arXiv:2603.01463v1 [cs.LO] 2 Mar 2026

依赖类型理论可居住性与合一的实现

Chase Norman[0000−0001−8954−3770] 和 Jeremy Avigad[0000−0003−1275−315X]

美国宾夕法尼亚州匹兹堡，卡内基梅隆大学
{chasen, avigad}@cmu.com

摘要。 依赖类型理论是许多现代证明助手的基础。可居住性与合一是不可判定问题，在定理证明和程序合成中具有实用价值。我们介绍 Canonical-min：一个用于依赖类型理论中可居住性与合一的可靠且完备的求解器，它由 185 行 Lean 代码实现。本文介绍了一种新颖的依赖类型理论实现，以及一个将类型检查器转化为高性能求解器的单子框架。最后，我们介绍 DTTBench，这是一个针对依赖类型理论中类型占据问题的基准测试。

关键词： 自动推理 · 依赖类型理论 · 可居住性 · 合一

1 引言

自动推理系统通过算法证明形式语言中的命题。如今，许多用于一阶和高阶逻辑的自动定理证明器都已得到广泛应用 [13, 21, 18, 1, 15]。然而，许多现代交互式定理证明器使用依赖类型理论（DTT）作为其形式语言 [16, 2, 3]。

选择这种做法有多种原因。交互式定理证明器允许用户使用带类型的编程语言定义函数。借助 DTT，同一种语言也可用于编写证明。证明 α 蕴含 β，可以表示为一个函数：将 α 的证明证书映射为 β的证明证书；证明 $P(z)$对所有整数 z 都成立，可以表示为一个函数：将整数 z 映射为$P(z)$的证明证书。DTT 甚至可以表示诸如 $\mathbb{R}^n$ 这样的带索引的结构族，将数字 n 映射到类型 $\mathbb{R}^n$。

依赖类型理论能够（但不局限于）进行构造性推理，因此可以表达程序合成问题。它甚至可以用于从头定义等式及其他基础概念，并证明它们的性质。然而，这种表达能力是有代价的：自动化难度相当高。类型占据，即寻找给定类型的表达式这一问题，是不可判定的，并且对应于定理证明和程序合成。合一，即求解项方程这一问题，同样不可判定 [10, 11]。现有的依赖类型理论可居住性求解器都不完备，因为它们使用一阶

模式合一 [19, 20, 14, 12, 6], , Canonical [17], 除外；从概念上说，它实现了 Dowek [9] 提出的一个可靠且完备的算法。

为了揭开 DTT 搜索的神秘面纱，我们介绍 Canonical-min，[1] 这是 Canonical 的一个简化版本，实现了 DTT 可居住性与合一的可靠性和完备性。本文包含全部 185 行源代码，但不包括编译器指令和类型类实例。第 2 节通过实现一个类型检查器，正式说明可居住性与合一；类型检查器是一种用于确定某个项是否属于给定类型的程序。第 3 节介绍 Canonical-min 中表示项与类型的数据结构。我们实现的每个方面都与现有类型检查器大相径庭，但这是支持我们对元变量进行新颖解释所必需的。第 4 节介绍一种新颖的单子框架，无需更改任何代码，即可将类型检查器转换为可居住性与合一求解器。第 5 节通过深度优先搜索将所有部分连接起来。最后，我们介绍 DTTBench，[2] 这是一个基准测试，包含来自 Lean 标准库 [16]和 Mathlib[5],的 DTT 可居住性问题，并据此评估 Canonical-min 与其他 DTT 可居住性系统。

2 依赖类型理论

2.1 项

类型论定义了一种语言，其中语法上有效的表达式集合称为项。通常，该集合由λ演算中的表达式组成，例如（$\lambda\ x\ y \mapsto y$）。我们将 x 和 y 称为绑定。

假设你希望判定两个λ项是否等价，这称为判断相等或定义相等。两个具有不同绑定的项，若行为相同，仍可被视为相等，例如（$\lambda\ x \mapsto f\ x$）和（$\lambda\ y \mapsto f\ y$）。为了考虑这种 α-等价，我们生成新鲜变量，并将这两个项都应用到它们上。

当我们调用第3.1节中定义的函数Term.apply来执行这些应用时，所得表达式将被 β-归约，直到它们处于弱头范式（弱头范式），以考虑 β-等价。处于弱头范式的项是一个函数符号（称为头符号），它应用于一列项（称为参数）。[3] 我们说，原来的λ项等价，当且仅当这些弱头范式项等价。

我们在 Lean 中将此过程实现为函数 Term.eq。在下面的代码中，左箭头（←）表示这是一次单子的函数调用，目前可以忽略；灰显的部分将在后文解释：

```
部分定义Term.eq (t1 t2 : 项) : 判断 := 判断 执行 let 变量 ←新鲜 (←
t1。应用 vars (other t2 vars))。eq (← t2。应用 vars rigid)
```

[1] 可在 https://github.com/chasenorman/Canonical-min 获取 [2] 可在 https://github.com/chasenorman/DTTBench 获取[3] λ 表达式也被视为弱头范式，但我们的算法不会直接处理此类项，因为它们总是应用于新鲜变量。

在 Lean 中，该函数被标记为“partial”，因为我们没有提供终止性证明。我们假设 t1 和 t2 具有相同的元数，稍后会看到我们的表示方式如何确保 vars 包含正确数量的变量。

要判断两个弱头范式表达式是否相等，我们会比较它们的头符号。如果头符号不相等，就抛出异常。否则，我们遍历这两个表达式中该头符号的参数，并用 Term.eq 比较它们。如果确定头符号和所有参数都相等，就不会抛出异常。

```
partial def WHNF.eq (w1 w2 : WHNF) : Judgment :=
do if w1.head  ==w2.head then for i in [0:w1.args.size]
do w1.args[i]!.eq w2.args[i]! else failure-- 抛出异常
```

使用异常至关重要，因为这样可以避免在两个项已确定不相等后仍进行不必要的 apply 调用。我们假设 w1 和 w2 的参数数量恰好等于其头符号的元数，因此当它们的头符号相同时，参数数量也相同。在 Lean 中，感叹号（!）是必需的，因为我们没有提供数组索引在有效范围内的证明。

2.2 类型

类型理论还定义了一组类型，以及一组规则，用于判定一个项是否属于给定类型。在依赖类型理论中，λ 表达式的类型由依赖函数类型给出，例如：

$$(T : \texttt{Type}) \to (t : T) \to T$$

我们将第一个 T 和 t称为绑定、类型，并将第二个 T 称为输入，最后一个 T 称为输出。此类型表示所有具有这些输入类型和输出类型签名的函数。

要判定项 t 是否属于类型 ty，我们再次通过创建新鲜变量来处理 α-等价。这些变量的类型将是 ty 的输入。将 t 应用于这些变量并规约至弱头范式后，我们会得到头符号、其类型 headType 以及参数。

我们的首要任务是确保这些参数具有头符号所要求的类型。为此，我们提取 headType 的输入。由于 headType 是依赖函数类型，此操作必然需要这些参数。接着，我们递归地检查每个参数的类型是否与相应输入匹配。

我们最后需要确保 headType 的输出与 ty 的输出相匹配。由于我们处理的是依赖类型，headType 的输出可能取决于参数，而 ty 的输出可能取决于新鲜变量。我们将在下一节看到，类型的输出是一个项，因此使用 Term.apply 将绑定替换为这些表达式。然后，我们使用 WHNF.eq 检查它们是否相等，从而完成类型检查器。同样，灰显部分暂时可以忽略。

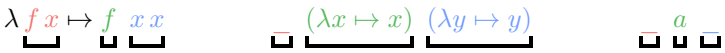
partial defcheck (t : Term) (ty : Typ) : Judgment := 判断 do let vars ← fresh ty let 〈_头部,头部类型, args〉 ← t.apply vars let 输入 := headType.get!.inputs args for i in [0:args.size] do check args[i]!inputs[i]! (← headType.get!.output.apply args (other ty.output vars)).eq (←ty.output.apply vars rigid)

我们假设 t 和 ty 具有相同的元数。

3 数据结构

3.1 项

我们将 λ 表达式表示为三个部分：一系列绑定、一个头部，以及由参数构成的脊柱。与必须以变量头符号为头部的弱头范式项不同，为了表示可归约表达式，我们允许项的头部本身也是一个项。观察这种分解在以下各项中的表现：



这种分解不同于项的标准归纳定义（其构造子包括变量、应用和 λ 绑定），因为它只有一个构造子，从而提高了统一性。此外，它将绑定和参数分别存放在连续的块中，便于高效随机访问。

我们将仅使用德布鲁因指标 [8]来编码项。我们在一个名为赋值的结构中存储头部的索引以及对参数的引用。索引由两个数字组成：debruijn 和 index，稍后会对此进行解释。

我们希望项可以改变，因此使用对赋值的引用，并将其称为元变量。下面是 Assignment 的定义，以及用于通过引用 MVar 访问赋值的函数声明：

结构体赋值 其中
  debruijn：自然数
  index：自然数 args:
  数组 MVar

部分定义MVar.assignment (m : MVar)(rigid : ConstraintM 布尔值) :
SearchM赋值：= 执行 ...

值得注意的是，赋值不会存储有关绑定的信息。我们不需要存储绑定的名称，因为我们使用德布鲁因指标（局部无名表示也采用了类似的技巧 [4]），而且我们

不需要存储绑定的数量，因为所有项都恰好有一个绑定块。

为了将索引与头部关联起来，我们维护一个包含局部上下文中所有头部变量和头部项的集合，称为显式替换 （ES）。ES 是一个由块组成的链表，其中每个块要么是变量块（带有块标识符，并可选地带有类型），要么是项块。

项和变量总是以块的形式直接添加到 ES 中，无论块的大小如何，都能确保高效且一致的行为。MVar 的德布鲁因字段表示链表中块的位置，索引字段表示头部在该块中的位置。由于块可以添加到 ES 中，德布鲁因的含义取决于它在项中的深度。ES 是链表，因为它必须是全持久化数据结构。使用块可以缩短链表的长度，从而提高访问速度。

项定义为一个元变量和一个 ES。

```
结构 项其中
  元变量: 元变量
  显式替换: 块列表
```

```
归纳 块其中
| 变量: 自然数 → Option 类型 → 块项:
  项 →块
```

我们必须将显式替换与德布鲁因指标分开，因为元变量的索引是可变的，甚至可能尚未赋值。为了在索引发生变化后正确解析头部，并对索引未赋值的项执行 β-归约，我们需要头部表达式库。

Terms 类型是一种高效的表示方式，用于表示共享同一个显式替换的项数组。索引 i 处的项定义为索引 i 处的元变量与该显式替换配对：

```
结构体 项其中
  元变量: 数组化元变量
  显式替换: 块列表

定义 项.大小(ts : Terms) : 自然数 := ts.元变量.大小
```

一个弱头范式项由一个头变量、该头变量的可选类型，以及作为项的参数组成。变量由其来源块的标识符和该块中的索引唯一确定。

```
结构体 弱头范式其中
  头部: 变量
  类型: Option 类型
  参数: 项
```

```
结构 变量其中块标识符:
  自然数 索引 : 自然数
```

现在，我们将所有内容整合起来定义 apply。要将项 t 应用到参数块 args，我们首先扩展 t 的 ES，使其在 de Bruijn 索引 0 处包含 args，从而替换 t 的绑定。接下来，我们访问 t.mvar 的赋值 assn，并根据 assn.debruijn 在 ES 中进行索引。如果该位置的块由变量组成，那么我们就到达了弱头范式（WHNF）。头符号是该块中索引为 assn.index 的变量，而参数

是扩展 ES 下的 assn.args。如果变量具有类型，则头符号的类型是索引 assn.index 处的类型。

否则，如果块由项构成，我们必须进行 β-归约。我们访问索引为 assn.index 的项，并在扩展 ES 下将其应用于 assn.args。

```
partial defTerm.apply (t : 项) (args : 块) (rigid : ConstraintM 布尔值 :=
pure false) : SearchM 弱头范式 :=do let es :=args :: t.es let assn ←
t.mvar.assignment rigid let args :项 := { mvars := assn.args, es }
match es[assn.debruijn]! with|.vars blockId types => pure { head :=
{blockId, index := assn.index }, type := types >>= (·[assn.index]!),
args }| .terms ts => ts[assn.index]!.apply args rigid
```

请注意，apply 不会创建新的元变量或赋值。

3.2 类型

与项一样，Typ 由一个德布鲁因索引的类型元变量和一个 ES 构成。Types 的定义方式与 Terms 类似，包含一个共享同一 ES 的类型元变量数组。索引为 i 的类型定义为索引为 i 的类型元变量与该 ES 配对。

```
结构 类型其中mvar:
类型元变量 es: 块列表
```

```
结构 类型其中
  mvars: 数组 类型元变量
  es: 块列表
```

```
定义 类型.size(ts : 类型) : 自然数 := ts.mvars.size
```

类型元变量与元变量的区别在于：类型元变量还包含一个用于输入类型的类型元变量数组，以及一个用于绑定的首选名称数组（这些名称仅用于打印，从不进行比较）。

```
结构 类型元变量 其中
  输入：数组 类型元变量 输出：元
  变量 首选名称：数组 字符串
```

请注意，输出本身并不是依赖函数类型，因为我们预期函数采用非柯里化形式。因此，它表示为元变量表达式。

获取给定类型的输入类型需要一组参数块，因为后续输入可能依赖于较早的输入。输入类型通过将 TypeMVar 的输入数组与显式替换配对得到，并将该显式替换扩展为用参数块实例化 de Bruijn 索引 0 处的绑定。

定义 Typ.inputs(类型 : 类型) (参数 : 块) : 类型: ={ 元变量: =
类型.元变量.输入, 显式替换: = 参数 :: 类型.显式替换 }
我们不会在结构上强制执行套叠，也就是说，索引 i 处的输入只能引用 args 中索引为 $j < i$ 的元素。相反，我们对所有输入使用相同的显式替换，其中包含全部 args。

类型的输出是与显式替换配对的输出元变量。这是一个 λ-项，其绑定必须使用 Term.apply 实例化。
定义 Typ.output(类型 : 类型) : 项: ={ 元变量: =
类型.元变量.输出, 显式替换: = 类型.显式替换 }
下面以一个示例类型为例，展示 Typ.output 的行为:

$$(T : \mathtt{Type}) \to (t : T) \to T \quad \overset{\text{output}}{\longmapsto} \quad \lambda\ T\ t \mapsto T$$

4 单子

到目前为止，我们已经展示了依赖类型理论的类型检查器的实现。接下来还需要说明如何实现可居住性和合一过程。事实上，只要巧妙地选择单子，我们展示的代码就已经能够实现这些功能。单子是一种类型变换器；如果 m 是一个单子，且 α 是一个类型，那么 m α 也是一种类型，表示返回 α的元素的计算所具有的类型。这些计算可能会根据 m 使用不同的效应。例如，Option 是一种单子，其中 Option α 是返回 α 的元素或抛出异常的计算所具有的类型。

我们希望扩展类型检查器，使其支持未赋值元变量。如果在类型检查过程中遇到未赋值元变量，我们应当针对该元变量生成一个约束，表示该元变量被赋值后仍需执行的检查。这样，我们就可以对元变量的赋值进行回溯搜索，并延后处理类型检查约束，直到能够对它们进行进一步求值。

我们将 ConstraintM α 定义为一种计算类型，它可以读取赋值数组，并返回一个 α 元素，或者抛出异常。
缩写ConstraintM: = ReaderT (数组 (Option Assignment)) Option
每个元变量都有一个标识符，用作该数组中的索引。如果此索引处的值为 none，我们就称该元变量未赋值。ReaderT 是一种单子变换器，这意味着它可以扩展现有单子，增加访问只读值的能力。在此处，我们扩展 Option 单子，使其能够在约束遭到违反时以失败的方式抛出异常。

现在，我们将约束定义为一个续延，以及阻止进一步推进的未赋值元变量，称为 stuck。续延是一种计算：如果约束遭到违反，就抛出异常；否则返回一个包含待处理剩余工作的约束数组。

结构约束 其中
续延：ConstraintM (Array Constraint)[4]stuck：元
变量刚性：ConstraintM 布尔值

接下来，我们定义用于 Term.eq、WHNF.eq 和检查的 JudgmentM 单子及 Judgment 类型：

```
缩写 JudgmentM := StateT (自然数 ×数组约束) ConstraintM缩写
Judgment:= JudgmentM Unit
```

JudgmentM 在 ConstraintM 的基础上扩展了两个状态变量。第一个是一个自然数，用于为变量的新块创建标识符。创建新鲜变量时，我们返回一个标识符等于状态变量 n 的块变量（Block.vars），并递增该状态变量。创建带类型变量时，我们将这些变量的类型设为给定类型 ty 的输入。[5]

```
定义 fresh (ty : Option 类型 := none) : JudgmentM 块 :=modifyGet
fun(n, l)  => (.vars n (ty >>= (·.输入 (.vars n none))), n.succ,
l)
```

JudgmentM 的第二个状态变量是一个 Constraint 数组，我们会在判断求值时不断累积其中的内容。Judgment 定义为 JudgmentM Unit，因为 Judgment 返回的唯一信息是是否抛出了异常、Nat 的当前值，以及累积的数组约束。

现在我们来定义该算法的核心机制：延续单子。考虑一个返回类型为 β 的函数，它在延续单子中调用一个返回类型为 α 的函数。调用方会向被调用方提供一个从 α 到 β 的回调函数，用来表示调用方打算在取得被调用方的返回值后执行的所有剩余代码。随后，调用方会立即让出控制权，由被调用方决定调用该回调函数多少次（如果要调用的话）。

请注意，下面的 ContT m r a 定义接受一个回调函数，用于描述应如何处理返回类型 a。计算不会返回类型 a，而是返回与回调函数相同的结果类型 r。

```
定义 ContT (m : 类型 → 类型) (ra : 类型) := (a → m r) → m r
```

我们定义了一个专门化版本的延续单子，它将判定单子扩展为具有固定结果类型 Unit：

```
缩写SearchM: = ContT 判定单子 Unit
```

[4] 实际上，我们使用等价类型 Array (Option Assignment) → Option (Array Constraint)，以使编译器确信该定义是健全的。[5] 访问 ty 的输入需要用到我们正试图创建的确切变量。为了解决这个问题，我们创建一个具有相同标识符 n、但没有类型的变量块。

我们的意图是将这个延续单子用于 MVar.assignment。

回想一下，MVar.assignment 函数旨在访问元变量的赋值。如果元变量已赋值，我们就会直接用该赋值调用回调函数，并继续正常执行。否则，我们会将回调函数作为一个新的 Constraint 存入状态数组，以便稍后调用。要访问赋值数组，ReaderT 提供了 read 函数。在下面的代码中，k 是回调函数，l 是约束数组。

```
部分定义MVar.assignment （m ：元变量)
    (rigid : ConstraintM 布尔值) : SearchM Assignment := do (← read)[m.id]!.getDM fun k (n, l) => return ((), n, l.push {
  continuation :=约束 (do k (← m.assignment rigid)) n, stuck := m,
  rigid })
```

采用这种实现时，如果对一个含有未赋值元变量的项调用 Term.apply，计算就会暂停，并以该元变量上的约束形式存储。

约束函数通过传入一个空续延、一个指定的状态自然数起始值，并且只返回累积的约束数组，将 SearchM T 转换为 ConstraintM (Array Constraint)。

```
定义 constraint(t : SearchM T)(start ：自然数) : ConstraintM (Array Constraint) := 执行 do，返回 (← t(函数 _ s => pure ((), s)) (start, #[]))。2.2
```

最后，我们定义一个从 SearchM Unit 到判断的转换，并使用一个平凡的续延：

```
定义 judgment(j : SearchM Unit) : 判断 := j pure
```

判断函数对于确保 WHNF.eq 和检查中的每次 for 循环迭代都彼此独立执行至关重要（而不是让每次迭代都成为前一次迭代的续延），这样即使前一次迭代陷入 stuck 状态，我们也能累积后续迭代产生的约束。

5 搜索

现在，我们的类型检查器能够为未赋值元变量生成约束，因此我们通过迭代加深深度优先搜索来实现项枚举。我们不使用树高来衡量深度，而是采用一个自定义的燃料参数，称为熵 [17], ，并在每次迭代时增加该参数。我们还限制生成项的大小，并在每次迭代时逐步放宽这一限制，然后运行深度优先搜索。

```
定义 迭代加深深度优先搜索 [单子m] [MonadLiftT CanonicalM m]
(root : 元变量) (cb step : m 单位) : m 单位: = 执行
```

```
令 start ←容量令 mut熵 : = 1000当
true 时执行扩展 4深度优先搜索
root cbstep 熵 start熵 : =熵 * 3
```

深度优先搜索函数接受一个根元变量，表示我们希望对其内部所有未赋值元变量进行赋值的表达式。我们调用启发式函数 next，从 root 的表达式树中选出一个未赋值元变量 mvar。若不存在这样的元变量，则说明已找到一个解，并调用用户提供的回调函数 cb。

next 函数还会预测求解其余元变量所需的熵。若该预测超过剩余熵，则回溯。否则，我们计算 mvar 的有效头部赋值数组，称为 定义域。对于每一个这样的赋值，我们执行该赋值，暂存所生成的约束，递归调用深度优先搜索，然后移除这些约束。最后，我们取消对 mvar 的赋值。为了分配新的元变量标识符，我们记录第一个未使用的标识符 pos。

```
variable ⟦Monadm⟧ (root : MVar) (cb step : m Unit) in partial
defdfs ⟦MonadLiftT CanonicalM m⟧(entropy : Float) (pos : Nat) : m
Unit := do step -- 用户提供的函数，用于日志记录、取消等。 let (some
(mvar, _), predicted) ← next root | cb if entropy <predicted then
return let domain ←mvar.domain pos for (assn, constraints) in
domain do mvar.assign assn process (·.push·) constraints dfs
(entropy/ domain.size.toFloat) (pos + assn.args.size) process (fun
_ => ·.pop) constraints mvar.assign none-- 取消对 mvar 的赋值
```

深度优先搜索中的计算是在 CanonicalM 单子中执行的。CanonicalM 是一个状态单子，可访问两个数组，每个数组都以元变量的标识符为索引。我们有与 ConstraintM 中相同的赋值，还有约束：对于每个元变量标识符，都会存储一个在该元变量上陷入 stuck 状态的约束数组。

```
结构CanonicalState其中
  赋值: 数组 (Option 可选赋值) :={} 约束: 数组 (数组 约
  束) :={}

abbrev CanonicalM := StateM CanonicalState
```

赋值给元变量时，我们只需修改赋值数组中由该元变量标识符所指定索引处的值。这可以通过 StateM 的 modify 函数来完成。

```
定义 MVar.assign : MVar → 可选赋值 → CanonicalM Unit := 函数 m
assn => 执行 修改 ({ · 并将 赋值[m.标识符] := assn })
```

process 函数会修改输入约束数组 cs 中 stuck 元变量对应的约束数组。修改函数 f 在添加约束时为 push，在移除约束时为 pop。

def process (f : 数组 约束 → 约束 → 数组 约束) (cs : 数组 约束) : CanonicalM Unit := do 对 cs 中的每个 c 执行 modify fun state => { state with constraints :=state.constraints.modify c.stuck.id (f · c) } 我们将容量定义为赋值的大小。

```
def 容量: CanonicalM 自然数 := do pure (← get)。赋值.size
```

extend 函数会向赋值中追加 none 值，并向约束中追加空数组。

```
def 扩展(大小 : 自然数) : CanonicalM Unit := domodifyfun状态 =>
{ 状态 with赋值: =状态。赋值 ++ replicate 大小 none 约束: =
状态。约束 ++ replicate 大小 #[]}
```

要理解定义域，我们需要对 MVar 进行恰当的定义。除了标识符之外，我们还会存储此 λ 表达式中绑定的首选名称（与 TypeMVar 一样），以及一个名为 lctx、由 TypeMVar 组成的链表。lctx 会收集那些输入已添加到局部上下文中的 TypeMVar，从而告知我们此 MVar 头部有效的德布鲁因索引对。

```
结构 元变量其中
  标识符: 自然数 lctx: TypeMVar 列
  表 preferredNames: 字符串数组
```

为了演示 MVar 的用法，我们将展示如何计算元变量 m 的定义域。m 的赋值所允许的德布鲁因值，是 m.lctx 中 TypeMVar 块类型的索引。对于该德布鲁因索引，允许的索引值是块类型.inputs 中 TypeMVar 头部类型的索引。我们的计划是构造一个赋值，其头部由德布鲁因索引和索引确定，参数则是新鲜元变量。首先，我们检查赋值数组是否有足够容量，以创建头部类型.inputs 所需数量的元变量。如果有，我们就为每个输入创建一个新的元变量，将该输入的绑定添加到局部上下文中，并将元变量绑定的首选名称设为该输入的首选名称。

然后，我们构造这个赋值，并将 m 赋给它。我们调用之前因 m 而处于 stuck 状态的约束的回调函数，以获得一个新的约束数组。如果没有抛出异常，则该赋值有效，我们便将此赋值及其约束添加到结果数组中。

```
定义 运行 (x: ConstraintM T) : CanonicalM (Option T) := 执行
返回 x。运行(← get)。赋值

def MVar.domain(m : MVar) (pos : Nat) := 执行 do let mut result :
Array (赋值, Array 约束) := #{} for (块类型, 德布鲁因) in m.lctx.zipIdx
do for (头部类型, 索引) in blockType.inputs.zipIdx do if pos  +头部类型.
inputs.size  > (← 容量) then continue let args :=头部类型.
inputs.zipIdx.map fun (input, k)  => ({ id := pos + k, lctx := input ::
m.lctx, preferredNames:= input.preferredNames } : MVar) let assn :赋
值 := { debruijn, index, args } m.assign assn if let someconstraints  ←
run ((←get).constraints[m.id]!.flatMapM 续延) then result:= res
ult.push (assn, constraints) return result
```

5.1 元变量选择

用于选择下一个要细化的元变量并估算剩余工作量的 next 函数，可以采用任意启发式实现。由于我们在 WHNF.eq 中以统一且相互独立的方式处理参数，并在 check 中处理输入，因此可以按任意顺序选择未赋值元变量。事实上，在依赖项尚未赋值时就先赋值后续元变量的能力，正是该算法在实践中性能出色的关键特性。确保完备性的唯一约束是：只有在所有元变量都已赋值时，next 才返回 none。

对于这个最小化实现，我们使用一种简单的启发式：选择项中最右边的元变量，除非更靠前的元变量具有一个作为刚性方程的约束，也就是说，一个方程 ?X $\equiv$ M，其中 M 不是元变量。我们优先选择具有刚性方程的元变量，因为它们受到严格约束；同时也优先选择最右边的元变量，因为它们会对更靠前的元变量产生强合一约束 [17]。

我们为一个由元变量和一个布尔值组成的二元组定义一个排序，该布尔值表示该元变量是否具有刚性约束：

```
实例 : Ord (元变量  × 布尔值) 其中比较
a b:= 比较 a.2 b.2
```

我们按照这个排序，选择最大的元变量作为下一个元变量。此外，我们估计每个元变量都会使预测熵乘以 5，除非该元变量具有刚性方程：

```
缩写 Next:= Option (元变量  × 布尔值)  × 浮点数部分定义
next (m : 元变量) : CanonicalM Next := 执行
```

如果 let someassn := (← get).赋值⁅m.id⁆! then assn.args.foldlM(init := (none, 1)) fun (fst, e1) s => 执行 let (snd, e2) ← next s pure (if compare fst snd == .gt then fst else snd, e1 * e2) else let rigid ←运行 ((← get).约束⁅m.id⁆!.anyM rigid) return (some(m, rigid.get!), if rigid.get! then 1 else 5)

要确定 m 是否具有刚性约束，我们对它的每个 stuck 约束调用刚性谓词。

刚性谓词由类型检查器传入 apply 函数，并通过赋值添加到约束中。为此，我们为该谓词准备了以下模板：

```
def rigid :ConstraintM 布尔值 := return truedef other (t : 项)
(args : 块) : ConstraintM 布尔值 :=return (← 约束 (do t.apply
argsrigid) 0).isEmpty
```

这里，rigid 表示该约束无条件刚性，而 other 表示当 t 应用 args 后没有因元变量而 stuck 时，该约束是刚性的。

至此，可居住性与合一的实现便告完成。最后补充一点性能建议：仅当块非空时，才将块添加到链表中；这可以通过对 Term.apply、Typ.inputs 和定义域进行以下修改来实现：

es := 如果 t.mvar.preferredNames.isEmpty，则 t.es，否则 args :: t.es ...
es := 如果 ty.mvar.inputs.isEmpty，则 ty.es，否则 args :: ty.es ...

局部上下文：= 如果(输入 input）。为空 则 m。局部上下文 否则 input :: m。局部上下文

6 个结果

要使用 Canonical-min 解决可居住性和合一问题，请创建要解决的元变量，调用所需的类型判断或定义等式判断，然后运行 iddfs。为方便起见，我们实现了一个 Lean 策略 canonical_min，它使用与 Canonical [17], 相同的 Lean 翻译层和接口，但使用 Canonical-min 使该类型可居住。为展示此策略的功能，我们将其与 DTT 中现有的可居住性系统进行比较：Twelf、sauto 和 mimer。

我们介绍了一个新的基准测试 DTTBench，其中包含 31 个改编自 Lean 标准库和 Mathlib 的问题。我们选取的定理都以非平凡的方式使用依赖类型（即证明中包含非恒定的 λ 表达式），不依赖超出 β-归约范围的定义性归约规则，并涉及等式、不等式、逻辑联结词、关系和集合等常见数学概念。我们在这些问题上测试了 Canonical-min、Twelf、sauto 和 mimer，并设置了 60 秒的超时限制。

	Canonical-min	Twelf	sauto	mimer
Eq	6/6	1/6	0/6	0/6
Leq	4/4	3/4	2/4	0/4
逻辑	6/6	3/6	3/6	2/6
关系	13/13	0/13	0/13	0/13
Set	2/2	1/2	1/2	0/2
总计	**31/31**	**8/31**	**6/31**	**2/31**

在大多数情况下，求解器在达到超时限制之前就报告失败，这表明预期的解并不在搜索空间中。此处所示的差异主要是定性的：Canonical-min 是完备的，而其他系统实现的是模式合一的变体。

现在，我们通过 Lean 定理示例展示 Canonical-min。借助依赖类型理论的完整表达能力，Canonical-min 可以证明等式的传递性等基本性质：

```
定理 Eq.trans {a b c : α} (h₁ : a = b)(h₂ : b = c): a = c :=-- 由 canonical 证明_min
  Eq.rec (motive := fun a_1 t => a = a_1) h₁ h₂
```

Canonical-min 可以对自然数上的不等式进行归纳推理：

定理 succ_le_succ (n m : 自然数) : n ≤ m → n.succ ≤ m.succ :=-- 通过 canonical_min fun a =>Nat.le.rec (motive := fun a t => n.succ.le a.succ) Nat.le.refl (fun{m} a a_ih => Nat.le.step a_ih) a

下面给出一个证明：对于关系 r和 p，如果 $f(a)$ 和 $f(b)$ 在 p 下相关，其中 a 和 b 在 r下相关，那么 $f(a)$ 和 $f(b)$ 在 p 的传递闭包下相关，其中 a 和 b 在 r的传递闭包下相关：

定理 TransGen.lift (f : α → β) (hab : 传递闭包 r a b) (h : ∀ a b, r a b → p (f a) (f b)) : 传递闭包 p (f a) (fb) :=-- 通过 canonical_min TransGen.rec (motive := fun b t => 传递闭包 (fun a c => p a c) (f a) (f b)) (fun {b} a _1 => TransGen.single (h a b a_1)) (fun {b c} a_1 a_2 a_ih => TransGen.tail a_ih (h b c a_2)) hab

最后，我们有一个康托尔对角线论证，证明从 A 到 A 的幂集的函数不可能是满射。

定理 Cantor(f : A → (A → Prop)) (f_inv : (A → Prop) → A) (Eq : Prop → Prop → Prop) (Not : Prop → Prop) (False : Prop) (f_surj : ∀ q x, Eq (f (f_inv q) x) (q x)) (P_ne_Not_P : ∀ P, Eq P (Not P) → False) : False :=-- 通过 canonical_min P_ne_Not_P

(f (f_inv fun a => 非 (f a a)) (f_inv fun a =>非 (f a a))) (f_满射 (fun a => 非 (f a a)) (f_inv fun a => 非 (f a a)))

这些内容均编码为独立的依赖函数类型，已包含在 DTTBench 中，并且其他任何系统都无法求解。

7 未来工作与结论

第2节所示的类型检查器与第5节所示的搜索逻辑在很大程度上彼此独立。类型检查器并不知道正在进行搜索，而搜索也不知道类型检查器如何判定约束违反。这表明该技术或许可以应用于更广泛的一类语言。例如，Canonical已经支持自定义的定义性归约规则。我们设想这对子结构、单价或模态类型理论会很有用。作为一个更简单的例子，若以布尔公式作为类型语言，将得到一个DPLL SAT求解器 [7], ，其中刚性约束即为单元传播。

我们可以确信 Canonical-min 是可靠的，因为算法要返回结果，类型检查器就必须运行至完成（包括处理延迟约束），且不能抛出异常。我们也可以确信它是完备的，因为我们会尝试为每个元变量赋予所有可能的值，并使用迭代加深搜索。不过，要将这些论证形式化，还需要针对类型论（例如正规化）以及我们的单子函数和数据结构给出许多证明。我们将这项工作留待未来完成。

我们已经证明，只要对依赖类型理论采用恰当的表示方式，并设计正确的控制流，就能用 185 行 Lean 代码编写出一个可靠且完备的可居住性与合一求解器。虽然 Canonical-min 的能力和优化程度不及 Canonical [17]„ 但我们希望这一最小化实现能够作为理解和分析该算法的参考，也能成为为依赖类型理论开发原型定理证明工具和程序合成工具的基准。

致谢。 本材料所述工作得到了美国国家科学基金会研究生研究奖学金计划（资助编号 DGE2140739）以及美国国家科学基金会项目 DMS 2434614 的支持。本材料中表达的任何观点、研究结果和结论或建议均属于作者，不一定反映美国国家科学基金会的观点。我们感谢 Tesla Zhang、Harrison Grodin 和 Wojciech Nawrocki 提供的宝贵见解。

利益披露。 作者声明不存在任何利益冲突。

参考文献

1. Barrett, C., Hadarean, L., King, T., Nötzli, A., Reynolds, A., Tinelli, C.：cvc5：一款多功能、达到工业级强度的 SMT 求解器。载于：国际计算机辅助验证会议（CAV）。《计算机科学讲义》，第 13371 卷，第 415–442 页。Springer（2022）。https://doi.org/10.1007/978-3-031-13185-1_5

2. Bertot, Y., Castran, P.：交互式定理证明与程序开发：Coq'Art《归纳构造演算》。Springer 出版公司，第一版（2010）3. Bove, A., Dybjer, P., Norell, U.：Agda 简介——一种支持依赖类型的函数式语言。载于：Berghofer, S., Nipkow, T., Urban, C., Wenzel, M.（编）《高阶逻辑定理证明》，第 73–78 页。Springer 柏林海德堡，柏林，海德堡（2009）。https://doi.org/10.1007/978-3-642-03359-9_64. Charguéraud, A.：局部无名表示。《自动推理杂志》**49**(3)，363–408（2012年10月）。https://doi.org/10.1007/s10817-011-9225- 2, https://doi.org/10.1007/s10817-011-9225-2 5. mathlib 社区，T.：Lean 数学库。载于：第九届 ACM SIGPLAN 国际认证程序与证明会议论文集，第 367–381 页。CPP 2020，计算机协会，纽约，纽约州，美国（2020）。https://doi.org/10.1145/3372885.3373824 6. Czajka, L.：基于类型占据的 Coq 实用证明搜索。载于：《自动推理：第十届国际联合会议 IJCAR 2020》，法国巴黎，2020年7月1日至4日，会议录，第二部分，第 28–57 页。Springer 出版社，柏林，海德堡（2020）。https://doi.org/10.1007/978-3-030-51054-1_37. Davis, M., Logemann, G., Loveland, D.：用于定理证明的机器程序。Commun. ACM **5**(7)，394–397（1962年7月）。https://doi.org/10.1145/368273.368557, https://doi.org/10.1145/368273.368557 8. de Bruijn, N.：带有无名占位符的 λ 演算记法：一种用于公式自动操作的工具及其在 Church–Rosser 定理中的应用。《数学研究》（会议录）**75**(5)，381–392（1972年）。https://doi.org/10.1016/1385-7258(72)90034-0 9. Dowek, G.：类型系统立方体的完备证明合成方法。《逻辑与计算杂志》**3**(3)，287–315（1993年6月）。https://doi.org/10.1093/logcom/3.3.287 10. Huet, G.P.：三阶逻辑中合一的不可判定性。《信息与控制》**22**(3)，257–267（1973年）。https://doi.org/10.1016/S0019-9958(73)90301-X 11. Huet, G.：用于类型化 λ-演算的合一算法。《理论计算机科学》**1**(1)，27–57（1975年）。https://doi.org/10.1016/0304-3975(75)90011-0 12. Kokke, W., Swierstra, W.：Agda 中的 Auto。载于：Hinze, R., Voigtländer, J.（编）《程序构造数学》，第 276–301 页。Springer 国际出版公司，查姆（2015年）13. Kovács, L., Voronkov, A.：一阶定理证明与 Vampire。载于：Sharygina, N., Veith, H.（编）《计算机辅助验证》，第 1–35 页。Springer 柏林海德堡，柏林，海德堡（2013）。https://doi.org/10.1007/978-3-642-39799-8_114. Lindblad, F., Benke, M.：Agda 中自动定理证明工具。载于：Filliâtre, J.C., Paulin-Mohring, C., Werner, B.（编）《证明与程序的类型》，第 154–169 页。Springer 柏林海德堡，柏林，海德堡（2006年）15. Moura, L.D., Bjørner, N.：Z3：一种高效的 SMT 求解器。载于：系统构造与分析工具和算法（TACAS）。《计算机科学讲义》，第 4963 卷，第 337–340 页。Springer（2008年）。https://doi.org/10.1007/978-3- 540-78800-3 24_16. Moura, L.d., Ullrich, S.：Lean 4 定理证明器与编程语言。载于：Platzer, A., Sutcliffe, G.（编）《自动演绎——CADE 28》，第 625–635 页。Springer 国际出版公司，查姆（2021年）。https://doi.org/10.1007/978-3- 030-79876-5 37_

17. Norman, C., Avigad, J.：用于 Lean 自动定理证明的 Canonical。载于：Forster, Y., Keller, C.（编）《第16届交互式定理证明国际会议》（ITP 2025）。《莱布尼茨国际信息学会议录》（LIPIcs），第352卷，第14:1–14:20页。达格施图尔城堡 – 莱布尼茨信息学中心，达格施图尔，德国（2025）。https://doi.org/10.4230/LIPIcs.ITP.2025.14, https://drops.dagstuhl.de/entities/document/10 14 18 4230 2025 2 3 . /LIPIcs.ITP. . . Schulz, S., Cruanes, S., Vukmirović, P.：更快、更高、更强：E . . 载于：Fontaine, P.（编）第27届 CADE 论文集，纳塔尔，巴西。第495–507页。人工智能讲义第11716号，Springer (2019 10 1007 978 3 030 29436 6 _ 29)。https://doi.org/ . / - - - - . Schürmann, C., Pfenning, F.：用于 LF 的简单元逻辑中的自动定理证明。载于：Kirchner, C., Kirchner, H.（编）《自动演绎——CADE-15》。第286–300页。Springer 柏林海德堡，柏林，海德堡（1998）。https://doi.org/10 0054266 20 1007. /BFB . Skystedt, L.：Agda 的新型综合工具。硕士论文，查尔姆斯理工大学和哥德堡大学，哥德堡，瑞典（2022）21. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.：让高阶超位置发挥作用。《自动推理杂志》**66**，541 564 2022 10 1007 10817 021 09613 – ( ). https://doi.org/ . /s - - -z