北京大学

硕士研究生学位论文

题目：**面向程序合成的领域特定语言调整技术**

姓　　名：	刘鑫远
学　　号：	1801213693
院　　系：	信息科学技术学院
专　　业：	计算机软件与理论
研究方向：	编程语言与程序分析
导师姓名：	熊英飞副教授

二〇二一年六月

版权声明

摘要

如何更加自动化地编写程序一直是软件工程和编程语言领域所关注的问题，程序合成有望成为该问题的最佳答案之一。程序合成根据输入的规约和给定的语言自动地生成该语言上符合规约的程序。近年来，程序合成技术发展迅速，已经在多个领域得到了成功的应用。

当前，程序合成技术的主要瓶颈在于求解速度。针对这一问题，研究人员已经采取了许多不同的策略和方案。本文尝试从一个新的角度来对待这一问题，即通过设计出更好的领域特定语言以加速程序合成的求解过程。程序合成高度依赖给定的语言，程序合成的过程实际上是在给定的语言上的所有程序中找到一个符合规约的程序，不同的语言会大幅影响程序合成的求解速度。

本文分析了语言对程序合成求解速度的影响因素，并提出了一种自动化地设计出更好的领域特定语言，用以加速程序合成的方法。该方法输入为某个特定领域内的若干个程序合成问题，以及一个较为通用的语言作为源语言，自动地在源语言的基础上设计出新的领域特定语言。

本文的方法首先利用已有的程序合成求解器，在领域内的问题上生成大量程序用作训练。再通过挖掘训练用程序上的特定语法结构信息，包括训练用程序中的频繁模式、语句出现频率等信息，抽象出一系列语法上的修改操作，进而根据这些修改操作，利用启发式搜索算法，生成一个新的领域特定语言。新的领域特定语言中添加了一些训练用程序中的频繁模式抽象出来的函数或语句，删除了一些训练程序中极少出现的语句，从而达到了加速程序合成求解的目的。

本文实现了该方法，并在字符串处理领域进行了验证。实验证明，该方法在字符串处理问题上设计的新的领域特定语言能够成功的加速程序合成的求解，在字符串处理领域目前速度最快的求解器 MaxFlash 上获得了平均 2.64 倍的加速效果。

关键词：程序合成，领域特定语言，频繁模式挖掘，启发式搜索

Automatic Domain-Specific Language Adjustment in Program Synthesis

Xinyuan Liu (Computer Software and Theory)
Directed by Prof. Yingfei Xiong

ABSTRACT

How to automate software programming has always been a concern in the field of software engineering and programming language. Program Synthesis is expected to be one of the best answers to this problem. Program Synthesis takes a specification of program as input and automatically generates a program in a given language satisfying the specification. In recent years, Program Synthesis techniques are developing rapidly and has been successfully applied in many domains.

At present, the bottleneck of Program Synthesis is speed. To solve this problem, researchers have developed many different techniques. This paper attempts to tackle this problem from a new perspective, that is, to design a better Domain-Specific Language to speed up the solution process of Program Synthesis. Program Synthesis highly depends on the given language. In fact, the process of Program Synthesis is to find a program satisfying the specification from all programs on the given language. Different languages will greatly affect the solution speed of Program Synthesis.

This paper analyzes the factors that affect the speed of Program Synthesis, and proposes an novel approach to design a better Domain-Specific Language automatically, in order to accelerate Program Synthesis. The input of this approach is a set of Program Synthesis problems in a specific domain, and a base language is used as the source language. A new Domain-Specific Language is automatically designed on the basis of the source language.

In this paper, existing Program Synthesis solver is used to generate a large number of programs for training. Then a series of grammatical modification operations are abstracted by mining grammatical structure information in the training program, including some frequent patterns and sentence frequency, etc. Finally, a new Domain-Specific Language is generated according to these modification operations by heuristic search algorithm. In the new Domain-Specific Language, some functions or statements abstracted from frequent patterns are added, and some rare statements in training programs are deleted, so as to accelerate the process of Program Synthesis.

This paper implements the approach above and verifies it on string processing. Experiments show that the new Domain-Specific Language can successfully accelerate Program Synthesis and obtain an average acceleration of 2.64 times on MaxFlash, the fastest solver on string processing in the state of art.

KEY WORDS: Program Synthesis, Domain-Specific Language, Frequent Pattern Mining, Heuristic Search

目录

第一章 引言

1.1 研究背景

如何更加自动化地编写程序一直是软件工程和编程语言领域所关心的问题。C 语言等高级语言的出现使得程序开发人员能够在更高的抽象层次编写程序，通过编译技术生成底层的汇编语言和机器码。这一过程大幅自动化了开发人员编写程序的过程，提高了程序开发人员的开发效率。

在这些已有技术的基础上，如何进一步地自动化程序开发过程成为研究人员关心的问题。程序合成技术就是这一问题可能的解决方案之一[1]。程序合成的目标是能够自动地生成程序[2]。程序合成的输入为：1）一个程序空间 Prog，通常可以用某种程序语言的语法来描述。2）程序要满足的规约 Spec，输出一个程序空间 Prog 中的程序 p，满足给定的规约 Spec[3]，即有：

$$p \in Prog \wedge p \mapsto Spec$$

程序的规约通常指的是程序的输入输出需要满足的约束条件。程序的规约可以由逻辑表达式给出，也可以由输入输出样例给出。根据某种算法，自动地找到在程序空间中的满足输入规约的程序，这一过程就是程序合成的过程。本文中为方便起见，也把这一过程称为程序合成的求解器求解某一个程序合成问题。

程序合成大多限定在一个领域特定语言（DSL, Domain Specific Language）所描述的程序空间内进行求解，即生成一个该领域特定语言上的程序。程序合成要求针对每一个或者每一类问题，预先给出相应的领域特定语言，求解器进而在这一语言所限定的语法空间内求解出一个能够满足规约的解。程序合成限定在领域特定语言上求解的原因是，目前的求解速度[4]严重限制着限制程序合成技术，是这一技术当前的主要瓶颈所在。领域特定语言对于特定领域内的任务有着足够的表达能力，同时又限制了程序合成过程的求解空间，使得程序合成求解过程能够在合理的时间之内完成。如果不对程序空间加以限制，由于目前的求解技术和计算能力的限制，程序合成过程的时间开销就会变得非常巨大。

程序合成过程非常依赖领域特定语言的设计，目前，已经有许多种人工定义的领域特定语言应用在了程序合成上面。在不同的领域需要不同的领域特定语言，在同一个领域也可以有多种不同的领域特定语言。人工设计新的领域特定语言需要足够多的领域知识，因此，设计领域特定语言是非常耗时和困难的。那么，如何自动地设计领域特定语言就成为了程序合成领域要解决的重要问题。

1.2 研究动机

目前，限制程序合成性能的瓶颈主要在于求解速度。程序合成问题的困难之处在于，即使领域特定语言相对通用语言较为精简，其所描述的程序的空间也仍然是非常大的，想要在巨大的程序空间中找到满足规约的解通常要花费很长的时间。

显然，领域特定语言描述了一个程序合成问题进行求解的程序空间，而程序空间的大小和结构直接影响着程序合成过程的求解速度。因此，对于同一个问题或同一个领域的问题，程序合成求解器在不同的领域特定语言上的求解速度也是不同的，已经有一些研究工作证明了这一点[5][6]。

下面，本文将用一个简单的程序合成的例子来直观地说明。该问题的程序规约由若干输入输出样例给出：

f("938-242-504") = "938242504"
f("308-916-545") = "308916545"
f("623-599-749") = "623599749"
f("981-424-843") = "981424843"
f("438-242-514") = "438242514"
f("218-916-545") = "218916545"
f("763-599-749") = "763599749"
f("981-435-843") = "981435843"
f("938-242-144") = "938242144"
f("228-916-545") = "228916545"
f("626-599-749") = "626599749"
f("981-424-883") = "981424883"

该问题的给出了程序 f 的若干输入输出样例，求解程序 f。f 的功能为去掉一串电话号码中的连字符“-”。接下来，考虑该问题在如下两个不同的字符串处理 DSL 上的求解，下面的表格中列出了每种领域特定语言支持的函数。

表 1.1 字符串处理 DSL1

(str.++ String String)	字符串连接
(str.at String Int)	根据下标取字符，字符视作 1 长度的字符串
(str.substr String Int Int)	根据下标取子串
(str.len String)	字符串长度
(str.indexof String String)	字符串检索，返回子串在母串中第一次出现的下标
(+ Int Int)	整数加法
(- Int Int)	整数减法

表 1.2 字符串处理 DSL2

(str.++ String String)	字符串连接
(str.at String Int)	根据下标取字符，字符视作 1 长度的字符串
(str.substr String Int Int)	根据下标取子串
(str.len String)	字符串长度
(str.indexof String String)	字符串检索，返回子串在母串中第一次出现的下标
(+ Int Int)	整数加法
(- Int Int)	整数减法
(str.replace String String String)	字符串替换，只进行一次替换

DSL2 与 DSL1 的区别只有增加了 str.replace 函数，该函数对第一个参数 String1 中的字符串进行一次替换，将其中第一次出现的 String2 替换为 String3。

在 DSL1 中求解上面的问题，需要用到一系列复杂的取子串和字符串连接函数：

f(s) = (str.++ (str.++ (str.substr s 0 3) (str.substr s 4 7)) (str.substr s 8 11))

而用 DSL2 进行求解，则只需要用到两次字符串替换函数：

f(s) = (str.replace (str.replace s “-” “”) “-” “”)

显然，使用 DSL2 求出的解结构简单得多，在大多数情况下，DSL2 求解该问题速度也比使用 DSL1 求解更快。这里值得注意的是，DSL2 中新增的 str.replace 函数是能够被 DSL1 中的函数表示出来的：

```
(str.replace a b c) =
    (str.++ (str.substr a 0 (str.indexof a b))
        (str.++ c (str.substr a (+ (str.indexof a b) (str.len b)) (str.len a))))
```

即将(str.replace a b c)用连接 a 中 b 首次出现之前的子串、c 和 a 中 b 首次出现之后的子串的方式实现。不难理解这两种实现方式是等价的。因此，DSL1 的表达能力和 DSL2 是相同的，即 DSL2 能够描述的所有程序功能都可以被 DSL1 实现。

但是，加入 str.replace 函数的 DSL2 能够显著地加快程序合成在该题目上的求解过程。考虑更多的字符串处理问题，不难发现，在字符串处理问题中字符串替换是经常使用到的操作。因此，可以直观地讲，在字符串处理领域，DSL2 是比 DSL1 更优的领域特定语言。

那么，如何设计出更好的领域特定语言用于程序合成？这将是本文研究的重点问题。本文中，将使用程序合成在某一特定领域的问题上的求解速度来评价不同 DSL 的优劣。本文的目标是找到一种方法，自动化地设计出新的领域特定语言，加速程序合成在领域内的问题上的求解速度。

本文提出的方法从一个基础的、通用的领域特定语言出发，输出一个新的、更加抽象的领域特定语言。继续以上文的两个 DSL 为例。DSL2 中新加入的函数 str.replace 函数完全可以被 DSL1 中的函数表达出来，那么，是否存在某种方法能够自动化地得到这样的函数？

本文采用挖掘的方法解决这一问题，即通过在该领域内的程序上寻找频繁出现的模式，将其抽象成新的语法功能加入 DSL 中。具体地，假设字符串替换操作在字符串处理领域是经常被使用的，那么，在已有的 DSL1 语言的字符串处理程序中，应该存在大量的相同模式来完成字符串替换的功能，那么这样的模式就有望被挖掘出来，作为新的语法功能加入 DSL 中。

1.3 主要贡献

本文的主要贡献有：

1）提出了一种自动化设计程序合成中的 DSL 的方法，利用现有的程序中挖掘到的程序结构信息设计新的 DSL。

2）深入分析了影响程序合成求解器求解速度的因素，并设计了估计函数对新 DSL 的加速效果进行估计。

3）实现了本文提出的方法，并在字符串处理领域进行了实验验证。

4）本文提出的方法对目前字符串处理问题上速度最快的求解器 MaxFlash 达到了

平均 2.64 倍的加速效果。

1.4 论文结构

本文一共分为七章，每一章的主要内容分别为：

第一章为引言部分，主要介绍了本文工作的研究背景、研究动机与主要贡献。

第二章介绍了程序合成的若干技术和与本文相关的一些研究工作。

第三章分析了本文研究的问题，并对本文提出的方法进行了简要概述。

第四章具体讲述了本文方法设计与实现。

第六章对本文的方法进行了实验验证。

第七章对本文的工作进行了总结并对未来的工作方向进行了设想与展望。

第二章 背景介绍与相关工作

2.1 程序合成

2.1.1 问题定义

程序合成的输入是一系列程序的输入输出要满足的约束条件，即程序的规约，在给定的 DSL 语法所描述的空间内，输出是一个满足规约的程序。程序的规约可以分为两大类：

1）输入输出的逻辑约束

给出程序的输入输出需要满足的逻辑条件作为程序的规约，以求两个数中最大值的 max 问题为例：

$$\max(x,y) \geq x \ \wedge \ \max(x,y) \geq y \ \wedge (\ \max(x,y) = x \ \vee \ \max(x,y) = y \ )$$

这个例子是 max 函数需要满足的逻辑约束。max 函数接受 x 和 y 两个参数作为输入，max(x,y)的输出结果要同时满足以下三个条件：

a）大于等于 x

b）大于等于 y

c）等于 x 或者等于 y

这样的输入输出逻辑约束完备地描述了 max 函数所需要实现的功能和语义。

2）输入输出样例

给定若干组程序的输入输出样例作为程序的规约，根据输入输出样例生成程序。根据输入输出样例生成程序的过程也被称作样例编程（Programming By Example, PBE）。例如，可以通过若干输入输出样例描述上文中 max 函数需要满足的规约：

$$\begin{gathered}\max(1,2) = 2 \ \wedge \\ \max(3,2) = 3 \ \wedge \\ \max(3,5) = 5 \ \wedge \\ \max(0,1) = 1 \\ \cdots\end{gathered}$$

上面这样的输入输出样例能够不完备的描述 max 函数所需要实现的功能和语义。

输入输出的逻辑约束往往是完备的，基于输入输出的样例的约束条件通常是不完备的。需要注意的是，这两者都有例外。基于逻辑条件的约束也可能是不完备的，某些情况下不完备的逻辑约束也可以被用于程序合成。相反，通过穷举所有合法输入空间中的输入对应的输出，基于输入输出样例的约束也可以是完备的。

不完备的规约会导致求解器给出错误的程序，即满足了给出的规约，但程序行为不符合或不完全符合用户的预期，这种现象称之为过拟合[7]。

虽然，上文中这种通常是不完备的、基于输入输出样例的规约会导致程序合成的求解器给出错误的程序。但是在实践中，基于输入输出的样例的程序合成仍然有着更大量的应用场景。这主要有以下几点原因：

1）给出较为完备地逻辑约束的过程往往较为复杂，很多情况下难以给出完备的逻辑约束，或者给出完备的逻辑约束过程费时费力。相反，给出输入输出样例通常是相对较为方便和容易的，用户只需要提供若干组程序的合法输入和相应的预期结果即可。

2）验证程序是否满足逻辑约束的过程可能较为复杂和耗时[8]，而验证程序是否满足给出的输入输出样例往往是容易和快速的。受限于时间开销的考虑，基于输入输出样例的约束往往有较大优势。

3）虽然不完备的输入输出样例约束会导致程序合成求解器给出错误的程序，但是这可以通过增加输入输出样例的数量来解决。上文中也提到过，给出输入输出样例是相对容易的。通过预先保证输入输出样例的数量，或者交互式地，在生成的程序不符合用户预期的时候要求用户提供更多样例都能够在一定程度解决这个问题。

4）程序合成往往限制在某一个领域特定语言所表达的程序空间内，领域特定语言也能在一定程度上限制过拟合的发生。

2.1.2 基于枚举的程序合成

按一定的搜索策略遍历给定的领域特定语言所描述的程序空间中的所有程序，并用输入的规约对程序逐一进行验证，这是朴素的枚举法解决程序合成问题的基本思想。虽然程序合成的枚举搜索空间受到领域特定语言的限制，相对在通用语言上搜索较小，但整个搜索空间仍然很大，所以使用朴素的枚举搜索方法效率仍然非常低下。在朴素枚举法的基础上，研究人员采用了许多方法对枚举法进行优化和加速，Euphony[9]是枚举法的一个比较有代表性的实例，Euphony 采用了如下几种策略来优化枚举算法和搜索过程：

1）采用概率模型对搜索空间进行赋权和排序，按照权重从高到低的顺序遍历 DSL

语法所描述的程序空间。

Euphony 使用了 PHOG [10](Probabilistic Higher-Order Grammar，概率高阶语法)模型来对语法空间赋权和排序。理论上，这一步可以使用许多模型，比如 N-Gram 模型[11]、PCFG [12](Probabilistic Context-Free Grammar，概率上下文无关文法)、决策树模型[13]、神经网络预测模型[14]等等。

$$A[context] \rightarrow \beta \qquad P$$

$$\begin{array}{lcll}
S[\text{“-”}, \texttt{Rep}] & \rightarrow & \text{“.”} & \mathbf{0.72} \\
S[\text{“-”}, \texttt{Rep}] & \rightarrow & \text{“-”} & \mathbf{0.001} \\
S[\text{“-”}, \texttt{Rep}] & \rightarrow & x & 0.12 \\
S[\text{“-”}, \texttt{Rep}] & \rightarrow & S+S & 0.02 \\
\cdots & & & P \\
S[\text{“.”}, \texttt{Rep}] & \rightarrow & \text{“.”} & 0.001 \\
S[\text{“.”}, \texttt{Rep}] & \rightarrow & \text{“-”} & \mathbf{0.002} \\
S[\text{“.”}, \texttt{Rep}] & \rightarrow & x & 0.01 \\
S[\text{“.”}, \texttt{Rep}] & \rightarrow & S+S & 0.19 \\
\cdots & & &
\end{array}$$

图 2.1　概率高阶语法

上图是 PHOG 的一个简单实例。PHOG 在上下文无关文法的基础上增加了每个非终结符展开到每个展开式的概率，同时考虑了上下文信息。上图中的例子中每一行表示一个展开式，展开式的上下文信息选用的是父节点和左兄弟，最右侧表示展开式的概率。例如：

$$S[\text{“-”}, Rep] \rightarrow \text{“.”} \qquad 0.72$$

这条展开式表示非终结符 S 在父节点为 “-”，左兄弟节点为 Rep 函数时，展开为“.” 这一个语法展开式的概率为 0.72。

搜索过程中的每个中间体或者最终的完整程序都可以看作用从初始节点按若干个展开式展开得到的，那么该中间体的权重就可以用这些展开式在 PHOG 中的概率的乘积或者对数和计算出来。

2）采用 A*算法[15]进行剪枝

剪枝也是搜索方法中常用的优化策略。A*算法是在求解最短路问题中常用的剪枝算法。按照权重从高到低遍历程序空间的问题也可以看作最短路问题，那么将 A*算法应用在这里就是非常直接的了。

A*算法的思想是将最短路问题中的距离函数划分为两部分，即当前搜索节点之前已经搜索过的距离和当前节点到目标节点还未搜索过的未知距离，即 *D(n) = f(n) + g(n)*。其中，n 表示搜索进行到当前的第 n 个节点，D(n)表示整条路径的长度，f(n)表示从初始节点到当前节点的路径长度，g(n)表示从当前节点到目的节点的路径长度。在实际的求解最短路问题的搜索过程中，从初始节点到当前节点的长度 f(n)是已知的，而当前节点到目的节点的路径长度 g(n)是未知的，于是采用估价函数 g*(n)来估计 g(n)，当 $g^*(n) \leqslant g(n)$时，可以保证搜索到最短路径。在寻路问题中，通常 g* (n)可以采用地图上的几何距离来估算 g(n)。

应用到程序综合问题上，A*算法被用在遍历程序空间这一过程中。按一定的权重遍历程序空间，程序空间中包含若干个待验证的程序，每次取出还未遍历到的权重最高的程序，这一问题也可以看作迭代求解最短路问题。

在遍历程序空间的问题上，将目标代价函数设定为中间体的部分程序或完整程序的所有展开式概率的对数和的相反数，即$f(n) = -\sum log_2 P(n)$ ，同时，将目的节点设定为所有的完整程序。按权重从大到小遍历程序空间的问题就转化成了使用 A*算法寻找最短路的问题。其中 g(n)的估计函数 g*(n)可以使用

$$g^*(n)\begin{cases} 0 & ,n\text{为终结符} \\ -\sum log_2 h(n_i) & ,n\text{为非终结符} \end{cases}$$

其中，n_i 表示 n 中的非终结符，$h(n_i)$表示从 n_i 展开成不含非终结符的完整表达式的概率上界，即从 n_i 展开的所有表达式中概率最高的一个。

使用 A*算法进行搜索和剪枝，能够高效地按权重从高到低的顺序遍历程序空间中的所有程序，并对这些程序逐一进行验证。使用 A*算法在基于枚举的程序合成中是一种很好的加速策略。

3）采用分治法[16]。

Euphony 采用了分治法进行搜索，即针对不同的输入输出生成多组程序，再生成出条件表达式判断不同的输入对应不同的程序。即先用枚举法对不同的输入输出对生成不同的子表达式，再用枚举算法生成一系列条件表达式，再用类似决策树训练的算法把子表达式用条件判断语句连接成一棵树的形式，形成一个完整的程序。这种方法有一定局限性，但也在某些情况下有比较好的效果。

2.1.3 基于约束求解的程序合成

约束求解是计算机科学领域的经典问题之一，随着 SMT[17]（Satisfiability Modulo

Theories，可满足性模理论）求解器的发展，约束求解器的求解速度得到了大幅的提升，能够求解的理论也逐渐丰富。

SMT 求解器接受一组约束，返回这组约束是否可以被满足。大多数求解器还可以在约束可以被满足的情况下额外返回返回一组赋值，在不能被满足的情况下则返回一个反例。

程序合成问题可以被转化成 SMT 求解问题。一种比较直接的方法是基于构件的程序合成[18]。基于构件的程序合成将程序合成问题转化为组成程序构件间的连接问题。即，将语法中的元素视作一个个语法构件，每个语法构件接受零到若干个输入，得到一个输出。使用 SMT 求解出一组满足所有约束条件的连接，再将这一组连接映射回程序空间得到一个程序作为程序合成的结果。

以上面提到的 max 问题为例，假设程序合成在下面的 DSL 上求解该问题。

```
(max ((x Int) (y Int)) Int
        ((Start Int (x
                     y
                     (+ Start Start)
                     (ite StartBool Start Start)))
      (StartBool Bool ((< Start Start)))))
```

图 2.2 Max 问题的 DSL

语法表达式(StartBool Bool ((< Start Start)))表示非终结符 StartBool 的类型为 Bool，共有一种展开方式，即展开为(< Start Start)。下文中还会使用这种方式介绍 DSL 的语法，不再对语法表达式的含义进行赘述。

max 函数接受两个参数 x 和 y，函数体为 Int 类型的表达式。该语法有两个非终结符 Start 和 StartBool。Start 为 Int 类型，可以展开为 x、y、加法和 if-then-else 表达式。StartBool 为 Bool 类型，可以展开为小于号表达式

基于构件的程序合成方法将程序合成为题转化为 SMT 问题，这一过程中将使用这些语法构件：

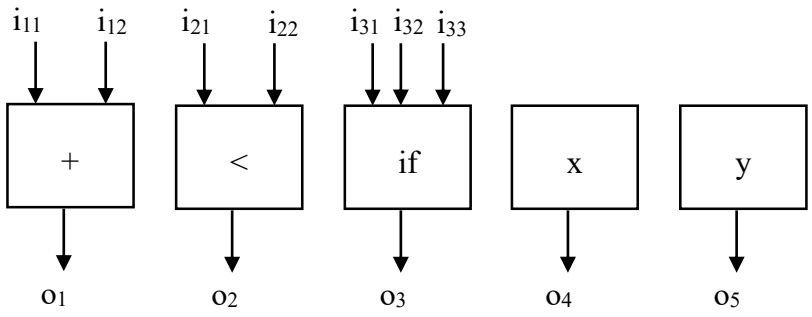

图 2.3 语法构件

并添加如下标签变量：

1）每个输入的标签变量：$L_{i11}, L_{i12}, \ldots$

2）每个输出的标签变量：$L_{o1}, L_{o2}, \ldots$

3）程序最终输出的标签变量 L_O

过程中产生如下几种约束：

1）规约约束

程序的输入输出要满足程序合成的规约。

$$O \geq x \ \wedge \ O \geq y \ \wedge (\ O = x \ \vee \ O = y \ )$$

2）语法构件的语义约束

用约束描述每个语法构件的语义，以加法为例：

$$o_1 = i_{11} + i_{12}$$

3）对输出标签赋予唯一的编号

$$L_{o1} \ \neq \ L_{o2}$$
$$L_{o1} \geq 1 \wedge \ L_{o1} \leq 6$$

4）输入输出标签之间的连接约束

$$L_{i11} = \ L_{o4} \rightarrow \ i_{11} = o_4$$

5）防环约束

每个构件输出的编号大于其所有输入的编号，保证最终程序无环：

$$L_{i11} < L_{o1} \wedge \ L_{i12} < L_{o1}$$

以上是基于构件的程序合成生成的连接逻辑约束，以及每种约束的一个例子。只要这一过程中产生的所有约束都可以被 SMT 求解器支持的理论覆盖，SMT 求解器就可以被用来求解程序合成问题。将求解出满足约束的一组连接映射回程序空间就得到了程序合成的解。

需要注意的是，一个语法构件往往会在一个程序中被使用多次，反映在连接逻辑中就是需要多个同一种语法构件，这时就需要将一种语法构件复制多次。实践中往往采用迭代的方法，先在每种语法构件只有一个的连接逻辑约束下求解，如果找不到解就尝试将所有语法组件复制一遍再求解，如此迭代直到求出满足所有约束的解。

另外，上面的连接逻辑中使用了全称量词的程序规约，而全称量词的求解过程往往是很慢的，因此这种方法实践中通常只用于规约为输入输出样例的情况。

假设规约为如下输入输出样例：

$$\max(1,2) = 2$$
$$\max(3,2) = 3$$

那么将生成如下规约约束：

$$x = 1 \wedge y = 2 \rightarrow 0 = 2$$
$$x = 3 \wedge y = 2 \rightarrow 0 = 3$$

也可以使用 CEGIS 方法（Counter-example guided inductive synthesis）将规约中的逻辑约束转化为输入输出样例，避免复杂的全称量词求解。

CEGIS 方法的大致思想是，利用约束求解给出的反例，将基于逻辑约束的规约转化为基于输入输出样例的规约。即使用约束求解器验证合成的程序的正确性，在程序不正确时给出反例，将一系列反例的集合作为程序的输入输出样例。CEGIS 方法简化了程序正确性的验证过程的计算。

2.1.4 基于空间表示的程序合成

空间表示法的思想是通过某种数据结构表示一系列程序的集合，即完整程序空间的一个子空间。空间表示法以程序的集合为单位进行操作，而非单个程序。

MaxFlash[19]是空间表示法的一个典型例子，MaxFlash 是目前字符串处理问题领域求解速度最快的求解器。MaxFlash 利用动态规划的思想，将整个程序合成问题划分为

若干个子问题，每个子问题的解对应一组程序的集合，并在计算过程中复用这些子问题的解。

考虑字符串处理程序 f(String1, String2)的例子，假设程序合成在如下语法上进行：

$$S \rightarrow Ns \mid Nz$$

$$Ns \rightarrow String1 \mid String2 \mid (+Ns\ \ Ns)$$

$$\mid CharAt\ \ Ns\ \ Nz \mid '.'$$

$$Nz \rightarrow 0 \mid 1$$

对于输入输出样例

$$f(\text{“John”}, \text{“Jonathan”}) = \text{“J. Jonathan”}$$

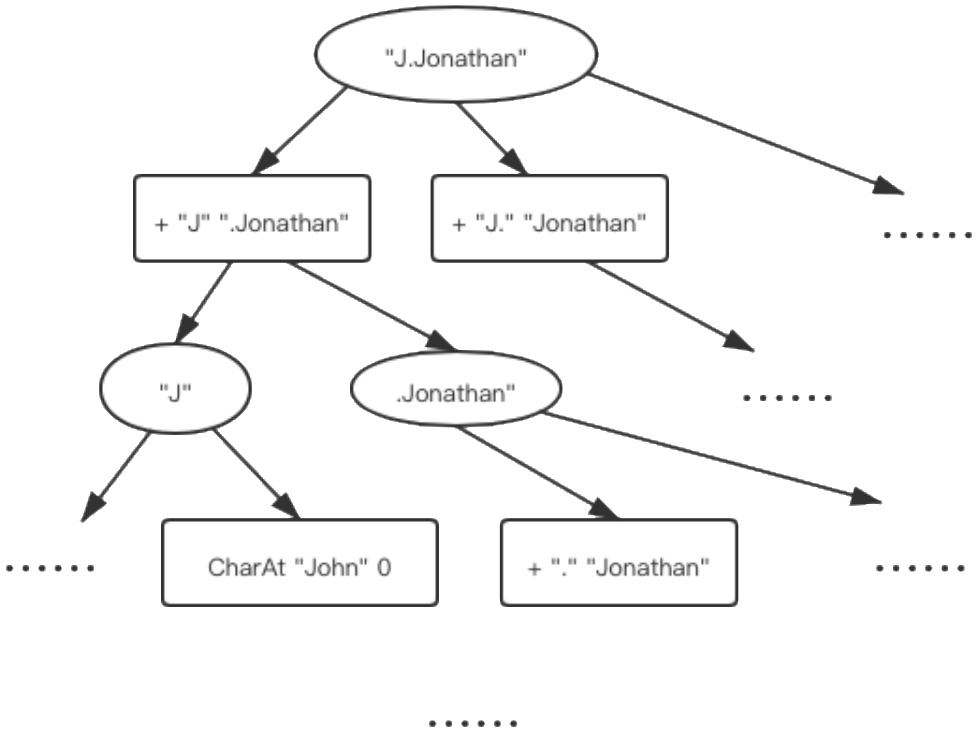

图 2.4 动态规划的搜索树

上述为动态规划算法的搜索树，具体来看，考虑输出为“J.Jonathan”，MaxFlash 将问题分解为

+ “J” “.Jonathan”

+ “J.” “Jonathan”

+ “J.J” “onathan”

+ “J.Jo” “nathan”

…

等若干个子问题组合。之后再分别求解“J”、“J.” 、“J.J” 、“J.Jo” 、“.Jonathan”、“Jonathan”等子问题。这样的子问题许多是无解的，最终得到一个可行的程序为：

$$(+\ (\text{CharAt}\ \text{String1}\ \ 0)\ (+\ '.'\ \text{String2}))$$

除此之外，MaxFlash 还使用了分支限界法，利用了程序的上下文语法信息和预训练的模型对子问题进行剪枝。

2.1.5 程序合成的应用

随着程序合成技术的日渐成熟，程序合成已经多个领域有了比较广泛的应用。目前，程序合成技术已经能够比较好地生成一些规模相对较小的程序，解决某些特定领域的问题。

1) 数据处理

数据处理是人们生产生活中经常遇到的需求，人们经常使用计算机完成数据处理任务，数据处理任务通常需要将原始数据处理成人们想要的格式。微软公司的产品 Excel 作为一款常用的办公软件，其中就实现了使用程序合成技术进行数据处理的功能。

新版 Excel 的快速填充（FlashFill）[20]功能是程序合成最为成功的应用之一。这一功能在 Excel 2016 版本中被添加，在表格中已有一些数据的情况下，用户只需要提供的一部分数据的手动处理结果作为样例，点击“数据”选项卡、 “数据工具”组中的“快速填充”按钮，Excel 就能直接给出其余数据处理之后的结果。这一功能背后的实现就采用了基于输入输出样例的程序合成技术，将用户提供的输入输出对作为规约，自动地生成一个程序用来处理其余的数据，直接给出处理完毕的数据。

图 2.5 FlashFill

上图是 FlashFill 功能的一个例子，截图自 2016 版 Excel。上图左侧是用户提供的数据和样例，右侧是自动快速填充的结果。可以看到，FlashFill 能够自动地提取出有用的信息，并处理成相同的格式。

Excel 中的 FlashFill 主要采用的是基于空间表示的程序合成，对于字符串处理等问题有着较为快速和准确的处理能力。在普通家用电脑上也能在毫秒级别完成程序合成过程并给出处理结果，这说明程序合成技术在实践中已经有了大规模应用的潜力。

2) 自动缺陷修复

自动缺陷修复也是近两年来软件工程领域的研究热点问题。自动缺陷修复接受一个有缺陷的程序和相应的测试集合作为输入，其中测试集合至少要包含一个失败测试，输出一个使程序通过所有测试的补丁[21]。自动缺陷修复问题也可以看做是一个程序合成问题。这里，程序的规约通常来自于程序的测试集，将要生成的补丁作为程序合成的目标程序。

S3[22]是使用程序合成进行缺陷修复的一个例子。S3 将缺陷程序的语法和语义转化成约束条件，使用约束求解的方法进行程序合成。

不过，由于现实中程序的测试集合往往是不完备的，使用程序合成的方法进行自动缺陷修复经常只能得到通过了所有测试，但在功能上不一定正确的补丁，这种补丁被称为似真补丁。现实中程序测试集不完备这一特点，限制了程序合成在自动缺陷修复领域的应用。

3) 测试生成

使用程序合成方法进行测试生成也是程序合成非常有意义的一个应用。

软件测试是软件开发过程中必不可少的一个环节，实际的软件生产过程中通常使用若干测试用例组成的测试集合对软件进行测试。编写测试用例和测试集合也是非常花费时间的，所以自动地生成测试也是软件工程领域非常关心的问题。

通常，测试生成方法都只能生成测试输入，而无法生成测试的断言。不过在某些情况下，测试断言也是可以自动生成的。例如在有正确的参考程序的情况下，可以用参考程序的输出作为测试的测试断言。

不过，仅仅是生成测试输入也是一个非常困难的问题。现代程序是高度结构化的，同时对输入的合法性有一定要求，生成符合被测试程序预期的、合法的结构化输入是测试生成的难点之一。

现代程序中的测试用例也是用一段程序来实现的，因此测试生成问题也可以被看作是一个程序合成问题。使用程序合成技术进行测试生成，可以将程序的输入的合法条件、结构要求等作为程序合成的规约，也可以额外添加其他的要求作为规约的一部分，例如测试用例的分支覆盖、语句覆盖信息等等。

2.2 相关工作

2.2.1 基于概率模型的程序合成加速

由于程序合成的瓶颈在于求解速度，已有的大部分程序合成求解器都会采用一些手段加速程序合成的求解过程。这类工作大多聚焦于在预先给定的 DSL 上，通过一系列模型给出程序的概率分布[25][26]，这类模型包括上文提到的 PHOG 模型、N-Gram 模型、神经网络模型等等。这类方法大多也依赖于训练程序，即从训练程序上学习程序的分布，已有的工作已经证明了，通过学习程序分布加速程序合成求解速度的方法是有效的，能够在一定程度上加速程序合成。

由于完整的程序空间非常大，难以直接刻画出空间内所有程序的分布情况，这类方法通常用一系列局部的概率分布去描述和刻画空间内所有程序的分布情况。这类思想来自于自然语言处理，自然语言中通常使用类似的方法刻画语言的局部概率分布信息。由于自然语言的空间同样非常大，且自然语言具有不确定性，这类刻画局部概率分布信息的方法在自然语言处理中被广泛应用。N-Gram 模型就是一个例子，N-Gram 模型的思想就是将语料中的内容以长度为 N 的滑动窗口分割为一个个长度为 N 的序列片段，每一个序列片段称为一个 gram，通过统计所有序列片段出现的频率，拟合出自然语言在局部的概率分布情况。

在编程语言中，类似的思想也可以被应用过来，同时，由于程序语言具有结构化的特点，一些新的模型被设计出来以利用这些结构信息。这其中较为简单的一个是概率

上下文无关文法，即 PCFG。概率上下文无关文法可以被认为是上文提到的概率高阶文法的简化版。PCFG 的思想是将上下文无关文法（CFG）的每条展开式赋以一个概率，即统计训练程序中从一个非终结符展开到其对应的每条展开式的频率，将其作为展开的概率，形成一个带概率的上下文无关文法。

$A \rightarrow \beta$		P
$S \rightarrow$	"."	**0.72**
$S \rightarrow$	"-"	**0.001**
$S \rightarrow$	x	0.12
$S \rightarrow$	$S + S$	0.02
...		

图 2.6 概率上下文无关文法

上图是概率上下文无关文法的一个简单例子，即非终结符 S 对应若干条展开式，以每条展开式在训练程序中出现的频率作为展开式的概率。在程序合成时，按展开式概率的大小为程序赋权，按权重从大到小的顺序遍历程序空间。

PHOG 与 PCFG 的区别在于，PHOG 在 PCFG 的基础上考虑了上下文信息，即考虑给定上下文条件下展开式被采用的概率。使用神经网络模型加速程序合成的思路也类似，只不过神经网络模型内部能够编码更多更复杂的上下文信息。神经网络预测模型每次输出的结果仍然是下一步语法展开或者下一个程序中的符号的概率或权重，这一概率或权重根据神经网络内部编码的上下文信息得到的。

这类方法大多集成在程序合成求解器上，这类方法被广泛采用，也从侧面说明了训练程序的语法结构信息是能够有效加速程序合成过程的。但另一方面，这些方法也存在一定的局限，通常，这类方法对程序语言的结构性特点利用有限。N-Gram 模型完全无法利用程序的结构信息，导致生成的程序容易违反程序的语法；PCFG 完全无法利用程序的上下文信息；PHOG 虽然利用了程序的上下文信息，但其能利用的信息十分有限，而且随着上下文的复杂程度增加，PHOG 的复杂程度也会大幅度增加，使其需要更多的训练数据和训练时间；神经网络模型虽然能利用比较多的上下文信息，但是其对于程序的语法结构信息利用非常有限，同时神经网络对于长程依赖的处理能力也不强，其模型的可解释性也不好。另外，这类方法可以和本文提出的方法同时使用，即在本文得到的新的领域特定语言上训练概率模型，用以进一步加速程序合成的求解过程。

2.2.2 挖掘程序可复用信息

已有一些工作致力于自动化挖掘程序中的复用信息，包括代码克隆检测[27]、API

调用序列学习[28]等等。

代码克隆检测目的是找出程序中的结构复用信息，即是否存在两段代码结构相似，其关注重点包括变量名的去特异化、程序结构的去特异化等，即判断不同的变量名、不同的程序结构（例如语句顺序、表达式组合方式等）是否可以被看作同一段程序的复用。本文方法中虽然也涉及程序结构复用信息的提取，但本文的训练用程序数量较大，且都是较为简单的程序，故本文方法不需要使用复杂的挖掘方法就能找到训练程序中的一些频繁模式。当然，与这些方法结合，也能进一步提高本文挖掘方法的挖掘能力，进而进一步加强本文方法的效果。

API 调用序列学习也是自动挖掘程序中的复用信息的一类研究，但是 API 调用序列学习不挖掘程序的结构信息，将程序中的 API 调用作为序列看待，之后可以再根据 API 的参数类型、返回值类型等重建程序的结构信息。

另外，也有一些工作研究如何自动学习和生成库函数。DreamCoder[29]自动学习特定领域内的库函数，并且也将得到的库函数应用到程序合成过程中，提高程序合成的求解能力。

这一系列工作证明了通过学习方法提取程序中的可复用结构是可行的。本文与这一系列工作主要的不同点在于，本文的方法着重于生成新的领域特定语言，而领域特定语言在库函数以外，还包括一些额外的语法限制，本文的实验证明，这些语法限制对加速程序合成的求解速度，即本文的生成 DSL 的优化目标是非常有用的。

第三章 问题分析与方法概述

本文的方法以一个已有的较为通用的编程语言为基础，设计出一个新的领域特定语言。本文中以程序合成求解器的在新的领域特定语言上的求解速度来衡量新语言的好坏，即设计新的领域特定语言加速程序合成的求解。本章将大致介绍方法的整体流程和框架，同时分析领域特定语言如何影响程序合成的求解速度。更为详细的方法的具体步骤和实现方案将在第四章详细介绍。另外，本章还将讨论本文方法的使用范围和限制。

3.1 流程概览

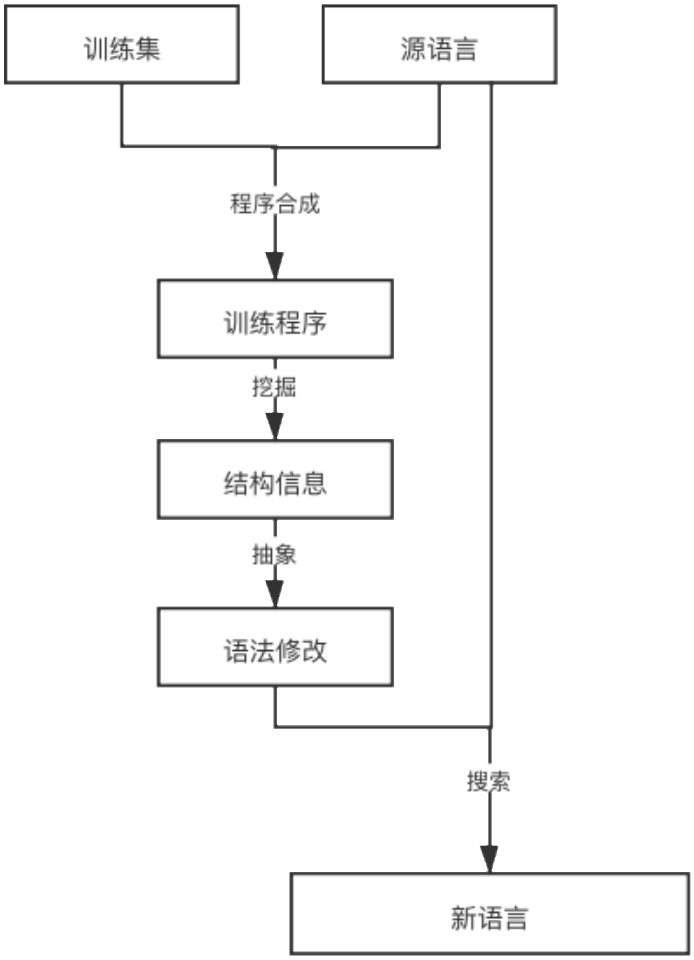

图 3.1 流程图

本文方法输入为一个源语言和一个训练集。训练集为某个特定领域内的若干程序合成问题，源语言是一个能够解决该领域问题的语言。本文方法的输出一个适用于该领域的新的领域特定语言。本文方法的目标是使得新语言能够加速程序合成在该领域问题上的求解。

本文的方法大体上分为四步：

1）在训练集上用已有的程序合成求解器求解出大量的源语言上的程序，称之为训练程序。

2）在训练程序上挖掘程序结构信息，包括训练程序中常见的模式、语法组件出现的频率等。

3）将上一步挖掘到的程序结构抽象成语法上的修改操作，这些操作主要包含添加语法组件、删除语法组件、对语法组件加以限制等。

4）利用搜索算法，找到一个由一系列修改操作组成的集合，使经过这些操作修改的新语法相较源语言更优。

具体而言，首先，本文的方法需要生成大量训练用程序。传统意义上来说，获取某个特定领域的大量程序是相对困难的。但是有了程序合成技术的帮助，这一问题变得容易了许多。只需要收集到若干某一领域的问题，就可以利用这些问题和程序合成技术生成领域内的程序，再在这些程序上进行有用信息的挖掘。这里本文利用已有的约束求解器，在某一个特定领域的若干个程序合成问题下，生成大量某一特定领域下的程序，用于后续的挖掘等步骤。

之后，本文方法利用挖掘方法从训练用程序上挖掘语法信息。本文在实现中采用了频繁模式挖掘等挖掘方法，通过频繁模式挖掘得到若干频繁模式，将这些频繁模式抽象出的语法组件加入新的语言中。同时还采集了语法的出现频率、展开深度等信息，用以调整语言的语法结构。

最后，本方法将挖掘到的程序结构信息抽象成语法上的修改操作，再利用搜索算法，根据这些修改操作自动地设计新的领域特定语言。

3.2 程序空间与程序合成求解速度

本节将从理论上分析同一个领域内的不同领域特定语言如何影响程序合成的求解速度。本文方法也是基于这一节的结论设计和实现的。

领域特定语言描述了一个程序空间，程序合成求解器在这一个程序空间内找到满足符合规约的程序。程序空间直接影响着程序合成求解器的求解速度，而领域特定语言通过定义出不同结构、大小的程序空间影响着程序合成的求解过程。

3.2.1 程序空间的定义

程序空间定义为，按照某语言的语法，从初始非终结符开始，经过若干步展开式的展开，得到的所有程序所组成的集合称为该语言定义的程序空间。本文方法得到的新语言定义的程序空间中的程序都能映射到源语言定义的程序空间中的一个程序，因此，我们称新语言为源语言的子语言。

3.2.2 程序空间的大小

新语言是源语言的子语言，即新语言所定义的程序空间中的所有程序都可以被映射到源语言定义的程序空间中的一个程序。但是相反的，源语言定义的程序空间中的程序不一定都能映射到新语言定义的程序空间中。源语言程序空间中的某些程序在新语言的程序空间中不存在，我们称新语言的程序空间小于源语言的程序空间。由此可知，源语言和源语言的所有子语言的程序空间大小构成偏序关系。

程序空间的大小直接影响程序合成的求解速度，这一点是显而易见的。通常情况下，程序空间越大，程序合成的求解速度就越慢。程序合成求解器在整个程序空间内寻找符合规约的程序，程序空间越大，程序合成求解器寻找到目标程序就越困难，需要的额外计算就越多。这些额外计算包括验证程序是否满足规约、程序空间的剪枝、程序的权重估计以及排序等等。

本文提出的方法主要通过删除语法中的某些不常用组件和限制组件的展开深度来缩小程序空间的大小。

缩小程序空间的大小减弱了语言的表达能力。这里需要注意的是，只有在最优解没有被删除的情况下，程序合成的求解时间和程序空间的大小才是严格正相关的。这里的最优解定义为程序合成求解器给出的第一个解。如果最优解在新语言的程序空间中被删除，程序合成求解器需要更多的额外计算才能找到次优解，最坏情况下甚至无法在新的程序空间中找到可行解。

3.2.3 程序空间的结构

程序空间的结构也影响程序合成的求解速度。不同于程序空间大小，其对求解速度的影响较为直观，而程序空间结构对程序合成求解速度的影响是复杂的。本节中将用例子来说明这一点。

回顾第一章提到的字符串替换操作的例子，这里使用一个简化的版本。考虑如下简单的字符串处理问题的例子：

该问题的规约由以下输入输出样例给定：

f("Launa-Withers") = "Launa Withers"

f("Lakenya-Edison") = "Lakenya Edison"

f("Brendan-Hage") = "Brendan Hage"

f("Harry-Potter") = " Harry Potter"

f("Michael-Neipper") = " Michael Neipper"

f("Robert-Zare") = " Robert Zare "

f("Kim-Shane") = " Kim Shane "

f("Amanda-Pinto") = " Amanda Pinto "

f("Steven-Thorpe") = " Steven Thorpe "

程序 f 将接收的字符串中的连字符"-"替换为空格。使用以下简单的 DSL 求解该问题。

表 3.1 简单字符串处理 DSL

(str.++ String String)	字符串连接
(str.substr String Int Int)	根据下标取子串
(str.len String)	字符串长度
(str.indexof String String)	字符串检索，返回子串在母串中第一次出现的下标
(+ Int Int)	整数加法

一个可行解为：

```
f(Param)=
  (str.++ (str.substr Param 0 (str.indexof String "-"))
      (str.++ " " (str.substr Param (+ (str.indexof Param "-") 1) (str.len Param))))
```

这个解在整个程序空间中的位置如下图中的箭头所示。

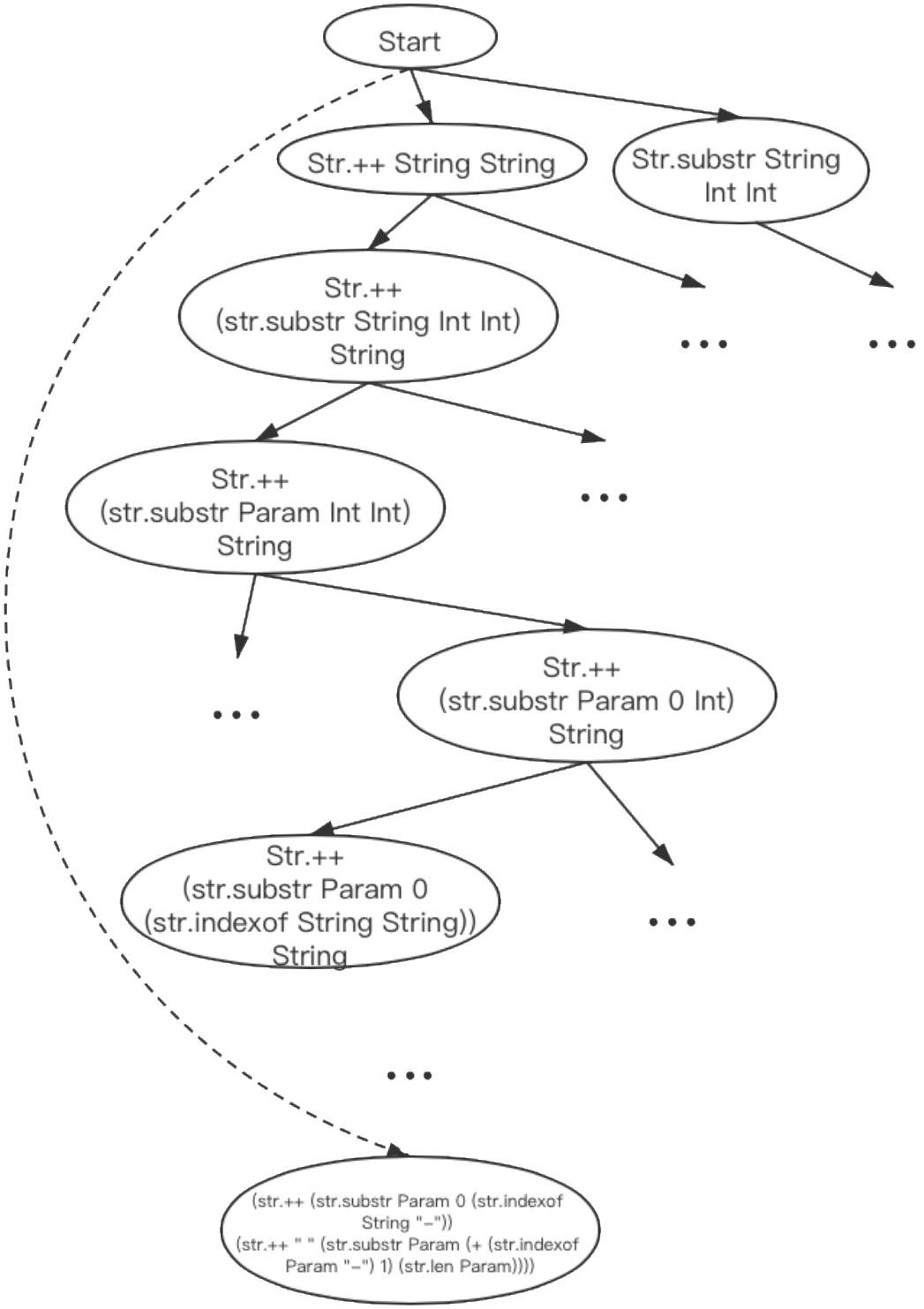

图 3.2 解在程序空间中的位置

从程序空间的起点开始，要经过极长的搜索路径才能到达，而在这条搜索路径上还有大量的无效分支。

相应的，若在程序语法中加入 str.replace 函数，那么 str.replace 函数可以将这条漫长的搜索路径缩短为一步或少数几步搜索过程，大大降低了搜索的深度和无效分支的

数量。加入 str.replace 函数在程序空间上可以理解为，在程序空间的某些可行位置添加了一个子节点，这个点是源语言程序空间上一条漫长的搜索路径在新程序空间上的映射，添加这样的点能够大幅节省某些程序合成问题求解时的计算，通过这样的点快速找到程序空间内的可行解。

需要注意，添加这样的点也会带来副作用。在求解某些问题的时候，这样的点会增大搜索路径上的无效分支的数量，虽然添加的点仍然能够映射回原有的程序空间，不会增加整个程序空间的规模。如上图所示，添加虚线前，初始符号 start 只有两种展开方式：str.++和 str.substr，但是虚线表示的 str.replace 的加入使得 start 符号多了一种展开方式，这种展开方式在一定比例的问题上都是无效分支。

形象地理解为，加入这样的点之后，程序空间变得更加“扁平”，即源语法程序空间中某些点的深度被大大降低了，这样的修改可能加速一部分程序的求解，也可能不利于一部分程序的求解。

3.3 语法修改操作

本文方法通过在源语言上应用若干语法修改操作得到一个新语法，使得新语法相较源语法更优，即程序合成在新语法上求解速度更快。

3.3.1 语法修改操作的定义

根据上一节得到的结论，程序空间的大小和结构影响程序合成的求解速度。本文提出的方法定义了三种语法上的修改操作，这三种修改操作目的是缩小程序空间的大小和优化程序空间的结构，这三种修改操作分别为：

1) 添加语法组件

为领域特定语言的语法添加一个新的组件。在本文中新添加的语法组件都是由若干源语言中的语法组件组成的，即添加的语法组件可以由源语言中的语法组件所表示，不引入新的语义和功能。添加语法组件会改变程序空间的结构，以上文中的字符串替换函数为例，添加进这样的函数能够缩短程序合成的搜索过程，有望能够加速程序合成的求解。

2) 删除语法组件

源语言内的部分语法组件在求解目标领域内的问题时可能不会被用到，或者很少被使用到，或者能够被别的组件替代，也就是说，即使不适用这部分组件，目标领域内的问题或大部分问题仍然能够被求解。删除这部分组件能够缩小程序空间，一定程度上也能够加速程序合成的求解。

例如，在求解字符串处理领域的问题是，条件判断语句和布尔表达式两种语法组件几乎不会出现。这在一定程度上说明了字符串处理领域的问题大多不需要使用条件判断语句和布尔表达式，即使部分问题的某些解程序会使用到条件语句和布尔表达式，往往也存在着不使用这些语法组件的程序可以求解该问题。因此在新语言的语法中尝试删掉这些语法组件，就有可能可以加速程序合成的求解。另外，源语言中的部分语法组件还有可能可以被新语法中添加的语法组件所替代，这些可以被替代的语法组件也可以在新语言中被删除。

注意，删掉语法组件会影响语法的表达能力，在删除语法组件的时候需要保证删除后语法仍然有足够的表达能力。考虑到本文方法是为某个领域设计领域特定语言，我们只需要考虑新语法在该领域内的表达能力。

3) 限制语法展开深度

进行深度限制这一操作主要是因为，在解决实际问题的程序中往往不会出现大量重复的递归和嵌套结构，而程序合成的求解器往往无法利用这一特点，程序合成求解器在求解过程中通常无法避免类似的无意义结构。另外，目前的程序合成求解器的求解能力也都比较有限，对于语法结构过于复杂的程序的求解速度很慢。限制语法深度也是避免了程序合成求解器在这类程序上耗费太多时间。总之，限制语法展开深度也是为了缩小程序空间，进而加速程序合成的求解。

通过以上三种修改操作，可以改变源语法的程序空间的大小、结构，通过若干修改操作的组合，得到一个新的语法，使新语法的程序空间更利于程序合成在目标领域问题上的求解。

3.3.2 语法修改操作的挖掘

本节讨论在定义了三类修改操作之后，如何获得具体的修改操作。直接使用搜索方法在每类修改操作定义的空间中搜索合适的修改操作是不现实的，尤其是针对添加组件操作。本文使用挖掘方法获得具体的修改操作。针对每一种修改操作，使用不同的方法，在训练用程序上挖掘出具体的修改方式。

这一步骤旨在获取若干修改操作，具体是否应用这些修改操作、采用哪些修改操作的组合由之后的搜索过程决定。

1) 添加组件操作

训练程序中往往会有许多频繁出现的模式，这些模式往往能够抽象成一个新的语法组件。在新的语言中加入这些语法组件，有可能改变程序空间的结构，达到加速程序合成求解速度的目的。

需要注意的是，本文的训练用程序是用程序合成求解器得到的较为简单的程序，

故不需要考虑变量名的抽象和统一等一系列较为复杂的问题。本文使用已有工作普遍采用的频繁子树挖掘算法挖掘频繁出现的模式。频繁子树挖掘算法是一种挖掘树结构中频繁模式的通用算法，通过与频繁模式挖掘类似的方法，由小到大地生成若干频繁出现的子树。这一步可以得到若干频繁模式，将这些频繁模式抽象成若干语法组件，这些语法组件将作为添加组件操作的备选，这一步的具体的实现方式将在第四章详细介绍。

2) 删除组件操作

某些语法组件在训练程序中不出现或出现频率非常低，这说明这些语法组件往往与这一领域不相关或者关系不大。本文中通过统计语法组件在训练程序中出现的频次决定哪些语法组件作为删除操作的备选。

3) 限制深度操作

本文方法统计非终结符在训练程序上的展开深度的 80%分位数，作为其在新语言上展开最大深度的限制。即要求训练程序中 80%情况下该终结符的展开深度都在限制范围内。

3.3.3 语法修改操作的组合

这一小节讨论如何找到最优的修改操作的组合。

这里需要注意到的是，上一小节讨论的方法挖掘到的语法修改操作并不一定都是正收益的。例如 3.2 节提到的，添加语法组件有可能会让程序合成求解器的无效搜索分支空间增多，程序合成求解器在求解过程中的需要进行的验证、剪枝等无效计算也更多，反而可能使程序合成减速。另外，删除组件和限制深度操作都会使程序空间减小，即使得一部分解被排除在新的程序空间之外。这就存在一种可能，即某问题的最优解在新的程序空间中被删除了，程序合成求解器在新的程序空间中只能找到原来的次优解，也会使得程序合成减速。另一方面，一些操作的组合可能会获得额外的增益效果，例如将字符串中取子串的操作替换为取前缀和取后缀的操作可以大幅提高程序合成的速度。这是因为，字符串处理问题中大部分的取子串操作都是在取前缀和取后缀，少部分取中间子串的操作又可以通过先去前缀再取后缀的方法组合得到。将取子串的替换为取前缀和取后缀的过程包含三个上文定义的修改操作：两个添加组件操作和一个删除组件操作，这说明多个操作的组合可能可以获得更好的增益效果。

本文中使用遗传算法自动搜索最优的修改操作组合。这里，遗传算法需要一个适应度函数，计算个体的适应度，在本文中即经过若干修改操作得到的新语言的加速效果。理论上可以通过直接测量求解器的求解时间来计算，但遗传算法搜索过程中需要计算大量样本的适应度函数，使得这一过程过于复杂和耗时，因此，本文使用程序空间估计

技术估计新语言的加速效果。

3.3.4 程序空间估计技术

程序空间估计技术通过语法修改操作对程序空间的影响估计程序合成在新语言上的加速效果。本文提出的方法考虑了程序空间的以下三个方面的性质对程序合成求解速度的影响：

1）程序空间的大小

领域特定语言的语法定义直接影响程序空间大小，这一点是较为直接的。生成新语言过程中进行的添加组件、删除组件、限制深度的操作都直接影响程序空间的大小。在程序空间结构不发生变化的情况下，程序空间越大，程序合成求解速度越慢；相反地，程序空间越小，程序合成求解速度就越快。添加组件会增大程序空间，删除组件、限制语法展开深度会减小程序空间。

2）程序空间的结构

程序空间的结构对程序合成求解速度的影响更大也更为显著。更优的程序空间结构能够让程序合成求解器更快地找到程序合成问题的可行解。在本文中使用了用求解器找到的第一个解在程序空间中的排序来估计程序空间结构对程序合成求解时间的影响。若程序空间中的程序是有权重的，则以权重进行排序；若程序空间中的程序是无权重的，那么简单的以大小进行排序。这一排序越大，程序空间的结构就相对更劣，即程序合成需要花费更多的时间排除若干不可行解；这一排序越小，程序空间的结构就相对更优，即程序合成过程中更容易找到可行解。

3）程序空间中是否包含可行解

本文方法中包含删除语法组件、限制深度等对程序空间进行剪枝的操作，这些操作会删除程序空间中的部分程序，因此，为保证语言的表达能力，需要保证剪枝后的程序空间中仍然有程序合成问题的可行解。

根据程序空间的这些性质，设计估计函数对新语言的加速效果进行估计。具体的估计函数和实现在第四章中详细介绍

3.4 方法的使用范围与限制

3.4.1 新语言的表达能力

本文方法输出一个新语言的语法，新语言是在源语言语法上经过若干修改得到的。在本文方法中，所有语法修改操作均不会为语言增加新的语义。即，新语言所能描述的程序都能够被源语言描述。

本文方法对源语言的修改包含删除源语言中的部分语法和对源语言的部分语法进行限制。因此，新语言的表达能力是源语言的子集。即可能出现部分程序能够被源语言表达而不能被新语言表达。不过，本文方法的目标是生成一个针对特定领域的领域特定语言，新语言的通用性并不是考虑的重点，只需要考虑新语言在我们关心的特定领域的表达能力。

本文的目标是在保证语言在特定领域的表达能力的情况下，加速程序合成在该领域的求解速度。

3.4.2 源语言的选取

源语言应该是一种较为通用的基础语言，拥有足够的表达能力。新语言的表达能力是源语言的子集，若源语言表达能力不够则会直接影响新语言的表达能力。因此，源语言的选取是有一定要求的。

首先，源语言应保证能够求解目标领域的绝大部分问题。否则，源语言无法求解的问题在新语言上一定也不能求解，这将大大限制新语言在目标领域的表达能力的上限。源语言应尽可能保证表达能力。

其次，受到程序合成技术的限制，源语言不能过于复杂。本文方法依赖从大量训练程序中挖掘到的结构信息生成新语言。目前程序合成技术在求解速度和求解能力上仍然有较大限制，很难在过于复杂的源语言上进行求解。因此，本文的方法选用了一个较为通用的领域特定语言作为源语言。

3.4.3 目标领域的设定

本文的方法针对特定领域的问题设计新的领域特定语言。那么，具体研究什么样的领域也是需要讨论的问题之一。领域的范围可大可小，例如，可以认为所有与数组相关的计算问题是一个领域，也可以认为数组中的搜索问题是一个特定的领域。

本文方法的目标领域应该设定为一个相对较小的领域。领域的范围越大，就要求针对该领域的领域特定语言的通用性更高；领域的范围越小，针对该领域的领域特定语言专用性就可以更强。

本文方法旨在利用领域内的信息，针对该领域设定专用性更强的领域特定语言，因此，选取的目标领域范围不应过大。若选取的目标领域范围过大，会要求能够求解该领域问题的领域特定语言仍然是一种通用较高的语言，为这样的领域设计新的领域特定语言意义不大。

另外，本文的方法还要求领域内的问题有一定相似性，解决该领域问题的程序有一定的可复用性，这样可以提高领域特定语言的抽象能力，使新语言能够以更高的抽象

层次、更加精简地描述该领域内的程序。

我们预期新语言应该在该领域应该有更好的专用性和更高的抽象能力，因而使得程序合成能够在该语言上更快地求解。

第四章　领域特定语言调整方法

本文实现了在字符串处理问题上的生成新的领域特定语言的方法。本章将具体描述方法的完整过程。

本文方法分为以下几步：

1）生成训练程序。在训练集上用已有的程序合成求解器求解出大量的源语言上的训练程序。本文中使用基于空间表示的程序合成求解器 MaxFlash 生成字符串处理领域的训练程序。本文所使用的源语言是一个用于求解字符串处理问题的、用上下文无关文法表达的、强类型的领域特定语言。

2）在训练程序上挖掘程序结构信息。包括训练程序中常见的模式、语法组件出现的频率等。本文使用了频繁子树挖掘的方法挖掘训练程序中的常见模式。

3）将上一步挖掘到的程序结构抽象成语法上的修改操作，这些操作主要包含添加语法组件、删除语法组件、对语法组件加以展开深度限制等。

4）利用搜索算法，找到一个由一系列修改操作组成的集合，使经过这些操作修改的新语法相较源语言更优。本文方法的搜索过程分为两部分，首先使用估计函数和遗传算法粗粒度地筛选出新语言的若干候选，再用直接测量求解时间的方法在候选语言中选出最优的一个。

本章将逐一介绍这些步骤的实现方案。

4.1　训练程序的生成

利用已有的约束求解器，在某一个特定领域的若干个程序合成问题下，生成大量某一特定领域下的程序，用于后续的挖掘等步骤。

需要注意的是，本文的方法从一个较为通用的源语言出发，在这一步的生成过程中，生成的程序也是基于源语言的。虽然理论上讲这里的源语言可以是任何能够解决某一领域问题的编程语言，但是在实践中，由于目前的程序合成技术的限制，基于通用编程语言的程序合成非常困难。本文的方法中采用较为了通用的领域特定语言作为源语言，而没有使用传统的通用编程语言。本文的实现中使用了一种强类型的上下文无关文法表达的语言作为源语言。

本文实现中使用的源语言包含基础的字符串运算、整数运算和正则表达式匹配等语法，源语法的具体细节见 5.2 节的内容。

这一步的目的主要是为了生成某一领域的大量程序用以挖掘领域内程序的结构特点。传统意义上来说，获取某个特定领域的大量程序是困难的。有了程序合成技术的帮

助，这一问题变得容易了许多。只需要收集到若干某一领域的问题，就可以利用这些问题和程序合成技术生成领域内的程序，再在这些程序上进行有用信息的挖掘。

需要注意的是，程序合成对于一个程序合成问题往往能够给出多个解，即有多个程序都能满足程序合成问题中的规约。

这些不同的程序对于挖掘程序结构信息都是有意义的。例如，考虑这样的程序合成问题，该问题的规约由如下输入输出样例给出：

f("938-242-504") = "242"
f("308-916-545") = "916"
f("623-599-749") = "599"
f("981-424-843") = "424"

程序 f 接受一个字符串，返回其中第 5-8 位组成的子串。假设程序合成在只有 str.substr 函数的领域特定语言上求解，能够实现这一功能程序可以有以下两种：

f(s) = (substr s 4 7)
f(s) = (substr (substr s 0 7) 4 -1)

其中第一个为程序合成的最优解，但是第二个程序对于挖掘程序结构信息也是有意义的。在第二个程序中，存在着取前缀和取后缀的模式，取前缀和取后缀在字符串处理领域可能是一个常见的操作。

因此，本文的方法中会考虑利用程序合成求解器使其输出多个可行解。对于这种情况，基于空间表示的程序合成求解器是最合适的。

基于枚举的求解器在枚举到第一个结果之后虽然也可以继续枚举和搜索直到找到下一个可行解，但是这一过程耗时较长。

基于约束求解的程序合成求解器也并不适合这类问题，这主要受限于现有的约束求解器。现有的约束求解器往往只能给出解的一组赋值而不是多个赋值或全部赋值，若想使用约束求解器求解出多组解往往只能额外添加排斥约束，即添加约束使约束求解器要求的新的解不等于已有解。虽然部分约束求解器在求新的解的过程中能够复用一部分之前的求解过程的中间计算，但是这也过程也是非常耗时的。

而基于空间表示的程序合成求解器则非常适合求解这个问题。空间表示法对程序集合进行操作，其中，基于 VSA[23]（变体空间代数，Version Space Algebra）的方法理论上可以以较小的额外开销求出程序的所有可行解。

变体空间代数法的思想是使用带约束的上下文无关文法表示程序的集合，对于每个样例产生一个上下文无关文法。

考虑在如下文法上的合成程序 f(x,y,z)例子，该文法只支持字符串拼接：

$$S \rightarrow S + S \mid x \mid y \mid z$$

对于如下输入输出样例：

f("a","cc","c") = "acc"

得到一个如下图所示的文法，表示出能满足该样例的所有程序的集合。

[acc]S→[a]S+[cc]S | [ac]S+[c]S	①
[ac]S→[a]S+[c]S	②
[cc]S→[c]S+[c]S | y	③
[a]S→x	④
[c]S→z	⑤

图 4.1 带约束的文法表示的程序空间 1

上图图中[acc]S 表示取值为 acc 的非终结符 S。展开式①表示[acc]S 有两种可行的展开方法，一种为[a]S+[cc]S，即取值为 a 的非终结符 S 与取值为 cc 的非终结符 S 连接；另一种为[ac]S+[c]S，即即取值为 ac 的非终结符 S 与取值为 c 的非终结符 S 连接。

这样的文法能够描述出对于这个输入输出样例所有的可行解。具体到上述例子，得到的可行解有 x+y 和 x+z+z 两个。

假设该问题还有另一个输入输出样例：

f("a","ac","c") = "aac"

那么对于这一个样例，可以得到如下图的文法描述满足该样例的程序集合：

[aac]S→[a]S+[ac]S | [aa]S+[c]S	①
[ac]S→[a]S+[c]S | y	②
[aa]S→[a]S+[a]S	③
[a]S→x	④
[c]S→z	⑤

图 4.2 带约束的文法表示的程序空间 2

上图的文法对应的可行解有 x+x+z 和 x+y 两种。

那么对以上两个文法定义求交运算，得到的结果应该只包含二者的公共部分，即 x+y 一个程序。

求交的运算可以被定义在 VSA 上，VSA 对求交是封闭的，即 VSA 求交之后一定还是 VSA。VSA 用有向图来表示一系列程序的集合，该有向图中只包含三种节点，VSA 中的每个节点都对应这一个程序集合。

1）叶节点

VSA 中的叶节点直接对应一个由一系列具体的程序组成的集合

2）并节点

VSA 中的并节点对应由它的所有子节点表示的集合合并起来的集合

3）交节点

VSA 中的交节点对应 DSL 中的一个函数 F，F 接收 k 个参数。该交节点有 k 个子节点，每个子节点对应 F 的一个参数。

VSA 也可以用类似上图中的文法的形式来表示，该文法是上下文无关文法的一个子集，该文法中只包含三种展开式：

1）由非终结符展开称为若干终结符，对应有向图中的叶节点。

$$S \rightarrow p_1 \mid p_2 \mid p_3$$

2）由非终结符展开称为若干非终结符，对应有向图中的并节点。

$$S \rightarrow N_1 \mid N_2 \mid N_3$$

3）由非终结符展开为一个接受 k 个参数的函数，对应有向图中的交节点

$$S \rightarrow F(N_1, N_2, N_3 \ldots N_k)$$

且每个非终结符只能在左边出现一次。

按如下规则定义 VSA 求交运算：

$$[\cup(N_1,\dots,N_k)]\cap N' = \cup(N_1\cap N',\dots,N_k\cap N')$$
$$F(N_1,\dots,N_k)\cap F'(N'_1,\dots,N'_k) = \emptyset$$
$$F(N_1,\dots,N_k)\cap F(N'_1,\dots,N'_k) = F(N_1\cap N'_1,\dots,N_k\cap N'_k)$$
$$F(N_1,\dots,N_k)\cap\{P_1,\dots,P_m\} = \{P_j \mid \exists P_1\in N_1,\dots,P_k\in N_k \text{ 使得 } P_j = F(P_1,\dots,P_k)\}$$
当N_1, N_2均为叶节点时，$N_1\cap N_2$定义为集合交

以上式子中，N 表示一个 VSA 中的节点，即一个程序集合。$\cup(N_1,\dots,N_k)$表示一个并节点，$F(N_1,\dots,N_k)$表示一个交节点，$\{P_1,\dots,P_m\}$表示一个叶子节点。

先对每个输入输出样例求出 VSA，再在所有输入输出样例的 VSA 上求交，得到的 VSA 即可以表示该问题的所有可行解的集合。

在本文的实现中，对 MaxFlash 求解器进行了修改，利用 VSA 算法，使其为每个问题求出最多 K_g 个可行解，将得到的所有可行解作为训练程序集合。

4.2 程序结构信息与语法修改操作

本方法将挖掘到的程序结构信息抽象成语法上的修改操作，根据这些修改操作自动地设计新的领域特定语言，那么如何挖掘有用的结构信息将是本方法的一个重点。本文中主要考虑根据训练程序中的以下几种特征抽象出语法上的修改操作：

1）根据频繁出现的模式添加新的组件

训练程序中往往会有许多频繁出现的模式，这些模式往往能够抽象成一个新的语法组件。在新的语言中加入这些语法组件，有可能改变程序空间的结构，达到加速程序合成求解速度的目的。

例如，可以在字符串处理领域的训练程序中可以找到诸如这样的模式：

MatchRegex(“[A-Z][a-z]*”, Str)

MatchRegex 函数是正则表达式匹配函数，上面这一个表达式的含义是匹配并返回字符串 Str 中第一个以大写字母开头，紧跟着若干个小写字母的子串。实际上上述例子中“以大写字母开头，紧跟着若干个小写字母”这样的字符串在实践中往往有着特殊的含义，例如用于人名、地名等等。

本文使用频繁子树挖掘算法挖掘频繁出现的模式。

需要注意的是，这里挖掘出的频繁模式一定是一颗子树，所以频繁模式挖掘一定是从一颗子树的根节点开始挖掘，向其子节点中寻找频繁模式。

采用迭代的方法，首先遍历训练程序集合，找到所有大小为 2，即包含两个节点

的频繁子树，再在已经找到的大小为 2 的频繁子树的基础上找出所有大小为 3 的频繁子树。这里本文的方法设定一个参数 K_f，出现次数大于 K_f 的子树被认为是频繁的。

由于本文的实现中的源语法采用了强类型的上下文无关文法语言，所有频繁子树中未被匹配的节点一定可以被一个非终结符表示。

那么每一个得到的频繁子树都可以用一个上下文无关文法的展开式表达。这些频繁出现的模式可以作为某个固定的语法组件加入新的领域特定语言中，把它当做新语法中的某个函数或者某种表达式。

这一步可以得到若干频繁模式，这些频繁模式抽象出的语法组件可以被加入新的语言中。

2）根据语法组件出现频率删除不常出现的组件

某些语法组件在训练程序中不出现或出现频率非常低，这说明这些语法组件往往与这一领域不相关或者关系不大。例如，在字符串处理领域的问题中，if 条件语句和布尔表达式几乎不会出现，说明字符串处理领域的问题大多不需要使用 if 条件语句和布尔表达式，即使部分问题的某些解程序会使用到条件语句和布尔表达式，但往往也存在不使用这些语法组件的解程序。在新语言的语法中就可以尝试删掉这些语法组件。

注意，删掉语法组件会影响语法的表达能力，在删除语法组件的时候需要保证删除后语法仍然有足够的表达能力。考虑到本文方法是为某个领域设计领域特定语言，我们只需要考虑新语法在该领域内的表达能力。

另外，上文中提到，新语法中会加入一些由源语言中常见模式组成的新的语法组件，在加入了这些新的语法组件后，源语言中的部分语法组件可能可以被新语法中的一个或多个语法组件所替代，这些可以被替代的语法组件也可以在新语言中被删除。

3）限制某些语法组件的展开深度

统计非终结符在训练程序上的展开深度的 80%分位数，作为该非终结符在新语法中的展开最大深度，从而对程序空间的大小进行限制。对每个非终结符的深度限制是固定的，是否对其进行限制取决于搜索算法得到的结果。

进行这一深度限制主要是因为在程序中往往不会出现大量重复的递归嵌套，而程序合成的求解器经常会出现类似的无意义结构。另一方面，目前的程序合成求解器的求解能力也都比较有限，对于语法结构过于复杂的程序的求解速度很慢。限制语法深度也是避免了程序合成求解器在这类程序上耗费太多时间。其次，由于添加的语法组件的存在，程序空间相对变得更“扁平”，限制语法的展开深度也有助于

求解器优先使用新添加的语法组件进行程序合成。

4.3 生成新的领域特定语言

在源语言上的语法进行若干修改操作，得到新的领域特定语言。新的领域特定语言是源语言的一个子集。这些修改操作都是基于本方法在前一步中挖掘到的信息得来的。主要有如下几种：

1）添加新的语法组件，将上一步挖掘到的所有频繁模式作为语法组件加入新的语言的语法中。

2）删除不常用的语法组件

3）为非终结符添加限制属性，在本文中的实现中主要是非终结符的展开深度。

本文方法的目标是经过若干次修改操作后，得到一个新的语言，使得程序合成在新语言上的求解速度更快。不过，这些操作本身并不一定都是正收益的。

例如，上文中提到的，添加语法组件会让求解器的无效搜索分支空间变得更多，程序合成求解器在求解过程中的进行的无效计算也更多，反而可能使程序合成减速。

又例如，删除某个不常用的语法组件，但是该语法组件是某一个问题最优解的组件之一，删除该语法组件使得该问题的最优解从程序空间中也被删除了，程序合成求解器只能找到原来的次优解，也使得程序合成减速了。

但是，某些操作的组合可能能大幅加速程序合成的速度，例如在字符串处理问题中，删除取子串的函数：

$$\text{str.substr(start, end)}$$

并同时加入前缀和后缀函数：

$$\text{str.prefix(end)} = \text{str.substr(0, end)}$$
$$\text{str.suffix(0)} = \text{str.substr(start, -1)}$$

可能可以大幅加速程序合成的过程。其中的原因在于，绝大部分字符串取子串的操作都是取前缀或者后缀的操作，而其中一少部分取中间子串的又可以被先取前缀再取后缀的操作组合代替。使用前缀函数和后缀函数代替取子串函数在绝大部分情况下减少了程序合成求解器要求解的参数，能够有效加速程序合成的求解过程。

4.4 启发式搜索

本文使用了遗传算法搜索出较优的操作组合。

遗传算法[24]是一种较为通用的启发式搜索算法。遗传算法利用模仿遗传学的思路进行搜索，其大致过程是：

1）先将搜索空间的参数编码为一个“基因”，每个“基因”对应搜索空间中的一组解。“基因”通常有若干位，每一位对应着搜索空间中的一个维度。

2）初始化种群，通常随机生成若干个个体，每个个体对应一个“基因”，即搜索空间中的一组解。

3）每次迭代，对种群中的个体进行交叉和变异，将新的个体加入种群。

交叉：将两个个体的部分“基因”替换重组，得到新的个体

变异：将某一个个体的部分“基因”值更改，得到新的个体

4）得到新的种群后，按适应度进行选择，选择出适应度最高的前若干个个体，淘汰其他个体，再进行下一次迭代，直到达到终止条件。

使用遗传算法搜索操作组合，进行“基因”编码的方式是非常直观的，即将所有操作集合中的每一种操作编码为一位，该位为 1 表示选取这种操作，为 0 表示不选取这种操作。

但是，使用遗传算法需要进行选择操作，即对某一个个体——即某些操作的集合评估适应度。在这里，遗传算法的优化目标是程序合成求解器在新语言上的求解速度。最直接的方式是直接测量，即新测量程序合成求解器在训练集上使用新语言进行求解的时间。但这种方法实际上是不可行的，实际搜索过程中搜索空间很大，且程序合成求解器的求解时间也比较长，而且这一时间将会随着源语言的语法复杂度的增长指数级地增长。所以，本文采取了估计的方法评估个体的适应度，即程序合成求解器在某个语法上的求解速度。

4.5 程序空间估计

估计程序合成求解器的求解时间是一个比较复杂的问题，本文使用了程序空间估计技术估计程序合成求解器的求解时间。本文提出并实现了程序空间估计技术，这一技术能相对准确地估计程序合成求解器的求解时间。

在本文的方法中使用的估计函数接受若干个参数，给出新语法相对源语法在程序合成求解器上的加速系数的估计。本文的方法中分别从程序空间的大小、程序空间的结构、程序空间中是否包含可行解三个方面估计程序合成在新语言上的加速效果，具体如下：

1）程序空间的大小

领域特定语言的语法直接影响这程序空间大小，这一点是较为直接的。生成新语言过程中进行的添加组件、删除组件、限制深度的操作都直接影响程序空间的大小。添加组件会增大程序空间，删除组件、限制语法展开深度会减小程序空间。在程序空间结构不发生变化的情况下，程序空间越大，程序合成求解速度越慢；相反地，程序空间越小，程序合成求解速度就越快。因此，添加组件的个数越多，这一部分的加速效果越小；删除组件的个数越多，这一部分的加速效果越大；限制展开深度的组件越多，这一部分的加速效果越大。

2）程序空间的结构

程序空间的结构对程序合成求解速度的影响更大也更为显著。更优的程序空间结构能够让程序合成求解器更快地找到程序合成问题的可行解。在本文中使用了用求解器找到的第一个解在程序空间中的排序来估计程序空间结构对程序合成求解时间的影响。若程序空间中的程序是有权重的，则以权重进行排序；若程序空间中的程序是无权重的，那么简单的以大小进行排序。这一排序越大，程序空间的结构就相对更劣，即程序合成需要花费更多的时间排除若干不可行解；这一排序越小，程序空间的结构就相对更优，即程序合成过程中更容易找到可行解。

3）程序空间中是否包含可行解

本文方法中包含删除语法组件、限制深度等对程序空间进行剪枝的操作，这些操作会删除程序空间中的部分程序，因此，为保证语言的表达能力，需要保证剪枝后的程序空间中仍然有程序合成问题的可行解。在本文的实现中，将这一项作为惩罚项来使用，即将这一项作为适应度函数的系数，只有 0 和 1 两个取值，当某一个个体的这一项的要求被违反时，直接将系数设为 0，遗传算法就会淘汰掉这一个个体。

另外，本文对这一项进行了近似处理。上文提到的基于 VSA 的程序空间表示法理论上能够表示出程序的完整解空间，即所有可行解。但实际上求出完整解空间以及验证是否空间内的所有可行解都被删除这两个过程的时间、空间开销还是很大。因此，本文使用训练程序中是否包含可行解来对这一项进行近似，同时可以接受一定比例的训练用问题对应的训练用程序中不包含未被删除的可行解。

因此，根据上述描述，在本文的实现中，使用了如下的公式来估计程序合成的加速效果：

$$f(x) = \left(\sum \frac{1}{Num(n_i) + 1} + \frac{R}{A} \sum \frac{1}{\mathrm{D_j} + 1} \right) \times \mathrm{B}$$

上式是本文中使用的估计函数 f(x)的公式。实际上针对不同的求解器，可以使用不同的估计函数或者在估计函数上加入不同的修正项。

第一项中， n_i 表示第 i 个训练问题上的大小最小的解的大小，$Num(n_i)$ 表示新语法上比 n_i 大小更小的程序的个数，即第一个解在程序空间中按从小到大顺序的排名。解的大小定义为这个解在新语法上的最小展开的展开式的个数。这一项主要用来估计程序空间的结构是否在当前训练问题集合上更优。

第二项中，A 表示添加的新语法组件的个数，R 表示进行深度限制的语法组件个数。D_j 表示第 j 个被删除的语法在训练程序上出现的次数多少，这一项用来估计程序空间的大小。

公式最后的 B 是表达能力惩罚项，只有 0 和 1 两个值，这是为了保证新语言的表达能力设定的。这是要保证训练集中某个训练问题下的所有训练程序都被删除的情况不能超过 K_t%，如果出现了这种情况，将 B 的值设为 0，否则为 1。

定性地看，$Num(n_i)$ 越小，求解器越容易在整个程序空间内找到可行解，加速系数越高；A 越大，即添加的语法组件越多，会使求解器的搜索空间变大，使得加速效果越不明显；D_j - R_j 越小，说明第 j 个被删除的语法越不重要，越应该被删除。

本文使用的是人工设计的估计函数 f(x)。实际上，f(x)可以根据测量得到的加速比和相关参数使用机器学习方法拟合出来。不过，由于估计函数参数比较多，测量实际加速比的开销也较大，难以得到准确的拟合结果。因此本文没有使用这种方法。

另外，在实际应用过程中需要注意的是，使用不同类求解方法的求解器在原理上有着根本性的不同，那么影响这些不同类别的求解器求解速度的因素也有不同。其次，即使是使用同一类求解方法的求解器，不同的求解器也使用了不同的优化方法，这些优化方法之间千差万别，使得这些使用同类方法的求解器之间的求解速度也有差别。根据不同求解器对估计函数进行修改也是必要的。

4.6 细粒度筛选

估计函数的准确性有限，遗传算法搜索出来的解可能相较最优解有一定差别。本文仅将遗传算法作为初步的粗粒度筛选，将遗传算法搜索出来的适应度最高的前 K_s 个语法再进行细粒度的筛选。由于遗传算法具有随机性，本文在实现中采取了重复三次遗传算法，取三次遗传算法运行结果中的前 K_s 个语法进行细粒度筛选。在细粒度的筛选

上，本文采用直接测量程序合成求解器求解速度的方法进行比较。

细粒度的比较选取的语言语法数量相对较少，使用直接测量求解时间的方法是可行的。由于估计函数得到的加速系数和实际新语言的加速比往往有着一定的出入，所以细粒度的筛选是必要的。否则，遗传算法按照估计函数搜索得到的最优解往往不是实际加速效果最好的解。遗传算法给出解与实际的最优解的差距与估计函数有关，估计函数越准确，遗传算法给出的最优解与实际最优解越接近。

第五章　实验与验证

本文实现了上述方法，并设计实验验证了在字符串处理这一领域上该方法的效果。本章中将讨论如下几个研究问题：

- **研究问题 1：**根据本文中的方法设计出的新的领域特定语言能否加速程序合成求解器的求解？
- **研究问题 2：**新语言的表达能力相比源语言有何变化？
- **研究问题 3：**实验中的各个参数对实验结果有何影响？
- **研究问题 4：**估计函数得到的加速系数与实际求解速度的加速比是否相关？相关度有多高？

5.1　数据集

本文实验中使用了 Lee 等人[25]收集的数据集，多项已有工作都使用了这一数据集进行实验。该数据集包含了 205 个字符串处理问题，其中包含 SyGuS 比赛字符串项目的 107 个程序合成问题，38 个收集自 StackOverflow 问答社区网站的问题，以及来自 ExcelJet 的 60 个问题。

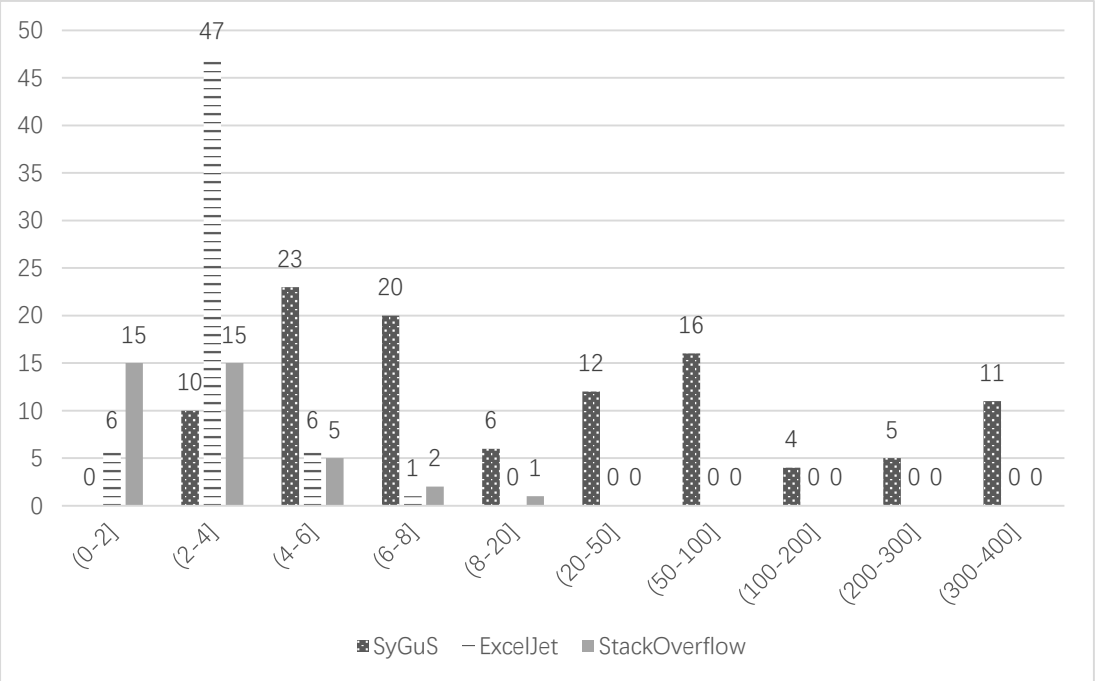

图 5.1　输入输出样例数量的频次

这些程序合成问题均为 PBE 问题，即给定输入输出样例作为规约。这些问题给出的输入输出样例从 2 对到 400 对不等，平均数为 42.7 对，中位数为 6 对，上图展示了输入输出样例数量的频次分布，横轴为输入输出样例数量的区间，纵轴为每个数据集中输入输出样例数量的频次。图中我们可以看到，数据集中绝大多数程序合成问题给出的输入数出样例的数量都在 8 对以下。通过进一步观察数据也可以发现，超过 50 对的输入输出样例大多是通过简单的重复得到的，即将少量几对输入输出样例重复多次，这一处理方式在 SyGuS 数据集中比较常见。SyGuS 数据集中进行这样的处理可能是为了区分程序合成求解器处理大量输入输出样例的能力。

需要注意的是，该数据集上的每个问题都给出了相应的 DSL，但是本文研究的对象是 DSL 对程序合成速度的影响，因此在本文的实验中不使用这些 DSL，而是分别使用实验中的源语言和生成的语言进行求解。

5.2 源语言

本文实验中采用的源语言是一个通用的字符串处理语言。该语言只有 Int 整型、String 字符串和 Bool 布尔值和 RegEx 正则表达式四种类型。

该语言包含正则表达式在内的四种基本数据类型，作为源语言，有较好的通用性，拥有足够的表达能力，能求解字符串处理领域的绝大多数问题。使用该语言作为源语言能够保证新语言的表达能力不受源语言的限制。

同时，该语言复杂程度适中，程序合成求解器能够在该语言上在合理的时间内求解出大部分字符串处理领域的程序合成问题，这使得利用程序合成求解器生成训练用程序成为可能。本文使用的 MaxFlash 求解器在 30 秒时限内在该语言上能够求解的问题数量如下表：

表 5.1 源语言上能够求解的问题数量

数据集	**SyGuS**	**ExcelJet**	**StackOverflow**	**总计**
求解数量/总数量	83/107	43/60	27/38	153/205

可以看出，在 30 秒时限内源语法能够求解数据集中约 75%的程序合成问题，且在各个数据集上能够求解的问题比例相差不大，均在 75%上下。

综上所述，用该语言作为本方法的源语言是合适的。

下图详细展示了本文使用的源语言的语法：

```
(ntString String   (str.++ ntString ntString)
                   (str.replace ntString ntString ntString)
                   (str.at ntString ntInt)
                   (int.to.str ntString)
                   (str.substr ntString ntInt ntInt)
                   (ite ntBool ntString ntString))

(ntInt Int         (+ ntInt ntInt)
                   (- ntInt ntInt)
                   (str.len ntString)
                   (str.to.int ntString)
                   (ite ntBool ntInt ntInt)
                   (Pos ntString ntRegEx ntInt ntInt)
                   (str.indexof ntString ntString ntInt))

(ntBool Bool       (str.prefixof ntString ntString)
                   (str.suffixof ntString ntString)
                   (str.contains ntString mtString)
                   )

(ntRegEx RegEx [0-9]
                   [a-z]
                   [A-Z]
                   ' '
                   ntRegEx | ntRegEx
                   ntRegEx ntRegEx
                   ntRegEx*
                   (str.toregex ntString))
```

图 5.2 源语言的语法

该语言有四种非终结符 ntString、ntInt、ntBool 和 ntRegEx，每个非终结符有若干条展开规则，其中各个函数的语义如下表：

表 5.2 函数的语义

(str.++ String String)	字符串连接函数，接受两个字符串，返回将参数中的两个字符串连接后的字符串。
(str.replace String String String)	字符串替换函数，返回将 Param1 中的所有 Param2 替换为 Param3 形成的新字符串。其中 Param1 表示第一个参数，Param2 表示第而个参数，以此类推（下同）。
(str.at String Int)	字符串索引函数，接受一个整数，返回该整数下标位置的字符。
(int.to.str String)	将整型数字转化为十进制字符串。
(str.substr String Int Int)	取子串函数，返回 Param1 中 Param2 位置开始到 Param3 长度的子串。
(str.len String)	字符串长度函数，返回参数字符串的长度。
(str.to.int String)	将十进制数字字符串转换为整型数字返回。
(str.indexof String String Int)	字符串检索函数，返回 Param1 中 Param2 第 Param3 次出现的位置。
(str.prefixof String String)	返回 Param2 是否是 Param1 的前缀。
(str.suffixof String String)	返回 Param2 是否是 Param1 的后缀。
(str.contains String String)	返回 Param1 是否包含 Param2。
(ite Bool String String)	字符串的 if-then-else 表达式，如果 Param1 为 true，返回 param2，否则返回 Param3
(ite Bool String String)	整数上的 if-then-else 表达式，如果 Param1 为 true，返回 param2，否则返回 Param3
(Pos String RegEx Int Int)	在 Param1 上进行正则表达式匹配，Param2 为要匹配的正则表达式，Param3 表示返回第几次匹配的位置，Param4 取值为 0 或 1，0 表示返回匹配开始的位置，1 表示匹配结束的位置。
(str.toregex String)	将字符串转换正则表达式

此外，该语言还支持若干基础正则表达式：[0-9]表示数字集合、[a-z]表示小写字母集合、[A-Z]表示大写字母集合、' '表示空格，以及正则表达式的或操作、连接操作、求闭包操作等常用操作。

5.3 训练程序生成

本文方法需要选取一部分程序合成问题作为训练集合。注意，由于本文的方法需要在训练集问题上找到足够多的可行解，出于时间和空间开销上的考虑，用于训练的程序应该是较为简单的程序，否则生成训练程序的过程开销过大，导致在有限的时间内找不到足够多的可行解用作训练程序。

这与本文方法在实际应用中的场景也是符合的。在实际应用中需要加速的往往也是比较复杂的问题。于是，可以从该领域比较简单的问题出发，设计适合该领域的领域特定语言，用于求解这一领域比较复杂的问题。

表 5.3 训练集与测试集的划分

数据集	SyGuS	ExcelJet	StackOverflow	总计
训练集数量	27	14	9	50
测试集数量	80	46	29	155
总计	107	60	38	205

本文在实验中选取了 50 个较为简单的字符串处理问题作为训练集，这些程序都能使用 MaxFlash 求解器求解出结果，在其余 155 个问题上进行测试。在训练集上，本文使用修改过的 MaxFlash 求解器在每个训练问题上求解得到 K_g 个可行解，将其作为训练程序。

5.4 实验设定

针对本章开头提出的研究问题，本文在上文描述的数据集和源语法上设计了一系列实验对这些问题进行回答。

在研究问题 2 以外的实验中，对方法中参数的设定为：

1）$K_g = 100$，即取 MaxFlash 在每个训练问题上的前 100 个可行解作为训练程序。

2）$K_f = 3$，即在频繁模式挖掘中，取至少重复出现三次的模式为频繁模式。

3）$K_t = 5$，即至多在 5%的训练问题下可以允许所有训练程序都被删除。

4）$K_s = 20$，即在遗传算法得到的前 20 种语法进行细粒度筛选。

对于**研究问题 1**，本文验证了本文所述方法加速 MaxFlash 求解器求解字符串处理问题上的效果。MaxFlash 是目前在字符串处理问题上求解速度最快的求解器，本文的方法实现了对 MaxFlash 的进一步加速。另外，本文还验证了本文方法在朴素枚举上的加速效果，本文使用的朴素枚举求解器只使用最简单的广度优先搜索进行枚举，使用 Z3 求解器验证搜索到的程序是否满足规约。在求解时限设置上，对于 MaxFlash 求解器，设置 30 秒的求解时限；对于朴素枚举求解器，设置 300 秒的求解时限。

对于**研究问题 2**，本文在实验中记录求解器在源语言上能够求解的问题，有多少在新语言上仍然能够求解，有多少在新语言上无法求解。

对于**研究问题 3**，本文将设置不同的参数重复进行实验，观察不同参数设置对于本文方法结果的影响。

对于**研究问题 4**，本文随机选取了若干生成的新语言，计算估计函数估计得到的加速系数和实际测量求解器求解时间的加速比。

5.5 实验结果

5.5.1 研究问题 1：本文方法的加速效果

表 5.4 整体加速效果

	求解数量		**求解速度**
	源语言	**新语言**	**平均加速比**
MaxFlash	103	106	2.64
朴素枚举	5	16	12.2

本文中计算加速比的方法为：对于同一个求解器，在源语法和新语法都能求解的问题下，计算新语法和源语法上的平均求解时间之比。对于 MaxFlash 求解器，设置 30 秒的求解时限；对于朴素枚举求解器，设置 300 秒的求解时限。

在 MaxFlash 求解器上，本文方法得到的新 DSL 相比源语言达到了平均 2.64 倍的加速比。在同一时限内的求解数量上也增加了三个。

朴素枚举求解器使用不采用任何优化手段的最基本的搜索方法。实验数据表明，本文方法对朴素枚举求解器也是有加速效果的，且相对加速效果比较显著，但是朴素枚举法由于程序空间过大，难以应对字符串处理领域一些比较复杂的问题。朴素枚举法能求解的问题数量过少，研究的意义不大。因此，在下文的实验中只使用 MaxFlash 求

解器。

表 5.5 各个数据集上的加速效果

数据集	求解数量		求解速度
	源语法	新语法	平均加速比
SyGuS	56	58	2.68
ExcelJet	29	30	2.63
StackOverflow	18	18	2.52
总计	103	106	2.64

上表展示了各个数据集上的加速效果。具体到各个数据集上的平均加速比相差不大，StackOverflow 数据集上的平均加速比稍低，可能是因为该数据集测试数据量较小带来的随机偏差，也有可能是训练集中来自该数据集的训练数据较少，有某些特征未被挖掘到，使得方法在该数据集上加速效果稍差。

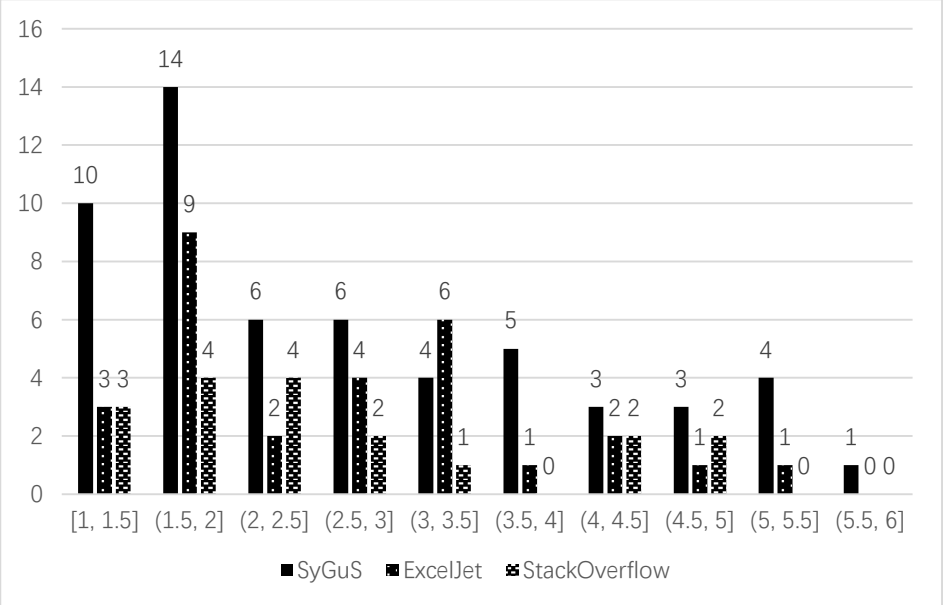

图 5.3 加速比分布直方图

上图为具体到每个问题的加速比分布的直方图，加速比在 1.5 到 2 这一区间的问题最多，加速比在 2 到 5 之间的分布相对比较均匀，测试集中的问题达到的最高的加速比为 5.82，来自 SyGuS 数据集。

5.5.2 研究问题 2：新语言的表达能力

表 5.6 新语言的表达能力

	求解数量		未能求解
	源语言	新语言	
MaxFlash	103	103	0

由于新语言是源语言经过变换得到的，因此新语言的表达能力是源语言的子集。本文的实验中表明，在源语言能够求解的 103 个问题中，求解器仍然能够在新语言上求解出全部 103 个问题。

要注意到，新语言的表达能力是源语言的子集，由于本文的方法删除了组件、限制了组件的语法展开深度，所以，从表达能力上来说，新语言的表达能力是下降了的，故可能存在某个测试集以外的问题可以用源语言求解，而不能用新语言求解，这在之后的调整参数 Kt 的实验中也有体现。不过，这一部分的实验说明了，本文方法得到的新语言的表达能力仍然是足够的，即在绝大多数情况下，新语法的表达能力足够解决字符串处理领域的程序合成问题。

5.5.3 研究问题 3：参数对实验结果的影响

本文分别采用不同的参数值进行实验，计算 MaxFlash 在得到的不同语法上的加速比。

1) 训练程序的数量 Kg

表 5.7 参数 Kg 的取值

Kg	**10**	**20**	**40**	**60**	**80**	**100**	**120**
加速比	1.01	1.07	1.93	2.45	2.62	2.64	2.64

Kg 为每个训练下的训练程序数量。在表中可以看出随着 Kg 的增大，本文方法所达到的加速比也越大，当 Kg 增大到一定程度之后，加速比的提升变得缓慢，甚至没有提升。在 Kg 较小时，程序合成在新的语言上几乎没有加速，进一步分析数据甚至可以发现，此时部分问题下使用新语言求解反而减速了，这可能是因为删除了过多的语法组件、对语法展开深度限制过大造成的。因此，实际应用本方法时，应保证足够数量的训练程序用于挖掘程序结构信息。在本文的实现中取 Kg=100，保证训练程序的数量足够。

2) 频繁模式挖掘中的阈值 Kf

表 5.8 参数 Kf 的取值

Kf	**2**	**3**	**4**	**5**	**6**	**10**	**+∞**
加速比	2.62	2.64	2.59	2.63	2.52	2.47	1.22

Kf 为频繁模式挖掘过程中的阈值，即出现频次超过 Kf 的模式在挖掘过程中被视为频繁模式。注意到本文方法得到的加速比随着 Kf 的变化有些波动，但在 Kf 一定范围内变化，方法的加速比变化都不大。这是因为本文方法还使用了遗传算法，对频繁模式挖掘得到的添加组件操作和其他修改操作一起进行进一步的筛选和组合。加速比在一定范围内的波动一定程度上是由随机误差带来的。

此外，此处还进行了将 Kf 设定为无穷大的实验，此时方法不会找到任何频繁模式，只会在语法中进行删除操作和限制深度操作。此时得到的加速比为 1.22，这一加速比是完全通过对程序空间进行剪枝得到的。这说明了除了提取程序中的可复用结构信息，对领域特定语言的语法进行限制也是能够有效加速程序合成求解的。

3) 可行解被删除比例的阈值 Kt

表 5.9 参数 Kt 的取值

Kt	**0**	**5**	**10**	**15**	**20**
表达能力	103	103	97	90	75
求解数量	105	106	100	94	79
加速比	2.18	2.64	2.71	2.94	2.92

Kt 是估计函数惩罚项中容忍可行解被删除的比例的阈值。根据表 6.5 可知，在 Kt 较小时，Kt 越大，新语言得到的加速比越高。但加速比数值不会随着 Kt 增大一直增高，当 Kt 较高时，对程序空间的过度剪枝反而会对程序合成的求解速度造成负面影响。这一点也体现在新语言的表达能力上。Kt 增大会影响语法的表达能力，使求解器能够求解的问题总数量减少。

这一步的实验结果证明了 3.2.2 中的一个结论，当对程序空间的剪枝涉及到程序合成求解器能找到的最优解的时候，即新语言的程序空间不包含源语言程序空间中的最优解时，程序空间大小与程序合成加速效果不再严格正相关。当程序空间中的最优解被排除，程序合成求解器需要花费额外的计算代价寻找剪枝后空间中的解，这在源语言的程序空间中可能是次优解或者次次优解。寻找次优解或者次次优解带来的额外开

销抵消了程序空间缩小带来的加速效果。

4) 细粒度筛选的样本数量 Ks

表 5.10 参数 Ks 的取值

Ks	**1**	**5**	**10**	**15**	**20**	**25**	**30**
加速比	2.33	2.58	2.64	2.64	2.64	2.64	2.64

Ks 经过遗传算法初筛之后再进行细粒度筛选的样本数量。根据表 6.6，可以看出 Ks 越大，新语言得到的加速比越大，这一结论是显而易见的。细粒度的筛选直接测量程序合成的求解速度，避免了估计函数不准确带来的偏差。

当 Ks 取 1 时，直接使用遗传算法得到的解，此时加速效果并不甚理想，但增大 Ks 取值，方法很快找到了一个最优的解。这说明估计函数还是相对准确的，但是仍然有不小的偏差，尤其在最优解的筛选中精确度不够。

5.5.4 研究问题 4：估计函数的准确性

估计函数是对新语法实际加速效果的估计，将其作为遗传算法的适应度函数。遗传算法的适应度函数需要计算样本的适应度，在本文中即新语言的加速效果，理论上使用直接测量得到的加速比是最准确的，但是直接测量的方法过于耗时，实际应用上的时间开销不可接受，故本文使用估计函数估计加速比。

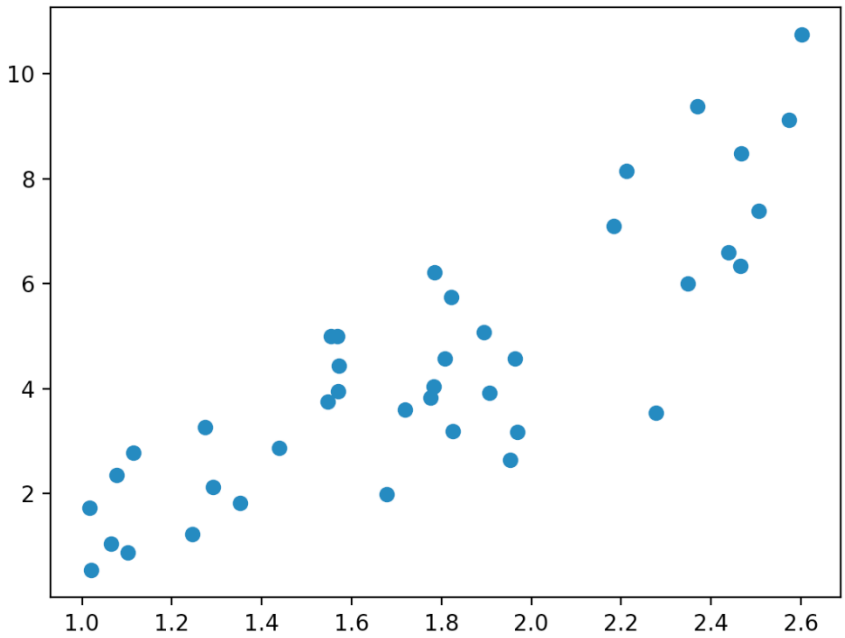

图 5.4 估计函数的准确性

在这一部分的实验中，取一定数量的新语法，计算其估计函数得到的加速系数和并测量实际加速比，得到的散点图如上图，横轴为实际加速比，纵轴为函数计算的加速系数，二者相关系数 r＝0.781，相关性显著。这说明估计函数对实际加速比的估计是较为准确的。

由于估计函数存在一定的偏差和局限，本文在遗传算法之后还使用了细粒度的筛选，细粒度的筛选中就直接使用测量得到的实际加速比进行筛选。有了细粒度筛选这一步骤，使得本文方法对估计函数准确性要求不高，只要在遗传算法得到的适应度最高的前 Ks 个解中包含最优解即可，之后细粒度筛选即可将前 Ks 个解中的最优解筛选出来。细粒度筛选为估计函数提供了一定的容错，这也是细粒度筛选这一步骤的必要性所在。

第六章　总结与展望

随着信息技术的不断发展，软件开发越来越成为人类的一项重要生产活动。现代社会的运行越来越依赖格式各样的软件，新的软件系统也越来越复杂，每年有越来越多的人力被投入软件开发活动中来。提高软件开发人员的开发效率一直是软件工程和编程语言领域广受关心的问题，其中，软件自动化是提高软件生产效率的核心。传统的软件自动化方法日臻成熟的今天，程序合成技术有望给软件开发的自动化程度带来新的突破。

6.1　本文工作总结

目前，程序合成技术的发展瓶颈在于程序合成的求解速度慢。为了解决这一问题，研究人员提出了多种方法、采取了多种策略来加速程序合成求解器的求解速度。不同于之前的大多数方法都关注于程序合成求解器的求解算法，本文尝试从一个新的角度来解决这一问题，即设计新的领域特定语言加速程序合成问题的求解。

本文分析了领域特定语言对程序合成求解速度的影响，并据此提出了一种自动化设计领域特定语言加速程序合成的技术。

领域特定语言描述了一个程序空间，程序合成过程就是在这个程序空间内找到满足规约的程序的过程。不同的领域特定语言描述了不同的程序空间，不同的程序空间直接影响着程序合成求解器的求解速度。

本文提出的方法以一个较为通用的领域特定语言作为源语言，通过挖掘领域内程序的特定语法结构信息，抽象出一系列语法上的修改操作，进而根据这些修改操作找到一个新的领域特定语言。新的领域特定语言是基于源语言的，但相较源语言在一定程度上更能反映出领域内程序的特征，更适合求解领域内的问题。

本文通过实验验证了本文提出方法的有效性。本文实现了本文提出的方法，并在字符串处理领域成功地加速了当前该领域速度最快的程序合成求解器 MaxFlash。本文提出的方法是一种通用的方法，也可以应用于加速其他领域、其他程序合成求解器的程序合成问题。

本位提出的方法达到了较好的加速效果，与源语言相比，MaxFlash 在本文方法找到的新的领域特定语言上的求解速度加速了 2.64 倍，在给定时限内求解出的程序数量也有所提升。

6.2 未来工作展望

本文的方法能够设计出新的 DSL 加速程序合成的求解，未来在本文工作的基础上，还有下面几个方向值得探索和研究：

1）本文只验证了方法在字符串处理领域、加速 MaxFlash 求解器的加速效果。未来应考虑在其他领域、其他类型的求解器上应用本文方法，验证方法在其他领域、其他类型求解器上的有效性。需要注意的是，不同的求解器的求解算法不同，新的领域特定语言应用在其上的加速效果也可能有不同，可能需要根据不同的求解器对方法的具体实现方式进行调整，例如修改搜索过程中的估计函数、定义新的语法修改操作等。

2）本文方法针对某一特定领域设计新的领域特定语言，加速程序合成在这一领域中的问题的求解速度。这一方法的应用场景假定了问题的领域已知且都属于该特定领域。但在实际过程中，新的问题往往不知道属于哪个领域。因此，自动对问题的领域进行归类或许是应用本方法之前需要考虑的问题。自动对问题领域归类有两个方向：一个是如何将问题划分入一个已知的领域，用该领域的已有领域特定语言进行求解，即新问题的分类；另一个方向是对若干未知领域的问题进行聚类，将其聚类成若干个不同的领域，在不同的领域内寻找不同的领域特定语言。

3）本文提出的方法是为熟悉的领域设计新的领域特定语言进行加速。类似的方法在未来能不能应用到为新的未知领域设计领域特定语言？在熟悉的领域本文的方法可以从一个已有的专用性较强的领域特定语言出发设计新的领域特定语言，但在未知的领域或许只能从通用编程语言出发设计领域特定语言。这对本方法依赖的程序合成求解器是一个考验，目前的程序合成求解器还很难在通用编程语言上进行求解。

4）本文方法的目标是加速程序合成的求解，类似的方法是否可以尝试应用于其他目标，例如提高程序合成的正确性是一个值得探索的问题。程序合成的规约通常是不完备的，不完备的规约会导致程序合成返回错误的程序，那么能否尝试通过领域特定语言的限制来尽可能避免程序合成返回不正确的程序。

5）本文方法为程序合成求解器设计新的领域特定语言，那么是否可以用类似技术为开发人员设计领域特定语言？与为程序合成求解器设计领域特定语言不同，为人设计领域特定语言需要考虑可读性和人的学习成本，需要用新的指标评价为人设计的领域特定语言的好坏。

6）自动设计领域特定语言的过程某些程度上来说也是一个机器学习任务。未来可以尝试将更多机器学习方法应用于设计领域特定语言的问题上。反之，领域特

定语言中可以包含一些领域知识，是否有机器学习任务能利用到这些领域知识。另外，诸如神经网络等机器学习方法缺乏可解释性，领域特定语言或许也能在这一点上予以补充。

参考文献

[1] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In POPL, pages 179–190. ACM, 1989.

[2] Syntax-Guided Synthesis. R. Alur, R. Bodik, G. Juniwal, P. Madusudan, M. Martin, M. Raghothman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak and A. Udupa. In 13th International Conference on Formal Methods in Computer-Aided Design, 2013.

[3] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh: Program Synthesis. Foundations and Trends in Programming Languages 4(1-2): 1-119 (2017)

[4] Oleksandr Polozov and Sumit Gulwani. Program synthesis in the industrial world: Inductive, incremental, interactive. In 5th Workshop on Synthesis (SYNT). 2016.

[5] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. Proc. ACM Program. Lang. 4, OOPSLA, Article 224 (November 2020), 29 pages.

[6] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 389–399.

[7] Yingfei Xiong, Bo Wang, Guirong Fu, Linfei Zang. Learning to Synthesize. GI'18: Genetic Improvment Workshop, May 2018.

[8] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In Proceedings of the 12th International Conference on Architectural Support for Programming LanguagesandOperatingSystems, ASPLOS2006, SanJose, CA,USA,October21-25,2006.404–415.

[9] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI2018, Philadelphia, PA, USA, June18-22,2018.436–449

[10] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2933–2942.

[11] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering

[12] AdityaKrishnaMenon,OmerTamuz,SumitGulwani,ButlerLampson, and Adam Tauman Kalai. 2013. A Machine Learning Framework for Programming by Example. In Proceedings of the 30th International Conference on International Conference on Machine Learning

[13] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented

Programming, Systems, Languages, and Applications

[14] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. 2017. DeepCoder: Learning to Write Programs. In 5th International Conference on Learning Representations

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4, 2

[16] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In Proceed- ings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems

[17] de Moura L., Bjørner N. (2008) Z3: An Efficient SMT Solver. In: Ramakrishnan C.R., Rehof J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg.

[18] S. Jha, S. Gulwani, S. A. Seshia and A. Tiwari, "Oracle-guided component-based program synthesis," 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, pp. 215-224

[19] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding dynamic programing via structural probability for accelerating programming by example. Proc. ACM Program. Lang. 4, OOPSLA, Article 224 (November 2020), 29 pages.

[20] Automating string processing in spreadsheets using input-output examples. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.317–330.

[21] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. Commun. ACM 62, 12 (December 2019), 56–65.

[22] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 593–604.

[23] A. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In OOPSLA, pages 107–126, 2015.

[24] Mirjalili S. (2019) Genetic Algorithm. In: Evolutionary Algorithms and Neural Networks. Studies in Computational Intelligence, vol 780. Springer, Cham.

[25] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI2018, Philadelphia,PA,USA,June18-22,2018.436–449.

[25] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. In Advances in Neural Information Processing Systems. 9169–9178.

[26]Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. ICML (2019).

[27] Chanchal K. Roy, James R. Cordy, Rainer Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming, Volume 74, Issue 7, 2009, Pages 470-495, ISSN 0167-6423, https://doi.org/10.1016/j.scico.2009.02.007.

[28] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). Association for Computing Machinery, New York, NY, USA, 631–642. DOI:https://doi.org/10.1145/2950290.2950334

[29] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, Joshua B. Tenenbaum, 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. arXiv preprint arXiv:2006.08381

附录 A 硕士期间参与的科研项目

[1] 深度学习系统测试技术，国家重点研发计划政府间合作项目，No.2019YFE0198100。

[2] 软件维护，国家自然科学基金优秀青年基金项目，No.61922003。

致谢

转眼间在燕园已经生活、学习了七年的时间，七年的时间里我增长了许多知识，收获了许多，也成长了许多。在这里遇到了许多优秀的老师和同学，此刻即将离开熟悉的校园，我在此对曾经帮助、陪伴过我的老师和同学们致以最真诚的感谢和祝福。

感谢杨芙清院士、梅宏院士领导的软件工程研究所，软工所为我们提供了丰富的资源以及优秀的环境。在这里我得以接触最前沿的学术研究，并亲身参与到科研之中，在科研道路上不断尝试和前进，相信这段经历会使我受益终生。

感谢我的导师熊英飞副教授。熊老师为人和善，是一名优秀的学者，也是一名出色的老师。从本科开始在熊老师的指导下接触科研以来，熊老师在学习、科研和生活中给了我许多中肯的建议和意见。很幸运能够成为熊老师的学生，熊老师是我们学习的榜样。

感谢张昕老师和吉如一同学在我完成硕士论文期间给与的指导和帮助。每周的讨论总能提出新的观点，得到一些新的有趣的结论。另外还要感谢吉如一同学提供的 MaxFlash 源代码以及在修改 MaxFlash 代码方面给与的帮助，吉如一同学的代码清晰、简洁、健壮，希望吉如一同学能够在未来取得更多更好的学术成果。

感谢程序设计语言实验室的胡振江老师。与胡老师接触时间不长，但胡老师广博的学识、谦和的为人、严谨的治学态度以及热忱的工作热情都给我留下的深刻的印象。胡老师是我们的朋友，我们的老师，也是我们敬佩的前辈。

感谢程序语言小组的同学们，感谢邹达明师兄、姜佳君师兄、王博师兄、梁晶晶师姐，在科研和生活中给了我许多帮助。感谢我的室友们，曾沐焓、谢佩辰、王子昌，从学习、生活到娱乐，互相帮助，共同分担，一起分享，你们使我三年的生活更加愉快和丰富。

最后，感谢我的家人们，你们是我坚实的后盾，是我前进的动力。你们给予我的支持始终是我最宝贵的财富。