具有可扩展类型的逻辑编程

Ivan PerezKBR @ 美国国家航空航天局艾姆斯研究中心，加利福尼亚州，美国
ivan.perezdominguez@nasa.gov

Angel Herranz马德里理工大学，马德里，西班牙
angel.herranz@upm.es

逻辑编程语言在声明式和简洁性方面具有明显优势。然而，逻辑编程的理念在其他编程社区中遭遇了阻力，通常也未被其他范式和语言所采用。本文提出了一种新颖的方法，可将逻辑编程融入现有的带类型函数式编程语言代码库。我们的方法能够与宿主语言集成，同时不牺牲静态类型，并发挥带类型函数式编程的优势，例如多态和高阶特性。为此，我们结合了三种思路。首先，我们采用可扩展类型技术，使宿主语言的值能够包含逻辑变量。其次，我们实现了一种合一算法，适用于支持特定操作的任意数据结构。第三，我们引入一种领域专用语言，用于定义和查询谓词。我们通过一系列示例展示了这一方案，并提供辅助工具，使用户能够方便地使用这种表示法，表明所提出的方法不仅在技术上可行，而且具有实用性。我们的想法已在 Haskell 语言中实现，并取得了非常好的效果。

1 引言

命令式语言和函数式语言中的定义围绕函数和过程构建，执行时需要提供输入并计算结果。例如，对于 Haskell 类型 data Nat = Zero | Suc Nat，我们可以将加法定义为：

```
加法 :: 自然数 -> 自然数-> 自然数 加法
零 y =y 加法 ( 后继 x) y = 后继 ( 加
法 x y)
```

尽管从抽象层面来看，函数是集合之间的关系，但在 Haskell 这样的语言中，通常无法将函数作为关系来处理。例如，我们无法用 plus 高效地计算减法函数，也无法通用地计算 plus 关系中所有输入和输出的元组。与之相比，我们可以看看如何在 Prolog 这样的逻辑编程语言中编写类似的谓词：

```
plus(zero, Y,Y). plus(后继(X), Y, 后继(Z)) :-
plus(X,Y,Z).
```

在 Prolog 中，如果 C 表示 A 与 B 的和，则 plus(A,B,C) 为真。这个谓词可用于对数字进行加法、用一个数减去另一个数、检查两个数的和是否等于给定的第三个数，或获取满足该关系的输入与输出元组：[1]

[1]我们手动停止为最后一个查询生成解。

M. Gebser、D. Inclezan、F. Ricca、M. Carro 和 M. Truszczynski：第41届国际逻辑编程会议（ICLP 2025），EPTCS 439，2026，第248–262页，doi:10.4204/EPTCS.439.18

?- plus(后继(后继(zero)),B,后继(后继(后继(zero)))). B =后继(zero). ?- plus(zero,zero,后继(zero)). false. ?- plus(A,后继(zero),C). A = zero, C =后继(zero) ; A = 后继(zero), C = 后继(后继(zero)) .

将逻辑编程中的理念集成到其他语言中，通常需要对现有代码库及其所用类型进行大量修改。根据所采用的方法和处理查询的引擎，这也可能导致静态类型安全性或宿主语言其他特性的丧失（例如多态、高阶特性）。

本文介绍了一种技术，可将逻辑编程风格的谓词集成到使用其他编程语言的现有代码库中。我们使用 Haskell 展示这一方案，但这些理念也适用于其他语言。通过示例，我们说明了这种方法只需程序员付出很少的工作，就能涵盖逻辑编程语言的许多用例，并且可以借助便捷的记法实现。具体而言，本文的贡献如下：

- 我们提出了一种逻辑编程接口，便于在代数数据类型中引入逻辑变量并表达合一约束（第3节）。
- 我们证明，所提出的方法适用于多态类型，并能利用宿主语言的类型推断和高阶机制，实现类型安全的高阶逻辑编程（第4节）。
- 我们通过引入剪枝扩展了该语言，使用户能够提升性能、实现确定性，并编码失败即否定（第 5 节）。

第6节讨论了一个展示我们方案的实现，第 7 节详细介绍了相关工作，第 8 节提出了未来工作。

2 背景

可扩展类型 [24] 是一种设计模式，其中数据类型由一个类型函数参数化，该函数应用于定义中的每个元素。例如，给定一个表示表达式的类型，如 data Expr = Const Double | Add Expr Expr | Neg Expr，我们定义相应的可扩展类型：

data ExprF f = ConstF (f Double)| AddF (f (ExprF f)) (f (ExprF f))|否定F (f (ExprF f))

如果我们将多态类型 Identity 用作类型函数 f，得到的表示就与原始 Expr 同构。其他参数化类型和类型函数则会产生不同的结果。例如，可以使用一种将元素与 Int 元组配对的类型，来为值标注其在输入文件中的行号和列号；这在编译器中很有用，可用于报告错误信息。应用 Maybe 或 Either 这样的可扩展类型，会使每个元素都变为可选项；这种表示在解析时很有用，可用于标记抽象语法树中解析失败的分支

（AST）。其他类型函数可以修改类型定义，以引入新分支、裁剪分支、替换元素等。可扩展类型的组合能够表示语言嵌入。

将特定类型函数应用于可扩展类型，并不能决定该类型应如何解释。例如，Either String 可用于为抽象语法树中解析失败的分支添加注释，说明无法从输入文件中解析出相应值的原因；也可用于将抽象语法树中的分支替换为具有给定变量名的变量。后续章节将运用这一思路，在谓词定义和逻辑编程查询中用变量替换数据类型的部分内容。

在本文余下部分中，我们将以这种方式参数化的类型简称为可扩展类型。第 7 节将进一步讨论其他通过定义中使用的类型函数来参数化类型的方法。

3 带可扩展类型的逻辑编程

本节介绍用于定义和组合谓词的原语，以及用于描述代数数据类型的值之间关系的方法。我们首先介绍基本类型以及简单的原语和连接词。随后将说明如何用逻辑变量替换值的部分内容，以及如何表达涉及含变量类型的关系。

3.1 目标、原语与布尔组合子

我们提出的基本类型是 Goal，它表示一个逻辑目标，简单来说，就是必须证明的某件事。目前我们将该类型保持为抽象类型，稍后再讨论实现细节。为了与 Goal 交互，我们提供函数 repl :: Goal -> IO ()。将 Goal 传入该函数后，它会逐个给出可能的解，类似于逻辑编程语言的交互式解释器。如果目标成立需要满足约束，repl 就会打印这些约束；否则，它会打印“true.”或“false.”。本文为便于阅读，对 repl 的查询和结果进行了对齐。

3.1.1 原语

我们提供 succeed :: Goal，它无需额外约束即可成立；与之对应的是 fail :: Goal，它总是失败。我们可以在会话中使用 repl 函数对任一目标求值，如下所示：

```
> repl fail
false.
```

3.1.2 布尔联结词

布尔联结词 和 和 或的对应运算，我们分别记作 (@@) 和 (@|)，允许用户组合目标：

```
(@@) :: 目标 -> 目标 -> 目标 (@|) ::
目标 -> 目标 -> 目标
```

示例 以下查询展示了如何使用布尔联结词组合 succeed 和 fail。结果应该很直观：

```
> 交互式解释器 (succeed @@ fail)
false.> 交互式解释器 (succeed @@ (f
ail @|succeed)) true.
```

3.2 项与逻辑变量

为了在值中引入逻辑变量，我们将类型函数应用于可扩展类型。我们引入一种自定义和类型 Term，它可以表示带有名称的逻辑变量，也可以表示某个给定类型的实际值：

```
data Term a  = 变量 String  | 复合项 a
```

当 Term 应用于可扩展类型时，后者中的每个元素都可能被逻辑变量替换，因此我们可以描述其中某些部分是具体值、某些部分尚未确定的值。

示例 给定使用如下数据类型定义来编码皮亚诺数：data Nat = Zero | Suc Nat，那么 Haskell 中等价的可扩展类型为：

```
数据 自然数F f  = 零构造子  |后继构造子 (f (自然数F f))
```

我们可以使用 NatF Term 表示一个自然数，其中定义的一部分被替换为变量。若要使整个数都可以替换为变量，我们需要在该类型外再包一层 Term：

```
类型 自然数项  =项 (NatF 项)
```

NatTerm 类型的值包括 Var "y"，表示由变量 "y" 指代的自然数；Compound (SucF (Var "x"))，表示 "x" 的后继；以及 Compound (SucF (Compound ZeroF))，表示 1。在 Prolog 中，这些项分别可以编码为 Y、suc(X) 和 suc(zero)。

3.3 项合一

我们设计了一种领域特定语言（DSL），用于编写类型上的谓词，并可对其执行项合一和变量替换。为合一两个项，我们提供了 (===) :: Term a -> Term a -> Goal.[2]

示例 我们可以如下定义后继谓词：[3]

```
isSuc :: 自然数项 ->  自然数项 ->  目标
isSuc x y  =  复合项 (后继构造子 x)  ===y
```

[2]该函数并非完全多态；我们会在第6节详细说明适用于参数 a 的约束。[3]请与 Prolog 程序比较：_ suc(X,Y) :- suc(X) = Y.

现在，我们在交互式解释器会话中查询这个谓词，以检查它对两个基项值是否成立：

```
[4]> repl $ isSuc (Compound (SucF (Compound ZeroF))) (Compound (SucF
(Compound (SucF (Compound ZeroF))))) true.> repl $ isSuc (Compound (SucF
(Compound ZeroF))) (Compound (SucF (Compound ZeroF)))

false.
```

我们这种方法真正强大的地方在于，现在可以使用变量来提供一个值，并“获得”另一个值，或者更准确地说，获得答案替换：

```
> 交互式解释器 $ isSuc (复合项 (后继构造子 (复合项 零构造子))) (变量"x")
x = 复合项 ( 后继构造子 ( 复合项 ( 后继构造子 ( 复合项零构造子 )))).>
交互式解释器 $ isSuc (变量"x") (复合项 (后继构造子 (复合项 零构造子)))
x = 复合项 零构造子.
```

3.4 存在量化

到目前为止所述的变量作用域默认是全局的。在函数内部使用的固定名称变量，会被视为与同一查询其他位置使用的同名（且类型相同）变量是同一个变量。

为了在谓词的主体中引入自由变量，类似于在逻辑编程语言的谓词定义中于前件引入自由变量，我们定义函数 exist :: (Term a -> Goal) -> Goal。当函数 exists 应用于一个谓词参数时，它会确保提供给该谓词的变量是自由的。为了避免名称冲突，我们建议用户始终使用 exists 引入变量。

示例利用目前为止给出的所有定义，我们可以实现谓词 leq（即小于或等于），用于比较两个自然数 x 和 y。比较的第一条规则是：如果第一个数 x 为零，那么 leq x y 必然成立，因为不存在比它更小的数，也就是说，x === 复合项 零构造子。第二条规则是：如果两个元素分别是其他元素 x’ 和 y’ 的后继，那么当 x’ 和 y’ 满足目标时，该目标也成立。结合这两条规则，我们得到：

```
leq :: 自然数项 -> 自然数项 -> 目标 leq x y = x ===复合项 ZeroF @|( 存在 $ \x’ -> 存在
$ \y’ ->x === 复合项 (SucF x’) @@ y === 复合项 (SucF y’) @@ leq x’ y’ )
```

在定义开头引入自由变量，有助于将规则分组并提升可读性：

```
leq:: 自然数项 -> 自然数项 -> 目标 leq x y = 存在 $ \x’ -> 存在 $ \y’ ->x ===
复合项 ZeroF @| x === 复合项 (SucF x’) @@ y ===复合项 (SucF y’) @@
leq x’ y’
```

如上所示，谓词可以是递归的。

[4]我们用 基项 指不包含变量的项。在讨论数据类型和泛型编程时，ground 一词也有其他含义，但我们这里只用它表示前一种含义。

3.5 记法

我们提出的想法也适用于其他语言，但 Haskell 重载记法和定义运算符的能力，可以让逻辑编程更加方便。具体来说，Haskell 允许我们定义新的运算符，并调整它们的结合性和优先级，使 (@@) 的绑定比 (@|) 更紧密，而 (===) 的绑定又比这两者都更紧密。我们分别为 Compound 和 Var 定义同义词 C 和 V，并定义模式同义词 Zero = C ZeroF 和 Suc x y = C (SucF x y)。借助这些特性，leq 现在可以更简洁地定义如下：

```
leq:: 自然数项 -> 自然数项 -> 目标 leq x y =
exists $ \x' -> exists $ \y' ->x ===Zero @| x ===
后继 x' @@ y ===后继 y' @@ leq x' y'
```

为了帮助理解我们方法的技术细节，在本次阐述中，我们尽量避免过度依赖语法糖。我们的实现提供了一些辅助工具，使逻辑编程更加便捷；我们将在第6节中讨论这些工具。

4 多态与高阶

迄今所述的方法也适用于使用多态类型编写谓词。让我们以多态列表的类型为例进行说明，这种类型常用于逻辑程序和函数式程序中。标准列表类型定义 data List a = 空列表 | Cons a (List a)，可以通过添加一个额外的类型函数来扩展，如下所示：

```
数据列表F f a =NilF | 列表构造子 (f a) (f (列表F f a))
```

通过应用类型函数 Term 到 ListF，我们可以在列表元素或列表尾部的任意位置使用逻辑变量：

```
类型 列表项 a = 项 (ListF 项 a)
```

示例 我们可以将 ListTerm 与 NatF 结合起来表示自然数列表：

```
类型 NatListTerm = 列表项 (NatF 项)
```

下面这个目前较为冗长的项编码了列表 `[0, 1, 2]`:

```
l1 ::自然数列表项 l1 = C $ 列表构造子 (C 零构造子) $ C $
列表构造子 (C (后继构造子 (C 零构造子))) $ C $ 列表构造子
(C (后继构造子 (C (后继构造子 (C 零构造子)))) 空列表
```

以下项编码了一个以 1 开头、尾部由变量“tl”表示的列表：

```
l2 :: 自然数列表项 l2 = C (列表构造子 (C (后继(C
Zero)))(V "tl"))
```

记法 和之前一样，我们引入模式同义词，以简化列表项类型的项的书写：Cons x y = C (列表构造子 x y) 和 Nil = C NilF。使用目前定义的所有模式同义词，可以将上一个示例中的列表 l1 定义为：

```
l1 :: 自然数列表项 l1 = Cons Zero $ Cons (SucZero) $ Cons (Suc (Suc
Zero)) Nil
```

我们的实现允许我们写出表达式，例如 `[0, "x"]`，表示一个列表，其第一个元素为 0，第二个元素为变量 "x"。我们将延迟表示法的辅助工具留到第 6 节介绍，以帮助读者直观地理解我们的方法如何运作。

4.1 多态与类型安全

使用可扩展类型并不会阻止宿主编程语言执行类型检查，包括检查那些非可扩展变体原本具有多态性的类型。如果我们在谓词定义或合一约束中使用了类型错误的项，类型检查器会像检测其他类型错误一样将其检测出来。

示例 以下谓词用于检查某个元素是否为列表的表头：

```
isHead :: 列表项 a -> 项 a -> 目标 isHead x y =
exists $ \tl -> x ===Cons y tl
```

如果我们使用错误类型的第二个参数调用 isHead，Haskell 编译器的类型检查器会发出警告：

```
> repl $ isHead l1 ((Var"x") :: Term Bool)<交互式解
释器>:9:19: 错误：无法将类型 'Bool' 与 'NatF Term' 匹
配 期望类型：Term (NatF Term) 实际类型：Term
Bool
```

这种级别的安全性在使用自由变量时尤其重要。具有强静态类型的宿主语言可以确保我们以类型一致的方式使用变量。

示例 我们可以通过检查列表头部或递归检查尾部来判断某个值是否在列表中。对于空列表没有相应的规则，因此在这种情况下目标会失败：

```
member ::项 a -> 列表项 a -> 目标 member x
xs =
    ( 存在 $ \tl -> xs ===Cons x tl ) @| ( 存在 $ \hd -> 存在 $ \tl -> xs ===Cons hd tl
  @@ member x tl )
```

第二条规则中的变量 hd 的类型是 Term a，而 tl 的类型是 List a。编译器可以推断这一点，因为它们都是 Cons 的参数，并且得到的项与 xs 合一，而 xs 的类型是已知的。例如，如果我们引入条件 xs === hd 或 member tl tl，编译器就会检测到我们以不一致的方式使用了变量。

让我们进一步通过一个用于检查列表是否有序的谓词，来展示我们方法的类型安全特性。对于包含零个或一个元素的列表，该谓词显然成立；如果元素更多，我们就检查前两个元素，并递归处理列表的尾部。

```
有序 v  =v  ===空列表 @|  ( 存在 $ \e1 ->  v  ===Cons e1 空
列表) @|  ( 存在 $ \e1 ->  存在 $ \e2 ->  存在 $ \ts ->v  ===
Cons e1 (Cons e2 ts) @@ leqe1 e2 @@ 有序 (Cons e2 ts) )
```

在这种情况下，编译器推断出 v 的类型为自然数列表，它被定义为 ListTerm (NatF Term)，并展开为 Term (ListF Term (NatF Term))。

4.2 高阶逻辑编程

Haskell 对一等函数的支持立即赋予我们的方法以高阶能力，使我们能够将谓词作为参数传递给其他谓词。

示例 我们通过 sortedWith 的一个广义版本来说明这一点，该版本将比较谓词 作为参数:

```
sortedWith :: (Term a ->  Term a ->  目标) ->  ListTerm a ->目标
sortedWith compare v  =
      v  ===空列表 @|  ( 存在 $ \x ->  v  ===Cons x 空列表)
  @|  ( 存在 $ \x1->  存在 $ \x2->  存在 $ \xs ->v  ===Cons
  x1 (Cons x2 xs) @@ compare x1 x2 @@ 带比较的有序
  compare (Cons x2 xs) )
```

由于 Haskell 采用强类型和静态类型，它提供了比大多数 Prolog 实现更高的安全性，[5] 因为这些实现无法在不添加手写运行时检查的情况下确保类型匹配。一般而言，如果调用 Prolog 谓词时传入了类型错误的参数，可能会得到不正确的结果，这使得这类类型错误难以识别。传入类型错误的参数可能会返回 false，就像该谓词对这些输入不成立一样（因为它确实不成立！），但也可能错误地返回 true（例如，即使 1 不是列表，append([],1,1) 仍然为 true）。

我们可以采用相同的方法来泛化函数并将其转化为谓词，例如标准函数 map，它会对列表中的每个元素应用一种变换，如下列谓词 mapP 所示:

```
mapP :: (项 a ->  项 b ->  目标) ->  列表项 a ->  列表项 b ->  目标 mapP f l1 l2  =l
1  ===  空列表@@ l2  ===空列表 @|  ( exists $ \l10->  exists $ \l1s->  exists $ \l20 ->
exists $ \l2s ->l1  ===Cons l10 l1s @@ l2  ===Cons l20 l2s @@ f l10 l20@@
mapP f l1sl2s )
```

[5] Som例如 Ciao Prolog 等 Prolog 实现支持通过编译时断言进行静态分析 .

给定一个谓词 isSuc，它将每个数字与其后继配对，我们可以使用它通过 mapP 将列表中的每个元素加 1，如下所示：

```
listPlusOne :: 列表项 (NatF 项) -> 列表项 (NatF 项) -> 目标 listPlusOne = 映
射P isSuc
```

Haskell 的类型检查器通过在编译时确保 isSuc 具有 Term (NatF Term) -> Term (NatF Term) -> 目标 类型来防止运行时错误。

5 剪枝

在逻辑编程中，剪枝 限制使用回溯来搜索其他解。请考虑以下（不正确的）求余算法实现：

```
remainder :: 自然数项 -> 自然数项 -> 自然数项 -> 目标 remainder n q r = lt
n q @@ n === r @| exists $ \diff -> plus q diff n @@ remainderdiff q r
```

如果 q 为零，这个定义就无法正常工作，因为第一条规则会失败，而执行第二条规则会使程序陷入无限循环。遗憾的是，添加一条类似 q === Zero @@ fail 的保护规则也无济于事，因为当 q 为零时，该规则会失败，求值随后回溯，最终又落入最后一条规则。

为避免这种情况，我们引入函数 scope 和 (@!)，用来控制回溯。受 Prolog 中剪枝的概念启发，我们将 (@!) 称为自己的剪枝运算符。我们用这两个函数重新编写 remainder，如下所示：

```
余数 :: 自然数项 -> 自然数项 -> 自然数项 -> 目标 余数 n q r =scope $ q
===Zero @! fail @| lt n q @@ n ===r @| (exists $ \diff -> plus q
diff n @@余数 diff q r)
```

只要 (@!) 的使用范围由 scope 限定，从而限制剪枝的作用范围，调用 remainder 的程序就无需知道该谓词是用剪枝实现的。如果 remainder 的定义中没有 scope，那么在某条规则中使用 remainder 的谓词会发现：如果 remainder 因 q 为 Zero 而失败，其他可选规则（即 (@|)）就会被跳过。

失败即否定 剪枝可用于实现一种否定形式，具体如下： h:

```
neg:: 目标 -> 目标 neg p = scope $ p
@! fail @| succeed
```

示例 通常需要声明两个项无法合一，为此我们将运算符(=/=)定义为：

```
(=/=) :: 项 a -> 项 a -> 目标 (=/=) x y =
neg(x === y)
```

类似地，我们可以实现一个谓词：当给定项不是给定列表的成员时，该谓词才成立：

```
notMember :: 项 a -> 列表 a -> 目标
notMember x xs  = 否定(member x xs)
```

这种称为失败即否定的否定，是一种较弱形式的否定。如果在合一算法尝试判断目标是否成立时，(=/=) 的任一参数仍是变量，那么合一 x === y 就会成立，导致其否定失败。这限制了这种否定形式的实用性。例如，不能使用上面定义的 notMember 来查找不属于某个列表的变量可能取值。相反，必须先将该项实例化为基项。为此，我们提供谓词 isGround；对于基项具有有限表示形式的任何类型，只有当给定参数不包含变量时，该谓词才成立。

6 实现

我们使用 Haskell 实现了本文中的构想，包括表示项和目标的类型、目标构造函数和组合子、定义类型必须支持哪些操作才能进行合一的类型类、合一算法，以及用于求值目标的执行函数。总体而言，我们的实现仅需 350 行代码（不计空格和注释）。我们实现这一方案，旨在展示文中阐述的能力，并探索设计决策、语法和嵌入方式。评估该方案的性能并将其与现有逻辑编程实现进行比较，不在本文讨论范围内，将留待未来工作。

高级描述 我们的实现定义了两种关键类型：第3节中介绍的多态类型项 Term，以及表示目标的类型 Goal。为了对任意类型的项进行合一，我们要求项支持三种操作：合一两个特定类型的项、检查某个变量是否被用于某个项中，以及在另一个项中用某个项替换一个变量。我们通过 Haskell 的类型类来封装这些操作，其中最关键的是 Logic 类型类，它表示已定义上述操作的类型。

为评估目标，我们提供了多个函数，包括：repl :: Goal -> IO ()，它会逐个打印解，让用户通过键盘控制解的生成；以及 findAll :: Logic a =>Term a -> Goal -> [Term a], ，它会提供满足某个目标时参数变量项的所有取值。

我们的实现使用带树状结构的目标表示法，并将合一约束置于节点中。内部函数 solve 会遍历这棵树，累积合一约束，将变量替换为其他项中的预期值，并丢弃无法合一的分支。函数 solve 会生成所有可能的解，每个解都会为变量项赋值。函数 repl 利用 Haskell 与生俱来的惰性，逐个生成并打印解。

记法与易用性 为了让我们的领域特定语言更加便捷，我们实现了一些功能，使记法简洁而熟悉，并减少用户需要编写的代码量。我们的实现使用泛型编程 [18] 来生成我们的合一函数在处理代数数据类型时所需的类实例。

[6]Our 实现已公开发布于 https://github.com/ivanperez-keera/telo s

例如，对于类型 NatF，用户需要声明若干类型类实例，但无需手动实现它们。我们的方案还便于书写一些常见类型的项。例如，作为基项时，佩亚诺自然数会显示为 1、2 等；否则则显示为一个数字加一个变量（例如，`1 + x`、`55 + z`）。我们也为 ListF 实现了类似的辅助功能，因此其值可以用熟悉的记法显示和书写（例如，1 : x : 5 : xs）。在需要项的地方，字符串字面量会被解释为变量项。总体而言，这使表达式非常简洁：

```
> 交互式解释器 (plus 1 "x" 5) x = 4.
> 交互式解释器 (isTail
[1，2，3] [2，3]) true.
```

其他支持高阶类型多态或动态地将值替换为和类型的语言，未必具备相同的语法重载机制，因此在这类语言中，记法可能会更加繁琐。

7 相关工作

函数逻辑编程语言 利用理论框架和高效实现来融合逻辑编程与函数式编程的语言，早已受到研究者关注 [4, 6]。这一类语言包括 BABEL [19], K-LEAF [8],ALF [11], Curry [10], 和 Escher [17], , 它们支持函数式风格；还包括 G¨odel [14], Mer- cury [28], 和 λProlog [21], , 它们采用逻辑编程风格。Curry 尤其深受 Haskell 启发，但也引入了逻辑变量，并使用窄化作为操作语义，以计算含有自由变量的表达式的值 [9]。我们的工作并非创建一种新语言，而是展示如何赋予现有函数式语言逻辑编程能力，无需额外的编译器扩展，也无需调用外部逻辑编程引擎。

逻辑编程语言中的“函数”Prolog 提供了有限的高阶能力，例如 call/N 和 apply/3 [22]。Prolog 实现 Ciao [12] 通过其元编程库，允许以函数式风格使用谓词，并将最后一个参数视为函数的结果。在我们的方案中，函数和高阶能力由宿主语言内置，程序员可以立即加以利用。此外，由于我们依赖静态类型语言，我们的方法能够提供一定程度的静态安全性。

带类型的逻辑编程语言Prolog 过去曾扩展出静态类型[20, 25, 3], , 但这些扩展并未集成到大多数广泛使用的 Prolog 系统中。Ciao Prolog [12] 实现提供了一种基于正则类型的断言机制，并对其进行静态检查。相比之下，我们的方案利用了宿主编程语言的类型系统。

逻辑编程嵌入 [29], [5], [27] 和 [7] 此前尝试将逻辑编程嵌入函数式语言，但需要手动调整类型才能使用它们。

逻辑谓词。 [15] 将关系嵌入 O'Caml 的工作需要引入投影和注入，以便在项和函数值之间转换。相比之下，我们的方法适用于任意代数数据类型，并借助泛型编程工具简化对其进行操作的过程。采用系统化的扩展模式，而非手动编写扩展，可以形成规律且可预测的方式，将变量添加到代数数据类型中。

在目标求值的实现方面，以往的嵌入式实现使用基于求值单子的解释器 [29, 5, 27, 26], make goals data 流，并在其他方面采用严格求值的环境中显式引入回溯惰性流单子 [15], , 或者使用延续传递风格和异常来更高效地求值逻辑程序 [7]。相比之下，我们将目标表示为合一约束树，并以一个用于生成自由变量的计数器作为参数。与使用（惰性）状态单子相比，我们的编码能够更精细地控制计数器，并引入了一个层次的惰性，这对高效生成解和实现交互式解释器至关重要。miniKanren [15] 与我们工作的另一个实现差异在于，前者使用了封装在类型安全 API 背后的类型不健全的内部表示。我们的方法在实现合一时不牺牲类型安全，而且项合一只能应用于类型相同的项。

这些嵌入在表达能力上也与我们的工作不同：例如，我们只支持相等约束，并通过剪枝实现一种有限形式的不等，采用的是一种（弱形式的）否定；而 miniKanren 和 [26] 的工作同时支持相等和不等。

一篇较新的论文介绍了 typedKanren [16], ，这是一种在 Haskell 中嵌入的、基于 miniKanren 的关系式语言。与我们的方案类似，typedKanren 支持在 Haskell 中编写强类型谓词，并采用了一种与我们的项类型相似的类型（区别在于它增加了严格性，并使用 Int 而不是 String 来表示变量）。一个关键区别是，typedKanren 要求为标准类型（例如二叉字典树）编写逻辑变体，而我们的表示基于可扩展类型，因此用途更广。与 miniKanren 一样，typedKanren 支持不等，而我们不支持一般情形下的不等。在 typedKanren 中，目标是单子，因此指定谓词时采用了不同的风格；而我们语言中的目标不是单子，因此我们提供了额外的构造和语法辅助，使记法更方便，也更接近 Prolog 的语法。

高阶类型参数化 我们采用可扩展类型，使用类型函数对类型进行参数化。 [23] 提出了一种替代技术，使用类型族，并在代数数据类型定义的每个分支中添加参数 f，而不是在类型内部的每个元素中添加参数。Najd 等人的工作还允许为抽象数据类型的所有元素引入类似 Term 的包装器。不过，我们发现可扩展类型易于使用，因为只需将 Term 应用于可扩展类型，就能轻松地把类型中任意 部分替换为变量。

barbies [1] 库实现了通用机制，用于处理由函子参数化、类似于可扩展类型的类型；而 Barbies-th [2] 或 Higgledy [13] 或许能帮助在 Haskell 中生成高阶类型。我们尚未研究如何利用这些方法。

8 未来工作

本文展示了如何在静态类型函数式编程语言中嵌入逻辑编程。为此，我们使用可扩展类型，用变量替换代数数据类型的任意部分，并提供了一种机制，用于表达含变量的值之间的合一约束。我们进一步扩展了语言，加入布尔联结词以及引入自由变量的方法。通过示例，我们展示了如何利用宿主语言的类型推断和高阶特性，在不牺牲类型安全的前提下实现代码复用。最后，我们概述了实现，并评估了它与其他方法之间的差异。

本文提出的方法可用于表示其他类型的约束，从而使我们能够在 Haskell 中实现约束逻辑编程。在语言中引入不等约束，将克服我们当前通过“失败即否定”实现不等的局限。

我们目前正在探索如何利用本文所提供的语言，在基于属性的测试中生成满足约束的值，而不是先生成值，再根据约束进行筛选（从而丢弃许多值）。

我们的实验表明，目标求值函数的语义与 Prolog 的语义一致。我们希望开展更详细、正式的评估，将我们的推理引擎与 Prolog 进行比较。此外，我们尚未讨论如何将合一算法与现有实现进行基准测试，因为这需要经过审慎而详细的评估，因此我们将其留待未来工作。

未来，我们希望让领域特定语言更易于使用，并使其记法更接近 Prolog 或其他逻辑编程语言，同时不牺牲其嵌入宿主语言的特性。Prolog 谓词通常由多条规则定义，规则头可以与参数合一。相比之下，在我们的宿主语言中，无法使用模式匹配来执行合一，因此我们需要在定义中使用 (===)，将输入拆分为各个组成部分。我们将利用编译器扩展来模拟 Prolog 子句头的合一，作为未来工作。

在我们的方案中，用户必须为想要使用的类型定义对应的可扩展类型，将值封装为项，并将基项转换回原始的非可扩展类型的值。尽管我们的表示法在一定程度上简化了这一过程，我们仍计划研究如何利用模板元编程，自动生成与给定代数数据类型相关联的可扩展类型。

Prolog 允许使用下划线代替项，表示相应位置的值必须合一，但不捕获该值；这相当于在使用下划线的位置引入自由变量。我们的实现支持在谓词定义的左侧使用通配符模式（即下划线）。未来，我们计划研究如何在谓词定义的右侧也支持通配符模式。

最后，我们计划评估如何消除在我们的语言中手动添加 scope 以限制剪枝影响的必要性。

参考文献

[1] Barbies。https://hackage.haskell.org/package/barbies。[2] barbies-th。https://hackage.haskell.org/package/barbies-th。[3] Jo˜ao Barbosa、M´ario Florido 与 V´ıtor Santos Costa（2022）：逻辑编程的数据类型推断。载于 Emanuele De Angelis 与 Wim Vanhoof 编：基于逻辑的程序综合与变换，Springer International Publishing，施普林格，页 16–37，doi:10.1007/978-3-030-98869-2 2。[4] Marco Bellia 与 Giorgio Levi（1986）：逻辑与函数式语言之间的关系：综述。《逻辑编程期刊》3(3)，页 217–236，doi:10.1016/0743-1066(86)90014-2。[5] Koen Claessen 与 Peter Ljungl¨of（2001）：Haskell 中的带类型逻辑变量。载于：2000 年 ACM Haskell 研讨会（PLI 2000 卫星会议）。[6] Doug DeGroot 与 Gary Lindstrom 编（1986）：逻辑编程：函数、关系与方程。Prentice-Hall。[7] Conal Elliott 与 Frank Pfenning（1991）：高阶逻辑编程语言的半函数式实现。《高级语言实现专题》，页 289–325。[8] Elio Giovannetti、Giorgio Levi、Corrado Moiso 与 Catuscia Palamidessi（1991）：Kernel-LEAF：一种逻辑与函数式语言。《计算机与系统科学期刊》42(2)，页 139–185，doi:10.1016/0022-0000(91)90009-T。[9]M. Hanus（2013）：函数逻辑编程：从理论到 Curry。载于：《程序设计逻辑——纪念 Harald Ganzinger 的论文集》，Springer LNCS 7797，页 123–168，doi:10.1007/978-3-642-37651-1 6。[10] M. Hanus、H. Kuchen 与 J.J. Moreno-Navarro（1995）：Curry：真正的函数逻辑语言。载于：ILPS'95 研讨会论文《逻辑编程的未来愿景》，页 95–107。[11] Michael Hanus（1991）：ALF 系统：一种函数逻辑语言的高效实现。载于 Harold Boley 与 Michael M. Richter 编：《声明式知识处理》，Springer Berlin Heidelberg，柏林，海德堡，页 414–416，doi:10.1007/BFb0013551。[12] M. V. Hermenegildo、F. Bueno、M. Carro、P. L´opez-Garc´ıa、E. Mera、J. F. Morales 与 G. Puebla（2012）：Ciao 及其设计理念概述。《逻辑编程理论与实践》12(1–2)，页 219–252，doi:10.1017/S1471068411000457。[13] Higgledy。https://hackage.haskell.org/package/higgledy。[14] P. Hill 与 J.W. Lloyd（1994）：G¨odel 编程语言。逻辑编程，MIT Press。[15] Dmitry Kosarev 与 Dmitry Boulytchev（2016）：在 OCaml 中对关系式语言进行带类型嵌入。载于：ML 家族研讨会 / OCaml 用户与开发者研讨会论文集，ML/OCAML 2016，日本奈良，2016 年 9 月 22–23 日，页 1–22。[16] Nikolai Kudasov 与 Artem Starikov（2024）：typedKanren：Haskell 中具有穷尽匹配的静态类型关系式编程。arXiv 预印本 arXiv:2408.03170，doi:10.48550/arXiv.2408.03170。发表于 miniKanren 2024。[17] John W Lloyd（1999）：在集成式函数逻辑语言中进行编程。《函数逻辑编程期刊》3(1-49)，页 68–69。[18] Jos´e Pedro Magalh˜aes、Atze Dijkstra、Johan Jeuring 与 Andres L¨oh（2010）：Haskell 的通用派生机制。载于：第三届 ACM Haskell 研讨会论文集，Haskell '10，美国纽约州纽约，页 37–48，doi:10.1145/1863523.1863529。[19] Juan Jos´e Moreno-Navarro 与 Mario Rodr´ıguez-Artalejo（1988）：BABEL：一种基于构造子规范与窄化的函数逻辑编程语言。载于：代数与逻辑编程国际会议，doi:10.1007/3-540-50667-5 74。[20] Alan Mycroft 与 Richard A. O'Keefe（1984）：Prolog 的多态类型系统。《人工智能》23(3)，页 295–307，doi:10.1016/0004-3702(84)90017-1。

[21] Gopalan Nadathur & Dale A. Miller（1986）：高阶逻辑编程。载于 Ehud Shapiro 编：第三届国际逻辑编程会议，Springer Berlin Heidelberg，柏林、海德堡，第 448–462 页，doi:10.1007/3-540-16492-8 94。[22] Lee Naish（1996）：*Prolog* 中的高阶逻辑编程。载于：多范式逻辑编程研讨会论文集，JICSLP，96，第 1–23 页。[23] Shayan Najd & Simon Peyton Jones（2017）：生长的树。《通用计算机科学期刊》21(1)，第 42–62 页。[24] Ivan Perez（2023）：变化的类型：可扩展类型设计模式。载于：第一届 ACM SIGPLAN 函数式软件架构国际研讨会论文集，FUNARCH 2023，纽约州纽约，美国，第 49–62 页，doi:10.1145/3609025.3609475。[25] Tom Schrijvers、V´ ıtor Santos Costa、Jan Wielemaker & Bart Demoen（2008）：迈向类型化 *Prolog*。载于：第 24 届国际逻辑编程会议论文集，ICLP ’08，Springer- Verlag，柏林、海德堡，第 693–697 页，doi:10.1007/978-3-540-89982-2 59。[26] Tom Schrijvers、Peter Stuckey & Philip Wadler（2009）：单子约束编程。《函数式编程期刊》19(6)，第 663–697 页，doi:10.1017/S0956796809990086。[27] Mehul Chandrakant Solanki（2012）：嵌入式编程语言：在 *Haskell* 中嵌入 *Prolog*。硕士论文，北不列颠哥伦比亚大学。[28] Zoltan Somogyi、Fergus Henderson & Thomas Conway（1996）：*Mercury* 的执行算法：一种高效的纯声明式逻辑编程语言。《逻辑编程期刊》29(1)，第 17–64 页，doi:10.1016/S0743-1066(96)00068-4。[29] J Michael Spivey & Silvija Seres（1999）：在 *Haskell* 中嵌入 *Prolog*。载于：Haskell 论文集，99，第 1999–28 页。