nu$），而在 $\Gamma, a:\tau, b:\nu$ 中则表示为索引（$2+\nu$）。

要是项能按其精确的支撑集 进行索引就好了（即仅包含项中实际使用变量的上下文）！这样我们就不必在意作用域中碰巧存在的其他无关变量。这里有原则的解决方案是在执行公共子表达式消除时切换到另一种表示。co-De Bruijn 表示 [20]正好提供了这一保证。

在 co-De Bruijn 表示中，每个项都严格按照其精确的支撑集进行索引。也就是说，每个子项都会显式丢弃其中未被提及的绑定变量。当我们到达一个变量节点时，作用域中只剩下一个绑定变量：恰好就是所引用的那个变量。

这个丢弃未使用变量的过程通过变薄来具体化，也就是使用具有单射性且保持顺序的重命名。我们可以将变薄看作由 0/1 位组成的序列，用来说明每个变量是被保留还是被丢弃。

下面，我们用余德布鲁因记法图示 S 组合子（λ项 $\lambda g.\lambda f.\lambda x.gx(fx)$），图示取自 [2]。其中，我们将变薄（即由比特组成的列表）表示为由 • (1) 或 ◦ (0) 组成的列表。



最初的三个 λ 抽象在变薄中只使用 $\bullet$，因为 g、f 和 x 都会出现在组合子的主体中。随后，第一个应用节点将上下文拆分为两部分：第一个子项（gx）丢弃 f，而第二个子项（fx）则去掉 g。接下来的应用节点为每个叶子子项选出仍在作用域内的唯一变量： g、 x、 f 和 x。

使用 co-De Bruijn 表示，我们可以识别共享子项：它们不能提及任何最局部的变量，并且在语法上相等。我们的编译过程成功地将左侧的程序转换为右侧的程序，其中重复表达式 (add m n) 和 (add n m) 已通过 let 绑定。

令 m = 为零，令 n = 为（inc 零），在（add（add m n）（add n m））中令 p = 为（add n m），令 q = 为（add m n），结果为（add（add q p）（add p q））

这个编译过程依赖于对变薄进行紧凑表示的能力（co-De Bruijn 表示大量使用变薄），此外还需要能够廉价地对它们进行相等性测试。我们在 V´elo 中提供的实现并非如此，不过这已经是一个解决了的问题 [2]。

5 执行

V´elo 交互式解释器让用户将项归约至头范式。我们可以通过定义式解释器 [4, 6],实现 V´elo 的动态语义，也可以提供更传统的类型可靠性句法证明 [30], 但使用归纳族将其机械化 [29,第二部分：性质] 。

我们选择了后者：通过使用归纳族，可以明确描述语言的操作语义。这使我们能够研究其元理论性质，尤其是证明一项进展性结果：每个项要么是一个值，要么能够执行一个归约步骤。反复应用进展性结果，直到得到一个值，或者最终用户失去耐心并终止进程，这个证明便能直接为我们提供一个求值器，并保证它符合 V´elo 的操作语义。

沿用现有方法 [29, 第 2 部分：性质], 我们定义了描述项如何归约的归纳族。

```
数据 归约 : (this,that : Term 上下文类型) -> 类型其中
  简化调用 : (操作 :Prim 类型 类型)-> (步骤 :归约关系 这些 那
  些)-> 归约关系 (调用p 这些)(调用p 那些)再次
```

归约函数应用 : {主体 : Term (上下文 :< 类型)return}-> {参数 : Term 上下文 类型}->(值 : Value 参数) -> 归约关系 (调用 App [函数 主体, 参数]) (替换 参数 主体)

如上所示，我们的设定强制采用值调用：正如规则 归约函数应用 $((\lambda(x) \cdot b)\ \$\ t)$ 仅当（ $b\{x \leftarrow t\}$) 已知 t 为一个值时，才会归约为（。此外，我们的代数设计（第 3.2 节）允许我们通过设置一个通用族来描述原始操作的参数如何归约，从而轻松强制执行从左到右的求值顺序。如下所示：考虑一个类型对齐列表的参数时，要么 头部 进行一步求值，而 其余部分 保持不变；要么 头部 已知是一个值，因此允许后续参数进行一步求值。

```
data Reduxes : (这些, 那些 : Terms ctxt tys) -> Type where (!:) :
(hd : Redux this that) -> (rest : Terms ctxt tys) -> Reduxes (t
his :: rest) (that :: rest) (::) : (值: Value hd) ->(tl : Reduxes 这些 那
些) -> Reduxes (hd :: 这些) (hd :: 那些)
```

不过，我们与标准方法有所不同：我们将进展性证明设计为通用形式，从而将归约项时计算自反传递闭包所需的样板代码整理到一个可共享的模块中。因此，我们顶层的进展定义以归约和值定义为参数：

```
data Progress : (0 值 : Pred a) -> (0 redux : Rel a) -> (tm: a) -> Type where Done :
 {0 tm : a} -> (val : 值 tm) -> Progress 值 redux tm

      Step : {this, that : a}
          -> (step : redux this that) -> Progress value redux this
```

而执行的结果也以类似的方式进行参数化，如下所示（其中RTList 是一个类型，它接受一个关系并返回其自反传递闭包）：

data Result : (0 值 : Pred a) -> (0 redux : Rel a) -> (this : a) -> Type where R : (that : a) -> (val : 值 that) -> (steps: RTList redux this that) -> Result 值 redux this

我们这种方法的好处在于，语言设计者只需说明归约是什么，以及如何计算单个归约，其余部分便可自动获得。此外，在得到求值结果的同时，我们还会得到所进行的归约步骤列表，并可选择将其打印出来，以展示执行跟踪。

6 结论

我们已经证明，依赖类型语言满足语言工作台挑战 [15]提出的核心要求。按照设计，V´elo 的记法作为一种领域特定语言是文本形式的，其内部核心则受 Idris 2 自身记法要求的约束。更重要的是，得益于依赖类型这一设定，V´elo 的语义（静态语义与动态语义）已作为实现的一部分得到验证。遗憾的是，支持最薄弱的核心标准是编辑器支撑。通过 Idris 2 创建的语言并不会自带编辑器；它们是自由形式的语言，需要手动编写解析器和细化器。未来的研究或许能改变这一点。Idris 2 支持细化器反射 [11]，这为派生解析器和细化器提供了途径。

不过，语言工作台特性模型中还有更多标准需要考虑：面向编辑器的语义与语法服务；测试与调试；以及可组合性。

随着语言服务器协议（LSP）的兴起，研究如何以通用方式派生出兼容 LSP 的语言服务器将是一个不错的想法，从而解决可选语义与语法服务缺失的问题。Idris 2 本身提供了 *IDE* 协议，并且 Idris 2 支持 LSP。

我们的语言也不具备测试和调试其实现的能力。我们介绍的一些特性已经得到了完整的形式化（例如执行），而另一些特性则仅知晓其保持作用域与类型（例如公共子表达式消除）。因此，依赖类型设置并不意味着我们从此不再需要测试。此前关于归纳族生成器的工作 [18] 应当能够让我们将基于属性的测试 [12] 引入核心处理阶段。

最后，还有语言的可组合性。如果在设计新语言时，能够支持重用现有语言及其类型系统，将会非常有益。这是一个棘手的问题：不仅要结合它们的语义，还要整合其余的工作台工具。语言片段 方法 [27]为内禀类型定义解释器提供了一种实现语言可组合性的方案，但无法扩展到工作台工具。将这种方法扩展到基于归纳族的语义定义，并据此构建可组合的工作台工具，是一个开放问题。

我们坚信，通过未来工程，我们能够满足这些缺失的标准，并使依赖类型语言成为一个极为出色的语言工作台。

参考文献

[1] Andreas Abel、Guillaume Allais、Aliya Hameer、Brigitte Pientka、Alberto Momigliano、Steven Sch¨afer 和 Kathrin Stark。POPLMark reloaded：用逻辑关系机械化证明。函数式程序设计期刊，29:e19，2019年。doi:10.1017/S0956796819000170。[2]Guillaume Allais。将内置类型视为归纳族。*CoRR*，abs/2301.02194，2023年。arXiv:2301.02194，doi:10.48550/arXiv.2301. 02194。[3] Guillaume Allais、Robert Atkey、James Chapman、Conor McBride 和 James McKinna。具有绑定的类型安全、作用域安全语法全集：其语义与证明。函数式程序设计期刊，31:e22，2021年。doi:10.1017/ S0956796820000076。[4] Nada Amin 和 Tiark Rompf。使用定义式解释器证明类型健全性。*SIGPLAN* 通讯，52(1):666–679，2017年1月。doi:10.1145/ 3093333.3009866。[5]Robert Atkey。定量类型理论的语法与语义。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。美国计算机协会，2018年。doi:10.1145/3209108. 3209189。[6]Lennart Augustsson 和 Magnus Carlsson。依赖类型练习：一个类型正确的解释器。载于哥德堡依赖类型编程研讨会，1999年。[7] Brian E. Aydemir、Aaron Bohannon、Matthew Fairbairn、J. Nathan Fos- ter、Benjamin C. Pierce、Peter Sewell、Dimitrios Vytiniotis、Geoffrey Wash- burn、Stephanie Weirich 和 Steve Zdancewic。面向大众的机械化元理论：PoplMark 挑战。载于 Joe Hurd 和 Thomas F. Mel- ham 编，高阶逻辑定理证明：第*18*届国际会议，*TPHOLs 2005*，牛津，英国，*2005*年*8*月*22*日至*25*日，论文集，《计算机科学讲义》第3603卷，第50–65页。施普林格，2005年。doi:10.1007/11541868\ 4。[8] Edwin C. Brady。使用 *Idris* 进行类型驱动开发。Manning Publications，2017年。[9] Edwin C. Brady。Idris 2：实践中的定量类型理论。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 信息学中心，2021年。doi:10.4230/LIPIcs. ECOOP.2021.9。

[10] James Chapman、Roman Kireev、Chad Nester 和 Philip Wadler。Agda 中的系统 F：乐趣与收益兼得。载于 Graham Hutton 编，程序构造的数学——第*13*届国际会议，*MPC 2019*，波尔图，葡萄牙，*2019*年*10*月*7*日至*9*日，论文集，《计算机科学讲义》第11825卷，第255–297页。施普林格，2019年。doi:10.1007/978-3-030-33636-3\ 10。[11] David R. Christiansen 和 Edwin C. Brady。elaborator 反射：在 Idris 中扩展 Idris。载于 Jacques Garrigue、Gabriele Keller 和 Eijiro Sumii 编，第*21*届 *ACM SIGPLAN* 函数式编程国际会议论文集，*ICFP 2016*，日本奈良，*2016*年*9*月*18*日至*22*日，第284–297页。美国计算机协会，2016年。doi:10.1145/2951913.2951932。[12] Koen Claessen 和 John Hughes。QuickCheck：用于 Haskell 程序随机测试的轻量级工具。载于 Martin Odersky 和 Philip Wadler 编，第五届 *ACM SIGPLAN* 函数式编程国际会议论文集（*ICFP '00*），蒙特利尔，加拿大，*2000*年*9*月*18*日至*21*日，第268–279页。美国计算机协会，2000年。doi:10.1145/351240.351266。[13]Nicolaas Govert de Bruijn。带有无名占位符的 λ 演算记法：一种用于公式自动处理的工具，并应用于 Church-Rosser 定理。《数学探究》（论文集），75(5):381–392，1972年。doi:10.1016/1385-7258(72)90034-0。[14] Peter Dybjer。归纳族。计算机形式化方面，6(4):440–465，1994年。doi:10.1007/BF01211308。[15] Sebastian Erdweg、Tijs van der Storm、Markus V¨olter、Meinte Boersma、Remi Bosman、William R. Cook、Albert Gerritsen、Angelo Hulshout、Steven Kelly、Alex Loh、Gabri¨el D. P. Konat、Pedro J. Molina、Mar- tin Palatnik、Risto Pohjonen、Eugen Schindler、Klemens Schindler、Ric- cardo Solmi、Vlad A. Vergu、Eelco Visser、Kevin van der Vlist、Guido Wachsmuth 和 Jimi van der Woning。语言工作台的最新进展——语言工作台挑战的结论。载于 Martin Erwig、Richard F. Paige 和 Eric Van Wyk 编，软件语言工程——第*6*届国际会议，*SLE 2013*，美国印第安纳州印第安纳波利斯，*2013*年*10*月*26*日至*28*日，论文集，《计算机科学讲义》第8225卷，第197–217页。施普林格，2013年。doi:10.1007/978-3-319-02654-1\ 11。[16] Paul Hudak。构建领域特定嵌入式语言。美国计算机协会计算调查（*CSUR*），28(4es):196，1996年。[17] Atsushi Igarashi、Benjamin C. Pierce 和 Philip Wadler。轻量级 Java：Java 和 GJ 的最小核心演算。美国计算机协会程序设计语言与系统汇刊，23(3):396–450，2001年。doi:10.1145/503502.503505。[18] Leonidas Lampropoulos、Zoe Paraskevopoulou 和 Benjamin C. Pierce。为归纳关系生成优质生成器。美国计算机协会程序设计会议录，2(POPL):45:1–45:30，2018年。doi:10.1145/3158133。

[19] Conor McBride。《我什么也不缺》。载于 Sam Lindley、Conor McBride、Philip W. Trinder 和 Donald Sannella 编，《一系列足以改变世界的成功——献给 *Philip Wadler* 六十岁生日的论文》，《计算机科学讲义》第 9600 卷，第 207–233 页。施普林格，2016 年。doi:10.1007/978-3-319-30936-1\ 12.[20] Conor McBride。《每个人都得有个去处》。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。doi:10.4204/EPTCS. 275.6.[21] Ulf Norell。Agda 中的依赖类型编程。载于 Pieter W. M. Koopman、Rinus Plasmeijer 和 S. Doaitse Swierstra 编，《高级函数式编程，第六届国际学校，*AFP 2008*，荷兰 *Heijen*，*2008* 年 *5* 月，修订讲义》，《计算机科学讲义》第 5832 卷，第 230–266 页。施普林格，2008 年。doi:10.1007/978-3-642-04652-0\ 5.[22] Cyrus Omar、Ian Voysey、Ravi Chugh 和 Matthew A. Hammer。使用类型化空洞进行实时函数式编程。*ACM* 程序语言论文集，3(POPL):14:1–14:32，2019 年。doi:10.1145/3290327.[23] Benjamin C. Pierce、Arthur Azevedo de Amorim、Chris Casinghino、Marco Gaboardi、Michael Greenberg、Cˇatˇalin Hrit¸cu、Vilhelm Sj¨oberg、Andrew Tolmach 和 Brent Yorgey。《编程语言基础》，《软件基础》第 2 卷。电子教材，2020 年。第 5.8 版，http: // softwarefoundations.cis.upenn.edu.[24] Arjen Rouvoet、Casper Bach Poulsen、Robbert Krebbers 和 Eelco Visser。线性、会话类型语言的内禀类型化定义解释器。载于 Jasmin Blanchette 和 Catalin Hritcu 编，《第九届 *ACM SIGPLAN* 认证程序与证明国际会议论文集，*CPP 2020*，美国路易斯安那州新奥尔良，*2020* 年 *1* 月 *20–21* 日》，第 284–298 页。美国计算机协会，2020 年。doi:10.1145/3372885.3373818.[25] Kathrin Stark、Steven Sch¨afer 和 Jonas Kaiser。Autosubst 2：基于多排序 de Bruijn 项和向量替换进行推理。载于 Assia Mah- boubi 和 Magnus O. Myreen 编，《第八届 *ACM SIGPLAN* 认证程序与证明国际会议论文集，*CPP 2019*，葡萄牙卡斯凯什，*2019* 年 *1* 月 *14–15* 日》，第 166–180 页。美国计算机协会，2019 年。doi:10.1145/3293880.3294101.[26] Coq 开发团队。Coq 证明助手，2022 年 1 月。doi: 10.5281/zenodo.5846982.[27] Cas van der Rest、Casper Bach Poulsen、Arjen Rouvoet、Eelco Visser 和 Peter Mosses。灵活组合的内禀类型化定义解释器。*ACM* 程序语言论文集，6(OOPSLA2)，2022 年 10 月。doi:10.1145/3563355。

[28] Guido Wachsmuth、Gabri¨el D. P. Konat 和 Eelco Visser。使用 Spoofax 语言工作台进行语言设计。*IEEE* 软件，31(5):35–43，2014 年。doi:10.1109/MS.2014.100.[29] Philip Wadler、Wen Kokke 和 Jeremy G. Siek。《*Agda* 编程语言基础》。2022 年 8 月。网址：https://plfa.inf.ed.ac.uk/ 22.08/.[30] Andrew K. Wright 和 Matthias Felleisen。类型健全性的句法方法。信息与计算，115(1):38–94，1994 年。doi:10.1006/inco.1994. 1093。