arXiv:2601.13224v1 [cs.PL] 19 Jan 2026

函数逻辑程序变换

Michael Hanus[1] 和 Steven Libby[2]

[1] 基尔大学信息学研究所，基尔，德国
mh@informatik.uni-kiel.de[2] 波特兰大学，波特兰，俄勒冈州，美国 libbys@up.edu

摘要。 许多用于处理程序的工具，如编译器、分析器或验证器，都会对其中间程序表示（如抽象语法树）执行变换。实现此类程序变换并非易事，因为需要遍历完整的语法树，并对树中的节点执行各种变换。本文展示了如何利用函数逻辑编程的特性，以紧凑且易于理解的方式实现程序变换。为此，我们提出将程序变换编写为部分定义的非确定性操作。由于与确定性定义的操作相比，非确定性的实现通常会带来一定的额外开销，因此我们将该方法与一种确定性变换方法进行了比较。我们针对函数逻辑语言 Curry 及其中间表示 FlatCurry 对这些方案进行了评估；FlatCurry 被各种分析、验证工具和编译器所使用。

1 引言

程序变换是计算机科学中一个由来已久的理念，最早可追溯到 McCarthy [22]。事实证明，它是程序分析、验证和编译中的一项重要工具。尤其是在函数式和逻辑式语言领域，人们利用程序变换来完成复杂任务的例子屡见不鲜。Peyton Jones 展示了如何将 Haskell 编译器构建为一系列程序变换 [27]，以及如何将优化实现为更多程序变换。 [25]。Appel 列举了若干通过程序变换构建 ML 编译器的示例 [8]，包括将抽象语法树变换为 CPS 形式，以及进行若干优化。Flanagan 等人最初将 A-范式表述为一组重写规则 [10],，而 Peyton Jones 等人则展示了通用重写如何助力轻松实现优化 [26]。许多优化都以简单的程序变换形式呈现。例如 Gill 的快捷消除中间结构 [11],、内联和 β 归约 [24],，以及部分求值等更复杂的优化 [19]。

Curry[3] 等函数逻辑语言也能从程序变换中受益。RICE 编译器由一系列程序变换构成，

[3] https://www.curry-lang.org/

包括这些变换 [21]。Hanus 还展示了如何利用程序变换来简化对 Curry 程序 [14]的验证。Peem¨oller 的部分求值器也使用了若干程序变换 [16]。

尽管变换理论通常被描述为处理复杂结构的一种简单而优雅的方法，但现实往往复杂得多。例如，A-正规形式转换 [10] 被表示为一组带有求值上下文的三条重写规则。然而，实际实现却是半页 Scheme 代码，使用一个颇为复杂的函数来构造续延，从而执行重写。实现只有半页之短固然令人惊叹，但它却失去了原始重写规则的可读性。这是程序分析和编译器中的一个反复出现的主题。GHC 编译器中的程序变换往往是冗长而晦涩的函数，与其实现的变换几乎毫无相似之处。

本文展示了如何利用函数逻辑编程的特性，以模块化、可组合的方式表示变换。后续各节将回顾函数逻辑编程和 Curry，介绍其名为 FlatCurry 的中间表示，阐述我们定义变换的方法，考察其结果，并作总结。

2 使用 Curry 进行函数逻辑编程

声明式语言 Curry [18] 兼具函数式编程（按需求值、强类型、高阶函数）的特性，以及逻辑编程（处理部分信息、合一、约束）的特性，综述请参见 [5,13] 。Curry 的语法与 Haskell [23]接近。与 Haskell 相比，Curry 会以（不作选择的）非确定性方式应用左侧相互重叠的规则（而 Haskell 总是选择第一个匹配的规则），并允许在定义规则的条件和右侧中使用自由（逻辑）变量。其操作语义基于按需求值，这种求值方式对大量程序而言是最优的 [3]。

与 Haskell 类似，Curry 采用强类型，因此程序由数据类型定义（用于定义这些类型的构造器）以及针对这些类型的函数或操作组成。以下简单示例定义了列表上的连接操作“++”，以及adjDup 操作，该操作会返回列表中一个相邻的重复数字：[4]

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

adjDup :: [Int] → Int
adjDup xs | xs =:= _ ++ [x,x] ++ _
          = x  where x free
```

adjDup 利用合一运算符“=:=”，当两个表达式求值并合一时，该运算符会实例化自由变量。adjDup也称为非确定性操作，因为对于给定的

[4] 为了检查一些意外错误，Curry 要求显式声明自由变量，例如 adjDup 规则中的 x。匿名变量则无需声明，它们用下划线表示。

参数，它可能会返回多个结果，例如，`adjDup [1,2,2,1,3,3,4]` 会返回2 和3。非确定性操作可解释为从值到值集合的映射 [12]，是当代函数逻辑语言的一项重要特性。尤其重要的一种操作是选择运算符“?”，它以非确定性的方式返回两个参数之一。正如我们将看到的，这类操作有助于以简洁的方式描述程序变换。

如果实际参数的形式符合条件等式右侧所指定的形式，则操作 `adjDup` 可约化。对于这类情况，Curry 支持一种更紧凑的记法：

```
adjDup' :: [Int] → Int
adjDup' (_ ++ [x,x] ++ _) = x
```

由于 `adjDup'` 中使用的模式包含定义的函数“++”，因此称其为函数模式。函数模式表示该函数模式经过求值后可得到的所有标准模式。函数模式匹配可通过特定的合一过程 [4] 高效实现。函数式模- 式可以表达任意深度的模式匹配，因此适用于指定程序变换。

由于 Curry 具有逻辑编程特性，即能够处理非确定性计算和失败计算，因此可以支持函数模式。为控制非确定性计算和失败，Curry 支持封装搜索运算符，它们可返回表达式的所有值或部分值。本文仅使用搜索运算符

```
oneValue :: a → Maybe a
```

如果参数没有值，则返回 `Nothing`；否则返回某个值 `Just`。[5]

3 FlatCurry：Curry 程序的中间表示

Curry 还包含许多上一节未介绍的特性，例如类型类、单子式输入/输出、模块等。为了避免在 Curry 的语言处理工具（编译器、分析器等）中处理所有这些特性，此类工具通常会使用一种中间语言，其中源语言的语法糖已被消除，并且模式匹配策略是显式的。这种中间语言称为 FlatCurry，也是本文所述示例程序变换的基础，因此我们将对其进行更详细的介绍。除了编译器之外，FlatCurry 还被用于描述 Curry 程序的操作语义 [1], 、实现用于分析 Curry 程序的模块化框架 [17], ，以及验证 Curry 程序的非失败属性 [14]。

[5] 具体选择哪个值并未规定，因此该操作被视为不安全（Curry 中还有其他更复杂的声明式封装运算符）。由于我们对程序变换的合流性并不感兴趣，这种略欠声明式的行为是可以接受的。

$P ::= D_1 \ldots D_m$ (程序)
$D ::= f(x_1, \ldots, x_n) = e$ (函数定义)
$e ::= x$ (变量)
| $f(e_1, \ldots, e_n)$ (函数/构造器应用)
| $e_1 \; or \; e_2$ (析取)
| $let \; x_1, \ldots, x_n \; free \; in \; e$ (自由变量)
| $let \; x = e \; in \; e'$ (let 绑定)
| $case \; e \; of \; \{p_1 \to e_1; \; 。 \; 。 \; 。 \; ; \; p_n \to e_n\}$ （case 表达式）
$p ::= c \; (x_1, \; 。 \; 。 \; 。 \; , \; x_n)$ （pattern）

Fi图 1. 中间语言 FlatC 的语法 urry

图 1 概述了 FlatCurry 的抽象语法。一个 FlatCurry 程序由一系列函数定义组成（此处省略数据类型定义），其中每个函数都由一条规则定义。源程序中的模式会被编译为 case 表达式，而重叠规则则通过显式析取连接起来。每个 case 表达式中的模式都必须互不重叠。

任何 Curry 程序都可以转换为这种格式 [2,7]。具体而言，Curry 编译器的前端会将源程序转换为 FlatCurry 程序，因此 FlatCurry 是许多 Curry 编译器的中间语言。所以，它是简化或优化 Curry 程序的程序变换的合理目标。

例如，考虑下面这个用于在列表中的任意位置插入元素的操作:

```
insert :: a → [a] → [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

这个定义包含重叠规则，可以转换为以下 FlatCurry 定义（其中使用标准列表表示法）:

```
insert(x,xs) =    x : xs
               or case xs of { y:ys → y : insert(x,ys) }
```

为了在 Curry 程序中处理 FlatCurry 程序，有一个 Curry 软件包 `flatcurry`[6] ，它定义了用于表示 FlatCurry 程序的数据类型，以及用于读取 Curry 程序并返回等价 FlatCurry 程序（以这些数据类型的项表示）的操作。为了理解后文讨论的变换示例，我们展示用于表示 FlatCurry 表达式的数据类型。为区分不同种类的应用，使用以下类型:

data 组合类型 = 函数调用| ConsCall函
数部分调用 Int构造器部分调用 Int

[6] https://cpm.curry-lang.org/pkgs/flatcurry.html

因此，函数调用 和ConsCall分别用于函数和构造器的应用，而函数部分调用 和构造器部分调用 用于部分应用，其中整数参数表示缺少的参数数量。然后，FlatCurry 表达式由以下类型表示（为便于阅读，与实际的flatcurry 软件包相比，此定义做了略微简化）：

```
data 表达式 =Var Int|Comb 组合类型 String [
表达式]Or 表达式 表达式| Free [Int]表达式|
Let (Int, 表达式) 表达式| Case 表达式 [分支表达
式]

data BranchExpr =Branch Pattern Expr
数据 Pattern =Pattern 字符串 [整数]
```

请注意，变量由唯一的整数表示。例如，考虑布尔否定操作 not。它的 FlatCurry 定义为

```
not(x) = case x of { False → True ; True → False }
```

它的右侧表达式由以下数据项表示（其中变量 x 的索引为 0）：

```
Case (Var 0)
     [Branch (Pattern "False" []) (Comb ConsCall "True" []),
      Branch (Pattern "True"  []) (Comb ConsCall "False" [])]
```

Prelude 操作“$”定义为

```
($) :: (a → b) → a → b
f $ x = f x
```

是一个常用于不使用括号来书写应用的中缀应用运算符。Curry 表达式“not $ not True”表示为以下数据项：

```
Comb 函数调用 "$" [Comb (函数部分调用 1) "not" [],Comb 函数调用 "not" [Comb
ConsCall "True" []]]
```

在下一节中，我们将讨论一种程序变换，通过移除对“$'”的调用来简化这个表达式。

4 FlatCurry 程序的变换

现在我们已经有了用于变换 Curry 程序的结构，可以定义变换它们的方法了。

4.1 函数逻辑变换

为了简化程序变换的开发，我们希望以一种简单、可组合的方式表示单个变换。借助这种表示方式，我们可以将遍历 FlatCurry 表达式的逻辑与

orFloat (Or (Let vs e1) e2) = Let vs (Or e1 e2) orFloat (Or e1 (Let vs e2)) = Let vs (Or e1 e2)

图 2。 变换：将 let 表达式从选择表达式中提取出来

unDollar (dollar f args miss x)| 缺失 == 1 = Comb 函数调用 f (args++[x])| 缺失 > 1 = Comb (函数部分调用 (miss-1)) f (args++[x]) dollar f args miss x = Comb 函数调用 "$"[Comb (函数部分调用 miss) f args, x]

图 3. 变换：移除表达式中对“$”的调用

变换本身。最自然的做法是将变换表示为作用于 FlatCurry 表达式的函数。我们允许变换具有非确定性和部分性。例如，我们希望实现一种将局部 let 绑定移出选择的变换。这可以通过以下变换规则来表达：

$$(let\ x = e\ in\ e_1)\ ?\ e_2 \ \Rightarrow\ let\ x = e\ in\ (e_1\ ?\ e_2)$$
$$e_1\ ?\ (let\ x = e\ in\ e_2) \ \Rightarrow\ let\ x = e\ in\ (e_1\ ?\ e_2)$$

在假设局部变量始终具有不同标识符（Curry 前端会确保这一点）的情况下，此变换是正确的。Curry 中的直接实现如图 2 所示。请注意，`orFloat` 在参数不匹配时会失败，并且在表达式

```
(let x = 1 in x) ? (let y = 1 in y)
```

这里的非确定性并无影响，因为最终两条规则都会应用，且应用顺序无关紧要。这使我们能够在控制流信息无关紧要时，避免对不必要的控制流信息进行编码。

非确定性还使我们能够在变换规则中使用函数式模式。图 3 中的去美元符号示例实现了以下变换

$$f\ \$\ x\ \Rightarrow\ f\ x$$

操作`dollar` 是 FlatCurry 表达式的一种简写形式，该表达式表示对“$”的应用，其中第一个参数是一个部分应用函数（而不是变量或其他表达式）。在`unDollar`中将`dollar` 用作模式，展示了如何利用函数式模式提高变换的可读性。虽然我们本可以在`unDollar`的定义中完整指定对“$”函数的应用模式，但函数模式使变换更加清晰。

我们还允许变换是偏函数。这意味着变换在某些情况下可能不适用。这并无大碍。如果变换不适用，我们就忽略它并继续尝试下一个。这一点尤其

```
caseCancel (Case (Comb ConsCall c []) (withBranch c e)) =e
withBranch c e =(_ ++ [Branch (Pattern c []) e] ++ _)
```

图 4. 针对无参数构造器的案例取消变换。

当我们需要搜索特定的子表达式时，这种方法很有用，例如图 4 中实现以下变换的案例取消示例：

$$case\ C\ of\ \{\ldots; C \to e; \ldots\}\ \Rightarrow\ e$$

我们使用函数模式来查找包含 Case 的被匹配项值的特定分支。若要求此函数必须是全函数，就需要手动遍历各个 Case，直到找到目标 Case；如果找不到，则发出失败信号。

部分变换的一个更微妙的用法出现在图 3 的去美元符号示例中。如果函数 `f` 是一个缺少 `0` 个参数的部分应用，那么我们不应将它转换为一个期望 `-1` 个参数的部分调用。省略这种情况，就巧妙地避开了这个问题。

到目前为止，我们的变换类型为 `Expr →Expr`。对于简单的变换，这已经足够，但为这种类型增添额外信息，有助于支持更复杂的变换。因此，我们再添加一个参数，其中包含下一个可用新变量的索引，以及当前正在变换的子表达式的路径（其类型为 `Path`，这是整数列表的类型同义词）。这样一来，我们便可以按需生成新变量，并确定一些有用的信息，例如当前是否位于右侧表达式的根节点。

除了变换后的表达式之外，变换还会返回变换后表达式中所使用的新变量数量。这样，我们就能高效地跟踪下一个新变量。因此，一般表达式变换的完整类型为

```
类型 ExprTransformation =(Int, 路径) → 表达式 → (表达式,Int)
```

我们可以使用以下函数，将类型为 `Expr → Expr` 的简单变换提升为 `ExprTransformation`

```
makeT :: (表达式 → 表达式) → ExprTransformation makeT f
  = \_ e → (fe, 0)
```

最后，我们的变换库提供了一个操作（其实现将在第 4.3 节中讨论）

```
transformExpr :: (() → ExprTransformation) → 表达式 → 表达式
```

它会反复应用变换，直到无法再应用任何变换为止。为了避免过早确定非确定性变换的具体选择，传入的不是一个常量，而是一个函数，该函数接受一个类型为`()`的占位单位参数。

诸如 `unDollar` 或 `orFloat` 这样的单独变换可能很有用，但这个系统真正的强大之处在于可组合性。我们支持两种类型的

组合。串行组合是依次应用一个变换，而并行组合则是同时应用两个变换。在我们的方法中，按顺序应用变换就是函数组合：例如，先应用变换 `t1`，再应用变换`t2`，可以在 Curry 中用函数组合运算符“.”表示：

```
transformExpr (\() → t2) . transformExpr (\() → t1)
```

`t1` 和`t2`的并行组合，即在可能的情况下同时应用`t1`和`t2` 这两个变换，可以表示为

```
transformExpr (\() → t1 ? t2)
```

这种组合具有很强的表现力，并让我们能够控制如何执行变换。例如，我们可能希望先将选择中的所有 let 表达式提取出来，然后移除所有“`$`”应用并消除简单 Case。这可以表示为

```
transformExpr (\() → makeT unDollar ? makeT caseCancel)。
transformExpr (\() → makeT orFloat)
```

4.2 纯函数变换

关注点分离和可组合性是我们系统中的强大工具。然而，我们可能希望在程序变换中避免所有非确定性。尽管这会牺牲便利性，我们仍然可以使用全定义的确定性函数来支持组合。

确定性变换必须完全定义，以便其类型为表达式 → Maybe 表达式。成功的变换返回`Just e`，而失败的变换返回`Nothing`。

下面的`caseCancelDET` 示例展示了定义确定性变换所需进行的修改。函数式模式被替换为完备的 case 表达式。不过，由于我们需要在列表中查找分支，因此需要编写一个辅助函数来查找正确的分支。case 表达式确保模式匹配按顺序进行，因此不会出现重叠模式。

```
caseCancelDET e = 对 e 进行 case 匹配
                        Case (Comb ConsCall c []) bs → 查找 c bs
                        _                              →Nothing
 其中 find _ [] = Nothing
      find c(Branch (Pattern p vs) e : bs)|c == p &
      & null vs = Just e| 否则 = find n bs
```

若要对所有子表达式反复应用确定性变换，直到无法再进行变换为止，我们的库提供了一个操作

```
transformExprDet :: ExprTransformation → 表达式 → 表达式
```

请注意，`transformExpr` 中使用的单位参数并非必要，因为变换参数始终是确定的。

顺序组合与之前的版本保持不变，但我们需要一个新的运算符来实现并行组合。`<?>` 运算符会尝试执行一个变换；如果第一个变换失败，则回退到第二个变换。

```
t1 <?> t2 = \env e  →  case t1 env e of
                          Nothing  →  t2 env e
                          answer   →  answer
```

借助这个新的组合运算符，我们可以定义先前变换的确定性版本如下（此处省略 `unDollear` 和 `orFloat`的确定性版本的定义）：

transformExprDet（makeT unDollarDet `<?>` makeT caseCancelDet）。
transformExprDet（makeT orFloatDet）

4.3 变换策略

为了将目前展示的变换规则应用于表达式的任意子表达式（通常是函数定义的右侧），我们需要一个变换引擎。这正是 `transformExpr` 操作的作用。为了评估我们的框架，我们使用三种不同的变换应用策略，下面将对它们作简要介绍。

第一种是混沌策略 （*CS*）。该策略以非确定性的方式选择一个子表达式，尝试对其应用变换；如果可行，则将其替换为变换结果。这与深度选择模式 [6] 类似。将变换规则应用于选定的子表达式时，会使用 `oneValue` 操作来检查变换是否适用，并确保只执行一次替换。这是最简单的实现方式，但对于大型程序来说可能效率低下。[7]

第二种策略是确定性策略 (*DS*)。该策略以自底向上的方式遍历表达式，并尝试在每个子表达式上应用纯函数式变换。如果应用成功，就替换该子表达式并再次尝试；否则继续处理下一个子表达式。

最后一种策略是前两种策略的组合，我们称之为混合策略(`MS`)。该策略仍以自底向上的方式遍历语法树，但在每一步都应用非确定性变换规则，并使用 `oneValue` 强制使结果具有确定性。与混乱策略相比，这种方式有所改进，因为每次应用变换时都无需重新计算路径。

在建立了用于创建、组合和应用变换的框架之后，我们将注意力转向评估不同策略的有效性。我们评估变换的性能以及开发周期。

[7] 附录 A 展示了用 Curry 实现该策略的方法。

模块	大小	函数	转换	PAKCS CS	PAKCS MS	PAKCS DS	KiCS2 CS	KiCS2 MS	KiCS2 DS
Prelude	72485	1285	3	955	1494	321	795	306	148
Data.Char	2190	9	0	22	49	7	18	7	4
Data.Either	1693	11	0	3	3	1	1	1	1
Data.List	14841	87	0	54	65	20	24	18	9
Data.Maybe	1809	9	0	4	6	1	3	2	1
Numeric	3494	7	4	9	18	4	17	17	2
System.Console.GetOpt	17328	47	0	66	119	21	38	23	11
System.IO	6223	51	0	15	19	7	5	5	3

表1。使用第4.1节中的变换来变换标准库

5 基准测试

我们介绍了在声明式语言中实现变换的不同方法。函数逻辑变换利用部分定义的非确定性操作，以易于理解的方式指定变换，并避免为不匹配的情况编写多余代码。其实际实现需要逻辑编程特性，例如通过封装搜索来控制失败和非确定性。相比之下，纯函数式变换要求以确定性的方式编码所有情况，但其实现不需要逻辑编程特性。由于使用逻辑特性会在运行时带来一些额外开销，因此了解支持简洁易懂的变换规约需要付出怎样的代价很有意义。因此，本节将评估上文讨论的各种方法。

我们在 Curry 中实现了混沌、混合和确定性变换策略，因此可以按照上文所示编写变换规则。具体而言，以函数逻辑风格编写的变换既可用于混沌策略，也可用于混合策略；而确定性策略则要求采用更复杂的确定性变换风格，即以全定义操作的形式编写。

在第一组基准测试中，我们将第4.1节所示的全部三条变换规则应用于 Curry 发行版中包含的各种标准库。对于每个模块，表1列出了 Curry 源文件的大小、相应 FlatCurry 程序中定义的函数数量（请注意，变换应用于每个函数的右侧）、在模块中执行的变换次数，以及采用第4.3节所述各种策略进行变换所需的时间（毫秒）。[8] 此外，我们还使用 Curry 实现 PAKCS [15]，（它将程序编译为 Prolog）和 KiCS2 [9]，（它将程序编译为 Haskell）运行了基准测试。由于标准库经过多年演进，已避免包含多余的代码，因此能够应用的变换很少。

[8] 我们在一台运行 Ubuntu 22.04 的 Linux 机器上测量了变换时间，该机器配备八核 Intel Core i7-1165G7（2.80GHz）处理器。

模块	大小	函数	转换	PAKCS CS	PAKCS MS	PAKCS DS	KiCS2 CS	KiCS2 MS	KiCS2 DS
Prelude	72485	1285	5779	39339	77830	3042	15660	5272	2740
Data.Char	2190	9	163	1358	3189	90	365	86	76
Data.Either	1693	11	12	6	10	2	2	3	3
Data.List	14841	87	237	481	870	70	117	61	56
Data.Maybe	1809	9	29	103	175	13	39	11	11
Numeric	3494	7	47	54	93	9	10	8	6
System.Console.GetOpt	17328	47	418	1483	2566	139	331	126	108
System.IO	6223	51	89	50	89	15	9	11	9

表 2. 将标准库变换为 A-范式

因此，这项基准测试主要评估了在不同策略下，尝试将变换应用于程序中所有位置所需的时间。

正如预期，确定性变换策略速度最快。不过，混合策略的耗时也没有慢太多。因此，没有必要将函数逻辑变换改写成更复杂的纯确定性形式，尤其是在使用 KiCS2 这类高效的 Curry 实现时。另一个有趣的现象是，使用 PAKCS 执行时，混沌策略比混合策略更快。原因尚不明确，但值得注意的是，Prolog 直接支持非确定性，而 KiCS2 则通过搜索树结构显式实现非确定性。

为了了解在许多变换都可实际应用时的变换行为，我们使用将 FlatCurry 程序变换为 A-范式（ANF）的操作测试了我们的实现 [10]。ANF 要求操作或构造器只能应用于变量。ANF 广泛应用于许多编译器中，也用于函数 [20] 和函数逻辑 [1] 程序的操作语义规范。ANF 变换会将非变量实参替换为新变量，并通过 let 绑定引入这些变量。例如，第 3 节中定义的操作 `insert` 的 A-范式为

```
insert(x,xs) =     x : xs
                 or case xs of { y:ys  → let z = insert(x,ys)
                                         in y : z }
```

函数逻辑 ANF 变换基本上只包含一条规则：用一个新变量和相应的 let 绑定来 replace 非变量实参。另一方面，纯函数式 ANF 变换也必须实现相同的变换，但还必须考虑所有其他表达式，以避免失败。

对标准库集合应用 ANF 变换的结果如表 2 所示。运行时间表明，当有许多变换可用时，非确定性变换与确定性变换之间的差异可能会增大；基于 Prolog 的 PAKCS 编译器便是这种情况——

而对于效率更高的、基于 Haskell 的 KiCS2 编译器，差异则没有那么大，尤其是在使用混合策略而非混沌策略时。由于两种策略可以使用相同的变换规则，因此始终可以优先采用混合策略。考虑到函数逻辑变换更易于实现，在使用 KiCS2 这类高效的 Curry 系统时，绝对运行时间上的差异是可以接受的。

6 结论

本文介绍了以声明式方式实现程序变换的方法。我们的示例针对中间语言 FlatCurry，但如果其他语言的抽象语法树表示为数据项，也可以为这些语言编写类似的变换。尽管此类变换可以用任何编程语言实现，但我们展示了 Curry 的函数逻辑编程特性有助于编写简洁且易于理解的变换，从而使实现代码与此类变换的规格说明非常相似。将变换定义为部分定义的非确定性操作，可以避免编写用于控制变换的多余代码。

从定性角度看，我们的程序变换系统相比手动编写变换具有多项优势。其中最显著的优势之一是，变换完全使用 Curry 编写。无需学习新语言，也无需安装新工具。相反，每个变换都是一个 Curry 函数。这样，Curry 开发人员无需学习新系统即可编写复杂的变换。具有函数式语言经验的开发人员通常只需有限的说明就能理解一个变换。

另一个优势是，变换具有可扩展性。由于变换是 Curry 函数，我们可以根据需要使其变得多么复杂。变换并不局限于简单的重写规则，还可以涉及任意计算。变换可以扩展为处理额外的信息。例如，β 归约变换可以接收一个映射，其中包含有关先前见过的函数的信息。我们只需为该变换添加另一个参数即可。

第三个优势是，我们的系统可以轻松扩展，以报告应用了哪些变换。通过为每个变换提供一个名称，就可以轻松重构完整的变换推导。这是调试变换的重要工具。它使开发人员能够看到表达式如何从原始形式演变为最终结果；在使用多个变换时，这通常很难实现。

我们的系统已广泛用于 RICE 编译器中 [21] ，以实现复杂的变换，包括转换为 A-范式、变量内联和 β 归约。这些优化的开发通常很直接，我们也能够了解这些优化之间的相互作用，

以及它们如何彼此影响。它还使我们更容易修改这些优化，使其更加有效。

我们的系统提供了一种便捷、简洁的方法，可完全使用 Curry 指定程序变换。通过支持部分操作和非确定性操作，我们能够专注于变换本身。该系统具有可扩展性，并已证明在 RICE 编译器等大型程序中行之有效。

参考文献

1. E. Albert, M. Hanus, F. Huch, J. Oliver 和 G. Vidal。声明式多范式语言的操作语义。*Journal of Symbolic Computation*，40 1 795 829，2005 ( ): – 。 2. S. Antoy。基于构造子的条件窄化。载于 *Proc. of the 3rd Inter-national ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)* 199 206 2001 ，第 – 页。ACM 出版社，。 3. S. Antoy、R. Echahed 和 M. Hanus。必要窄化策略。*Journal of the ACM* 47 4 776 822, 2000 ，( ): – 。 4. S. Antoy 和 M. Hanus。使用函数模式进行声明式编程。载于 *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)* 6 22 3901, 2005 ，第 – 页。施普林格 LNCS 。 5. S. Antoy 和 M. Hanus。函数逻辑编程。*Communications of the ACM* 53 4 74 85, 2010 ，( ): – 。 6. S. Antoy 和 M. Hanus。新的函数逻辑设计模式。载于 *20th International Workshop on Functional and (constraint) Logic Programming (WFLP 2011)*，第 19–34 页，丹麦欧登塞，2011。施普林格 LNCS 6816。 7. S. Antoy、M. Hanus、A. Jost 和 S. Libby。ICurry。载于 *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)* 286 307 12057, 2020 ，第 – 页。施普林格 LNCS 。 8. A. Appel。 *Compiling with Continuations*。剑桥大学出版社，美国纽约州纽约市，2007。 9. B. Braßel、M. Hanus、B. Peem¨oller 和 F. Reck。KiCS2：一个从 Curry 到 Haskell 的新编译器。载于 *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)* 1 18 ，第 – 页。施普林格 LNCS 6816，2011。 10. C. Flanagan、A. Sabry、B. F. Duba 和 M. Felleisen。《使用延续进行编译的本质》。载于 *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* 93, 237 247, ，PLDI ' 第 – 页，1993。 11. A. Gill、J. Launchbury 和 S. L. Peyton Jones。《消除中间数据结构的捷径》。载于 *Proceedings of the Conference on Functional Programming Languages and Com-puter Architecture* 93, 223 232, 1993 ，FPCA ' 第 – 页，美国纽约州纽约市，。 12. J.C. Gonz´alez-Moreno、M.T. Hortal´a-Gonz´alez、F.J. L´opez-Fraguas 和 M. Rodr´ıguez-Artalejo。一种基于重写逻辑的声明式编程方法。*Journal of Logic Programming*，40:47–87，1999。 13. M. Hanus。函数逻辑编程：从理论到 Curry。载于*Logics - Essays in Memory of Harald Ganzinger* 123 168 *Programming* ，第 – 页。施普林格 LNCS 7797，2013。

14. M. Hanus。推断声明式程序的非失败条件。 *Science of Computer Programming* 250 103416 2026 , ( ), 。 15. M. Hanus、S. Antoy、B. Braßel、M. Engelke、K. H¨oppner、J. Koj、P. Niederau、R. Sadre、F. Steiner 和 F. Teegen。PAKCS：波特兰-亚琛-基尔 Curry 系统。网址：https://www.curry-lang.org/pakcs/，2025。 16. M. Hanus 和 B. Peem¨oller。Curry 部分求值器，载于 *Proc. of the 23rd Int. Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)* 55 71 ¨ 2014 ，第 – 页。哈勒-维滕贝格大学，。 17. M. Hanus 和 F. Skrlac。用于函数逻辑程序的模块化通用分析服务器系统。载于 *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)* , 181 188 第 – 页。ACM 出版社，2014。 18. M. Hanus（编）。Curry：一种集成函数式逻辑语言（版本 0.9.0）。网址：http://www.curry-lang.org，2016。 19. N. Jones、C. Gomard 和 P. Sestoft。*Partial evaluation and automatic program generation* 1993 。

Prentice-Hall, Inc.，美国，。 20. J. Launchbury。惰性求值的自然语义。载于 *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)* , 第 – 页。144 154 ACM 出版社，1993。 21. S. Libby。RICE：一个优化型 Curry 编译器，载于 *25th International Symposium on Practical Aspects of Declarative Languages (PADL 2023)* 3 19 ，第 – 页。施普林格 LNCS 13880，2023。 22. J McCarthy。计算数学理论的基础：初步报告。载于 *Papers Presented at the May 9-11 1961 Western Joint IRE-AIEE-ACM ,* , ，第 – 页，*Computer Conference* 225 238, 1961 美国纽约州纽约市，。计算机协会。 23. S. Peyton Jones（编）。*Haskell 98 Language and Libraries—The Revised Report*。剑桥大学出版社，2003。 24. S. Peyton Jones 和 S. Marlow。格拉斯哥 Haskell 编译器内联器的秘诀。*J. Funct. Program.*，12(5):393–434，2002 年 7 月。 25. S. Peyton Jones 和 A. Santos。Haskell 的基于变换的优化器。*Science of Computer Programming*，32(1)，1997 年 10 月。 26. S. Peyton Jones、A. Tolmach 和 T. Hoare。按规则行事：在 GHC 中将重写作为一种实用优化技术。 *Haskell 2001*，2001 年 4 月。 27. S.L. Peyton Jones。通过程序变换编译 Haskell：一线实践报告。载于 *Proc. ESOP'96*，第 18–44 页。施普林格 LNCS 1058，1996。

混沌策略的实现

下面我们介绍 Curry 中混沌变换策略的实现。首先，定义非确定性操作 `subExpOf` ，对于给定的 FlatCurry 表达式，它会返回一个子表达式及其路径（由整数列表表示，表示从零开始计数的参数位置）。

```
subExpOf :: Expr → (Path,Expr) subExpOf e = ([],e) -- 子表达式就是整个表达式 subExpOf (Comb _ _ args) =uncurry extendPath $anyOf (zip [0..] (map subExpOf args)) subExpOf (Let (_,e)_) = extendPath 0 (subExpOf e) subExpOf (Let _ e) = extendPath 1 (subExpOf e) subExpOf (Free _e) = extendPath 1 (subExpOf e) subExpOf (Or e1 e2) =extendPath 0 (subExpOf e1) ? extendPath 1 (subExpOf e2) subExpOf (Case ce _) = extendPath 0 (subExpOf ce) subExpOf (Case _ bs) = uncurry extendPath $ anyOf (zip [1..] (map (subExpOf . branchExp) bs)) where branchExp (Branch _ be) = be
```

辅助操作 `extendPath` 将路径组件延长一个位置:

```
extendPath :: Int → (Path,Expr) → (Path,Expr)
extendPath pos (p,e) = (pos:p, e)
```

现在，我们可以将混沌策略定义为操作`transformExpr` ，它通过迭代地将给定的变换规则应用于某个子表达式，来简化 FlatCurry 表达式。因此，变换步骤通过使用 `subExpOf` 选择一个可以应用规则的子表达式来实现（由局部操作`tryTransExpr`实现）。为了使整体策略具有确定性且不会失败，我们使用`oneValue`控制变换。操作`newVar` 返回表达式中未出现的下一个新变量；操作`replace` 则在给定位置替换表达式中的子表达式（其定义采用与`subExpOf`类似的分情况处理）。

```
transformExpr :: (() → ExprTransformation) → 表达式 → 表达式
transformExpr trans ne = runTrExpr trans (newVar e) e
runTrExpr :: (() → ExprTransformation) →Int → 表达式 → 表达式 runTrExpr trans nvar exp =
  case oneValue (tryTransExpr nvar 表达式) of
    Nothing → exp -- 没有适用的变换 Just (p, (e',nvs)) → runTrExpr 转换 (nvar+nvs) (replace exp pe')
```

其中

```
tryTransExpr v e = let (p,se) = subExpOf e
                   in (p, trans () (v,p) se)
```