HOTGP—高阶类型化遗传编程

Matheus Campos Fernandes
fernandes.matheus@ufabc.edu.br
ABC联邦大学（UFABC）圣安德烈，圣保罗，巴西

Fabrício Olivetti de França
folivetti@ufabc.edu.br ABC联邦大学（UFABC）圣安德烈，圣保罗，巴西

Emilio Francesquini
e.francesquini@ufabc.edu.br ABC联邦大学（UFABC）圣安德烈，圣保罗，巴西

arXiv:2304.03200v1 [cs.NE] 6 Apr 2023

摘要

程序合成是根据一组规格说明生成计算机程序的过程，这些规格说明可以是对问题的高层描述和/或一组输入输出示例。程序合成可以建模为一个搜索问题，其中搜索空间是在某种语法下有效的所有程序的集合。由于搜索空间十分庞大，暴力搜索通常不可行，而遗传编程等搜索启发式在没有任何指导的情况下也难以在其中导航。本文介绍了 HOTGP，这是一种新的遗传编程算法，用于合成纯函数式、类型化的函数式程序。HOTGP利用规格说明中丰富的数据类型以及内置语法所提供的知识来约束搜索空间，从而提升合成性能。该语法基于 Haskell 的标准基础库（生成的代码可以直接使用任意标准 Haskell编译器进行编译），并支持高阶函数、 λ-函数和参数多态。实验结果表明，与使用标准基准测试集的6种最先进算法相比，HOTGP具有竞争力，并且合成出正确程序的频率高于其他任何受评估算法。

CCS概念

• **计算方法** → **遗传编程**； • **软- 件及其工程** → 函数式语言。

关键词

归纳式程序合成, 遗传编程, 函数式编程

致谢

本研究部分由圣保罗州研究支持基金会（FAPESP）资助，项目编号为 #2021/12706-1、#2019/26702-8、#2021/06867-2，另获高等教育人才培养协调机构—巴西（CAPES）资助。

1 引言

程序合成（PS）是根据一组规格说明 [6]，以算法形式创建计算机程序的任务。程序规格说明是对程序目标的高层次描述 [19]。这种规格说明可以采用不同格式，从自然语言到更形式化的记法。为解决问题而指定程序的一种常见方法是提供一组输入输出示例，并特别关注边界情况，以缩小有歧义的解的范围。这称为归纳合成，或示例编程（PBE）[7]。在这种情况下，合成器的任务是找出一个程序，使其能够正确映射示例提供的每一对输入和输出。取决于

算法1： 示例规范的一种解法。

```
1 函数solution(x):
2     s ← 0
3     对于 i ← 0; i < length(x); i ← i + 1 执行
4         如果 x[i] < 100 则
5             s ← s + x[i]
6     返回s
```

所提供示例的完整性，规范可能存在歧义，从而导致许多行为不符合预期的替代程序，即使它们能够正确映射这些示例。

程序的生成可以建模为一种搜索，其中搜索空间包含预先指定的语法下所有有效程序的集合。目标是找到满足规范的程序。搜索空间的规模使得采用朴素方法从所有可能程序的枚举中选择最佳候选程序变得不切实际。因此，程序合成通常使用（元）启发式方法完成，通常是遗传编程（GP）[18]。

尽管这种方法在标准程序合成基准测试 [11–14, 17], 中取得了一些成功，但仍无法找到某些对人类而言轻而易举的任务的解决方案。原因包括：

(1) 遍历搜索空间颇具挑战性，因为有时程序代码的微小变化会对其输出产生重大影响；(2) 在缺乏有关程序的额外信息时，搜索依赖于示例的完整性；(3) 一些合成器可以创建有状态程序，而这些程序的行为可能会根据状态的变化方式而变得不可预测。

为了说明这些困难，我们以这样一个程序的规格说明为例：给定一个列表x，返回所有小于 100 的元素之和。算法1给出了一个有效的命令式风格程序。改变s的任何初始值（第 2 行）、i的任何初始值（第3 行）或常量 100（第 4 行），都可能使准确率从 100% 降至0%。另一方面，例如，如果输入示例列表不包含95到 100 之间的值，那么第 4行中的98 之类的常量就足以在训练集上达到 100% 的准确率，尽管当考虑所有可能的输入时，该代码会产生错误的结果。最后，循环体内任何会影响 i 或 s 的值的附加语句，也可能降低所达到的准确率。

缓解这些问题的一种可能方案是采用带类型的纯函数式范式。在这种范式中，程序是一个纯函数，并定义为纯函数的组合。

纯函数根据定义具有引用透明性 [29]这一基本属性。这意味着任何表达式（包括程序本身，它本身也是一个表达式）都可以安全地替换为其求值结果。这一属性使输入转换明确且可预测，并将搜索空间限制为不含副作用的函数。最后，类型化语言包含有关输入和输出类型的信息，有助于进一步约束搜索空间。

此外，如果我们还允许参数多态1，就可以有效地将搜索空间限制为仅包含格式良好的程序。例如，取类型签名 $\forall a.\ a \rightarrow a$。该签名只允许一种实现，即恒等函数。任何其他实现要么违反引用透明性，要么违反类型信息。尽管这个例子比较极端，但它展示了这些属性的结合如何限制搜索空间，从而减轻程序合成算法 [22] 的任务。

尽管上述特性已经足以支持表达能力丰富的语言，典型的函数式编程语言还会提供用于实现高阶函数的构造 [15]。在此语境下，高阶函数是指将函数作为参数之一接收的函数。常用的高阶函数包括映射、过滤器和折叠，它们概括了程序所需的许多常见模式。

本文提出了一种新的遗传编程算法，名为 HOTGP（高阶类型遗传编程），用于搜索纯粹的、带类型的函数式程序。其语法支持高阶函数、函数中的参数多态，以及参数化类型（如列表和元组）。我们使用 29 个基准问题对 HOTGP 进行了评估，并将结果与文献中的另外 6 种算法进行了比较。结果表明，与标准遗传编程算法相比，纯函数式方法能够显著提高找到正确程序的频率。

本文其余部分安排如下。第 2 节介绍相关工作。第 3 节描述 HOTGP。第 4 节概述实验评估，第 5 节总结全文。

2 相关工作

据我们所知，进化自动设计算法（ADATE） [23] 是最早以函数式代码为目标的程序合成示例。该研究旨在通过增量变换合成递归式 ML 语言程序。算法从一个由标记“?”表示的初始程序开始，该程序始终返回一个不知道[2] 值。随后，ADATE 系统地将表达式扩展为针对输入类型的模式匹配，为模式匹配的每个分支合成一个程序，并将一般情况替换为递归调用。

蒙大拿 [22] 提出强类型遗传编程（STGP）算法，这是遗传编程的一种改进形式，在程序合成过程中考虑每个函数和终端的类型。考虑类型的目的是进一步约束搜索空间通过只允许类型正确的类型化程序存在（即，所有函数都对具有适当数据类型的值进行操作的程序）。与标准遗传编程不同，在标准遗传编程中，给定的非终结符必须能够处理任何数据类型，STGP 施加了额外的约束以强制实现类型正确性。STGP 的另一项重要贡献是，非终结符的类型可以采用参数多态。

参数多态的主要优点是，无需编写多个相似的函数，而这些函数之间的区别仅在于类型。针对四个不同问题（涉及矩阵和列表操作）的实验表明，STGP 通常优于无类型遗传编程。

Haynes 等人 [8] 使用“追捕问题”比较了 STGP 与标准无类型遗传编程。该问题模拟了一场四只捕食者追捕一只猎物的游戏。目标是为捕食者创建一种算法，使其尽可能快地捕获猎物。猎物总是逃离最近的捕食者，而捕食者只能获得自身和猎物的信息，无法得知其他捕食者的情况。结果表明，生成优良的 STGP 程序比生成优良的遗传编程程序更快。此外，最佳 STGP 程序的捕获率高于最佳遗传编程程序。

PolyGP [32, 33] 通过支持高阶函数和 λ-函数扩展了 STGP。它与 STGP 的另一个区别是：使用类型合一算法而非查找表，在使用多态函数时确定具体类型。λ-函数采用与主程序合成过程相同的初始化过程，但可用终结符仅限于输入参数。由于这些λ-函数没有任何类型限制，因此可能无效；在这种情况下，必须将其丢弃并重新生成。整体算法是在用户定义的终结符和非终结符集合中，像 STGP 一样，简单地搜索由 λ-函数组成的组合。

Katayama [16] 提出了 MagicHaskell，这是一种广度优先搜索方法，利用 SKIBC [31] 组合子搜索类型正确的函数式程序。这通过缩小搜索空间简化了程序合成。MagicHaskell 还引入了德布鲁因λ项来查找等价表达式，并采用记忆化提升性能 [2]。此外，它还实现了融合规则，以进一步简化合成的程序。据报道，这种方法在处理较大的问题时效果不佳 [24]。

强结构遗传编程（SFGP） [1] 是对 STGP 的扩展。SFGP 不仅为终结符指定已知的数据类型，还为函数指定节点类型。节点类型用于标识给定节点是变量、表达式还是赋值。函数的每个子树也可以被要求具有特定的节点类型。作者认为，这些额外信息有助于构建类型正确的命令式程序（例如，赋值的第一个子节点必须是“变量”节点类型，并且与第二个子节点的数据类型匹配）。他们在 3 个数据集上进行了实验，使用的语法经过精简，主要处理整数；实验结果显示，与其他竞争方法相比，该方法在计算开销更低的情况下取得了较高的成功率。

Santos等人 [28] 讨论了程序合成方法的理想特性，方法是进一步约束搜索空间，类似于 STGP 所做的工作。他们提出使用精化类型来实现这一目标。据我们所知，该项目仍在进行中，尚无实验评估或比较结果。

[1]在某些编程语言中也称为泛型。[2]这等价于一个始终返回 null 的函数。

Pantridge 等人 [25] 提出了一种代码构建遗传编程（CBGP）[26] 的改编方法，旨在融入函数式编程的元素，例如高阶函数和 λ-函数。CBGP 使用与 PushGP 相同的表示方式，包含三种主要构造：APP，用于应用函数；ABS，用于定义零个或多个参数的函数；LET，用于在当前作用域中引入局部变量。它还运用类型理论中的概念来确保多态类型的正确性。对于一部分基准问题，CBGP 实现了更高的泛化率。然而，对于其他问题，泛化率接近于 0。作者指出，演化搜索会避免使用 λ-函数，而倾向于在映射等高阶函数中使用预定义函数。这些结果为类型安全对程序合成（PS）所带来的益处提供了一些间接证据，尤其是关于不同搜索算法执行过程中解的稳定性。

在这一研究方向上，Garrow 等人 [5] 使用语法引导系统比较了 Python 和 Haskell 程序的生成 [21]。与我们的工作类似，他们针对每一组类型使用不同的语法，而不是针对每个基准测试问题使用不同的语法。他们的方法支持高阶函数，但将函数参数限制为预先定义的常用函数。实验结果表明，在大多数选定的基准测试中，Haskell 版本的表现始终优于 Python。作者将通用 λ函数的实现留作未来工作，因为这会增加搜索空间的复杂度，而且必须将其作为不同于主程序的一种构造谨慎处理。

He 等人 [9] 研究了将已合成的程序重用为子程序并纳入非终结符集合的方法。其核心思想是，如果算法已经为较简单的任务合成了解决方案，就可以在增量过程中利用这些方案构建更复杂的解决方案。他们的结果表明，在 4 个选定的基准测试中，添加手工编写的模块能够带来显著收益。

Forstenlechner 等人 [3] 批评了遗传编程（GP）中的一种常见技术，即为每个问题提供不同的语法。他们认为，这会导致语法难以复用，因为这些语法是专门针对每个问题定制的。他们为 G3P 算法提出了一种通用语法，并在 Helmuth和 Spector [12] 引入的基准测试上进行了实验。由于所提出的语法难以处理涉及字符和字符串的基准问题，作者提出了一种改进且扩展的语法，从而形成了 G3P+ [4]。

3 高阶类型遗传编程

本节介绍高阶类型化遗传编程（HOTGP）。据我们所知，STGP 是最早提出并在遗传编程（GP）中采用类型的技术之一。因此，它自然影响了后续工作，例如 Castle 和 Johnson [1], Santos等人 [28], 以及HOTGP。现在，我们介绍类型化遗传编程所需的主要概念，这些概念由所有这些合成器共享。

在强类型遗传编程（SGTP）中，每个终结符都有对应的数据类型，而非终结符则有对应的输入类型和一个输出类型。为确保正确性，算法施加了两项约束：i）树的根节点必须与预期程序输出类型具有相同的输出类型；ii）每个非根节点的输出类型必须符合其父节点所要求的类型。

由于这些限制，进化搜索的主要组件必须进行调整。在初始化过程的每一步中，只有当一个节点与其父节点所期望的类型匹配时，才会考虑该节点。STGP 还会构建数据类型可能性表，以跟踪每种深度的树能够生成哪些数据类型；对于每种初始化方法（生长 和 完全），各有一张表。这些表是字典，其键表示深度，值表示对于 完全 或生长方法而言，该深度的树所能表示的类型。在深度 0 时，它们只包含终结符。在深度 i 时，它们包含所有接受 $i-1$ 处类型作为参数的函数的输出类型；对于生长方法，它还包含 $i-1$处的类型。

变异算子使用*grow* 方法，通过与初始化过程相同的算法生成一棵新子树，并用它替换一棵随机子树。正如预期，交叉也会考虑类型。第一个亲本的交叉点完全随机选取，而第二个亲本的交叉点则限定为类型与第一个亲本相同的候选程序。如果不存在这样的候选程序，则返回其中一个亲本。原始遗传编程算法在演化过程中还进行了一些额外改动，例如采用稳态替换 [30]和指数适应度归一化，根据亲本的排名选择用于繁殖的亲本。选中排名第 n_{th}的个体的概率为 $p(n) = P_{\text{标量}} \times p(n-1)$，其中 $0 < P_{\text{标量}} < 1$ 是一个超参数。

蒙大拿 [22] 还主张将运行时错误作为进化过程的一部分来处理，并对出现这些错误的个体进行惩罚。这与最初的遗传编程方法相反， [18], 该方法要求始终返回一个值。

STGP 还为不返回任何内容的函数（即过程）引入了空类型，以及可在计算过程中通过状态改变的局部变量。

与 STGP 相比，HOTGP引入了高阶函数和 λ-函数，取消了对非纯函数的支持，并使用从Haskell 的基础库中提取的通用语法。主要差异将在接下来的小节中详细介绍，具体如下：

- HOTGP 使用纯函数式程序范式（Haskell 编程语言的一个子集）构建程序，而 STGP 则以类型化 LISP 和 ADA 的组合为模型，允许使用非纯函数（见第 3.1 节）；
- 由于HOTGP 仅设计为支持纯函数，因此所有副作用（包括局部变量（可变状态）和 IO）在设计上均被禁止（参见第 3.1 节）；
- 随着我们转向另一种语言，我们对语法进行了适当的修改（见第 3.1 节）；
- 我们不为每个问题指定一组严格的专用终结符和非终结符，而是根据输入和输出类型指定泛型集合[3] （见第 3.1 节）；
- 此外，我们使用一组更通用的非终结符（均可在标准 Haskell 基础库中找到），而非通常需要由用户实现的非常具体的函数。这一特性与 Haskell 语言子集的使用相结合，使得所有合成的代码都能.....

[3]这是 STGP 和 PolyGP 均未探索过的一种设计选择。

可供 Haskell编译器立即使用，无需修改（参见第 3.1 节）；

- 最后，HOTGP支持高阶函数（接受 λ函数作为输入的函数），以处理综合程序中的高级构造（见第3.2节）；

3.1 函数式语法

尽管 HOTGP 和 STGP 都采用了强类型，但在 STGP 的两次实验评估中，作者针对每个基准问题分别设计并使用了有限的语法。例如，为了解决多维最小二乘回归问题，他们使用了一组最小函数集，其中包含矩阵和向量运算符，如 *matrix_transpose, matrix_inverse, mat_vec_mult, mat_mat_mult*。而本文使用了一组更通用的函数，适用于所有问题，且全部摘自标准 Haskell 基础库。

我们认为，在实际场景中，为每个问题仅提供所需的函数并不可取，因为这会向算法提供过多信息。在我们看来，这并不理想，因为事先可能无法轻易获得这类信息。对用户提出更合理的要求，是请他们为每个问题指定可接受的结果类型。这类信息通常只需要用户对问题具备与提供示例同等程度的直觉。

HOTGP 的基本类型目前包括 32 位整数、单精度浮点数、布尔值和 UTF8 字符。还支持以下参数化类型：对（2 元组）、链表以及 λ-函数。类型可以组合成更复杂的类型，例如，由 λ-函数构成的对的列表；也可以组合成更简单的类型，例如字符串（表示为字符列表）。

由于使用了 Haskell 语言的一个子集，HOTGP 不允许使用非纯函数。使用纯函数通常与可能减少错误数量相关 [27]。纯函数的一个重要性质是，由于没有副作用，它们更容易组合。因此，只要一个函数的返回类型与另一个函数的输入类型相同，就可以将二者组合成一个新的、更复杂的纯函数。

HOTGP 语法所允许的全部函数列于表14中。大多数函数都是针对特定类型的常见操作。由于我们采用强类型语言，因此还需要转换函数。其他常用函数包括对数字列表（整数和浮点数）求和与求积；Range，用于生成数字列表（等价于 Haskell 的 `[x,y..z]`）；Zip，用于将输入的两个列表中的元素配对；Take，用于返回列表的前 *n* 个元素；以及 Unlines，用于将字符串列表转换为单个字符串，并使用换行符将它们连接起来。特别是，对于要求程序将文本打印到标准输出的基准测试，Unlines 是必需的（在我们的例子中，由于使用的是纯语言，我们只需返回输出字符串）。

值得注意的是，我们在语法中加入了三个构造函数：ToPair、Cons 和 Singleton。这是经过刻意

[4]为节省篇幅并提高可读性，本文采用受ML 启发的约定来表示对与列表：(7, 42) 是一个包含 7与42 的对，而 `[42, 7, 6]` 是一个元素为42、7 和 6 的列表。同样，我们写作 `[a]` 来代替类型List a，并用 (a, b) 来代替类型 Pair a b。

表1：HOTGP 支持的函数。

函数类型	函数名称
整数 → 整数 → 整数	加法整数, 减法整数, 乘法整数, 整数除法，整数取模，最大整数，MinInt
布尔值 → 布尔值	`Not`
布尔值 → 布尔值 → 布尔值	`And, Or`
布尔值 → a → a → a	`If`
浮点数 → 浮点数	`Sqrt`
浮点数 → 浮点数 → 浮点数	浮点数加法，浮点数减法，浮点数乘法，浮点数除法
a → [a]	单例
a → [a] → [a]	`Cons`
[a] → a	`Head`
[a] → [a]	逆序
[[a]] → [a]	连接
a → b → (a,b)	转换为对
(a,b) → a	`Fst`
(a,b) → b	`Snd`
字符 → 字符 → 布尔值	字符相等
字符 → 布尔值	IsLetter, IsDigit
整数 → 浮点数	整数转浮点数
浮点数 → 整数	取整
整数 → 整数 → 布尔值	整数大于比较、整数小于比较、整数相等比较
[a] → 整数	`Len`
整数 → [a] → [a]	`Take`
整数 → 整数 → 整数 → [整数]	范围
[整数] → 整数	整数求和，整数乘积
[浮点数] →浮点数	浮点数求和，Product浮点数
[[字符]] → [字符]	Unlines
整数 → [字符]	整数转字符串
[a] → [b] → [(a,b)]	`Zip`
(a → b) → [a] → [b]	`Map`
(a → 布尔值) → [a] → [a]	过滤器

作出的选择，旨在简化语法。以二元组（数对）为例。我们的语法必须能够处理诸如 (1, 2) 或 (1 + 2, 3 * 4) 这样的结构（字面量数对和表达式数对）。不过，这些数对也可以轻松表示为 ToPair 的应用。第一个例子可以表示为 ToPair 1 2，也就是将 ToPair 函数应用于参数 1 和 2。按照同样的表示方式，第二个例子就变成了 ToPair (AddInt 1 2) (MultInt 3 4)。

换句话说，构造一对元素就是一次简单的函数应用，我们的语法无需对其进行特殊处理。这样做还有一个额外好处：它与已为常规节点定义的变异和交叉算子直接兼容。基于同样的道理，进化过程可以结合 Cons构造函数 和 单例 函数生成链表。例如，字面量 1、2、3 组成的列表可表示为 Cons 1 (Cons 2 (Singleton 3))；而表达式1 + 2、3 * 4、5 - 6 组成的列表则可表示为 Cons (AddInt 1 2) (Cons (MultInt 3 4) (Singleton (SubInt 5 6)))。与构造一对元素的情况一样，这样做还有一个额外好处，就是可以进行交叉

和变异仅作用于此类列表的头部或尾部。

HOTGP 还允许用户选择程序合成算法可以使用的类型，以进一步约束搜索空间。每当用户选择可用类型的一个子集时，系统便从表1中筛选出仅支持所选类型的函数，据此推断非终结符集合。例如，如果我们只选择 Int 和 Bool 类型，就可以使用 AddInt、And、GtInt 等函数，但不能使用 Head、Floor、ShowInt 等函数。

在该算法的未来实现中，我们计划使用 Haskell 的类型类来支持特设多态，这样便可以只使用 Add、Mult、Sub，而由上下文决定它们的类型，无需为每种类型分别设置专用符号。

STGP 与 HOTGP 的另一个重要区别是，HOTGP 不包含空类型，也没有用于创建局部变量的构造。因此，HOTGP 的语法无法表示非纯函数和可变状态。HOTGP 从构造上杜绝了副作用，因此只能表示纯程序。另一方面，与 STGP 类似，仍可能发生运行时错误（例如除以零）。发生此类错误时，适应度函数会为该解赋予无穷差的适应度值。

3.2 高阶函数与 λ-函数

HOTGP 的主要新颖性在于使用高阶函数。为此，添加对 λ-函数的支持至关重要。λ函数，或匿名函数，或简称 λ函数，是一种不与名称绑定的函数定义。作为一等值，它们可以用作高阶函数的参数。

引入 λ函数 时，在创建或修改程序时需要格外谨慎。求值时，HOTGP 的 λ函数 只能访问自身的输入，无法访问主程序的输入。换言之，它们不会捕获创建或执行它们时所处的环境。这意味着 λ终结符 基本上可以视为程序内部的“子程序”，并以此方式生成。我们采用与主程序相同的初始化流程，使用当前节点所需的函数类型，并采用 *grow* 方法。不过，还必须遵守另外两个约束。

约束1要求所有λ函数至少在其一个子树中使用其参数，这显著降低了生成仅返回常量值的λ函数的可能性。我们认为，对于高阶函数用途而言，λ函数必须使用其参数才能产生有趣的结果；否则，可以简化程序，去掉对该高阶函数的使用并返回一个常量[5]。

约束2的形式是可配置的 lambda 最大深度，用于防止我们的程序变得过大。然而，由于这些 lambda 可以嵌套，仅靠这个超参数不足以恰当地限制程序的大小。例如，取一个简单的 lambda：λx → map otherLambda x。根据允许的类型，otherLambda =λx → map yetAnotherLambda x 也可能是一个有效函数，以此类推，这可能导致 lambda 的大小任意增长。因此，

[5]这之所以成立，仅仅是因为 HOTGP 的语法排除了生成带有副作用的表达式。

为防止 λ函数嵌套过深，我们将嵌套函数限制为始终为 $\lambda x \rightarrow x$（恒等函数）。

为了强制执行这些约束，与 STGP 类似，HOTGP 使用类型可能性表来生成 λ函数。对于主程序树，由于参数类型和输出类型已预先确定，STGP 和 HOTGP 都只需创建两个表：一个用于 生长 方法，另一个用于 完整 方法。然而，HOTGP 需要生成涉及当前程序所允许的每一种可能类型的 λ函数。由于表具有递归性质，不同的参数类型可能会产生差异极大的类型可能性表，因此我们需要为每一种可能的参数类型分别保留一张表。作为约束1的推论，这些表也能保证不会变得过大，因为它们无需计算深度超过 λ函数最大深度的可能性。这些表与主表的另一个区别在于，只有当某个节点至少有一棵子树能够将参数叶节点作为后代时，该节点才会被视为有效，从而强制执行约束2。

在变异和交叉中，λ函数被视为普通终结符。它们总是被丢弃并重新生成（使用上述过程），或作为一个整体移动，本质上被视为单一单元。

3.3 代码精化

遗传编程算法面临的一个众所周知的难题是膨胀 [20],程序在没有为适应度函数带来任何益处的情况下，出现不必要且无法控制的增长。随着一些显然不影响程序输出的构建模块在交叉和变异的连续应用中存活下来，这种现象会自然发生。这些膨胀不仅会使生成的程序变得更长、更难以阅读，还可能影响其在测试集上的性能。例如，考虑将一个数字加倍的任务，以及候选程序 x0 * (min x0 900)。如果训练集不包含满足 $x_0 > 900$ 的输入案例，那么从算法的角度来看，这将是一个正确的解。

Helmuth 等人 [10] 通过实验证明，更简单的程序通常具有更强的泛化能力，而且也更容易理解和推理。例如，Pantridge 等人 [25], 会在搜索结束时应用一个精化步骤，反复尝试移除程序中的随机片段，并检查是否有所改进。

为减轻膨胀的影响，我们还会结合训练数据，对找到的最佳树应用一个细化过程。细化从应用化简规则开始，这些规则会移除代码中的冗余：

- 常量求值：如果某个树分支中不涉及任何参数终结符，就可以安全地将其求值为一个常量，例如`head [4*5, 1+2]` → `20`;
- 一般定律应用：简化器可以访问一个手工编写的简化过程表，这些过程已知为真（定律）（例如条件判断 True then a else b ≡ a，a > a ≡ False，长度 (单例 b) ≡ 1，等等）。

完成这一步后，HOTGP 会执行局部搜索过程，旨在移除树中那些对程序在训练集上的正确性没有贡献，甚至会降低正确性的部分。局部搜索会依次用每个节点的一个子节点替换该节点；如果修改后的版本有所改进或结果相同，则保留该版本。

算法2： 局部搜索过程

```
1  函数localSearch(tree):
2      条件判断 非 hasNext(tree.node) 则返回tree
3      bestTree ← tree
4      遍历 child ∈ tree.node.children 执行
5          条件判断 child.outputType =tree.node.outputType 然后
6              新树 ←替换(tree.node, 子节点)
7              条件判断 准确率(newTree) ≥ 准确率(bestTree)
8              && nNodes(newTree) < nNodes(bestTree)
9              then
10                 bestTree ←newTree
11     条件判断 bestTree ≠ tree 那么
12         返回 局部搜索(bestTree)
13     返回 局部搜索(nextPreOrder(树))
```

算法2描述了这一过程。它以树作为输入，其中包含正在检查位置的内部表示，可通过 *tree.node*访问。该过程首先调用 localSearch，将简化规则处理后得到的树作为输入，并将当前位置设为树根。接下来，算法扫描当前节点中输出类型与其父节点相同的子节点，并通过替换父节点来创建一棵新树（第6行）。保存最佳树后，若树未发生变化，过程便递归继续，将当前位置推进到前序遍历中的下一个节点；否则，继续使用当前位置。当没有更多位置需要检查时，过程结束（第2行）。

4 实验结果

在本节中，我们将 HOTGP与文献中提出的最先进的基于GP的程序合成算法进行比较。对于此次比-较，我们采用“通用程序合成基准测试套件” [12], 其中包含总计 29 个用于归纳式程序合成的基准问题[6]。

遵循 Helmuth 和 Spector [12], 提供的建议说明，我们针对每个基准测试问题使用 100 个不同的种子执行了算法。我们使用了建议数量的训练实例和测试实例，并将固定的边界情况纳入训练数据。我们还使用了其论文中描述的相同适应度函数。

对于进化搜索，我们采用每步替换 2 个个体的稳态替换方式，初始种群规模为 1 000，并使用 99.93% 的 Parent-Scalar。主程序的最大树深度设为 15， λ-函数的最大树深度设为 3。交叉率和变异率均根据经验设为 50%。最多允许进行 300000 次评估；只要算法根据训练数据找到完全准确的解，就提前停止。

我们报告在综合考虑训练数据和测试数据的情况下，在100次执行中找到的正确解决方案所占的百分比。

[6]HOTGP 的完整源代码可从以下网址下载：https://github.com/mcf1110/hotgp。

细化过程前后的基准测试集。为将这些结果置于当前文献背景下，我们将所得结果与 PushGP [12], 语法引导遗传编程（G3P） [3], 、G3P 的扩展语法版本（此处称为 G3P+） [4], ，以及一些近期提出的方法进行比较，例如代码构建遗传编程（CBGP） [25],和使用 Haskell 与 Python 语法的 G3P（G3Phs 和 G3Ppy） [5]。我们没有与 STGP 和 PolyGP 进行比较，因为它们的原始论文 [22, 32] 发表于这套基准测试集问世之前。所有结果均列于表2中。在此表中，所有无法使用我们当前函数集解决的基准测试，在 HOTGP 列中均标记为“–”。对于其他方法，短横线表示作者未针对该特定基准测试运行其算法。

4.1 结果分析

与其他算法相比，HOTGP 在 9 个基准问题的测试集上取得了最高成功率，其次是 PushGP 和 CBGP，分别在 7 个和 5 个基准测试中取得最高成功率。值得强调的是，HOTGP 在 4 个问题上取得了 100% 的成功率，在 7 个问题上取得了 $\geq 75\%$ 的成功率；只有 CBGP 达到了同样的结果。此外，在 29 个问题中，HOTGP 至少在 10 个问题上取得了 50% 的成功率，这一点是其他任何被比较的方法都未能做到的。这为我们的初始假设提供了证据：在程序合成中加入类型信息确实可以缩小搜索空间，从而提高进化过程的效率。

例如，在 比较字符串长度问题中，输入参数的类型为 String，输出为布尔值，但允许使用中间整数类型。查看表1可知，将字符串转换为布尔值的方法有限，因为我们只支持字符级函数。我们能做的最好办法是使用 Head 提取第一个字符，再用 IsLetter 或 IsDigit 将该字符转换为布尔值。例如，我们可以生成一个程序，对两个输入都执行这一操作，并使用不同的布尔运算符比较结果。也可以先应用 Map函数，再应用 Head。此外，要将字符串转换为整数，唯一的办法是使用 Length函数；而将两个整数转换为布尔值的方法也只有几种。一个得到的解例如：((length x1) > (length x0)) && ((length x1) < (length x2))。

另一方面，对于 最后一个零的索引 问题，使用我们的语法可以得到一种可能的正确解fst(head (reverse (filter(λy → 0 ==(sndy)) (zip (range 0 (length x0) 1)x0)))). 因此，合成器必须首先枚举输入，应用过滤器，仅保留包含0 的元素，将列表逆序，取第一个元素，并返回其索引。获得的最佳解之一是 ((length x0) + (if((head(reverse x0)) == 0)then 1 else0)) - 2，准确率为 32%。它只需检查最后一个元素是否为 0；如果是，则返回列表长度减一，否则返回长度减二。这是递归解的一种可能的一般情况，其中它检查

表2：在考虑训练（Tr）和测试（Te）数据集时，各问题找到成功解的比例（执行次数百分比）。HOTGP *列出了HOTGP经过简化过程后获得的结果。每个问题在测试数据集上的最佳值以突出显示。由于没有算法能够为这些问题找到结果，因此省略了校验和**、*collatz-numbers*、*string-differences*、*wallis-pi* 与*word-stats* 问题。**

	HOTGP		**HOTGP ***		PushGP	G3P	G3P+		CBGP	G3Phs		G3Ppy	
基准测试	Tr	Te	Tr	Te	Te	Te	Tr	Te	Te	Tr	Te	Tr	Te
比较字符串长度	100	100	100	100	7	2	96	0	22	94	5	12	0
计数奇数	46	46	50	50	8	12	4	3	0	–	–	–	–
数字	–	–	–	–	7	0	0	0	0	–	–	–	–
双字母	0	0	0	0	6	0	0	0	–	–	–	–	–
even-squares	0	0	0	0	2	1	0	0	–	–	–	–	–
for 循环索引	73	39	73	59	1	8	9	6	0	–	–	–	–
等级	37	32	39	37	4	31	63	31	–	–	–	–	–
最后一个零的索引	0	0	0	0	21	22	97	44	10	0	0	2	2
中位数	82	73	100	99	45	79	99	59	98	100	96	39	21
镜像	1	1	1	1	78	0	89	25	100	–	–	–	–
负数转零	100	100	100	100	45	63	24	13	99	0	0	68	66
数字输入输出	100	100	100	100	98	94	95	83	100	100	99	100	100
猪拉丁语	–	–	–	–	0	0	4	3	–	–	–	–	–
将空格替换为换行符	38	38	38	38	51	0	29	16	0	–	–	–	–
拼字游戏得分	–	–	–	–	2	2	1	1	–	–	–	–	–
小或大	28	59	28	59	5	7	39	9	0	30	4	0	0
最小的	98	95	100	100	81	94	100	73	100	100	100	99	89
反向字符串长度	87	87	89	89	66	68	20	18	–	0	0	35	34
平方和	1	1	1	1	6	3	5	5	–	–	–	–	–
超级字母异序词	–	–	–	–	0	21	43	0	–	30	5	51	38
音节	0	0	0	0	18	0	53	39	–	–	–	–	–
向量平均值	78	79	80	80	16	5	0	0	88	67	4	0	0
向量求和	34	34	37	37	1	91	28	21	100	100	68	0	0
x 字词行	–	–	–	–	8	0	0	0	–	–	–	–	–
最佳结果数量		4		9	7	2		3	5		1		2
= 100%		3		4	0	0		0	4		1		1
≥ 75%		6		7	3	4		1	7		3		2
≥ 50%		8		10	5	6		3	7		4		3

最后一个元素，并且条件判断其是否非零，使用列表的剩余部分递归。

如第3.3节所述，代码精炼步骤总能得到相同或更好的解。这种改进在中位数 和*for* 循环索引 问题上尤为明显。这是因为代码精炼有时能够舍弃误用的数值常量。例如，在中位数 问题中，有一个解在训练集上的准确率为 99%，即 max -96 (min (max x1 x2) (max (min x1 x2) x0))，但它仅在三个参数的中位数大于 –96时才有效，否则总会返回一个常量值。经过代码精化后，HOTGP 找到了最终的正确解：min (max (min x2 x1) x0) (max x1 x2)。

代码精炼的另一个好处是减少程序大小，这可以提高生成程序的可读性。图1显示了细化后程序大小的下降率，几何平均数为52%。细化过程的有效性取决于解决方案的性质

问题。对于大多数问题，上四分位数的末端都远低于 > 75% 的缩减幅度，这意味着某些解决方案被大幅简化并不罕见。然而，在诸如 *counts-odds*、*even-squares* 和平方和之类的问题中，结果更加明显；这些问题涉及的类型较少（因此语法也更精简），并且通常会达到最大评估次数，因此更容易发生膨胀。值得注意的是，数字输入输出 和负数转零几乎没有缩减，这表明该算法能够直接找到完美且近乎最简的解决方案。

为了进一步了解正确解法实际上有多精简，以及各个问题容易出现膨胀的程度，表3列出了 HOTGP 为每个问题找到的最小正确解法，并将其与作者编写的手写解法进行比较。即使在细化步骤之前，大多数解法的节点数也与手写解法相同，而且几乎所有解法都相当接近。平方和最初比人工解法大得多，但细化后，规模显著缩小。我们找到的唯一正确解法

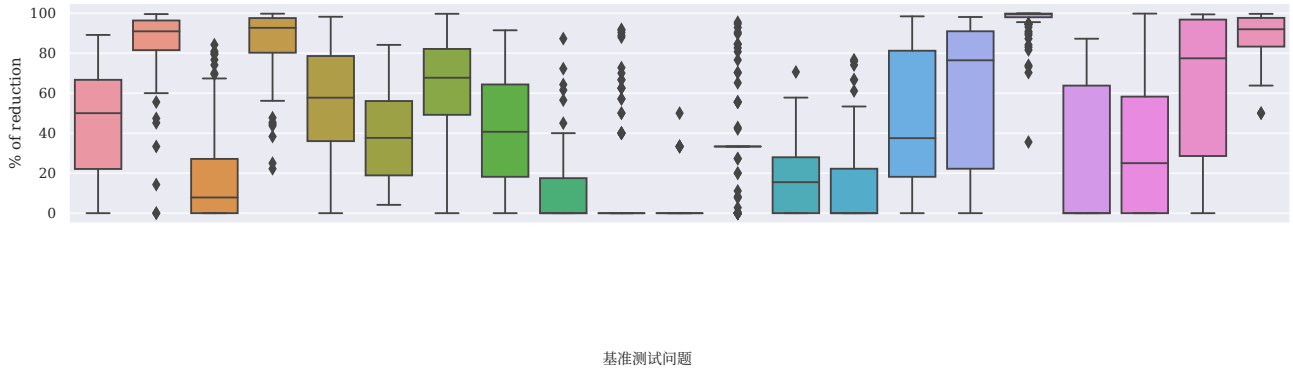

图1：细化过程导致节点数量减少的百分比

s.

镜像也实现了这一批次中最大的缩减，整体缩减了 87%。

表3：手工构造的解（HC）以及 HOTGP 和 HOTGP* 找到的最小正确解的节点数。括号中显示的是相对于 HC 的节点数。

基准测试	HC	HOTGP	HOTGP *
比较字符串长度	11	11 (1.0x)	11 (1.0x)
计数奇数	4	4 (1.0x)	4 (1.0x)
for 循环索引	7	25 (3.6x)	9 (1.3x)
等级	27	45 (1.7x)	29 (1.1x)
中位数	9	9 (1.0x)	9 (1.0x)
镜像	10	102（10.2 倍）	13（1.3 倍）
负数转零	3	3（1.0 倍）	3（1.0 倍）
数字输入输出	4	4（1.0 倍）	4 (1.0x)
将空格替换为换行符	8	8（1.0x）	8 (1.0x)
小或大	11	12（1.1 倍）	12 (1.1x)
最小	7	7 (1.0x)	7 (1.0x)
反向字符串长度	4	5 (1.2x)	5 (1.2x)
平方和	7	163 (23.3x)	30 (4.3x)
向量平均值	6	6 (1.0x)	6 (1.0x)
向量求和	5	5（1.0倍）	5（1.0 倍）

5 结论

本文介绍了 HOTGP，这是一种支持高阶函数、λ-函数、多态类型以及利用类型信息约束搜索空间的遗传编程算法。它还采用了一种基于 Haskell 语言的语法，其中非终结符集合仅使用纯函数。我们支持这一方法的主要论据是：i）将程序限制为纯函数可以避免不良行为；ii）使用类型层面的信息和参数多态性可以缩小搜索空间，引导遗传编程算法找到正确的解决方案；iii）高阶函数消除了对若干命令式结构的需求（例如，for 循环）。

HOTGP 与大多数遗传编程实现不同：它会主动利用输入和输出类型的信息，在创建新解或修改现有解时约束候选终结符和非终结符，并选择可行的重组位置。

我们使用 29 个基准问题评估了该方法，并将结果与文献中的 6 种最先进算法进行了比较。总体而言，结果令人满意：对于 10 个问题，该方法大多数时候都能稳定地返回正确程序，这一成绩是所有受测方法都未能达到的。此外，HOTGP 达到最高成功率的次数也多于最先进算法。

我们还对算法找到的最佳解决方案应用了代码精化，以减少膨胀 代码的出现。该过程进一步改善了结果，同时提高了最终程序的可读性。

尽管我们取得了具有竞争力的结果，但我们观察到仍有可能改进的地方。首先，我们的非终结符集合比某些最先进算法小得多（例如，PushGP）。未来的工作包括仔细研究向语法中添加新函数的影响。这种纳入可能会进一步简化程序合成，或使我们找到当前尚未找到的解决方案。另一方面，纳入新函数也会扩展搜索空间，并可能妨碍我们当前的一些结果。

我们的 PS 方法也可以从更模块化的视角中受益。在模块化方法中，首先将问题划分为更简单的任务，分别独立求解，然后组合起来生成完整的合成程序。这种方法需要支持不同形式的函数组合，并修改基准测试，为不同的子任务创建训练数据。这样的合成器还可以与 Wingman[7]（目前用于高级 Haskell 代码生成的实现）结合使用；Wingman[7] 既可以合成整个程序，也可以仅利用类型信息和代码空洞来引导这一过程。

还需要进一步研究更高级的类型级信息，例如广义代数数据类型（GADTs）、类型族、精化类型和依赖类型。更多类型信息可以进一步约束搜索空间，并在某些情况下为合成正确程序提供额外提示。显然，这必须伴随对当前基准测试的修改，并纳入新的基准测试，以提供有关目标程序的这类高级信息。

[7]https://haskellwingman.dev/

参考文献

[1] Tom Castle 和 Colin G. Johnson。2012 年。使用强结构遗传编程演化高级命令式程序树。载于《遗传编程》，Alberto Moraglio、Sara Silva、Krzysztof Krawiec、Penousal Machado 和 Carlos Cotta（编）。Springer Berlin Heidelberg，德国柏林，1–12。[2] N.G de Bruijn。1972 年。带有无名占位符的 λ演算记法：一种用于自动处理公式的工具，并应用于丘奇-罗瑟定理。《Indagationes Mathematicae（会议录）》75，5（1972），381–392。https: //doi.org/ 10 1016 1385 7258 72 90034 0 。/ - ( ) - Stefan Forstenlechner 、 David Fagan[3] 、 Miguel Nicolau 和 Michael O'Neill。2017 年。遗传编程中任意程序合成问题的语法设计模式。载于《遗传编程》，James McDermott、Mauro Castelli、Lukas Sekanina、Evert Haasdijk 和 Pablo García-Sánchez（编）。Springer International Publishing，尚，262–277。[4] Stefan Forstenlechner、David Fagan、Miguel Nicolau 和 Michael O'Neill。2018 年。扩展程序合成语法以实现语法引导的遗传编程。载于《自然界并行问题求解——PPSN XV》，Anne Auger、Carlos M. Fonseca、Nuno Lourenço、Penousal Machado、Luís Paquete 和 Darrell Whitley（编）。Springer International Publishing，尚，197–208。[5] Fraser Garrow、Michael A. Lones 和 Robert Stewart。2022 年。为什么函数式程序合成很重要（在遗传编程领域）。载于《遗传与进化计算会议伴会》（美国马萨诸塞州波士顿）《GECCO '22》。美国计算机协会，纽约，纽约州，美国， 1844 1853 10 1145 3520304 3534045 –。https://doi.org/ 。/ 。[6] Sumit Gulwani。2010 。程序合成的维度。载于《第 12 届 ACM 声明式编程原理与实践国际研讨会论文集》（奥地利哈根贝格）《PPDP '10》。美国计算机协会，纽约，纽约州，美国， 13 24 10 1145 1836089 1836091 –。https://doi.org/ 。/ 。[7] Sumit Gulwani、Oleksandr Polozov、Rishabh Singh 等。 2017 。程序合成。《Foundations and Trends® in Programming Languages》4，1-2（2017），1–119。[8] Thomas Haynes、Roger L. Wainwright、Sandip Sen 和 Dale A. Schoenefeld。1995 年。演化合作策略中的强类型遗传编程。载于《第 6 届国际遗传算法会议论文集》。Morgan Kaufmann Publishers Inc.，美国加利福尼亚州旧金山，271–278。[9] Yifan He、Claus Aranha 和 Tetsuya Sakurai。2022 年。通过 PushGP 和自适应替换变异将子程序作为知识融入程序合成。载于《遗传与进化计算会议伴会》（美国马萨诸塞州波士顿）《GECCO '22》。美国计算机协会，纽约，纽约州，美国， 554 557 10 1145 3520304 3528891 –。https://doi.org/ 。/ 。[10] Thomas Helmuth、Nicholas Freitag McPhee、Edward Pantridge 和 Lee Spector。2017 年。通过自动简化改进演化程序的泛化能力。载于《遗传与进化计算会议论文集》（德国柏林）《GECCO '17》。美国计算机协会，纽约，纽约州，美国， 937 944 10 1145 3071178 3071330 –。https://doi.org/ 。/ 。[11] Thomas Helmuth、Nicholas Freitag McPhee 和 Lee Spector。2018 年。使用增删式均匀变异进行程序合成。载于《遗传与进化计算会议论文集》（日本京都）《GECCO '18》。美国计算机协会，纽约，纽约州，美国，1127–1134。https://doi.org/ 10 1145 3205455 3205603 。/ 。Thomas Helmuth 和 Lee Spector。[12] 。通用程序合成基准测试套件。载于《GECCO '15：2015 年遗传与进化计算年度会议论文集》。ACM，西班牙马德里，1039–1046。https://doi.org/10. 1145/2739480.2754769[13] Thomas Helmuth 和 Lee Spector。2022 年。下采样词法选择的问题求解优势。《《人工生命》》27，3–4（2022），183–203。[14] Erik Hemberg、Jonathan Kelly 和 Una-May O'Reilly。2019 年。利用领域知识和新颖性通过语法演化改进程序合成性能。载于《遗传与进化计算会议论文集》（捷克共和国布拉格）《GECCO '19》。美国计算机协会，纽约，纽约州，美国， 1039 1046 10 1145 3321707 3321865 –。https://doi.org/ 。/ 。[15] Mark P Jones。1995 。构造器类系统：重载与隐式高阶多态。《函数式编程杂志》5，1（1995），1–35。

[16] Susumu Katayama。2005。λ函数的系统化搜索。《函数式编程趋势》6（2005），111–126。[17] Jonathan Kelly、Erik Hemberg 和 Una-May O'Reilly。2019。通过新颖的探索—利用控制改进遗传编程。载于《遗传编程》，Lukas Sekanina、Ting Hu、Nuno Lourenço、Hendrik Richter 和 Pablo García-Sánchez（编）。Springer International Publishing，尚，64–80。[18] John R Koza。1992。《遗传编程：通过自然选择实现计算机编程》。Bradford Books，马萨诸塞州剑桥。[19] John R. Koza 和 Riccardo Poli。2005。遗传编程。载于《搜索方法：优化与决策支持技术导论教程》，Edmund K. Burke 和 Graham Kendall（编）。Springer US，波士顿，马萨诸塞州，127–164。https://doi.org/ 10 1007 0 387 28356 0_5 。/ - - - Sean Luke 和 Liviu Panait。[20] 。遗传编程膨胀控制方法比较。《进化计算》14，3（2006），309–344。[21] Daniel Manrique、Juan Ríos 和 Alfonso Rodríguez-Patón。2009。语法引导遗传编程。载于《人工智能百科全书》。IGI Global， 767 773 10 4018 978 1 59904 849 9 114 – 。https://doi.org/ 。/ - - - - 。[22] 1995 。ch David J Montana。 。强类型遗传编程。《进化计算》3，2（1995），199–230。[23] Roland Olsson。1995。使用增量式程序转换的归纳式函数式编程。《人工智能》74，1（1995），55–81。[24] Edward Pantridge、Thomas Helmuth、Nicholas Freitag McPhee 和 Lee Spector。2017。归纳式程序合成方法的基准测试难度。载于《遗传与进化计算会议伴会论文集》（德国柏林）（GECCO '17）。美国计算机协会，纽约，纽约州，美国， 1589 1596 10 1145 3067695 3082533 – 。https://doi.org/ 。/ 。[25] Edward Pantridge、Thomas Helmuth 和 Lee Spector。2022 。函数式代码构建遗传编程。载于《遗传与进化计算会议论文集》（波士顿，马萨诸塞州）（GECCO '22）。美国计算机协会，纽约，纽约州，美国，1000–1008。https://doi.org/10.1145/ 3512290.3528866[26] Edward Pantridge 和 Lee Spector。2020。代码构建遗传编程。载于《2020年遗传与进化计算会议论文集》（坎昆，墨西哥）（GECCO '20）。美国计算机协会，纽约，纽约州，美国， 994 1002 10 1145 3377930 3390239 – 。https://doi.org/ 。/ 。[27] Baishakhi Ray、Daryl Posnett、Premkumar Devanbu 和 Vladimir Filkov。2017。GitHub中编程语言与代码质量的大规模研究。《ACM通讯》 60, 10 2017 91 100 10 1145 3126905 （9月），– 。https://doi.org/ 。/ Paulo Santos、Sara Silva 和 Alcides Fonseca。[28] 2020 。作为遗传编程用户界面的精化类型化遗传编程。载于《2020年遗传与进化计算会议伴会论文集》（坎昆，墨西哥）（GECCO '20）。美国计算机协会，纽约，纽约州，美国， 251 252 10 1145 3377929 3390042 – 。https://doi.org/ 。/ 。[29] Harald Søndergaard 和 Peter Sestoft。1990 。引用透明性、确定性与可展开性。《Acta Informatica》27（1990），505–517。[30] Gilbert Syswerda。1991。代际遗传算法与稳态遗传算法中的繁殖研究。载于《遗传算法基础》，GREGORY J.E. RAWLINS（编）。第1卷。Elsevier，94–101。https://doi.org/10.1016/B978-0-08- 050684-5.50009-4[31] David A Turner。1979。应用式语言的一种新实现技术。《软件：实践与经验》9，1（1979），31–49。[32] Tina Yu。2001。多态性与遗传编程。载于《遗传编程》，Julian Miller、Marco Tomassini、Pier Luca Lanzi、Conor Ryan、Andrea G. B. Tettamanzi 和 William B. Langdon（编）。Springer Berlin Heidelberg，柏林，海德堡，218–233。[33] Tina Yu 和 Chris Clack。1998。PolyGP：Haskell中的多态遗传编程系统。载于《遗传编程1998：第三届年度会议论文集》，John R. Koza、Wolfgang Banzhaf、Kumar Chellapilla、Kalyanmoy Deb、Marco Dorigo、David B. Fogel、Max H. Garzon、David E. Goldberg、Hitoshi Iba 和 Rick Riolo（编）。Morgan Kaufmann，美国威斯康星州麦迪逊，416–421。http://www.cs.mun.ca/~tinayu/index_files/addr/public_ html/pgp.new.pdf