研 究 生 毕 业 论 文

（申请专业硕士学位）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 论 | 文 | 题 | 目 | 面向浮点程序误差检测的用例生成系统 |
| 作 | 者 | 姓 | 名 |  |
| 学科、专业名称 专业硕士（软件工程领域） | | | | |
| 研 | 究 | 方 | 向 | 软件工程 |
| 指 | 导 | 教 | 师 |  |

学 号 ：

论文答辩日期 ：

指 导 教 师 ： （签字）

**Use Case Generating System for Floating-point Program Error Detection**

## By

**(Author Name)**

## Supervised by

(Supervisor’s position)**(Supervisor’s Name)**

A Thesis

Submitted to the XXX Department and the Graduate School

of XXX University

in Partial Fulfillment of the Requirements for the Degree of

**Master of Engineering**

# 目录

[图 目 录](#_bookmark0) **xi**

[表 目 录](#_bookmark1) **xiii**

|  |  |  |
| --- | --- | --- |
| [第一章](#_bookmark2) | [引言](#_bookmark2)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **1** |
| [1.1](#_bookmark3) | [背景](#_bookmark3)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 1 |
| [1.2](#_bookmark4) | [相关工作](#_bookmark4)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 2 |
| [1.3](#_bookmark12) | [本文工作](#_bookmark12)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 5 |
| [1.4](#_bookmark13) | [论文结构](#_bookmark13)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 6 |
| [第二章](#_bookmark14) | [相关技术综述](#_bookmark14) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **9** |
| [2.1](#_bookmark15) | [浮点数](#_bookmark15) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 9 |
|  | [2.1.1 IEEE 754 标准](#_bookmark16) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 9 |
|  | [2.1.2 浮点数表示](#_bookmark18)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 9 |
|  | [2.1.3 舍入规则](#_bookmark20) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 10 |
| [2.2](#_bookmark21) | [训练数据生成技术](#_bookmark21) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 11 |
|  | [2.2.1 Herbie 工具](#_bookmark22)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 11 |
|  | [2.2.2 iRRAM](#_bookmark26) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 13 |
| [2.3](#_bookmark28) | [LLVM 与符号执行技术](#_bookmark28) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 14 |
|  | [2.3.1 LLVM 架构](#_bookmark29)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 14 |
|  | [2.3.2 LLVM 命令](#_bookmark32)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 15 |
|  | [2.3.3 符号执行技术](#_bookmark34) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 16 |
| [2.4](#_bookmark37) | [深度学习与神经网络](#_bookmark37)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 17 |
|  | [2.4.1 深度学习](#_bookmark38) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 17 |
|  | [2.4.2 神经网络](#_bookmark39) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 18 |
| [2.5](#_bookmark42) | [本章小结](#_bookmark42)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 19 |

|  |  |  |
| --- | --- | --- |
| [第三章](#_bookmark43) | [浮点用例生成系统的需求分析](#_bookmark43) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **21** |
| [3.1](#_bookmark44) | [浮点用例生成系统的总体规划](#_bookmark44) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 21 |
| [3.2](#_bookmark46) | [浮点用例生成系统的功能性需求](#_bookmark46)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 22 |
|  | [3.2.1 程序预处理模块的需求](#_bookmark48)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 23 |
|  | [3.2.2 中间代码转换模块的需求](#_bookmark51) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 25 |
|  | [3.2.3 训练数据集生成模块的需求](#_bookmark54) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 26 |
|  | [3.2.4 神经网络训练模块的需求](#_bookmark56) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 27 |
|  | [3.2.5 数值用例生成模块的需求](#_bookmark60) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 29 |
| [3.3](#_bookmark63) | [浮点用例生成系统的非功能性需求](#_bookmark63) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 30 |
| [3.4](#_bookmark65) | [本章小结](#_bookmark65)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 31 |
| [第四章](#_bookmark66) | [浮点用例生成系统的设计](#_bookmark66) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **33** |
| [4.1](#_bookmark67) | [浮点用例生成系统的概要设计](#_bookmark67) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 33 |
|  | [4.1.1 系统架构设计](#_bookmark68) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 33 |
|  | [4.1.2 逻辑视图设计](#_bookmark70) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 35 |
|  | [4.1.3 进程视图设计](#_bookmark72) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 36 |
|  | [4.1.4 开发视图设计](#_bookmark74) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 37 |
|  | [4.1.5 物理视图设计](#_bookmark76) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 38 |
| [4.2](#_bookmark78) | [浮点用例生成系统的详细设计](#_bookmark78) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 39 |
|  | [4.2.1 程序预处理模块的详细设计](#_bookmark79) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 39 |
|  | [4.2.2 中间代码转换模块的详细设计](#_bookmark83) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 41 |
|  | [4.2.3 训练数据集生成模块的详细设计](#_bookmark87) · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 43 |
|  | [4.2.4 神经网络训练模块的详细设计](#_bookmark92) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 47 |
|  | [4.2.5 数值用例生成模块的详细设计](#_bookmark96) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 49 |
| [4.3](#_bookmark101) | [数据库设计](#_bookmark101) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 53 |
| [4.4](#_bookmark106) | [本章小结](#_bookmark106)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 54 |
| [第五章](#_bookmark107) | [浮点用例生成系统的实现](#_bookmark107) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **55** |
| [5.1](#_bookmark108) | [程序预处理模块的实现](#_bookmark108) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 55 |
| [5.2](#_bookmark114) | [中间代码转换模块的实现](#_bookmark114) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 58 |
| [5.3](#_bookmark117) | [训练数据集生成模块的实现](#_bookmark117)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 60 |
| [5.4](#_bookmark120) | [神经网络训练模块的实现](#_bookmark120) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 60 |

|  |  |  |
| --- | --- | --- |
| [5.5](#_bookmark122) | [数值用例生成模块的实现](#_bookmark122) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 61 |
| [5.6](#_bookmark124) | [本章小结](#_bookmark124)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 62 |
| [第六章](#_bookmark125) | [实验评估与测试](#_bookmark125)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **63** |
| [6.1](#_bookmark126) | [实验评估](#_bookmark126)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 63 |
|  | [6.1.1 实验环境](#_bookmark127) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 63 |
|  | [6.1.2 实验设计](#_bookmark129) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 64 |
|  | [6.1.3 实验结果评估](#_bookmark133) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 66 |
| [6.2](#_bookmark141) | [功能性测试](#_bookmark141) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 68 |
| [6.3](#_bookmark147) | [非功能性测试](#_bookmark147) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 71 |
|  | [6.3.1 易用性](#_bookmark148) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 71 |
|  | [6.3.2 性能](#_bookmark149) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 71 |
| [6.4](#_bookmark151) | [本章小结](#_bookmark151)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 72 |
| [第七章](#_bookmark152) | [总结与展望](#_bookmark152) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | **73** |
| [7.1](#_bookmark153) | [总结](#_bookmark153)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 73 |
| [7.2](#_bookmark154) | [展望](#_bookmark154)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | 74 |
| [参考文献](#_bookmark155) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | | **75** |
| [致谢](#_bookmark199)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | | **81** |
| [简历与科研成果](#_bookmark200)· · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | | **83** |
| [学位论文原创性声明](#_bookmark201) · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · · | | **85** |

# 图 目 录

* 1. [穷尽表达式生成的主要算法](#_bookmark9) 4
  2. [数值扰动算法](#_bookmark11) 5
  3. [IEEE 754 浮点数标准](#_bookmark17) 9
  4. [Herbie 工具测试结果](#_bookmark23) 11
  5. [Herbie 工具误差改进结果](#_bookmark24) 12
  6. [浮点函数计算值和真实值的对比](#_bookmark25) 12
  7. [函数 *f* (*x*) = √*x* + 1 − √*x* 及其误差函数](#_bookmark27) 13
  8. [传统编译器架构](#_bookmark30) 14
  9. [llvm 编译器架构](#_bookmark31) 14
  10. [llvm 编译命令](#_bookmark33) 15
  11. [llvm 工具常用命令](#_bookmark35) 16
  12. [一种符号执行的基本算法流程](#_bookmark36) 16
  13. [RNN 神经网络模型](#_bookmark40) 18
  14. [CNN 神经网络模型](#_bookmark41) 19
  15. [用例生成系统整体流程](#_bookmark45) 21
  16. [用例生成系统的功能性需求划分](#_bookmark47) 23
  17. [浮点程序预处理模块用例图](#_bookmark49) 24
  18. [中间代码转换模块用例图](#_bookmark52) 25
  19. [训练数据集生成模块用例图](#_bookmark55) 27
  20. [神经网络训练模块用例图](#_bookmark58) 28
  21. [数值用例生成模块用例图](#_bookmark61) 30
  22. [系统架构图](#_bookmark69) 34
  23. [系统逻辑视图](#_bookmark71) 35
  24. [系统进程视图](#_bookmark73) 36
  25. [系统开发视图](#_bookmark75) 37
  26. [系统物理视图](#_bookmark77) 38
  27. [程序预处理模块流程](#_bookmark80) 39
  28. [程序预处理模块类图](#_bookmark81) 40
  29. [浮点程序预处理模块时序图](#_bookmark82) 41
  30. [中间代码转换模块流程](#_bookmark84) 41
  31. [中间代码转换模块类图](#_bookmark85) 42
  32. [中间代码转换模块时序图](#_bookmark86) 43
  33. [训练数据集生成模块流程图](#_bookmark88) 43
  34. [训练数据集生成模块类图一](#_bookmark89) 44
  35. [训练数据集生成模块类图二](#_bookmark90) 45
  36. [训练数据集生成模块时序图](#_bookmark91) 46
  37. [神经网络训练模块流程图](#_bookmark93) 47
  38. [神经网络训练模块类图](#_bookmark94) 48
  39. [神经网络训练模块时序图](#_bookmark95) 49
  40. [数值用例生成模块流程图](#_bookmark97) 49
  41. [数值用例生成模块类图一](#_bookmark98) 50
  42. [数值用例生成模块类图二](#_bookmark99) 51
  43. [数值用例生成模块时序图](#_bookmark100) 52
  44. [独立字符单元生成代码](#_bookmark109) 55
  45. [Token 生成代码](#_bookmark110) 56
  46. [抽象语法树生成入口函数代码](#_bookmark111) 56
  47. [抽象语法树生成代码](#_bookmark112) 57
  48. [中间代码 IR 生成的代码](#_bookmark113) 58
  49. [转换算法的部分代码](#_bookmark115) 59
  50. [转换 IR 的代码](#_bookmark116) 59
  51. [验证数值区间的代码](#_bookmark118) 60
  52. [生成训练数据的代码](#_bookmark119) 60
  53. [神经网络模型的代码](#_bookmark121) 61
  54. [数值用例生成功能的代码](#_bookmark123) 62
  55. [实验过程的问题](#_bookmark132) 65
  56. [实验过程使用的浮点程序](#_bookmark139) 67

表 目 录

* 1. [程序预处理模块功能性需求列表](#_bookmark50) 25
  2. [中间代码转换模块功能性需求列表](#_bookmark53) 26
  3. [训练数据集生成模块功能性需求列表](#_bookmark57) 28
  4. [神经网络训练模块功能性需求列表](#_bookmark59) 29
  5. [数值用例生成模块功能性需求列表](#_bookmark62) 30
  6. [用例生成系统的非功能性需求列表](#_bookmark64) 31
  7. [sourceCode 表字段](#_bookmark102) 53
  8. [IR 表字段](#_bookmark103) 53
  9. [byteCode 表字段](#_bookmark104) 54
  10. [userData 表字段](#_bookmark105) 54
  11. [实验环境](#_bookmark128) 63
  12. [有效数值用例生成效率以及用例生成耗时对比](#_bookmark134) 66
  13. [浮点程序的部分数值用例评估](#_bookmark140) 67
  14. [中间代码生成的测试用例](#_bookmark142) 68
  15. [中间代码转换的测试用例](#_bookmark143) 69
  16. [训练数据生成的测试用例](#_bookmark144) 70
  17. [神经网络模型生成的测试用例](#_bookmark145) 70
  18. [数值用例和运行时间生成的测试用例](#_bookmark146) 71
  19. [用例生成系统任务耗时性能测试](#_bookmark150) 72

# 第一章 引言

## 背景

随着计算机科学技术的不断发展，浮点程序不仅应用于航空航天、国防军事等关键领域，同时也常常被应用于高铁、汽车、能源、通信等工业软件的关键组件中。这些关键领域和关键组件对于可靠性的要求极高，因此，保障浮点程序计算结果的准确性至关重要。据统计，这些关键领域中 65% 的问题都是由浮点程序的计算误差而引起的 [[1](#_bookmark156)]，为此美国将“浮点程序准确性作为软件可靠性的主要发展方向之一”，致力于保障浮点计算的结果准确。

绝大部分浮点程序按照 IEEE 754 国际标准 [[2](#_bookmark157)] 来定义和处理浮点运算。按照这一标准，计算机会按照舍入规则 [[3](#_bookmark158)] 对不能直接表示的浮点数值进行近似处理。例如，单精度型的浮点数在计算机内部由 32 个比特位来表示，仅能精确表示实数轴上 232 个实数。对于不能精确表示的实数，舍入规则会选择距离该实数最接近的浮点数值来代替，这将不可避免地引入舍入误差 [[4](#_bookmark159)]。这些舍入误差会随着程序运行而累积，并最终引发严重后果。因此，保障浮点计算的准确性是十分困难的，历史上有大量因未能将误差累积控制在一定范围内而酿成的事故 [[5](#_bookmark160)]。例如，在计算布朗常数的过程中，奔腾处理器的误差累积造成计算结果出现错误，最终导致Intel 公司召回了市面上所有的奔腾处理器，并损失了近 4.75 亿美元[[6](#_bookmark161)]；1991 年的海湾战争中，由于美国导弹防御系统使用的浮点数精度仅有 24位，系统需要将准确值舍入为 24 位的近似值，从而造成在计算防御导弹发射时间过程中的舍入误差不断累积，最终导致防御系统未能成功拦截敌方导弹而造成了 28 名军人死亡 [[7](#_bookmark162)]。

目前，华为公司采用了自动化的误差检测技术来控制其工业软件中的浮点误差。该技术通过数值用例来触发误差累积，并结合数值扰动与算式扰动等技术自动检测程序中的数值缺陷。在误差检测过程中，数值用例作为浮点程序的输入值来驱动程序执行，并在一系列复杂的计算过程中同时引入数值扰动和算式扰动。当发生误差累积时，输出结果与实际结果之间的偏差会不断增大，扰动引起的数值波动也会随之增加，并可由此自动判断是否存在数值缺陷，保障程序的准确性。因此，可触发误差累积的数值用例是该技术的关键组成部分。

然而，现有的数值用例生成技术仍然存在以下不足：一方面，现有的用例生成系统主要通过提取软件中的算术表达式，并针对各算术表达式的局部性质来

生成使某一表达式误差较大的用例。基于这一原理，生成的用例仅面向软件中的局部表达式，而难以适应浮点程序全局的用例生成要求，从而造成用例适应性不足。另一方面，能触发浮点程序明显误差累积的有效用例通常分散在一些较小范围的区间内，现有系统命中这些小区间的概率较低，因此需要生成大量用例才能找到这些小区间中的有效用例，这导致系统生成有效用例的效率较低。

本文面向华为工业软件中保障浮点计算程序准确性的需求，在改进已有技术的基础上，设计并实现了一个面向全局浮点程序误差检测的用例生成系统。本系统基于LLVM 编译器对任意的C++ 浮点程序生成对应的中间代码，然后利用自主设计的转换规则将中间代码转换为可用的二进制码，并通过获取该二进制码与有效数值用例之间的映射关系，最终生成面向浮点程序全局误差检测的数值用例。同时，本系统改进了现有系统随机选取数值用例的方法，改进后的方法需要获取由浮点程序二进制码和触发浮点程序误差累积的有效数值用例组成的训练数据，并利用神经网络模型学习这些训练数据中的有效数值用例生成规则，最终获得浮点程序的具体数值用例。此方法保证这些用例大部分都落在触发浮点程序明显误差累积的区间内，提高了系统生成有效用例的效率。目前本系统已经在华为内部上线。我们针对一系列浮点软件，共生成了 16800 个数值用例来进行实验评估。评估结果表明本系统能够针对浮点程序进行全局用例生成，生成用例中能触发浮点程序误差累积的有效用例约占总数的 74.42%，且生成用例的平均耗时约为 2.58s，符合工业软件的用例生成要求。

## 相关工作

浮点程序的误差控制是一个较为长期的研究课题。1963 年，Wilkinson 在书中介绍了浮点运算中舍入误差累积效应问题的分析理论 [[8](#_bookmark163)]。1996 年，Higham在书中根据之前的Wilkinson 理论，详细分析了浮点程序中舍入误差的累积对程序结果准确性和稳定性的影响 [[9](#_bookmark164)]。1998 年，Rump 在其论文中就各种不同浮点程序的优化进行了探讨，通过变化浮点数处理顺序等操作，使获得的程序产生的误差尽可能小甚至避免误差 [[10](#_bookmark165)]。2005 年，周毓麟构建了一种概率模型，这种模型能够考虑舍入误差的积累，验证其对浮点程序准确性的阻碍作用 [[11](#_bookmark166)]。同年，Eggert 实现了一个数值稳定性分析程序，来分析浮点舍入模式的随机变化对浮点程序的影响[[12](#_bookmark167)]。两年后，Martel 对基于语义的浮点运算表达式的转换原理进行了深入的讨论 [[13](#_bookmark168)]，并在之后提出了能够提高浮点运算数值精度的表达式转换方法[[14](#_bookmark169)]。但是这些研究大多注重误差产生原理，无法生成误差检测所需的数值用例。

针对上述问题，工业界出现了一些用例生成系统，但这些系统仍存在不足。

下面对已有系统采用的方法进行具体描述，并分析其不足之处。现有系统采用表达式扰动方法，通过对单一浮点表达式的转换分析其对浮点运算误差的影响，并生成此表达式的数值用例 [[15](#_bookmark170)]。表达式扰动是将浮点表达式变换为语法不同但语义相同的等价表达式，并观察不同形式对于浮点运算误差的影响。此方法首先为其算法假设的表达式定义了规范形式，如公式([1.1](#_bookmark5))所示，并将需要处理的浮点表达式转换为这一规范形式。在这种规范形式中，*e* 需要是最简表达式，即它不能够存在加零或乘一的多余操作。

*e* ::= −*e* | *e* + *e* | *e* − *e* | *e* × *e* | *e* ÷ *e* | *ε* | *ν* (1.1)

得到 *e* 的最简表达式后，我们需要对其进一步重写，通过公式([1.2d](#_bookmark7))使其没有二元的减法操作，通过公式([1.2f](#_bookmark8))使其只使用倒数形式的除法，同时由于 *e* 是完全展开的，因此也不能被乘法分配律改写。具体规则如公式([1.2a](#_bookmark6))到([1.2f](#_bookmark8))所示。通过循环应用此规则，最终得到 *e* 的规范形式。

|  |  |
| --- | --- |
| Θ(*ε*) → *ε* | (1.2a) |
| Θ(−(−*e*1)) → Θ(*e*1) | (1.2b) |
| Θ(−(*e*1 + *e*2)) → Θ(−*e*1 − *e*2) | (1.2c) |
| Θ(*e*1 − *e*2) → Θ(*e*1) + (Θ(−*e*2)) | (1.2d) |
| Θ(*e*1 × (*e*2 ± *e*3)) → Θ(*e*1 × *e*2) + (Θ(±*e*1 × *e*3)) | (1.2e) |
| Θ(*e*1*/e*2) → Θ(*e*1) × (1*/*Θ(*e*2)) | (1.2f) |

表达式 *e* 转换为规范形式后，需要经过一系列算法处理，穷尽产生其所有等价表达式，主要算法如图[1.1](#_bookmark9)所示。其中算法 1 定义了函数 *tac* : *E* → 2*E*，此函数使用结合律与交换律生成实数域上 *e* 的所有等价表达式的集合。该函数首先在第 4 行通过 *units* 方法提取出表达式的所有算术单元并返回，并在第 7 行通过调用 *z* : *S* × {+*,* ×} → 2*E* 方法得到这些算术单元所有可能结合的方式。在第 10 行，生成那些在每次单元组合时递归调用算法 *tac* 的算术单元的所有可能重写。最后在第 11 行，返回用所有可能重写代替每个算术单元形成的集合。算法 2 定义了函数 *f* : *E* → 2*E*，此算法利用分布律生成实数域上 *e* 的所有等价表达式。该函数在第 7 行的 *f p* 方法返回 *e*′ 的所有公因式提取点的集合，然后通过第 9 行的操作考虑所有可能的组合情况，同时 *f* 会在这些组合中迭代寻找新的公因式提取点，然后重新组合，直至无法生成新的表达式。最终现有系统使用前面所述的两个算法，生成表达式 *e* 在数学范围内所有的等价表达式，并生成针对这些表达式的数值用例。

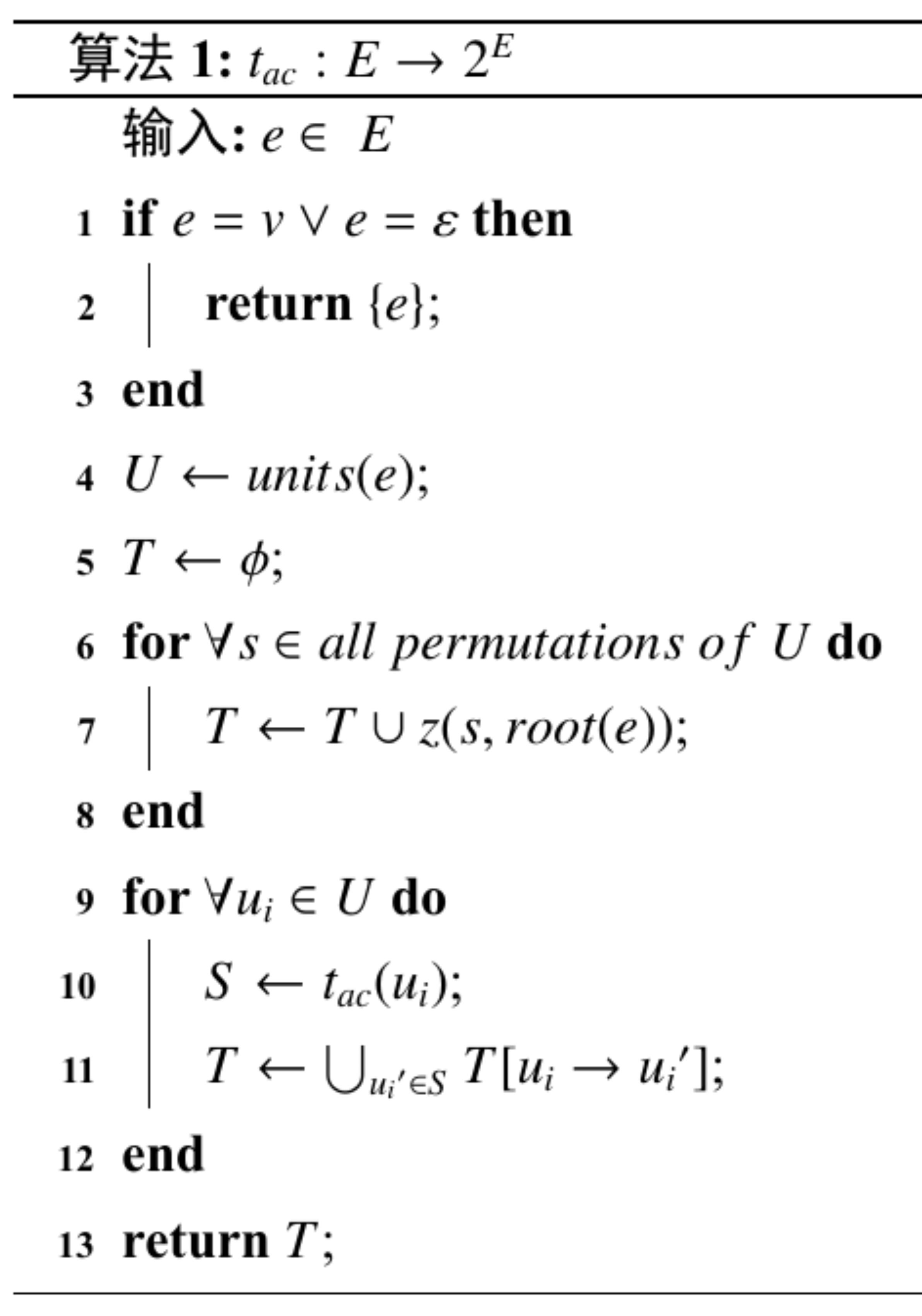
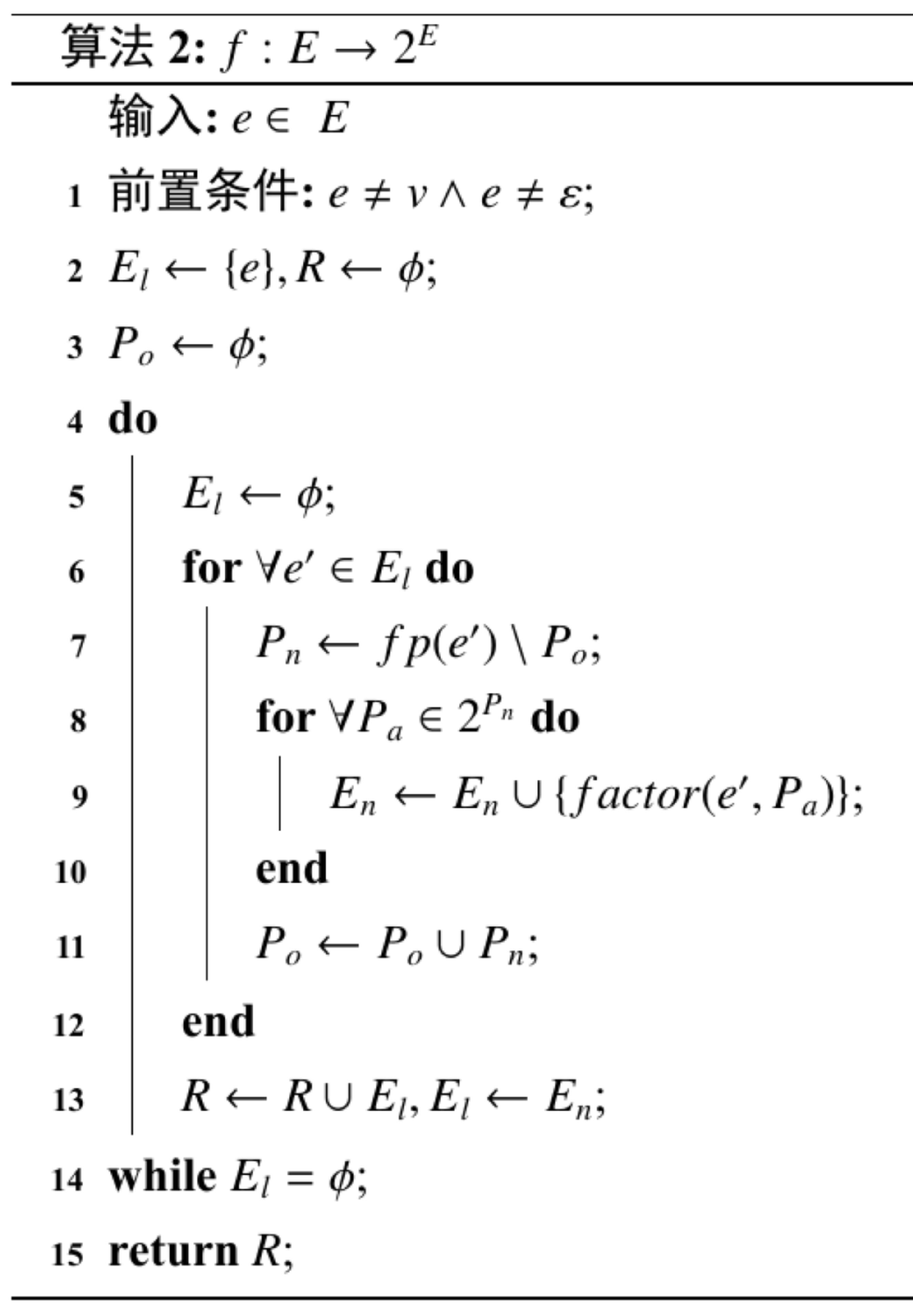
 

图 1.1: 穷尽表达式生成的主要算法

从上述分析来看，现有系统实际上仅仅对单一浮点表达式进行处理，这导致它们只能针对单个浮点表达式进行用例生成工作，而无法针对浮点程序全局进行用例生成。除此以外，Muller 等学者的研究，如使用 Harrisson 方法计算 *ab* + *cd*的舍入误差[[16](#_bookmark171)]、浮点数乘法运算的舍入误差 [[17](#_bookmark172)]、浮点数求逆的舍入误差 [[18](#_bookmark173)]等，大多也只是关注浮点表达式，而非浮点程序全局。

不仅如此，现有系统还采用数值扰动方法随机选取数值用例，并改变其最低有效位来进行浮点运算的误差检测 [[15](#_bookmark170)]。数值扰动通过改变输入用例来夸大舍入误差，从而判断浮点运算的稳定性。数值扰动研究中，规定 *s* 为符号位，*b*基值为 2，*p* 为有效位的位数，*x* 为指数，*f* 为有效位的整数值，因此可以得到公

式([1.3](#_bookmark10))。

( 1)*s*  *f bx* (1.3)

– × ×

*bp*−1

每个实数 *R* 都有对应的浮点表示 *f* ，此 *f* 因舍入会不可避免地存在表示误差。为了估算 *f* 的误差大小，现有工作使用一个随机的 *k* 位字符串替换 *f* 有效位的 *k* 位后缀。实际上，用浮点表示的 *r* 作为 *f* 有效位的前缀，并将其与任意 2*k* 个可能后缀连接，最终模拟了表示误差的均匀分布。由于研究中扰动了一个 *k*位的后缀，所以此方法比计算 *r*(1 ± *k*) 需要更少的运算步骤且此方法主要集中在有效位上，因此更加高效。当 *F* 需要以特定格式表示所有可能的浮点值时，使

用 *pv* : *N*1 × *F* → *F* 来表示此过程，如图[1.2](#_bookmark11)所示。

随机扰动浮点值的后缀使得其能够对由浮点近似操作引入的误差进行统计建模。由于方法还干扰了运算的中间结果，因此不仅对表示误差进行建模，而且对运算误差进行建模。但是，由于已有技术在生成数值用例方面是随机获取的，因此需要生成大量用例才能找到那些能够触发浮点程序明显误差累积的有效用例，整体效率较低。

double p\_v(double value, int bits){

byte \*byte\_array = (byte\*) (&value);

for(i = 0; i < bits/8; ++i){ byte\_array[i] ^= (byte) rand(); } byte\_array[i] ^= (0xFF >> (8-bits%8)) & ((byte) rand()); return value;

}

图 1.2: 数值扰动算法

针对现有系统仅面向单一表达式生成用例的适应性不足，以及有效用例生成效率低等问题，本文面向华为工业软件的需求，设计并实现了一种面向浮点程序误差检测的用例生成系统。对于任意浮点程序，本系统均可生成用于该浮点程序全局误差检测的数值用例，且生成的数值用例绝大部分都能够触发该浮点程序的明显误差累积，提高了系统生成有效用例的效率。

## 本文工作

本文面向华为需求，设计并实现了浮点程序误差检测的用例生成系统。该系统基于 LLVM 编译器，并利用神经网络模型自动生成用于浮点程序误差检测的数值用例。系统针对现有方法仅面向单一表达式生成用例而造成的适应性不足问题，设计并实现了针对浮点程序全局的用例生成技术，该技术基于二进制码的数值映射逻辑，通过符号执行技术将不同表达式的浮点计算链接起来，从而使生成的用例面向全局的数值关系。针对现有系统有效用例生成效率低的问题，本文引入了一种基于神经网络的数值区间预测技术。该技术以神经网络模型来预测有效用例的数值区间，并以浮点程序的二进制码与触发程序明显误差累积的数值用例为训练数据，由人工智能方法来自动学习用例生成分布规则，使生成用例能够命中有效区间的概率有效增加，从而大幅度提高了用例的生成效率。本系统通过五个模块来实现这些功能，包括程序预处理模块、中间代码转换模块、训练数据集生成模块、神经网络模型训练模块和数值用例生成模块，下面将对这五个模块进行详细说明：

* + 1. 程序预处理模块主要进行浮点程序对应中间代码的生成工作。此模块基于

LLVM 编译器，通过分析源码对于浮点程序的处理过程，找到其中与中间代码生成有关的源代码，并对出于适应性要求对中间码实施编码，使其生成可用的中间代码文件，并将此中间代码数据存储进数据库，为后续特征提取提供基础设施。

* + 1. 中间代码转换模块主要进行将中间代码转换为神经网络模型可用的二进制码。此模块需要利用程序预处理模块生成的中间代码文件，通过重新设计中间代码转二进制码的转换规则，最终生成浮点程序对应的二进制码文件，并将此二进制码数据存储进数据库中，供其他模块使用。
    2. 训练数据集生成模块主要进行模型训练数据的生成工作。此模块需要利用浮点程序预处理模块和中间代码转换模块实现的功能，生成浮点程序的二进制字节码，并与通过 Herbie 工具生成、误差函数验证的有效数值用例共同组成一条训练数据。重复此过程获得最终用于模型训练的训练数据集，并将训练数据存储进数据库中。
    3. 神经网络模型训练模块主要进行可用模型的生成工作。此模块首先需要搭建一个神经网络模型，然后由生成的训练数据集数据对模型进行训练，检测训练效果并根据效果进行神经网络模型的调整。我们采用了效果最优的卷积神经网络 CNN 作为预测模型，并重复模型训练和模型调整的过程，直到达到满意的效果为止，并输出可用模型。
    4. 数值用例生成模块主要进行用于浮点程序误差检测的数值用例输出工作。此模块需要利用预处理模块和转换模块实现的功能以及神经网络模型训练模块生成的可用模型，通过转化用户提供的待误差检测的浮点程序，得到对应的中间代码和二进制码，并驱动二进制码投入训练完成的模型中，从而直接生成能触发该浮点程序明显误差累积的数值用例，用户可以得到整体耗时等相关信息。

本系统目前已经在华为内部上线，通过上述五个模块相互协作，共同实施了面向浮点程序误差检测的用例生成过程。同时，我们对系统进行了实验评估，评估结果表明本系统能够针对浮点程序全局进行用例生成，且生成用例中能触发浮点程序明显误差累积的有效用例约占总数的 74.42%。

## 论文结构

本文的组织结构如下：

第一章是引言部分。本章主要阐述了项目的背景和意义，描述了现阶段针对浮点运算误差提出的理论和用例生成系统，详细说明了本文为解决已存在的问题所提出的解决方案以及本人主要的工作内容。

第二章是技术概述。本章主要介绍了浮点数以及浮点运算误差的产生的原因，并阐述了测试数据中用于浮点程序误差检测的数值用例的来源与验证，并在最后介绍了系统所使用的 LLVM 编译器以及神经网络和深度学习技术。

第三章是系统需求分析。本章首先从系统背景、系统开发目的、以及系统提供的功能等方面介绍项目的整体情况。然后，本章将分析系统的功能需求，并给出系统各模块的用例图、用例描述以及需求描述。最后，本章将给出系统的非功能性需求。

第四章是系统概要设计与详细设计。首先，本章以需求分析和用例描述为基础，给出系统的总体架构。然后，结合四个视图进一步说明本系统的概要设计。最后，通过绘制具体的时序图和详细类图等，对系统的模块进行详细设计。

第五章是系统的具体实现。本章主要依据系统的概要设计和详细设计，给出各模块所实现功能的关键代码，并对代码进行一定的解释。

第六章是实验评估与测试。本章通过对实验过程的描述以及对实验结果的分析，对系统效果进行了评估。同时详细描述系统功能性测试与非功能性测试的过程，并根据结果对系统进行分析和总结，对项目进行一定程度的优化。

第七章是总结与展望。本章将总结系统的开发过程，并对项目的优点和不足进行说明，并对项目的未来发展作出展望。

# 第二章 相关技术综述

## 浮点数

浮点数是精确数值的有理数表示。在计算机中，浮点数通常需要被舍入到一定精度。浮点运算是指浮点数参与的数学运算，这种数学运算通常伴随着因为无法准确表示而进行的近似或舍入误差。

* + 1. **IEEE 754** 标准

提到浮点数，当然得提及浮点数标准。浮点数标准，也称为IEEE 浮点数算术标准(IEEE 754 标准)。此标准从上世纪 80 年代以来被广泛使用，直到现在仍应用于绝大部分浮点运算器以及计算机。

**单精度（32位）**

0 1 8 9 31

数符 阶码 尾数

0 1

**双精度（64位）**

11 12 63

数符 阶码 尾数

图 2.1: IEEE 754 浮点数标准

IEEE 754 浮点数标准定义了浮点数的几种表示格式，其中最常见的包括单精度格式 (32 位) 与双精度格式 (64 位)，如图 [2.1](#_bookmark17)所示。这两种格式均包含数符、阶码以及尾数，这三部分共同组成了一个常见的浮点数。

* + 1. 浮点数表示

结合图 [2.1](#_bookmark17)，我们可以发现 IEEE 754 浮点数标准将浮点数的表示分为三大部分，分别为符号位，指数位 (阶码) 和尾数位 [[19](#_bookmark174)]。逻辑上是使用包含符号位，指数位和尾数位的三元组来表示浮点数 *N*。其中，规定使用 2 作为待表示浮点数的基数，使用 *P* 作为待表示浮点数的有效数字位数；规定使用 *S* 作为待表示浮点数的符号位，*S* 可以是 0 或者 1，分别表示该浮点数大于 0 或者小于 0；规定使用 *E* 作为待表示浮点数的指数位，*E* 需要通过对原数据的指数 *e* 进行处理，为

其加上一个固定的偏移量后得到，这种移码的做法既可以有效避免出现指数小于 0 的情况，又可以保持原数据的大小顺序，便于进行原数据大小的比较；规定使用 *M* 作为待表示浮点数的尾数位，根据浮点数标准规定，尾数部分的最高有效位必须为 1；同时浮点数标准规定最高有效位无需存储，可以认为其存在于小数点左侧，但被隐藏导致无法查看。因此，尾数部分实际表示的数值为 1*.M*(计算机中实际存储的是 *M*)，表示范围比实际存储的数值要大。根据以上规定，浮点数 *N* 最终可以表示为公式([2.1](#_bookmark19))：

*N* = ( 1)*S*  *M* 2*E* (2.1)

– · ·

2(*P*−1)

* + 1. 舍入规则

不仅如此，IEEE 754 浮点数标准还规定了浮点数运算中的数值舍入规则。在计算机所使用的浮点数系统中，浮点数之间的运算操作或者函数计算往往得到的结果并不一定是一个浮点数，因此在计算机所使用的浮点数系统中往往不能直接存储中间运算操作所得到的结果，计算机需要先对此结果进行舍入操作，这就使得输出的数值与原来的数值相比缺失一定的精度。

浮点数舍入指的是一种计算机操作，它是将需要被存储进计算机的浮点数按一定规则进行判断并截取计算机能够存储的部分。总而言之，就是将一个准确数值 (无限精度) 转化成此准确值附近的两个估算数值 (有限精度) 之一，使其能够被计算机使用。这个操作要么是截去了所规定的精度范围以外的其它有效数字，要么是在计算机能存储的有效数字位的下一位往前进一位。在 IEEE 754 浮点数标准中是这样定义舍入模式的概念的：一个浮点数的数值结果被舍入到一个有限精度的浮点数的方式是由浮点数的舍入方向来规定的。因而这就产生了一种舍入模式，常用“◦”表示。比如，当要求计算两个浮点数 *a* 和 *b* 的和 *a* + *b*时，计算机返回的计算结果实际是在进行舍入操作后的值◦(*a* + *b*)。在IEEE 754浮点数标准中，存在很多不同种类的舍入模式 [[20](#_bookmark175)]，其中部分舍入模式的具体定义如下：

第一种是舍入到最接近：舍入为最接近浮点数 *x* 的浮点数。在此情况下，需要遵循一种特殊的原则：若 *x* 值恰好落在两个连续浮点数的正中间时，我们需要选择舍入到哪一个的浮点数作为结果，此时需要采用“Ties to Even”原则(“就近取偶”原则，这是此情况下计算机默认的舍入方式)，也就是说浮点数 *x* 的舍入结果返回的是这两个连续浮点数中整型的有效数字是偶数的那个浮点数 (在二进制式中以 0 结尾) [[21](#_bookmark176)]；第二种是朝零方向舍入：舍入为最接近浮点数 *x* 的浮点数，但与第一种舍入模式不同的是，舍入后浮点数的绝对值要小于或等于

原浮点数的绝对值[[22](#_bookmark177)]；第三种是朝正无穷方向舍入：舍入为大于或等于浮点数 *x* 的最小浮点数，此浮点数可能是正无穷[[23](#_bookmark178)]；第四种是朝负无穷方向舍入：舍入为小于或等于浮点数 *x* 的最大浮点数，此浮点数可能是负无穷 [[24](#_bookmark179)]。

由于计算机的存储问题，我们需要对浮点运算中的浮点数进行舍入操作。舍入操作的本质是将一个浮点数的准确值变为近似值，而近似值与准确值之间必定存在一定的误差。并且，这种误差会随着舍入操作而累积，最终使得浮点运算的结果完全不准确 [[25](#_bookmark180)]。

## 训练数据生成技术

* + 1. **Herbie** 工具

针对训练数据中浮点程序数值用例的确定，我们选取了Herbie 开源工具[[26](#_bookmark181)]， Herbie 工具主要提供检测函数浮点误差并寻找替代函数的功能，该工具可自动尝试查找可减少舍入误差的替代表达式以提高原浮点函数准确性。Herbie 工具主要使用一系列启发式搜索技术来查找可减少错误的重写，它使用一系列浮点值 (采样点) 测试表达式，并凭经验确定错误。它可以找到在不同数值范围内具有良好效果的重写，并将它们组合在一起，最终生成能够产生良好效果的单个结果。

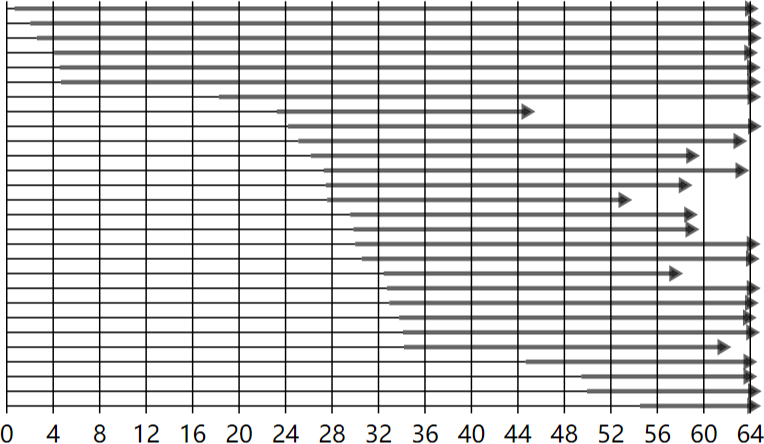


图 2.2: Herbie 工具测试结果

开发人员将 Herbie 工具应用到 Richard Hamming 的“科学家和工程师的数

值方法”的 28 个测试案例中，得到了令人欣喜的结果：其中大部分测试用例经过Herbie 工具的函数替换，准确性均有所提高，如图 [2.2](#_bookmark23)所示。图中每个箭头起点表示原始函数的精度，终点表示 Herbie 工具替换后函数的精度。每个箭头表示由于 Herbie 工具替换函数导致的准确性的提高。最高约提高了 64 位。

**Average Error：**

**30.1 → 0.2**

**Times： 3.9s**

**Precision： binary64**

图 2.3: Herbie 工具误差改进结果

下面将以函数 *f* (*x*) = √*x* + 1 − √*x* 为例，具体描述 Herbie 工具所做的工作。 Herbie 工具首先经过一系列的数学转换，先将函数 *f* (*x*) = √*x* + 1 − √*x* 转换为函数 *f* (*x*) = ( √*x* + 1 · √*x* + 1 − √*x* · √*x*)*/*( √*x* + 1 + √*x*)，进而准换为最终函数

√ √

*f* (*x*) = 1*/*( 1 + *x* + *x*)，通过将最初的浮点函数减法运算变为除法运算，我们可以看到该函数的平均误差从 30.1 降低为 0.2，如图 [2.3](#_bookmark24)所示。其中 Times 表示 Herbie 工具改进函数所需的时间，Precision 表示该浮点数运算所假定的精度为双精度 IEEE 754 算法。

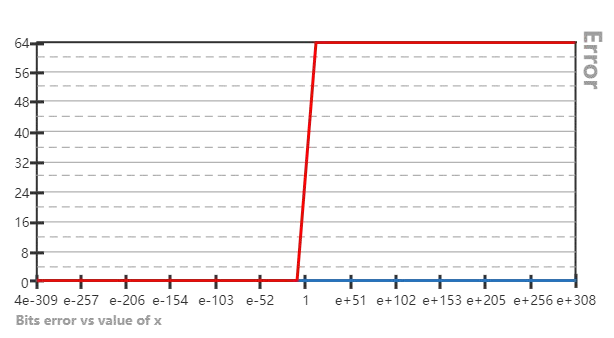


图 2.4: 浮点函数计算值和真实值的对比

再结合浮点函数计算值和真实值的对比图像，如图 [2.4](#_bookmark25)所示，我们就可以直

观地找出能触发原函数大误差的区间。图 [2.4](#_bookmark25)是浮点数误差与输入值的关系图，用红线显示原始函数的误差，用蓝线显示替换函数的误差。两者可以通过按钮切换。如果表达式具有多个变量，则左下角的变量选择器选择将哪个变量放置在水平轴上。图中就将变量 *x* 作为水平轴。此图有助于了解 Herbie 工具为改善其准确性进行的各种输入。有时，Herbie 工具会以某些你更关心的其他输入的准确性为代价来提高某些输入的准确性。这种情况下，你可以在运行工具前添加前提条件，来将 Herbie 工具限制为你关心的某些输入。因此我们可以通过 Herbie工具进行具有浮点运算误差的函数及其数值用例的搜集，但是这些数值用例并不一定能够触发浮点程序的大误差，因此我们还需要借助其他技术对已生成的数值用例的正确性进行验证。

* + 1. **iRRAM**

iRRAM 是基于 Real-RAM 概念的C++ 软件包，用于无错误的实数运算 [[27](#_bookmark182)]。在 iRRAM 之前，存在很多基于从三角函数的普通算术到线性代数甚至微分方程式不等的实数运算实现，例如对线性代数变换的研究 [[28](#_bookmark183)]、对迭代运算的精度研究 [[29](#_bookmark184)]。最终，基于交互式的 Real-RAM 概念 [[30](#_bookmark185)]，iRRAM 技术被最终提出并投入使用。

iRRAM 的程序使用普通的C++ 进行编码，此过程中需要使用特殊的类 RE- AL，其行为类似于实数，而没有任何错误。它允许将非常小的一组操作直接用于此类型的变量：如常规算术运算，测试以及与整数或其他类型变量的相互转换。在它们之上，程序员几乎可以使用 C++ 中的所有编程方法，例如定义自己的数据类型。

f\_origin(const &x){

return sqrt(x + 1) - sqrt(x);

}

REAL f\_real(const REAL &x) {

return REAL(1.0) / (REAL(sqrt(x + 1)) + REAL(sqrt(x)));

}

//实际函数

//误差函数

图 2.5: 函数 *f* (*x*) = √*x* + 1 − √*x* 及其误差函数

例如，对于函数 *f* (*x*) = √*x* + 1 − √*x*，我们可以编写出它的原函数和误差函数，如图 [2.5](#_bookmark27)所示。其中 f\_origin 是原函数，f\_real 是误差函数。通过使用 iRRAM包中的 REAL 类，我们可以编写如图所示的函数，通过上述两个函数，我们可以计算获得原函数的计算值和精确值，根据获取到的原函数的计算值和精确值，

我们可以对 Heribe 工具生成的数值用例的准确性进行验证，从而提高用于神经网络模型训练的数据集的准确性和可靠性。

* 1. **LLVM** 与符号执行技术
     1. **LLVM** 架构

LLVM 是构建传统编译器的框架系统。LLVM 源码主要使用的编写语言为 C++。LLVM 计划启动于 2000 年，经过数年来的一系列更新，目前LLVM 编译器已经被各大公司采用，如Apple、Microsoft、Google、Facebook 等。LLVM 编译器主要用于优化各类程序的编译过程、链接过程、运行过程以及空闲过程，以节约整个编译所花费的时间。目前，LLVM 编译器对所有用户保持开放，同时对已有脚本保持兼容。

**源代码 **  **机器码**

|  |  |  |  |
| --- | --- | --- | --- |
| 前端 | 优化器 | 后端 |  |
|  |

图 2.6: 传统编译器架构

传统的编译器架构分为三阶段，如图 [2.6](#_bookmark30)所示。用户需将源程序代码投入编译器进行编译，其中前端主要进行源程序代码的解释工作，通过词法、语法、语义分析来生成代码的中间表示，优化器则需要对此中间表示进行一系列的优化操作，最终，后端将优化后的中间表示转换为有实际意义的机器码并输出至对应平台。

**C X86**

GHC

Frontend

llvm-gcc Frontend

LLVM PowerPC Backend

Clang C/C++/ObjC Frontend

Other Frontend

LLVM ARM

Backend

LLVM X86

Backend

LLVM

Optimizer

LLVM Other Backend

**Fortran PowerPC**

**Haskell ARM**

**Other Other**

图 2.7: llvm 编译器架构

LLVM 编译器也分为三个阶段，但在设计上与传统编译器略有不同 [[31](#_bookmark186)]： LLVM 编译器对于不同语言编写的代码编译均提供了同一种中间代码的表示。如图 [2.7](#_bookmark31)所示。其中不同语言编写的代码需使用不同的编译工具进行编译，但是编译完成后，会生成统一的中间代码表示 LLVM Intermediate Representation(LLVM IR)。此种编译方式的好处是：如果需要编译一种新编程语言编写的代码，实现方面只需要构建一个对应的新前端即可；中间的优化器进行对中间代码表示 IR的优化处理操作，由于优化器处理的 LLVM IR 具有统一的表示，因此不论构建了新前端，还是构建了新后端，我们都不需要对优化器进行修改；最后，对应后端将优化好的中间代码转换为对应的机器码，出现新硬件的处理方式与前端处理方式相同。

下面以 C++ 语言编写的代码文件为例，详细叙述 LLVM 具体的编译过程：首先对源代码进行预处理，接着通过词法分析生成 Token，再经过语法分析的过程生成抽象语法树 (AST)，AST 经过处理生成统一的中间代码IR，对IR 进行进一步的优化生成汇编代码，最后通过 Link 生成目标文件。

在 LLVM 编译过程中，最为关键的就是统一表示的 IR。此 IR 主要具有三种不同格式，第一种是可读格式，这种情况下 IR 与汇编语言相类似，这种可读格式的IR 在计算机中存储为.ll 文件，主要是为了方便编程人员查阅；第二种是不可读格式，此情况下，IR 虽使用二进制表示，却无法被编程人员查阅。这种中间代码在计算机中存储为.bc 文件；第三种是内存格式，此情况下，IR 存在于内存中，因此可以加快LLVM 编译器的编译速度。这三种格式虽表现形式不同，但在内容上是完全等价的，因此我们在本文中主要会使用可读IR 和不可读二进制 IR 这两种格式。

* + 1. **LLVM** 命令

通过对 LLVM 源码进行修改编译，我们得到可用的 LLVM 工具，编译过程命令如图 [2.8](#_bookmark33)所示。第一条命令表示进入源码所在的文件夹。第二条命令表示根据 LLVM 源码创建 makefile 文件。最后一条命令表示根据 makefile 文件编译源码并构建可用 LLVM 工具。

cd LLVM\_Source/

cmake /xxx/llvm cmake --build .

//进入待编译的 LLVM 源码文件夹

//创建对应的makefile 文件

//对 LLVM 源码进行编译

图 2.8: llvm 编译命令

我们在本文中主要会使用可读 IR 和不可读二进制 IR 这两种格式，常用的

llvm 命令如图 [2.9](#_bookmark35)所示。第一条命令表示获取对应代码的可读 IR。第二条命令表示获取不可读的二进制码IR。第三条命令表示使用编译好的lli 工具将二进制 IR 转换为可用的二进制码形式。

clang -O0 -emit-llvm -S xxx.cpp -o xxx.ll

clang -O3 -emit-llvm xxx.cpp -c -o xxx.bc

//将C++源码转换为可读的中间代码IR

//将C++源码转换为不可读的二进制IR

bin/lli xxx.bc //通过编译好的 lli 工具将二进制 IR 转换为最终的二进制码形式

图 2.9: llvm 工具常用命令

* + 1. 符号执行技术

本文系统通过在 LLVM 平台上引入符号执行技术来实现数值软件的全局用例生成。符号执行是一种动静态相结合的程序分析技术，近年来受到学术界与工业界的广泛关注 [[32](#_bookmark187)–[34](#_bookmark188)]。该技术的基本思想是将程序输入定义成可代表任意值的符号，并基于这些符号推导程序执行过程中的各中间数据和程序的执行结果，即使用符号值替代具体值来模拟程序的执行，其中的核心步骤是变量的符号化和程序执行的模拟。变量的符号化是指用一个符号值表示程序中的变量，例

* + - 1. makeSymbolic(); //将关键变量符号化
      2. PathSet = {InitPath}; //用于存储已经发现的路径，初始路径指向程序开始位置
      3. while PathSet not empty
      4. Path = PathSet.selectPath();
      5. Instruction = Path.nextInstruction;
      6. if Instruction.type = ArithmeticOperation or LogicOperation or BitOperation
      7. //算术运算、逻辑运算、位运算等
      8. ResultValue = makeExpression(Operator, Operand1, Operand2, …);
      9. else if Instruction.type = Branch //分支指令
      10. NewPath = Path.fork();
      11. Path.addPathConstraint(Branch.getTrueCondition());
      12. NewPath.addPathConstraint(Branch.getFalseCondition());
      13. PathSet.add(NewPath);
      14. else if Instruction.type = EndOfProgram
      15. PathSet.remove(Path);
      16. solvePathConstraint(Path); //求解约束条件
      17. else if Instruction.error
      18. PathSet.remove(Path);
      19. solvePathConstraint(Path);
      20. else //其它指令
      21. executeInstruction(Instruction);
      22. SymbolicExecutionCompleted();

图 2.10: 一种符号执行的基本算法流程

如本文系统中主要将浮点变量进行符号化。在后续的执行过程中，所有与被符号化的变量相关的变量取值均被表示成了输入符号的表达式。而在模拟程序执行的过程中，由于符号执行使用符号值替代具体值，所以无法直接计算得到一个明确的结果，而是使用符号表达式的形式来表示变量的值。符号表达式包括操作符和操作数，操作数既可以是常量也可以是另一个符号表达式。对于分支语句，每当遇到分支语句，原先的一条路径就会分裂成多条路径，分裂出来的各条路径除约束条件和下一条指令外完全相同。每条路径分别进入该分支条件下的不同分支，即从当前执行状态中复制出一个新的执行状态，并在新状态中记录条件为假的分支条件约束，而当前的执行状态会被记录为条件为真的分支条件约束而继续推导。最终当某一个执行状态的推导到达程序出口时，符号执行平台通过约束求解器求解当前状态的条件约束，并生成覆盖对应路径的用例。

图 [2.10](#_bookmark36)描述了一种典型的符号执行算法，它首先将程序中的关键变量符号化，并将初始路径并加入到路径集合中（1-2 行）。初始路径在初始状态下没有任何变量值，仅将执行指令位置设置为程序开始位置，并不断地选择路径执行指令直到程序末尾。当执行遇到运算语句时自动记录符号运算表达式来表示结果（6-8 行）。每当遇到分支语句则克隆路径状态，并分别将分支条件为真和为假的条件加入到约束条件中，并以不同的状态进入到不同的分支中（9-13 行）。当某条路径在执行过程中出现错误导致无法继续执行时，需要将其从路径集中删除并求解进入该路径的约束条件（17-19 行）。当某条路径执行到程序结尾时，同样需要将其从路径集中删除并求解出进入该路径的约束条件（14-16 行）。当所有路径都执行到程序末尾或发现错误，且没有新的路径被发现时，则符号执行过程全部结束（22 行）。

本文利用符号执行的模拟过程来推导符号运算表达式，并以此来链接浮点程序不同表达式的计算过程，构建软件全局的浮点计算逻辑。由于符号执行会推导执行过程中不同执行路径下各变量的符号表达式，该技术可以很好地将程序中前后浮点表达式形成的全局语义表达出来，从而帮助我们实现面向全局的浮点程序用例生成。

## 深度学习与神经网络

* + 1. 深度学习

机器学习(Machine Learning) 学科主要负责对已有数据进行分析，获得数据之间存在的规律，并利用此规律预测未知信息。而深度学习 (Deep Learning) 是属于该学科中的一门分支学科，它被引入机器学习领域，来使机器学习更专注于对人工智能 (AI) 的研究 [[35](#_bookmark189), [36](#_bookmark190)]。

深度学习主要负责建立像人脑一样的神经网络，通过神经网络来学习已有数据集中存在的规律，从而有效找出文字、图像等未知数据中包含的信息。其本质上属于一种机器学习算法，该算法使得计算机能够像人类一样识别各种不同的数据。目前，深度学习技术被广泛运用于机器翻译、数据挖掘以及平台个性化推荐等领域中，取得了一定的成果。通过训练机器，让机器模仿人类的行为，深度学习解决了很多以前无法解决的难题 [[37](#_bookmark191), [38](#_bookmark192)]，这在某种程度上也意味着人工智能正逐步走向我们的生活 [[39](#_bookmark193)]。

* + 1. 神经网络

深度学习研究最初主要基于人工神经网络 [[40](#_bookmark194)]。随着深度学习技术的发展，人工神经网络被直接称为神经网络。神经网络 (Neural Networks) 由大量简单的神经元组成，这些神经元之间互相连接形成一种网络结构，这种网络结构类似于人脑，因此能够进行类似人脑的学习和识别工作。神经网络模型大致可以分为两类：有监督学习神经网络模型和无监督学习神经网络模型 [[41](#_bookmark195)]。本文主要利用有监督学习神经网络模型，这种模型又主要分为两种：循环网络 (RNN) 和卷积网络 (CNN)。

RNN 循环网络[[42](#_bookmark196)] 是一种递归神经网络，即在其模型中存在循环操作，数据可以通过神经网络回到初始的位置，如图 [2.11](#_bookmark40)所示。这种神经网络模型主要用来处理序列数据，序列数据是指序列中的前后数据之间存在联系，而非一个个独立的个体。在此模型下，前一个数据在隐藏层中生成的值 *ht*−1 会被存放到延迟器中，等到下一个数据进入隐藏层获取数值 *ht* 时，*ht* 会受到之前存放于延迟器中的 *ht*−1 的影响，这种处理方式类似于将不同时刻的隐藏层进行连接，以达到处理序列数据的作用。

ht Xt



ht-1

输入层

ht

隐藏层

延迟器

输出层

图 2.11: RNN 神经网络模型

而 CNN 卷积网络与普通神经网络极为相似，主要由三部分构成[[43](#_bookmark197)]。第一部分是输入层，主要负责对数据进行重定型处理。第二部分包含多个卷积层和池化层，卷积层主要负责提取输入数据的特征，池化层主要负责将特征进行细

化。第三部分为全连接层，主要负责将细化后的特征重新组合变为可用特征。具体结构如图 [2.12](#_bookmark41)所示。本文中主要使用 CNN 神经网络模型, 主要原因是 CNN 可以将输入数据的特征提取出来，这在本文中有利于得到浮点程序和数值用例之间的关系特征。

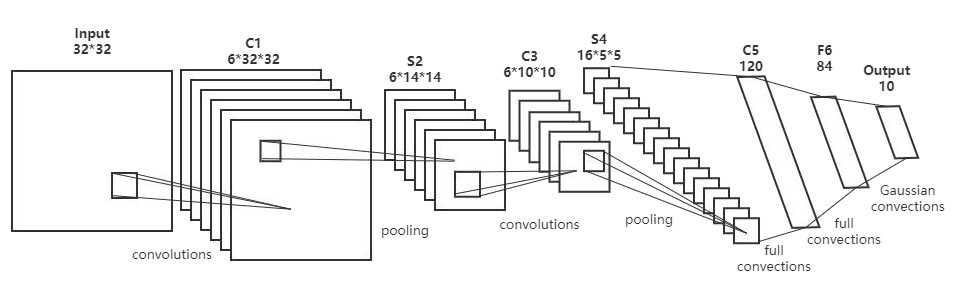


图 2.12: CNN 神经网络模型

CNN 中，卷积层与池化层负责提取数据的特征。在卷积层中，包含多个由神经元组成的特征平面，这些平面内部的神经元相互连接，同时共享权值(卷积核)。卷积核初始为一个数值矩阵，经过数据集训练后，卷积核将逐步得到完善并趋于稳定。卷积核的共享有利于减少卷积层内部特征平面间的连接，同时能降低过拟合出现的概率。池化 (pooling) 层通常采用两种采样方式，分别为均值采样(mean pooling) 和最大值采样(max pooling)。池化实际上也可以看作卷积，但两者之间略有不同。这两层所做的操作大大降低了模型的复杂度，使得相关人员无需关注过多的参数问题。同时，CNN 可以被看作是过滤效率逐步提高的过滤器：信息需穿过多层过滤网，同时过滤器会根据训练数据来提高过滤的效率，最终获得的信息纯度会越来越高，对于任务的帮助也就越来越高。

## 本章小结

在本章中，主要介绍了面向浮点程序误差检测的用例生成系统中需使用到相关技术与工具。在本章节中，介绍了浮点数的相关知识、提供浮点程序数值用例的Herbie 工具、提供数值用例验证的 iRRAM、提供浮点程序编译的 LLVM编译器以及负责深度学习的神经网络模型。

对于浮点数的介绍，本文先给出了浮点数的一般性定义，由此引出浮点数运算标准：IEEE 754 标准。然后，对IEEE 754 标准中浮点数格式以及舍入规则进行了详细的描述。接着，根据 IEEE 754 标准中浮点数格式重新定义了浮点数，并给出了浮点运算存在的问题。

对于训练数据的生成，本文在第二节中给出了训练数据中数值用例的生成

技术。主要通过给出具体浮点程序的例子，详细介绍了获取浮点程序的数值用例的工具：Herbie 与验证该数值用例的 iRRAM 技术。对于浮点程序的编译，本文则详细介绍了 LLVM 编译器的整体架构以及运行流程，并给出了详细步骤的命令。

最后，对于系统所使用的深度学习和神经网络模型技术作出详细介绍，通过并给出各种不同神经网络模型的架构组成，阐述了不同神经网络模型的优势所在。

# 第三章 浮点用例生成系统的需求分析

本章将总体概述面向浮点程序误差检测的用例生成系统的需求分析。首先，我们将概述本系统的开发目标与总体工作流程；然后，我们将具体给出本系统的功能性需求并根据模块给出对应用例图；最后，我们将分析系统并给出其具体的非功能性需求。

## 浮点用例生成系统的总体规划

现有的用例生成系统主要是基于国内外一些检测浮点运算误差的理论和算法，这些理论和算法能够帮助行业内的部分软件工程师发现并解决一些与之相关的问题。但是这些理论与算法大多针对的是单一的浮点运算表达式，不能很好地进行浮点程序全局的用例生成；同时现有系统生成的用例是通过随机选取生成的，需要生成大量用例才能获取那些能触发浮点程序明显误差累积的有效数值用例，系统生成有效用例的整体效率较低。针对传统用例生成系统的这两点问题，本文设计并实现了面向浮点程序误差检测的用例生成系统，结合自主设计的浮点程序转换规则和神经网络模型，在提供尽可能多的训练数据集的情况下进行训练，使得用例生成系统生成的大部分用例都能够用于浮点程序全局的误差检测。

中间代码



修改过的LLVM编译器

浮点程序

数值用例

卷积神经网络模型

训练数据集

|  |  |
| --- | --- |
| 中间代码转换算法 | |
|  | 二进制码 |

用户

程序员

训练结果

图 3.1: 用例生成系统整体流程

图 [3.1](#_bookmark45)展示了本系统基于自主设计的转换算法和神经网络模型进行面向浮点程序误差检测的用例生成的整体流程。本系统主要分为三部分：修改过的 LLVM

编译器，自主设计规则的转换算法以及卷积神经网络模型。用户向系统提交需要生成准确用例的浮点程序，首先由被修改过的 LLVM 工具对此浮点程序进行处理，得到此浮点程序的中间代码 IR，并将中间代码信息存储进数据库，中间代码 IR 的形式为汇编码 (最初的想法是通过 LLVM 编译器直接得到浮点程序的二进制码，但因为由原生 LLVM 编译器生成的二进制码存在相似以及冗余等问题，选择了当前处理方法，具体情况在后续章节详细介绍)。其次，根据自定义的转换算法，系统对中间代码IR 进行分析处理，生成可用的二进制码，并存储进数据库。然后，程序员将训练数据集投入神经网络模型进行迭代，生成可用的神经网络模型。最后，将之前生成的二进制码投入训练好的卷积神经网络模型，得到能触发该浮点程序明显误差累积的数值用例。

由于自定义转换算法的实现，使得二进制码中减少了很多与浮点程序无关的信息，提高了训练结果的准确性；同时大量不同的训练数据集保证了模型的训练效果，使得用例生成系统生成的用例能够用于完整浮点程序的误差检测，扩大了系统的适用范围。

## 浮点用例生成系统的功能性需求

需求分析是软件开发中的关键过程，同时也是系统开发的根本依据。本节将分模块明确本系统的功能性需求，同时使用相关用例来描述各模块对应的功能性需求。根据系统各个部分的职责以及具体需求的不同，本系统主要分为程序预处理模块、中间代码转换模块、训练数据集生成模块、神经网络训练模块以及数值用例生成模块这五个模块。系统的功能性需求主要这五个模块进行分析。主要的功能性需求如表 [3.2](#_bookmark47)所示。本系统中，各个模块在功能性需求上存在一定的联系，但是各模块的分工明确，模块与模块之间需求区分非常明显。

在程序预处理模块中，系统主要的工作是对浮点程序代码进行预处理，使其转换成类似于汇编码的格式。首先系统会先对浮点程序代码进行词法分析；然后会对生成的 token 进行处理生成抽象语法树结构；最后系统根据抽象语法树，将其转换为类似于汇编码的中间代码 IR，并存储到数据库中。

在中间代码转换模块中，我们的主要工作是获取对应程序的二进制码。首先本系统需要自行设计具体的中间代码转换算法；然后系统需要根据此转换算法，将预处理完生成的中间代码 IR 转换成可供系统使用的二进制码，并二进制码信息存储到数据库中。

在训练数据集生成模块中，系统的主要工作是生成并保存准确的神经网络模型训练数据集。首先本系统利用 Herbie 网站工具，得到大量不同的浮点程序以及其数值用例；然后通过使用 iRRAM 包中的REAL 类，编写各测试函数的误

面向浮点程序误差检测的用例生成系统

中间代码转换

训练数据集生成

神经网络训练

数值用例生成

数值用例生成

程序预处理

中间代码生成

抽象语法树生成

词法分析

神经网络调整

训练数据验证

中间代码转二进制码

迭代训练

训练结果评估

训练数据生成

转换算法设计

神经网络搭建

图 3.2: 用例生成系统的功能性需求划分

差函数，对由 Heribe 工具得出的浮点程序数值用例进行验证，提升测试数据的准确性和可靠性；最后将用例验证正确的浮点程序的二进制码以及其数值用例存入数据库的训练数据表中。

在神经网络训练模块中，系统主要需要获取神经网络模型，并保证其准确性。首先我们需要搭建一个用于用例生成的卷积神经网络模型；然后使用训练数据表中的训练数据集进行神经网络模型的迭代训练，训练完成后，我们需要对每次迭代的训练结果作出评估，并依据评估结果调整模型参数；调整完毕后之后继续进行迭代训练，最终得到可供使用的神经网络模型。

在输入区间用例生成模块中，我们会使用之前各模块的功能，生成能够触发浮点程序明显误差累积的数值用例。我们首先给出待处理的浮点程序；然后对其进行预处理和转换工作，使其变为可使用的二进制码；接着将二进制码投入训练好的神经网络模型，产生使得该浮点程序输出误差最大的数值用例。

* + 1. 程序预处理模块的需求

根据上述程序预处理模块功能性需求的相关分析，得到本模块的用例图，如图 [3.3](#_bookmark49)所示。该模块的主要用户为两种，分别是程序员与系统使用者，他们的用例主要为获取浮点程序中间代码，其中包括程序词法分析、程序语法分析、抽象语法树生成、中间代码IR 生成以及中间代码存储。接下来，我们将给出各用例的具体描述，并在最后给出本模块的具体功能性需求与需求描述。

获取浮点程序中间代码用例主要发生在系统使用者打算进行浮点程序的输

程序预处理模块

程序词法分析

程序语法分析

系统使用者

《包含》

《包含》

浮点程序中间代码获取 《包含》

抽象语法树生成

《包含》

《包含》

程序员

中间代码存储

中间代码IR生成

图 3.3: 浮点程序预处理模块用例图

入区间用例生成之时或程序员打算获取神经网络模型训练数据集之时。具体流程包括词法分析、语法分析、生成抽象语法树、生成中间代码IR 以及中间代码存储。

首先，用户给出需要进行处理的浮点程序本地路径；然后，系统在获取到该浮点程序时，首先对其进行词法分析处理，词法分析主要是将该浮点程序的代码拆分为一个个token，并将其存入文件中，每一个token 前有标识表明该token的具体类型，每一个 token 后存在数字表示此 token 在某一行中的具体位置；接着，系统会在之前词法分析的基础之上，将各token 组成一系列的语法短语，以此来判断源程序代码在结构上是否正确；然后，根据语法分析的结果，将组成的语法短语插入到树结构当中，形成可用的抽象语法树；最后，系统根据该抽象语法树，生成可读的中间代码 IR，并将其存入对应的数据库表中，其形式类似于汇编码，但有具体的格式区分。

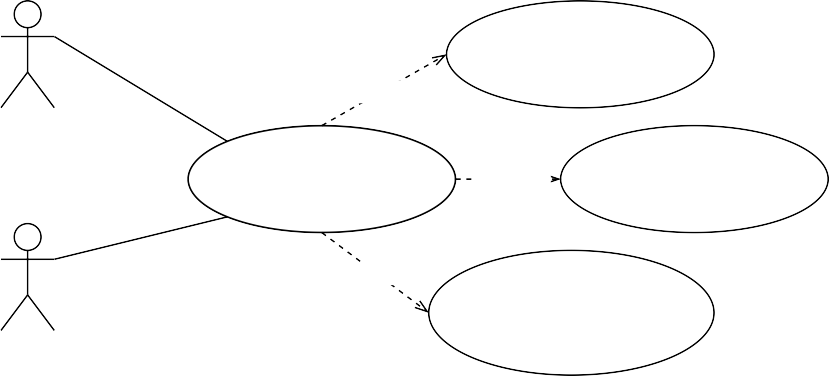
本模块的功能性需求主要包括词法分析、抽象语法树生成以及中间代码生成，他们的具体需求描述如表 [3.1](#_bookmark50)所示。词法分析是指通过系统可以得到待处理的浮点程序的 token，同时 token 包含其类型以及位置等信息。抽象语法树生成是指系统可以根据词法分析所得到的 token，进行 token 的组合，验证源程序代码的正确性，同时根据 token 的组合，生成抽象语法树来表示浮点程序代码的抽象语法结构。中间代码生成是指系统可以根据已生成的抽象语法树，对其进行语义分析，最终生成可读的中间代码 IR。

表 3.1: 程序预处理模块功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 词法分析 | 程序员与系统使用者可以通过系统得到待处理的浮点程序的 token，同时 token 包含其类型以及位  置等信息 |
| 抽象语法树生成 | 系统可以根据词法分析所得到的token，进行 token的组合，验证源程序代码的正确性，同时根据 token的组合，生成抽象语法树来表示浮点程序代码的抽  象语法结构 |
| 中间代码生成 | 系统可以根据已生成的抽象语法树，对其进行语义分析，来生成可读的中间代码 IR（此中间代码类似汇编语言的形式），并将其存入对应数据库表中，  方便后续模块进行中间代码转二进制码的工作 |

* + 1. 中间代码转换模块的需求

根据上述中间代码转换模块功能性需求的相关分析，得到本模块的用例图，如图 [3.4](#_bookmark52)所示。该模块的主要用户为两种，分别是程序员与系统使用者，他们的用例主要为中间代码 IR 转换，其中主要包括转换算法设计、中间代码到二进制码的转换以及二进制码的存储。接下来，我们将给出各用例的具体描述，并在最后给出本模块的具体功能性需求与需求描述。



中间代码转换模块

转换算法设计

《包含》

系统使用者

中间代码IR转换

《包含》

二进制码存储

《包含》

中间代码转二进制码

程序员

图 3.4: 中间代码转换模块用例图

中间代码IR 转换用例主要发生在系统使用者或程序员提供的浮点代码经过系统的预处理模块产生中间代码IR 之后。具体流程包括中间代码转换算法设计、

转换中间代码到二进制码以及二进制码存储。

首先，用户所提供的浮点程序代码已经经过了系统的预处理，生成了可用的中间代码 IR；然后，程序员可以设计具体的中间代码的转换算法：如一行中间代码转换为 64 位二进制码，中间代码 IR 中的操作数用 4 位或 8 位，具体的操作符用 4 位或 8 位，不同的操作符规定不同的表示方式，同时确定中间代码转换完成后具体的行数，不足用 0 补足等；最后，系统取出数据库中存储的中间代码 IR，并根据设计出来的中间代码转换算法，对其进行转换工作，将其转换为可用的二进制码，并将其存储入对应的数据库表中，方便其他相关模块的使用。

本模块的功能性需求主要包括设计转换算法以及中间代码转二进制码，他们的具体需求描述如表 [3.2](#_bookmark53)所示。转换算法设计是指程序员通过规定转换后的二进制码文件的行数与列数、规定操作数和操作符的具体二进制码表示等多个方面来设计具体的中间代码转换规则。中间代码转二进制码是指系统根据前一部分设计好的中间代码转换规则，将浮点程序中间代码转换为具体的二进制码，并存储到数据库中。

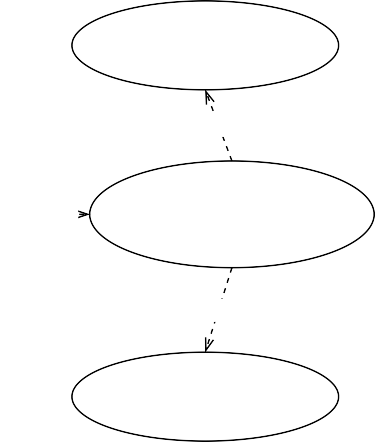
表 3.2: 中间代码转换模块功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 转换算法设计 | 程序员可以通过规定转换后的二进制码的行列数、  规定操作数和操作符的表示、不足补 0 等方面来设计具体的中间代码转换规则。 |
| 中间代码转二进制码 | 系统可以取出存储的中间代码 IR，根据前一部分设计好的中间代码转换规则，对其进行转化工作，  并将转化好的二进制码存储入数据库对应表中。 |

* + 1. 训练数据集生成模块的需求

根据上述训练数据集生成模块功能性需求的相关分析，得到本模块的用例图，如图 [3.5](#_bookmark55)所示。该模块的主要用户为两种，分别是程序员与系统使用者，他们的用例主要为生成训练数据集，主要包括生成浮点程序的二进制码，确定使浮点程序输出误差最大的数值用例以及对浮点程序的数值用例进行验证，其中训练数据集主要是用于神经网络模型的训练工作。接下来，我们将给出各用例的具体描述，并在最后给出本模块的具体功能性需求与需求描述。

训练数据生成用例发生在用户需要训练数据集来训练神经网络模型之时。具体流程包括通过系统生成浮点程序的二进制码，确定使浮点程序输出误差最大的具体数值用例以及验证浮点程序的数值用例。



训练数据集生成模块

浮点程序的数值用例获取

中间代码获取

系统使用者

《包含》

《包含》

训练数据生成

《包含》

生成浮点程序的

二进制码

《包含》

《包含》

程序员

浮点程序的数值用例验证

中间代码IR转化

图 3.5: 训练数据集生成模块用例图

首先，用户需要提供浮点程序的本地路径；然后，系统使用浮点程序预处理模块获取该浮点程序的中间代码，并存储入数据库表中；然后，系统根据获得的中间代码 IR，依据转换算法，将其转换为对应的二进制码，同时存储入数据库表中；接着，程序员使用 Herbie 工具来获取使浮点程序运算误差最大的数值用例，并借助 iRRAM 类中的REAL 包编写浮点程序的误差函数，来验证获取到的数值用例的正确性，若正确，系统则将这些相关信息存储入数据库中的训练数据表中。

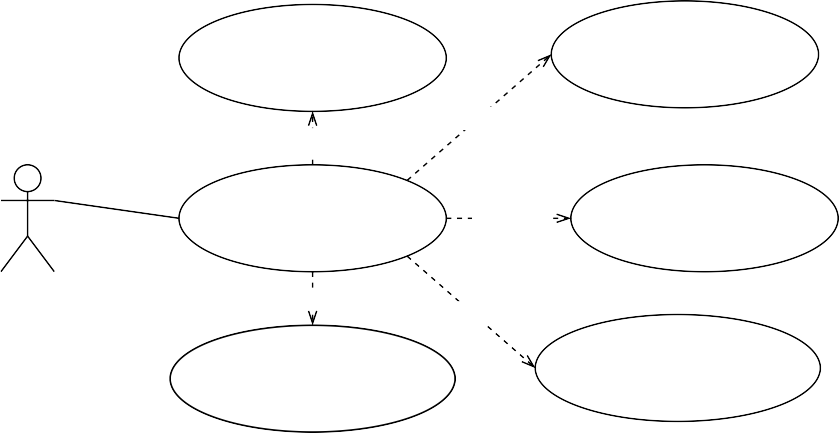
本模块的功能性需求主要包括训练数据生成以及训练数据验证，他们的具体需求描述如表 [3.3](#_bookmark57)所示。训练数据生成包括两部分：一是通过浮点程序预处理模块获取浮点程序的中间代码 IR，再通过中间代码转换模块获取浮点程序对应的二进制码；二是通过 Herbie 工具生成浮点程序对应的数值用例。训练数据验证是指借助 iRRAM 编写对应浮点程序的误差函数来验证获取到的数值用例的正确性，最终用于神经网络模型训练的训练数据主要由二进制码和数值用例两部分构**3.2**成**.4**。神经网络训练模块的需求

根据上述神经网络训练模块功能性需求的相关分析，得到本模块的用例图，如图 [3.6](#_bookmark58)所示。该模块的主要用户为程序员，其用例主要为获得可用的神经网络模型，主要包括搭建神经网络模型，训练神经网络模型，神经网络模型训练结果评估以及神经网络模型参数调整再训练。接下来，我们将给出各用例的具体描

表 3.3: 训练数据集生成模块功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 训练数据生成 | 程序员可以通过浮点程序预处理模块获取浮点程序的中间代码 IR，再通过中间代码转换模块获取对应的二进制码；同时程序员可以通过 Herbie 工  具生成对应浮点程序的数值用例。 |
| 训练数据验证 | 程序员可以借助 iRRAM 编写对应浮点程序的误差函数来验证数值用例的正确性，若正确，则将中间代码，二进制码以及具体数值用例存入数据库对应  的训练数据表中。 |

述，并在最后给出本模块的具体功能性需求与需求描述。



神经网络训练模块

CNN神经网络模型搭建

神经网络模型迭代训练

《包含》

《包含》

获得可用神经网络模型 《包含》

神经网络模型

训练结果评估

程序员

《包含》

《包含》

神经网络模型生成

神经网络模型参数调整

图 3.6: 神经网络训练模块用例图

获得可用的神经网络模型用例发生在程序员需要稳定可用的神经网络模型来生成使浮点程序输出误差最大的具体数值用例之时。具体流程包括 CNN 神经网络模型的搭建，CNN 神经网络模型的迭代训练，CNN 神经网络模型训练结果的评估以及评估后对 CNN 神经网络模型参数的调整，这种调整需持续到模型稳定可用为止。

首先，程序员先根据神经网络模型教程，搭建一个 CNN 神经网络模型，模型具有数据重定型、卷积层、全连接层以及输出层四部分；然后系统取出数据库中存储的训练数据集，包括浮点程序的二进制字节码以及其数值用例，投入到搭建好的 CNN 神经网络模型进行迭代训练；迭代训练主要包括训练完成后，系

统投入用于测试的浮点程序的二进制码，通过验证生成的数值用例是否落在能触发该浮点程序明显误差累积的区间上，评估此 CNN 神经网络模型的优劣；若不足，则对该 CNN 神经网络模型的参数进行修改，并重新投入训练数据集训练，并在训练完成后重复上述操作直到该 CNN 神经网络模型相对稳定可用。

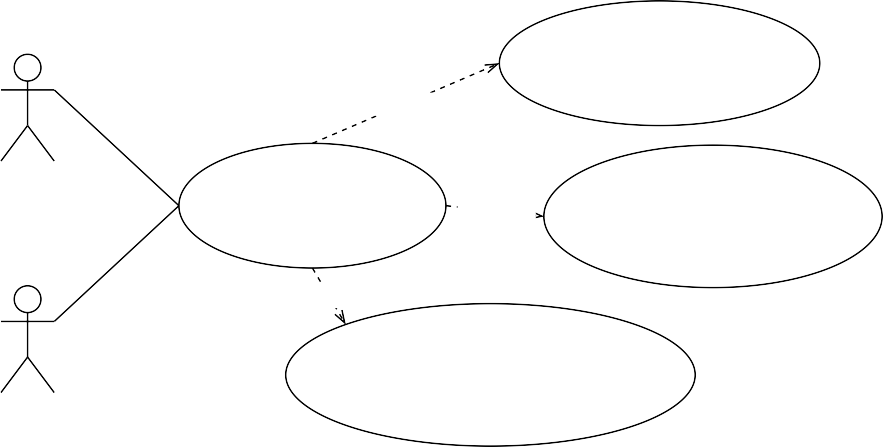
表 3.4: 神经网络训练模块功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 神经网络搭建 | 程序员通过神经网络教程，可以搭建出一个初始的  CNN 神经网络模型。 |
| 迭代训练 | 系统取出数据库中的训练数据，投入到初始 CNN  神经网络模型进行训练。此训练过程需要被重复多次。 |
| 训练结果评估 | 程序员向训练完成的 CNN 神经网络模型中投入用于测试的浮点程序的二进制码，可以得到对应程序的数值用例；程序员还可以通过验证此用例是否落在能触发该浮点程序明显误差累积的区间中来评估此 CNN 神经网络模型的优劣，若训练结果  满意则生成可用的模型。 |
| 神经网络调整 | 程序员可以调整 CNN 神经网络模型的参数，并再  训练。 |

本模块的功能性需求主要包括搭建神经网络、迭代训练、训练结果评估以及调整神经网络，他们的具体需求描述如表 [3.4](#_bookmark59)所示。神经网络搭建是指搭建出一个初始的用于用例生成的 CNN 神经网络模型。迭代训练是指将数据库表中的数据取出，投入到初始的 CNN 神经网络模型进行训练，此过程需要重复多次。训练结果评估是指投入用于测试的浮点程序的二进制码，获取浮点程序对应的数值用例，并通过验证此数值用例是否落在能触发该浮点程序明显误差累积的区间中来评估此 CNN 神经网络模型的优劣，此过程同样需要重复多次。神经网络调整是指调整模型参数重新训练。

**3.2.5** 数值用例生成模块的需求

根据上述数值用例生成模块功能性需求的相关分析，得到本模块的用例图，如图 [3.7](#_bookmark61)所示。该模块的主要用户为两种，分别是程序员与系统使用者，其用例主要为数值用例生成，主要包括生成浮点程序的中间代码 IR，生成浮点程序的二进制码以及利用神经网络模型生成数值用例。接下来，我们将给出各用例的具体描述，并在最后给出本模块的具体功能性需求与需求描述。



数值用例生成模块

生成浮点程序的中间代码IR

《包含》

系统使用者

数值用例生成

《包含》

生成浮点程序的二进制码

《包含》

程序员

用于浮点程序误差检测

的数值用例生成

图 3.7: 数值用例生成模块用例图

数值用例生成用例发生在用户需要获取用于某浮点程序误差检测的数值用例之时。具体流程包括通过系统生成浮点程序的中间代码 IR，通过系统生成浮点程序的二进制码，最终利用训练好的神经网络模型获得数值用例。

首先，用户需要给出待处理的浮点程序；然后，系统会通过预处理模块对此浮点程序进行处理，经过词法分析、语法分析、抽象语法树生成等步骤，最终转换为类似于汇编语言的可读中间代码 IR，并存储进数据库的中间代码表中；接着，系统通过转换模块对该可读中间代码 IR 进行处理，根据事先设定的转换算法，将此可读中间代码 IR 转换为可用的二进制码，并存储进数据库的二进制码表中；最后将此可用的二进制码投入已经训练好的 CNN 神经网络模型中，获得用于此浮点程序误差检测的具体数值用例。

本模块的功能性需求为数值用例生成，其具体需求描述如表 [3.5](#_bookmark62)所示。数值用例生成是指用户可以通过系统得到浮点程序的中间代码 IR，以及该浮点程序的二进制码，最终基于训练完成的神经网络模型获得用于该浮点程序误差检测的数值用例。

表 3.5: 数值用例生成模块功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 数值用例生成 | 用户可以通过系统得到浮点程序的中间代码IR，以  及对应的二进制码，同时用户可以得到用于该浮点程序误差检测的数值用例。 |

## 浮点用例生成系统的非功能性需求

本系统除了需要满足上述各模块的功能性需求外，还需要在可用性、易用性、可靠性、可拓展性以及性能方面有一定的要求。经过仔细的分析讨论，本系统的非功能性需求如表 [3.6](#_bookmark64)所示。

表 3.6: 用例生成系统的非功能性需求列表

|  |  |
| --- | --- |
| 需求名称 | 需求描述 |
| 可用性 | 本系统应保证在遇到断电等意外情况时，在电力恢复的  15 分钟之内重新启动并恢复服务 |
| 易用性 | 1、本系统该保证新用户可以在 15 分钟之内熟练使用系统的所有功能  2、本系统应保证所有的用户操作步骤均不会超过 5 步 |
| 可靠性 | 1、本系统应保证对于输入的源程序进行检查，保证输入源程序的正确性  2、本系统遇到自身故障时，应保证 1 个小时之内可以恢复系统的正常运转  3、本系统的自身故障发生率应小于 2% |
| 可拓展性 | 本系统类似组件应保持设计一致性，保证简单修改即可  立即使用。 |
| 性能 | 1、本系统的主要功能响应时间不应超过 3 秒  2、本系统的神经网络模型训练时间应不超过 12 小时  3、本系统应能保证多个任务同时运行  4、本系统得出的用于浮点程序误差检测的数值用例落在实际区间上的比例应高于 80% |

由于用例生成系统的主要目的是避免传统浮点运算误差检测工具的高耗能，因此我们需要将系统设置的尽量简单易用：系统中各功能少量的操作步骤能让刚接触到本系统的新用户很快摸索清楚具体的操作流程。同时本系统会对用户输入的源程序进行检查，大大避免了由于用户提供的源程序有误而导致的系统问题。不仅如此，本系统对于神经网络模型进行迭代训练，大大增加了训练后神经网络模型的准确性，也就提高了输出结果的准确性。

## 本章小结

本章首先通过对面向浮点程序误差检测的用例生成系统的整体流程进行描述，阐述了本系统的功能性需求。然后通过对各模块中具体包含的用例的描述，

详细说明个各模块的功能性需求。最后通过对系统目的和功能进行分析，阐述了本系统的非功能性需求。

# 第四章 浮点用例生成系统的设计

本章基于第三章对于系统五个模块的功能性需求、相关用例以及系统整体非功能性需求的描述，首先将对本系统进行整体性设计，并结合四视图进一步说明系统的概要设计；接着分系统模块通过流程图、时序图以及详细类图阐述本系统各模块的详细设计；最后给出本系统的数据库设计。

## 浮点用例生成系统的概要设计

根据上一章需求分析得到的功能性需求、用例描述以及非功能性需求，本节将详细阐述本系统的总体概要设计。首先，通过对本系统的架构进行设计，展示本系统的整体框架与结构层次；最后，通过四个系统视图：逻辑视图、进程视图、开发视图以及物理视图的角度，进一步对详细阐述本系统的总体概要设计。

* + 1. 系统架构设计

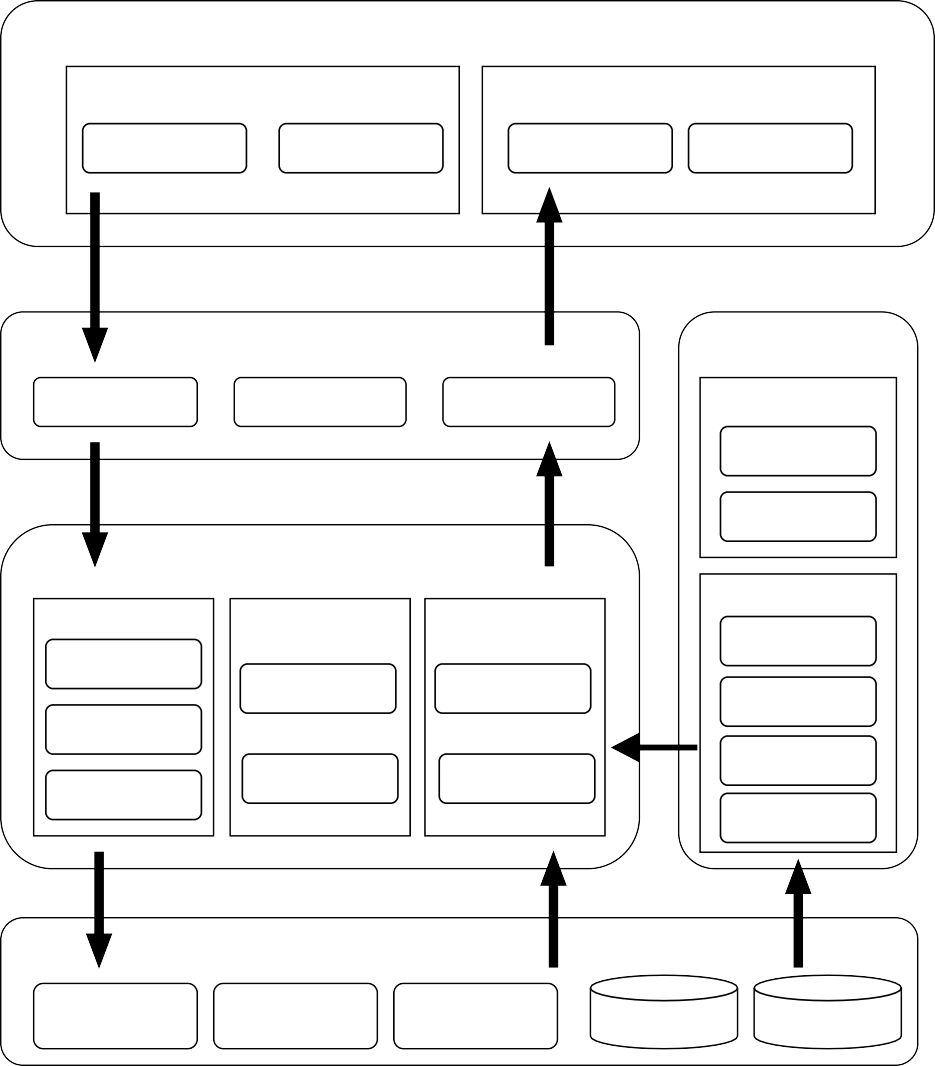
图 [4.1](#_bookmark69)描述了用例生成系统系统的总体架构图，包括与系统用户直接进行交互的交互展示层、系统主要服务的接入层、系统主要功能实现的核心业务层、为核心业务数值用例生成提供深度学习模型的神经网络模块以及为上述所有功能模块提供数据支持的基础数据层。

本系统主要负责面向浮点程序误差检测的数值用例生成功能，其中包括数值用例生成任务的创建以及调度分发、对浮点程序进行预处理得到中间代码、对中间代码进行转换得到二进制码、模型训练数据集生成、深度学习模型生成、数值用例生成以及该任务耗时计算等相关功能。

交互展示层中，系统使用者首先进入浮点程序误差检测的数值用例生成任务的创建：用户只需要提交待处理的浮点程序代码的本地绝对路径，供系统查找到源程序文件来使用即可。数值用例生成任务执行完成后，系统会生成任务执行的结果——可用于该浮点程序误差检测的有效数值用例以及该任务执行所花费的总时间。

服务接入层中，系统可以通过 GET/POST 请求为交互展示层提供核心业务的结果。此层中，主要包括一些核心服务的调度接口，可以对核心业务层中的实现功能进行调度，最终达成实现系统主要功能的目的。

核心业务层是本系统的核心部分。程序预处理主要包括对于浮点程序的词法进行分析，将词法分析的结果组合成抽象语法树，最终生成我们需要的可读



**交互展示层**

**任务管理**

**结果展示**

任务创建 任务发布 数值用例查看 任务耗时查看

**任务发布**

**结果生成**

任务调度分发

**服务接入层**

数值用例生成

任务耗时生成

**神经网络模块**

**训练数据**

训练数据生成

**误差分析检测 结果返回**

训练数据验证

**核心业务层**

**程序预处理**

词法分析

**中间代码转换**

**数值用例生成**

**模型训练**

模型构建

转换算法设计 生成数值用例

模型迭代训练

AST生成

**模型使用**

中间代码生成

二进制码转换

任务耗时计算

模型评估

模型调整

**基础模块层**

中间代码信息 二进制码信息 训练数据信息

Mysql

Mybatis

图 4.1: 系统架构图

的中间代码 IR。中间代码转换主要包括转换算法的设计以及根据该算法对中间代码进行转换，最终生成可用的二进制码。数值用例生成功能则主要是使用了前两个服务以及深度学习模型，最终能得出用于浮点程序误差分析的数值用例以及该功能的耗时。

神经网络模块主要是为数值用例生成功能提供深度学习模型，此模块主要分为两个部分，训练数据部分主要通过 Herbie 工具等将大量浮点程序的数值用例和二进制码作为训练数据集，模型训练部分则是将这部分训练数据集放入构建好的模型进行迭代训练，不断评估和调整模型，最后得到供核心业务使用的神经网络模型。

基础数据层则主要是为系统提供基本数据的支撑，包括预处理模块需要使

用的浮点程序的路径信息、中间代码模块需要使用的对应程序中间代码的路径信息和对应程序二进制码的路径信息以及训练、测试和用户使用的数据信息。

需求分析中按模块将本系统划分为五个，其中图 [4.1](#_bookmark69)中的核心业务层的三类业务分别代表系统的程序预处理模块、中间代码转换模块以及数值用例生成模块，图 [4.1](#_bookmark69)中的神经网络模块的两类工作分别代表训练数据集生成模块以及神经网络训练模块。

从工程角度来说，本系统实现前后端分离，前端页面主要使用 CSS 等样式组件，服务端主要采用 SpringBoot 框架进行开发，该框架的可拓展性使其他服务组件能够被快速集成进来。同时使用 Mybatis 进行数据库的连接工作，把 sql与业务逻辑进行分离，相对而言更加简洁明了。

* + 1. 逻辑视图设计

逻辑视图提供用户的最终视角，将系统拆分成一系列抽象功能，同时展现各功能之间的依赖关系，以此描述本系统的功能性需求。类图在面向对象程序分析之中主要被用来描述类以及类之间的关系，因此本小节使用类图来描述本系统的逻辑视图，将本系统分解为一些列关键功能类，如图 [4.2](#_bookmark71)所示。

逻辑视图

**ModelTrainService**

+ run()

**StorageService**

+ compile()

**CompileService**

+ errorDisplay()

+ runTimeDisplay()

**ResultDisplayService**

+ fileLocation

**DispatchService**

+ run()

**ModelAdjustService**

+ errorAnalysis()

+ runTimeCount()

**ErrorAnalysisService**

+ generateData()

+ dataVerify()

**TrainingDataService**

errorInterval

cnnModel

runTime

**ModelService**

+ run()

trainingData

|  |
| --- |
| **PreprocessService** |
| + fileLocation |
| + excutePreprocess() |

|  |
| --- |
| **LexicalAnalysisService** |
| + fileLocation |
| + excuteLexicalAnalysis( |

|  |
| --- |
| **GrammarAnalysisService** |
| + tokenfile |
| + generateAST() |

|  |
| --- |
| **IRGenerateService** |
| + AST |
| + generateIR() |

图 4.2: 系统逻辑视图

DispatchService 主要进行数值用例生成任务流程以及模型训练流程的调度工作，它需要调度多个模块来完成任务。数值用例生成任务中，ErrorAnalysis- Service 负责创建任务；PreprocessService 负责源程序的预处理工作，调用 Lexi- calAnalysisService 进行词法分析、GrammarAnalysisService 生成抽象语法树、IR-

GenerateService 生成中间代码；CompileService 主要负责中间代码转二进制码； ResultDisplayService 负责展示任务最终结果。模型训练中，TrainingDataService负责训练数据集的生成与验证；而 ModelService 则会调用 ModelTrainService 和 ModelAdjustService 进行模型训练与调整。StorageService 提供的是整个系统的存储信息功能。

* + 1. 进程视图设计

进程视图侧重于描述整个系统运行流程的设计。它主要是将逻辑视图中的功能进行抽象，使其适合系统的进程结构。因此使用顺序图来表示本系统的进程视图，如图 [4.3](#_bookmark73)所示。

return errorInterval&runTime

Bytecode

compile()

return Byte

startErrorAnalysisService

runTimeCount()

errorAnalysis() Preprocess Service

return IR

runModelTrainingService() return Model

runModelAdjustService()

return cnnModel

loop

compile()

excuteLexicalAnalysis() return tokens generateAST() return AST generateIR()

return IR

return Bytecode

generateData()

数值用例生成 程序预处理线 中间代码转换 神经网络模型任务 程 线程 线程

测试数据生成任务

startTraining

DataService

用户

进程视图

图 4.3: 系统进程视图

用户首先开始生成训练数据，测试数据生成线程会首先调用预处理线程进行词法分析、抽象语法树生成以及中间代码生成的工作；得到中间代码后，预处理线程会调用转换线程，使其转化为二进制码，然后和数值用例组成训练数据集，用于模型的迭代训练与评估调整，最终得到可使用的 CNN 模型。用户还可以开始数值用例生成任务，先后调用两个线程进行预处理和转换两个过程，最后调用神经网络模型线程获取可用的神经网络模型，并将中间结果放入模型来获取数值用例生成任务的结果。

* + 1. 开发视图设计

开发视图，也叫模块视图，是从开发者的角度，描述系统模块的组织和管理以及其在开发环境下的静态结构，充分考虑了系统内部的功能需求，如图 [4.4](#_bookmark75)所示。



**Service**

**Preprocess Service**

**Compile Service**

**Storage Service**

**TrainingData Service**

**IRMap**

**Dao Bytecode**

**Map**

**TrainingData Map**

开发视图

**展示层**

**业务逻辑层**

**数据存储层**

Mysql

Mybatis

Utils

Models

VO

**Controller**

**ErrorAnalysis Controller**

CSS

components

container

assets

app

**ErrorAnalysis**

**Service**

**ModelTraining**

**Service**

**ResultDisplay**

**Service**

**TrainData**

**Controller**

图 4.4: 系统开发视图

系统遵循前后端分离的设计，从上向下分为三层：展示层、业务逻辑层、数

据存储层。展示层主要包括用户交互页面的相关组件模板等，其中 app 包中存放的是页面代码，assets 包中存放的是前端页面所需的静态资源文件，如 json 文件和图片等，Container 存放前端所需的容器组件，Components 包存放前端所需的展示组件，CSS 包存放前端部分样式文件，而 VO 包中存放的是业务逻辑层与展示层交互的 VO 对象。业务逻辑层采用分层架构，Controller 层包括各类监听器，用于监听前端请求，并启动对应的线程执行业务操作，并返回结果给前端；Service 层则负责处理具体的业务逻辑，并返回结果给 Controller 层；Dao 层负责业务数据的存取工作以支持Service 层工作；而Model 包中存放的是数据信息的对象实体，Utils 包中存放各种静态工具供使用。数据存储层主要提供系统数据信息的存储与检索服务，包括 Mysql 与 Mybatis 存取数据，以保证数据的持久化。

* + 1. 物理视图设计

物理视图

《PC》

User Client

《server》

Server

Browser

HTTP

《artifact》

jar

TCP

《server》

DB Server

《database system》 Mysql

《schema》

project

《schema》

workspace

《schema》

data

图 4.5: 系统物理视图

物理视图，又称为部署视图，该视图从部署的角度，考虑如何把软件映射到硬件之上，来解决系统拓扑结构等问题，如图 [4.5](#_bookmark77)所示。用户通过浏览器进入本系统，发起误差分析检测任务，此时客户端会发起 HTTP 请求，该请求经过一定操作转发到服务端；同时应用服务端和数据库服务端使用 TCP 进行通信，来存

取系统的各类数据信息。

## 浮点用例生成系统的详细设计

上文已经从系统架构、逻辑视图、进程视图、开发视图以及物理视图五个角度对系统进行了整体的概要设计，本节将对系统的各模块进行进一步分析，从每个模块的流程图、详细类图以及时序图入手，对每个模块所实现的功能进行详细设计。

* + 1. 程序预处理模块的详细设计

程序预处理模块主要进行的是对源浮点程序的预处理工作，主要负责将浮点程序代码文件，转化为系统可使用的中间代码文件，并将其存储进数据库的对应表中。具体的工作流程如图 [4.6](#_bookmark80)所示。



解决源代码问题

提供源程序路径

开始程序预处理

词法分析

语法分析

不正确

中间代码存储

中间代码生成

抽象语法树生成 源代码是否正确

正确

图 4.6: 程序预处理模块流程

要开始源程序的预处理工作，首先，用户需要向系统提供源程序的本地绝对路径，系统根据此路径找到待处理的源程序代码文件。首先系统需要对源代码文件进行词法分析，将源代码分成一个个 token。得到 token 后，系统进行语法分析，对token 进行语法组合。根据token 的组合，可以判断源代码是否正确，若不正确则需要用户修改源代码文件后再重新开始预处理工作，若正确系统会生成抽象语法树。最后系统会根据此抽象语法树生成可读的中间代码 IR，并且将中间代码存储进数据库中，以供其他模块进行使用。

图 [4.7](#_bookmark81)展示了程序预处理模块的类图设计。ErrorAnalysisController 是对外提供不同实现的控制器，在本模块中向外提供执行预处理任务功能。PreprocessSer- vice 将预处理任务细分为两部分。第一部分是执行预处理操作得到中间代码，此部分需要依赖 LexicalAnalysisService 提供词法分析服务、GrammarAnalysisSer- vice 提供语法分析服务以及 IRGenerateService 提供中间代码生成服务。第二部分是将生成的中间代码转换成 IR 对象存储进数据库表中，此部分需要依赖 IR

实体类和 IRGenerateService 提供中间码存储服务。



浮点程序预处理模块类图

**LexicalAnalysisService**

+ sourceCode\_path: String

+ excuteLexicalAnalysis(String):String

+ generateIR(sourceCode\_path, AST\_path): IR

+ storeIR(ir: IR): void

**«interface» IRMapper**

+ generateAST(String):String

**«interface» GrammarAnalysis**

+ generateAST(String):String

+ token\_path: String

**GrammarAnalysisService**

+ excuteLexicalAnalysis(String):String

**«interface» LexicalAnalysis**

|  |
| --- |
| **ErrorAnalysisController** |
| + preprocessService: PreprocessService  + lexicalAnalysisService: LexicalAnalysisService  + grammarAnalysisService: GrammarAnalysisService  + irGenerateService: IRGenerateService |
| + getIR(sourceCode\_path: String): IR |

|  |
| --- |
| **PreprocessService** |
| + lexicalAnalysisService: LexicalAnalysisService  + grammarAnalysisService: GrammarAnalysisService  + irGenerateService: IRGenerateService |
| + getIR(sourceCode\_path: String): IR  + storeIR(ir: IR): void |

|  |
| --- |
| **IR** |
| + IR\_path:String  + sourceCode\_path:String |
| + IR()  + IR(IR\_path, sourceCode\_path)  + getIRPath(): String  + setIRPath(IR\_path:String)  + getsourceCodePath(): String  + setsourceCodePath(sourceCode\_path) |

|  |
| --- |
| **IRGenerateService** |
| + irMapper: IRMapper |
| + generateIR(sourceCode\_path, AST\_path): IR  + storeIR(ir: IR): void |

图 4.7: 程序预处理模块类图

程序预处理模块的时序图如图 [4.8](#_bookmark82)所示，当系统接收到用户获取中间代码的请求时，ErrorAnalysisController 会使用getIR 方法调用PreprocessService 来启动整个程序预处理流程。然后，PreprocessService 会调用LexicalAnalysisService 中的 excuteLexicalAnalysis 方法获取源代码的token。接着，PreprocessService 会继续调用GrammarAnalysisService 中的generateAST 方法来获得源代码的抽象语法树。最后，PreprocessService 会调用IRGenerateService 中的generateIR 方法获得对应的中间代码。同时，PreprocessService 会调用IRGenerateService 中的storeIR方法来存储获得的中间代码 IR 对象。上述所有任务均完成后，系统会返回任务结果给用户。



getIR

getIR

(source\_path) cuteLexicalAnalysis

(source\_path) return

generateAST(token\_path) return

generateIR(AST\_path) return

storeIR(ir) return

return

return

IRGenerate Service

GrammarAnalysis Service

ex

LexicalAnalysis Service

Preprocess Service

ErrorAnalysis Controller

图 4.8: 浮点程序预处理模块时序图

* + 1. 中间代码转换模块的详细设计

中间代码转换模块主要进行的是对源浮点程序中间代码的二进制码转换工作，主要负责将已生成的中间代码文件转换为系统可用的二进制码文件，并存储进数据库中。具体的工作流程如图 [4.9](#_bookmark84)所示。



设计转换规则

提供中间代码路径

二进制码存储

二进制码生成

开始中间代码转换

图 4.9: 中间代码转换模块流程

要对源浮点程序代码文件的已生成的中间代码进行转换工作，首先，我们需要设计一种将中间代码转换成二进制码的转换规则，此规则需要规定以下内容：最终生成的二进制码的行数和列数、每一行中间代码转换成的二进制码位数、每一行中间代码里的数学操作数转换成的二进制码位数、每一行中间代码里的数学操作符转换成的二进制码位数、中间代码中特殊操作所需转换为的特殊二进制码以及二进制码总位数不足时用 0 补足等。然后，用户需要向系统提供中间代码的本地绝对路径，系统会根据此路径寻找对应的中间代码文件。找到中间代码文件后，系统会根据之前设计的转换规则，将中间代码转换为对应

的二进制码，并且将该二进制码文件的信息存储进数据库中，以供其他模块进行使用。

图 [4.10](#_bookmark85)展示了中间代码转换模块的类图设计。ErrorAnalysisController 在中间代码转换模块中向外提供的是执行浮点程序中间代码转二进制码任务的功能。 IRCompileService 主要提供两个功能，第一个功能就是将中间代码转成二进制码，第二个功能是将生成的二进制码转换为ByteCode 对象存储进数据库的对应表中。第一个部分主要依赖 IR 实体类和 ByteCode 实体类，第二个部分需要依赖 ByteCode 实体类。

中间代码转换模块类图

**IRCompileService**

+ compileIR(sourceCode\_path: String, IR\_path: String): ByteCode

+ storeByteCode(byteCode: ByteCode): void

**«interface»**

**ByteCodeMapper**

+ compileIR(sourceCode\_path: String, IR\_path: String): ByteCode

+ storeByteCode(byteCode: ByteCode): void

+ irMapper: IRMapper

+ byteCodeMapper: ByteCodeMapper

+ getByteCode(ir: IR): ByteCode

+ IR()

+ IR(IR\_path:String, sourceCode\_path:String)

+ getIRPath(): String

+ setIRPath(IR\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path:String)

+irCompileService: IRCompileService

+ IR\_path:String

+ sourceCode\_path:String

**ErrorAnalysisController**

**IR**

|  |
| --- |
| **ByteCode** |
| + ByteCode\_path:String  + sourceCode\_path:String |
| + ByteCode()  + ByteCode(ByteCode\_path:String, sourceCode\_path:String)  + getByteCodePath(): String  + setByteCodePath(ByteCode\_path:String)  + getsourceCodePath(): String  + setsourceCodePath(source\_path:String) |

图 4.10: 中间代码转换模块类图

中间代码转换模块的时序图如图 [4.11](#_bookmark86)所示。首先，用户会发出中间代码转二进制码的请求，当系统接收到获取二进制码的请求时，ErrorAnalysisController 会启动整个中间代码转换的流程。ErrorAnalysisController 会调用IRCompileService中的compileIR 方法来获得转换后的二进制码，此转换方法根据之前设计的转换规则书写。同时，ErrorAnalysisController 也会调用 IRCompileService 中的 store- ByteCode 方法来存储之前获得的二进制码 ByteCode 对象。所有任务均完成后，会返回任务结果给用户。



IRCompileService

getByteCode(ir)

compileIR(sourceCode\_path, IR\_path)

return

storeByteCode(byteCode)

return

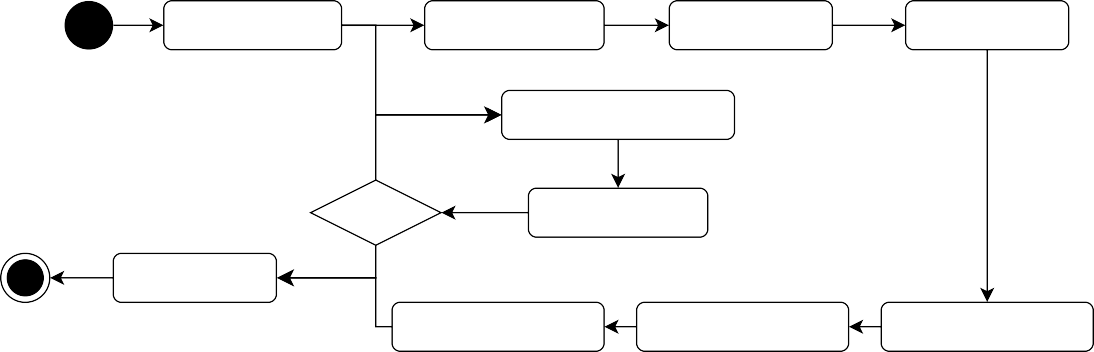
return

ErrorAnalysisController

图 4.11: 中间代码转换模块时序图

* + 1. 训练数据集生成模块的详细设计

训练数据集生成模块主要进行的是生成供神经网络模型训练的训练数据集工作，具体的工作流程如图 [4.12](#_bookmark88)所示。



提供源程序路径

开始程序预处理

中间代码生成

中间代码存储

获取源程序数值用例

不正确

正确与否 数值用例验证

训练数据存储

正确

二进制码存储

二进制码生成

开始中间代码转换

图 4.12: 训练数据集生成模块流程图

要生成可供神经网络模型训练的训练数据集，首先，用户需要向系统提供源浮点程序的本地路径，系统取得对应的浮点程序代码文件后，该工作主要分为

两个部分。第一部分中，系统先开始进行浮点程序预处理工作，生成对应的中间代码，并将该中间代码信息存储进数据库中；然后，系统依据先前设计的转换规则，对此中间代码进行转换工作，将此中间代码转换为可用的二进制码，并将该二进制码信息存储进数据库中。第二部分中，系统利用 Herbie 工具获取用于对应浮点程序误差检测的具体数值用例；获取该数值用例后，使用自己编写的误差函数对该数值用例进行验证，若不正确，则重新获取对应数值用例；若正确，则将该数值用例和对应浮点程序的二进制码等相关信息存储进数据库中的训练数据表中。



训练数据集生成模块类图1

+ ByteCode()

+ ByteCode(ByteCode\_path, sourceCode\_path)

+ getByteCodePath(): String

+ setByteCodePath(ByteCode\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path:String)

+ compileIR(source\_path, IR\_path): ByteCode

+ storeByteCode(byteCode: ByteCode): void

+ irMapper: IRMapper

+ ByteCode\_path:String

+ sourceCode\_path:String

**IRCompileService**

**ByteCode**

+ generateIR(sourceCode\_path, AST\_path): IR

+ storeIR(ir: IR): void

+ irMapper: IRMapper

**IRGenerateService**

+ IR()

+ IR(IR\_path:String, sourceCode\_path:String): IR

+ getIRPath(): String

+ setIRPath(IR\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath (sourceCode\_path:String): void

+ IR\_path:String

+ sourceCode\_path:String

**IR**

+ compileIR(source\_path, IR\_path): ByteCode

+ storeByteCode(byteCode: ByteCode): void

**«interface»**

**ByteCodeMapper**

+ irGenerateService: IRGenerateService

+ getIR(sourceCode\_path: String): IR

+ storeIR(ir: IR): void

**PreprocessService**

+ generateIR(sourceCode\_path, AST\_path): IR

+ storeIR(ir: IR): void

**«interface» IRMapper**

|  |
| --- |
| **TrainDataController** |
| + preprocessService: PreprocessService  + irGenerateService: IRGenerateService  + irCompileService: IRCompileService |
| + getTrainData(sourceCode\_path): void |

图 4.13: 训练数据集生成模块类图一

训练数据集生成模块的主要工作分为两部分，因此本文将本模块的类图也

分为两个部分，图 [4.13](#_bookmark89)展示了训练数据集生成模块第一部分的类图设计。此部分中，TrainDataController 是本模块的控制器，负责生成训练数据集的任务。用户需要先向系统提供生成训练数据集的源浮点程序代码文件的本地路径。Train- DataController 接收到代码路径后，会调用预处理和中间代码转换服务。其中， PreprocessService 提供 getIR 方法，IRGenerateService 提供 generateIR 方法来获得对应中间代码，同时 PreprocessService 还肩负着将中间代码存储进数据库的功能。IRCompileService 主要功能是获得对应的二进制码，同时它需要将 ByteCode对象存储进数据库中。其中，IRGenerateService 仅需要依赖IR 实体类，IRCom- pileService 则需要依赖 IR 实体类和 ByteCode 实体类。

训练数据集生成模块类图2

+ storeTrainData (sourceCode\_path: String, ir: IR,

byteCode: ByteCode, error\_interval: String): void

+ ByteCode()

+ ByteCode(ByteCode\_path, source\_path)

+ getByteCodePath(): String

+ setByteCodePath(ByteCode\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path)

+ irMapper: IRMapper

+ byteCodeMapper: ByteCodeMapper

+ trainingDataMapper: TrainingDataMapper

+ ByteCode\_path:String

+ sourceCode\_path:String

**TrainingDataService**

**ByteCode**

+ storeTrainData(sourceCode\_path): void

+ trainingDataService: TrainingDataService

**TrainDataController**

+ storeTrainData (sourceCode\_path: String, ir: IR,

byteCode: ByteCode, error\_interval: String): void

**«interface»**

**TrainingDataMapper**

|  |
| --- |
| **TrainingData** |
| + ByteCode\_path:String  + sourceCode\_path:String  + IR\_path:String  + Error\_interval:String |
| + TrainingData()  + TrainingData  (IR\_path:String, ByteCode\_path:String, sourceCode\_path:String, Error\_interval:String)  + get(); + set(); ······ |

|  |
| --- |
| **IR** |
| + IR\_path:String  + sourceCode\_path:String |
| + IR()  + IR(IR\_path:String, sourceCode\_path:String)  + getIRPath(): String  + setIRPath(IR\_path:String)  + getsourceCodePath(): String  + setsourceCodePath(sourceCode\_path:String) |

图 4.14: 训练数据集生成模块类图二

图 [4.14](#_bookmark90)展示了训练数据集生成模块第二部分的类图设计。此部分中，Train- DataController 依旧是本模块控制器，主要负责将生成的训练数据存储进数据库

的对应表中。其中 Controller 调用的TrainingDataService 主要提供storeTrainData方法，负责将已经生成的训练数据的实体对象存储进数据库的对应表中，而训练数据中所包含的浮点程序数值用例信息则来源于 Herbie 工具并且已经通过根据 iRRAM 自行编写的误差函数进行了正确性验证。类图中显示，TrainingDataSer- vice 需要依赖 IR 实体类、ByteCode 实体类以及 TrainingData 实体类来进行数据存储工作。



storeTrainData (source\_path) getIR

(source\_path)

generateIR (source\_path, AST\_path)

return storeIR(ir)

return

compileIR(sourceCode\_path,IR\_path) return

storeByteCode(byteCode) return

storeTrainData(sourceCode\_path,ir，byteCode, error\_interval)

return

return

TrainingData Service

IRCompile Service

IRGenerate Service

Preprocess Service

TrainData Controller

图 4.15: 训练数据集生成模块时序图

训练数据集生成模块的时序图如图 [4.15](#_bookmark91)所示。用户会发出生成训练数据集的请求，同时向系统提供源浮点程序代码文件的本地路径。当系统接收到生成训练数据集请求时，会使用 TrainDataController 启动整个训练数据集生成的流程。首先，TrainDataController 会调用PreprocessService 的getIR 方法，PreprocessSer- vice 会调用IRGenerateService 的generateIR 方法获得对应的中间代码，取得中间代码后调用storeIR 方法将中间代码存储到数据库中；接着，TrainDataController会调用 IRCompileService 的 compileIR 方法将中间代码转换成二进制码，再调用 IRCompileService 的 storeByteCode 方法将二进制码存储到数据库中。最后， TrainDataController 会调用TrainingDataService 的storeTrainData 方法，该方法中的数值用例参数使用 Herbie 工具获得并经过自行编写的误差函数验证，最终将训练数据信息存储在数据库对应表中。

* + 1. 神经网络训练模块的详细设计

神经网络训练模块主要进行的是神经网络模型的生成工作，包括搭建神经网络模型、迭代训练模型以及模型的评估和调整。具体的工作流程如图 [4.16](#_bookmark93)所示。



神经网络模型调整

搭建神经网络

开始神经网络迭代训练

神经网络模型评估

神经网络模型生成

合格

神经网络模型是否合格

不合格

图 4.16: 神经网络训练模块流程图

要生成可供使用的神经网络模型，用户首先需完成神经网络模型的搭建工作。然后，神经网络模型的训练实际上是一个迭代过程，主要包括三个部分：第一部分是训练部分，主要是将所有的训练数据投入到搭建好的神经网络模型中进行训练；第二部分是评估部分，此部分需等到神经网络模型训练完成后，向神经网络模型投入用于测试的浮点程序二进制码，并评估神经网络模型输出结果的正确性，若输出结果正确则该神经网络模型可用，若输出结果不正确则需要进行第三部分的工作；第三部分是神经网络模型参数调整，调整完毕后重新训练直至模型可用。

图 [4.17](#_bookmark94)展示了神经网络训练模块的类图设计。ModelTrainingController 是本模块的控制器，负责执行神经网络模型的生成任务。用户首先需要搭建一个神经网络模型供系统训练，然后 ModelTrainingController 会调用 TrainingDataService，此服务主要负负责从数据库中获取到之前生成的所有训练数据的集合；接着， Controller 会调用ModelTrainingService，此服务主要负责对神经网络模型进行训练；同时在训练完成后，ModelTrainingService 还负责对神经网络模型进行评估，判断其是否合格，若不合格，则ModelTrainingService 需要继续负责调整神经网络模型参数的任务，同时ModelTrainingService 实现的几个功能需要不停地被循环使用，直到神经网络模型评估合格为止。其中，TrainingDataService 需要依赖 TrainingData 实体类，ModelTrainingService 需要依赖 ByteCode 实体类，因为其需要使用用于测试的浮点程序的二进制码来生成具体数值用例从而对神经网络模型的优劣进行评估。

神经网络训练模块类图

**ModelTrainingController ModelTrainingService**

+ modelTrainingService: ModelTrainingService + trainingDataService: trainingDataService

+ trainingDataService: trainingDataService

+ getModel(): Model

+ modelTrain(trainDatas: Arraylist<>): Model

+ modelAssess(ArrayList<ByteCode>): boolean

+ modelAdjust(assessResult: boolean): void

**TrainingDataService**

+ trainingDataMapper: TrainingDataMapper

+ getAllTrainingData(): ArrayList<TrainingData>

**«interface»**

**TrainingDataMapper**

+ getAllTrainingData(): ArrayList<TrainingData>

**«interface» ModelMapper**

+ modelTrain(trainDatas: Arraylist<>): Model

+ modelAssess(ArrayList<ByteCode>): boolean

+ modelAdjust(assessResult: boolean): void

**TrainingData**

+ ByteCode\_path:String

+ sourceCode\_path:String

+ IR\_path:String

+ Error\_interval:String

**ByteCode**

+ ByteCode\_path:String

+ sourceCode\_path:String

+ TrainingData()

+ TrainingData(IR\_path, ByteCode\_path, sourceCode\_path, Error\_interval)

+ getAllTrainingData(); + set(); ······

+ ByteCode()

+ ByteCode(ByteCode\_path, sourceCode\_path)

+ getByteCodePath(): String

+ setByteCodePath(ByteCode\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path:String)

图 4.17: 神经网络训练模块类图

神经网络训练模块的时序图如图 [4.18](#_bookmark95)所示。用户会发出获取可用的神经网络模型的请求，同时用户会提供一个事先搭建好的神经网络模型供系统训练。当系统接收到获取神经网络模型的请求时，会使用 ModelTrainingController 启动整个神经网络模型生成的流程。ModelTrainingController 首先会调用 Training- DataService 的 getAllTrainingData 方法来获得之前生成的所有训练数据的集合；然后，ModelTrainingController 会调用 ModelTrainingService 的 modelTrain 方法对当前的神经网络模型进行训练，训练完成后，控制器会继续调用 modelAssess方法对训练完成的神经网络模型进行评估。若评估合格，则会直接生成可用的神经网络模型；若评估不合格，控制器则会调用 modelAjust 方法对神经网络模型的参数进行调整；神经网络模型的参数调整完成后，控制器将继续调用 modelTrain方法进行训练，直到评估结果满意，系统能够获得可使用的神经网络模型为止。此过程为一个循环迭代的过程。



ModelTrainingController

TrainingDataService

ModelTrainingService

getModel()

getAllTrainingData()

return

loop

modelTrain(trainDatas)

return

modelAssess(byteCodes)

return

modelAdjust(assessResult)

return

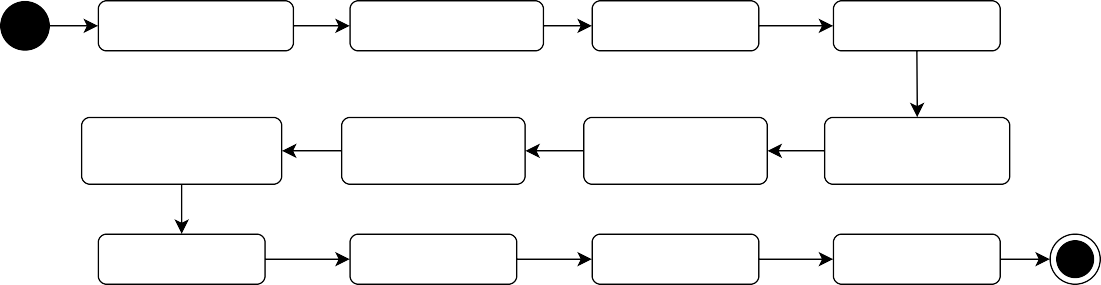
return

图 4.18: 神经网络训练模块时序图

* + 1. 数值用例生成模块的详细设计

数值用例生成模块主要面向系统用户，进行的是用于浮点程序误差检测的数值用例的生成工作，具体的工作流程如图 [4.19](#_bookmark97)所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 提供源程序路径 | 开始程序预处理 | 中间代码生成 | 中间代码存储 |
| 投入神经网络 | 二进制码 | 二进制码 | 开始中间代码 |
| 模型 | 存储 | 生成 | 转换 |
| 获得数值用例 | 获得运行时间 | 用户数据存储 | 结果显示 |

图 4.19: 数值用例生成模块流程图

要获得可用于浮点程序误差检测的数值用例，首先需要用户向系统提供源浮点程序的本地路径，系统取得对应的浮点程序代码文件后，进行后续一系列工作。首先，系统要开始进行浮点程序预处理工作，生成对应的中间代码 IR，并将该中间代码信息存储进数据库中；然后，系统依据先前设计的转换规则对此

中间代码进行转换工作，将此中间代码转换为可用的二进制码 ByteCode，并将该二进制码信息存储进数据库中；接着，系统将此二进制码投入训练完成的神经网络模型中，获得可用于对应浮点程序误差检测的数值用例，并通过系统时间的计算来获得任务的总体运行时间；最后，系统将该数值用例、二进制码以及任务运行时间等信息存储进数据库中的用户数据表中，并将结果显示在系统页面上。



输入区间用例生成模块类图1

+ preprocessService: PreprocessService

+ irGenerateService: IRGenerateService

+ irCompileService: IRCompileService

+ ByteCode()

+ ByteCode(ByteCode\_path, sourceCode\_path)

+ getByteCodePath(): String

+ setByteCodePath(ByteCode\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path:String)

+ compileIR(source\_path, IR\_path): ByteCode

+ storeByteCode(byteCode: ByteCode): void

+ irMapper: IRMapper

+ ByteCode\_path:String

+ sourceCode\_path:String

**IRCompileService**

**ByteCode**

+ errorAnalysis(sourceCode\_path): String

+ getIR(sourceCode\_path: String): IR

+ storeIR(ir: IR): void

+ irGenerateService: IRGenerateService

**PreprocessService**

**ErrorAnalysisController**

+ generateIR(sourceCode\_path, AST\_path): IR

+ storeIR(ir: IR): void

+ irMapper: IRMapper

**IRGenerateService**

+ IR()

+ IR(IR\_path:String, sourceCode\_path:String): IR

+ getIRPath(): String

+ setIRPath(IR\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath (sourceCode\_path:String): void

+ IR\_path:String

+ sourceCode\_path:String

**IR**

+ compileIR(source\_path, IR\_path): ByteCode

+ storeByteCode(byteCode: ByteCode): void

**«interface»**

**ByteCodeMapper**

+ generateIR(sourceCode\_path, AST\_path): IR

+ storeIR(ir: IR): void

**«interface» IRMapper**

图 4.20: 数值用例生成模块类图一

数值用例模块的主要工作分为两部分，因此本文将本模块的类图也分为两个部分，图 [4.20](#_bookmark98)展示了数值用例生成模块第一部分的类图设计。此部分中，Er- rorAnalysisController 是本模块的控制器，负责生成浮点程序对应的中间代码 IR和二进制码 ByteCode 的任务。用户需要先向系统提供待处理的浮点程序代码

文件的本地路径。PreprocessService 中的 getIR 方法以及 IRGenerateService 中的 generateIR 方法一起使用可以提供获取对应浮点程序的中间代码 IR 的功能；IR- GenerateService 中的 storeIR 方法主要是用来将生成的 IR 对象存储进数据库中。 IRCompileService 中的 compileIR 方法可以提供获取对应浮点程序的二进制码的功能，同时此服务中的 storeByteCode 方法可以用来将生成的ByteCode 对象存储进数据库中。其中，IRGenerateService 仅需要依赖IR 实体类，IRCompileService则需要依赖 IR 实体类和 ByteCode 实体类。

输入区间用例生成模块类图2



**ErrorAnalysisController**

+ errorAnalysisService: ErrorAnalysisService

+ resultDisplayService: ResultDisplayService

+ errorAnalysis(sourceCode\_path): String

+ runTimeCount(): Integer

+ display(error\_interval, runTime): void

**ErrorAnalysisService**

+ irMapper: IRMapper

+ byteCodeMapper: ByteCodeMapper

+ userDataMapper: UserDataMapper

+ errorAnalysis(byteCode, model): String

+ runTimeCount(): Integer

+ storeUserData(sourceCode\_path, ir, byteCode, error\_interval, runTime): void

**IR**

+ IR\_path:String

+ sourceCode\_path:String

+ IR()

+ IR(IR\_path:String, sourceCode\_path:String): IR

+ getIRPath(): String

+ setIRPath(IR\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath (sourceCode\_path:String): void

**ResultDisplayService**

+ errorAnalysis(byteCode, model): String

+ runTimeCount(): Integer

+ storeUserData(sourceCode\_path, ir: IR, byteCode, error\_interval, runTime): void

**«interface»**

**UserDataMapper**

+ display(error\_interval:String,runTime:Integer): void

+ display(error\_interval, runTime): void

**«interface» ResultDisplay**

**UserData**

+ ByteCode\_path:String

+ sourceCode\_path:String

+ IR\_path:String

+ Error\_interval:String

+ runTime: Integer

+ UserData()

+ TrainingData(IR\_path, ByteCode\_path, sourceCode\_path, Error\_interval, runTime)

+ get(); + set(); ······

**ByteCode**

+ ByteCode\_path:String

+ sourceCode\_path:String

+ ByteCode()

+ ByteCode(ByteCode\_path, sourceCode\_path)

+ getByteCodePath(): String

+ setByteCodePath(ByteCode\_path:String)

+ getsourceCodePath(): String

+ setsourceCodePath(sourceCode\_path:String)

图 4.21: 数值用例生成模块类图二

图 [4.21](#_bookmark99)展示了数值用例生成模块第二部分的类图设计。此部分中，ErrorAnal- ysisController 仍是本模块的控制器，负责生成用于浮点程序误差检测的数值用例、计算任务耗时以及显示任务结果等任务。ErrorAnalysisService 中的 errorAnal- ysis 和 runTimeCount 方法分别被使用来生成用于浮点程序误差检测的数值用例以及计算整个数值用例生成任务的运行耗时，同时还存在 storeUserData 方法将这些相关信息存储进数据库的用户数据表中。ResultDisplayService 的 display 方法主要是用来将任务运行结果显示给用户。其中，ErrorAnalysisService 需要依赖 IR、ByteCode 以及UserData 三个实体类，ResultDisplayService 依赖UserData 实体类。



resultDisplay()

errorAnalysis (source\_path)

ErrorAnalysis ResultDisplay Service Service

ErrorAnalysis

Controller

Preprocess

Service

IRGenerate

Service

IRCompile

Service

return

return

runTimeCount() return

display(error\_interval,runTime)

runTimeCount()

return

return

return

return

compileIR (sourceCode\_path,IR\_path) return

storeByteCode(byteCode) return

errorAnalysis(byteCode, model)

generateIR (source\_path, AST\_path)

return

storeIR(ir)

getIR (source\_path)

图 4.22: 数值用例生成模块时序图

数值用例生成模块的时序图如图 [4.22](#_bookmark100)所示。用户会发出用于源程序误差检测的数值用例生成请求，同时向系统提供 C/C++ 源程序代码。当系统接收到该请求时，会使用 ErrorAnalysisController 启动整个用例的生成流程。该控制器会调用PreprocessService 的getIR 方法，PreprocessService 会调用IRGenerateService 的 generateIR 方法获得对应中间代码，并调用 storeIR 方法将中间代码存储到数据库中；接着 ErrorAnalysisController 会调用ErrorAnalysisService 中的 errorAnalysis 和 runTimeCount 方法来生成数值用例和运行耗时，并且调用storeUserData 方法将这

些信息存储进数据库的用户数据表中。最后，控制器将调用 ResultDisplayService

的 display 方法将结果显示出来。

## 数据库设计

为了保证本系统各类数据信息的一致性和持久化，本系统使用关系型数据库 Mysql 对数据进行存储。本系统设计的数据主要包括浮点程序代码信息、中间代码信息、二进制码信息、训练数据信息和用户数据信息，同时为了对系统功能进行更好的测试，因此还增加了测试数据信息。下面将给出各类信息的详细数据库设计。

如表 [4.1](#_bookmark102)所示, 浮点程序的源代码信息存储在 sourceCode 表中，主要包括源代码id 以及源代码路径。其中源代码id 唯一标识该浮点程序的代码，是表的主键。源代码路径则存储的是浮点程序的源代码文件在本机上的绝对路径信息，便于系统寻找到该文件并进行使用。

表 4.1: sourceCode 表字段

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 含义 | 类型 | 描述 |
| sourceCode\_id | 源代码 id | Integer | 自增主键，唯一 |
| sourceCode\_path | 源代码路径 | String | 源代码文件绝对路径 |

如表 [4.2](#_bookmark103)所示，中间代码信息存储在 IR 表中，主要包括中间代码 id、源代码 id 以及中间代码路径。其中中间代码id 唯一标识该中间代码文件，是表的主键。源代码 id 则唯一标识该中间代码对应的源浮点程序代码文件，中间代码路径则表示此中间代码文件在本机上的绝对路径信息，便于系统寻找到中间代码文件并进行使用。

表 4.2: IR 表字段

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 含义 | 类型 | 描述 |
| IR\_id | 中间代码 id | Integer | 自增主键，唯一 |
| sourceCode\_id | 源代码 id | Integer | 该中间代码对应的源代码 id，唯一 |
| IR\_path | 中间代码路径 | String | 中间代码文件绝对路径 |

如表 [4.3](#_bookmark104)所示，二进制码信息存储在 byteCode 表中，主要包括二进制码 id、源代码id 以及二进制码路径。其中二进制码id 唯一标识该二进制码，是表的主键。源代码 id 则唯一标识该二进制码对应的源代码文件，二进制码路径则表示此二进制码文件在本机上的路径信息。

表 4.3: byteCode 表字段

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 含义 | 类型 | 描述 |
| byteCode\_id | 二进制码 id | Integer | 自增主键，唯一 |
| sourceCode\_id | 源代码 id | Integer | 该二进制码对应的源代码 id，唯一 |
| byteCode\_path | 二进制码路径 | String | 二进制码文件绝对路径 |

用户数据信息、训练数据信息以及用于测试的数据信息字段类似，下面仅以用户数据信息为例，描述用户数据信息的数据库详细设计，如表 [4.4](#_bookmark105)所示。用户数据信息存储在userData 表中，该表主要有用户数据 id、源代码 id、中间代码 id、二进制码 id、数值用例、运行时间六个字段。其中用户数据 id 唯一标识用户数据信息，是表的主键。源代码id 唯一标识对应的源代码文件，中间代码id 唯一标识对应的中间代码文件，二进制码 id 唯一标识对应的二进制码文件；数值用例是系统生成的用于浮点程序误差检测的具体用例，该用例能够触发浮点程序的明显误差累积；运行时间则是每一次数值用例生成任务所花费的时间，单位为秒。

表 4.4: userData 表字段

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 含义 | 类型 | 描述 |
| userData\_id | 用户数据 id | Integer | 自增主键，唯一，系统实际用户的数  据 id |
| sourceCode\_id | 源代码 id | Integer | 对应源代码 id |
| IR\_id | 中间代码 id | Integer | 对应中间代码 id |
| byteCode\_id | 二进制码 id | Integer | 对应二进制码 id |
| error\_interval | 数值用例 | String | 用于浮点程序误差检测的数值用例 |
| runTime | 运行时间 | Integer | 数值用例生成功能具体运行时间，单  位为秒 |

## 本章小结

本章首先通过介绍本系统的系统架构图对系统进行了整体性的设计，并从四视图的视角对系统进行了功能、运行流程、结构以及硬件部署等方面的分析和设计。之后，分模块的通过流程图、详细类图以及时序图对每个模块的功能以及流程进行详细的设计。最后，通过对数据库各表字段的详细解释给出了本系统的具体数据库设计。

# 第五章 浮点用例生成系统的实现

本章基于第四章对于整个系统的总体设计与对系统五大模块的详细设计，通过具体代码来阐述面向浮点程序误差检测的用例生成系统各模块所实现的功能，并在最后展示本系统实现的界面。

## 程序预处理模块的实现

程序预处理模块主要实现对源代码文件的预处理工作，主要包括对代码文件进行词法和语法的分析，以及中间代码 IR 的生成。图 [5.1](#_bookmark109)表示的是词法分析过程的第一部分，在此部分中，函数会根据提供的路径 path 找到浮点程序的代码文件，生成一个进程p 来进行浮点程序的代码处理。首先函数定义 err 参数来存储代码文件的所有内容，定义 tokens 列表来存储生成的所有独立字符单元。然后通过 for 循环将源代码文件按照行来划分，每一行数据存储为 ln 参数。最后根据不同情况进行处理每一行数据，最终将源代码分解为一个个独立的字符单元，存储进 tokens 对象中并返回。

def getTokens(path):

p = subprocess.Popen(['clang','-dump-raw-tokens',path],stdin=subprocess,……) err = p.communicate() tokens = []

for ln in err.split('\n'):

if 'Loc=<' in ln and ln.endswith('>'): ln = None

tokens.append(Token(\*kTokenRE.match(ln).groups()))

return tokens

图 5.1: 独立字符单元生成代码

图 [5.2](#_bookmark110)表示的是词法分析的第二部分。第一部分中系统会获取到独立字符单元集合 tokens，接着会调用此方法进行对于这些独立字符分别进行处理：首先进行 while 循环，根据独立字符的具体类型对这些字符进行处理，最终生成 ReturnedToken 对象；然后系统会使用 setCodeCompletionIdentifierInfo 方法存储生成的 ReturnedToken 之前的独立字符所携带的标识符，并通过 setIdentifierInfo方法将已处理对象的IdenfitierInfo 设置为null，以避免混淆已处理和未处理的字独立符单元。

void Preprocessor::Lex(Token &Result) { do {

switch (CurLexerKind) { case CLK\_Lexer:

ReturnedToken = CurLexer->Lex(Result); break;

}

} while (!ReturnedToken);

if (Result.is(tok::code\_completion) && Result.getIdentifierInfo()) { setCodeCompletionIdentifierInfo(Result.getIdentifierInfo()); setCodeCompletionTokenRange(Result.getLocation(), Result.getEndLoc()); Result.setIdentifierInfo(nullptr);

}

}

图 5.2: Token 生成代码

图 [5.3](#_bookmark111)表示的是语法树生成的第一部分，此部分中的 ParseAST 方法相当于语法树生成的入口函数，首先函数会判断参数 HaveLexer 判断是否已经通过预处理器进行了词法分析。若词法分析已经进行, 函数会使用ParseTopLevelDecl 函数调用ParseExternalDeclaration 函数，此函数又会在不同情况下调用 ParseDecla- rationOrFunctionDefinition 或者 ParseSimpleDeclaration 方法，以这两个方法为入口可以进入到 lib/Parse 路径下的各类辅助函数中，这些函数能够处理 C/C++ 编程语言所定义的所有结构和语法。

void clang::ParseAST(Sema &S, bool PrintStats, bool SkipFunctionBodies) { if (HaveLexer) {

for (bool P.ParseFirstTopLevelDecl(); !AtEOF; P.ParseTopLevelDecl()) {

if (ADecl && !Consumer->HandleTopLevelDecl(ADecl.get())) return;

}

}

}

bool Parser::ParseTopLevelDecl(DeclGroupPtrTy &Result, bool IsFirstDecl) { switch (Tok.getKind()) { //根据 Token 的类型进行相应的操作处理 } Result = ParseExternalDeclaration(attrs);

return false;

}

图 5.3: 抽象语法树生成入口函数代码

文件 lib/Parse 路径下的各类辅助函数大部分都会调用 Sema 类的 ActOnIf- Stmt 函数来进行语义分析和 AST 的生成，如图 [5.4](#_bookmark112)所示。图中首先显示的是 ParseStmt.cpp 文件中的 ParseIfStatement 函数，我们可以发现此函数返回了 Ac- tOnIfStmt()，实际上就是调用了 Sema::ActOnIfStmt() 函数，而 ActOnIfStmt() 函数开始部分就是分析代码语法是否有误，然后返回经过 Owned 方法处理过的 If- Stmt 函数结果。IfStmt 函数主要负责根据ASTContext 和其他参数创建一个AST对象，并对其中的部分属性进行修改以获得可用的抽象语法树。

StmtResult Parser::ParseIfStatement(SourceLocation \*TrailingElseLoc) { return Actions.ActOnIfStmt(IfLoc, IsConstexpr, InitStmt.get(),Cond, );

}

StmtResult Sema::ActOnIfStmt(SourceLocation IfLoc,FullExprArg CondVal, ) {

//分析语义

return Owned(new (Context) IfStmt(Context, IfLoc,ConditionVar, ));

}

IfStmt::IfStmt(ASTContext &C, SourceLocation IL, VarDecl \*var,Expr \*cond, ){

setConditionVariable(C, var);

SubExprs[COND] = reinterpret\_cast<Stmt\*>(cond); SubExprs[THEN] = then;

SubExprs[ELSE] = elsev;

}

图 5.4: 抽象语法树生成代码

图[5.5](#_bookmark113)首先显示了中间代码生成的入口函数，此函数位于CodeGenModule.cpp中，主要包含了一个 Switch 语句，这个 Switch 语句会根据参数 D 的类型调用不同的函数来生成中间代码，如果该参数是模板类声明则直接返回不需要生成中间代码，此部分代码中 Switch 方法重点选择的是与 C++ 程序有关的参数类型。接下来的 getIR 方法展示了面向浮点程序误差检测的用例生成系统中的中间代码生成服务的具体实现，包括在系统的预处理模块中对 Linux 命令行的调用等。用户需给出待处理的源代码文件路径，即 sourceCode\_path，方法中首先会规定中间代码文件的生成路径，该生成路径由固定部分和计数器IR\_count 组成， IR\_count 属于全局变量，会随着任务的增加而增加。然后，方法中会规定具体命令行，该命令行同样由不变部分加上源代码路径和生成路径组成。最后系统会新开一个进程去执行该命令行，此时中间代码生成，同时系统会将该命令行执行的结果打印在控制台上，并在日志文件中留下记录。这些工作全部完成后会最终返回一个新的 IR 对象，此 IR 对象可用于中间代码的数据库存储。

void CodeGenModule::EmitTopLevelDecl(Decl \*D) { if (D->isTemplated()) return;

switch (D->getKind()) { }

}

public static IR getIR(String sourceCode\_path) { IR\_count += 1;

String end\_path = “ ” + IR\_count + “.ll”;

commandStr = “~/clang -O0 -emit-llvm ” + sourceCode\_path + “ -S -o “+end\_path; try {

Process p = Runtime.getRuntime().exec(commandStr);

//此处还会将命令行结果显示在控制台显示出来，并打印到日志文件中

} catch (Exception e) { e.printStackTrace();

}

return new IR(sourceCode\_path, end\_path);

}

图 5.5: 中间代码 IR 生成的代码

## 中间代码转换模块的实现

中间代码转换模块主要是负责将中间代码IR 转换为神经网络模型可用的二进制码。此工作主要分为两个部分：设计中间代码 IR 转二进制码的转换规则，并根据此规则对浮点程序的中间代码文件进行处理并生成二进制码文件。两部分全部完成后，会最终返回一个新的 ByteCode 对象，此对象可用于二进制码的数据库存储。

图 [5.6](#_bookmark115)代码展示了笔者所设计的部分转换规则。此转换规则主要针对中间代码中每一个字符串进行判断，判断其是否属于操作数，还是操作数以外的操作符。转换规则实现了针对不同情况的不同处理方式：若此字符串属于操作数，其整数部分和小数部分都需占 8 位二进制位数，但具体转换方式不同；若此字符串非操作数，则判断其是操作符还是参数，操作符或者内存操作则直接使用 switch方法进行转换，占 8 位二进制位数，若其第一个字符为“%”，则表示需要建立新的变量，在接下来的行数中进行赋值和使用的操作，同时也占据 16 位二进制数等。

图 [5.7](#_bookmark116)代码展示了根据转换规则进行中间代码转换的过程。此转换过程中也涉及到规则的定义，如一行有 128 位二进制码，一共有 145 行，不足均补 0 等。此方法主要使用java 读取和写入文件方法，读取中间代码文件，在读取到“entry：”

public static String compileRules(String str){ String[] result= new String[2];

Pattern pattern = Pattern.compile("^[-\+]?\d\*[.]\d+$"); if(pattern.matcher(str).matches()){

String[] strs = str.split("."); strs[1] = "0." + strs[1]; int one = Integer.valueOf(strs[0].toString());

double two = Double.valueOf(strs[1].toString()); for(int i = 0;i < 8;i++){

result[0] = result[0] + one % 2;

if(two \* 2 >= 1) {result[1] = result[1] + 1; two = two \* 2 - 1; } else{ result[1] = result[1] + 0; two = two \* 2; }

} return result[0] + result[1];

}else{ switch(str){ case "+" : return "00000001"; default: return "1111111"; } }

}

图 5.6: 转换算法的部分代码

下一行时开始转换工作，转换时调用 compile 方法。Count 作为计算行数的变量，当count 小于 145 时，进行补足操作。最后方法返回根据中间代码路径和输出文件路径新建的 ByteCode 对象。

public static ByteCode compileIR(String sourceCode\_path, String IR\_path){ BufferedInputStream in = new BufferedInputStream(FileInputStream(IR\_path)); String end\_path = " ";

BufferedOutputStream out = BufferedOutputStream(FileOutputStream(……));

byte[] b = new byte[1024]; int len = -1; int count = 0; int startState = 0; while ((len = in.read(b, 0, b.length)) != -1) {

String str = new String(b, 0, len, "UTF-8"); String[] strs = str.split(" ");

if(startState == 1){

//处理每一个字符串，并将最后字符串进行合并为 result，少于 128 位补 0 count++;

}if(strs.equals("entry:")) { startState = 1; }

}

//不足 145 行用 0 补足足行数

out.flush(); in.close(); out.close();

return new ByteCode(sourceCode\_path, end\_path);

}

图 5.7: 转换 IR 的代码

## 训练数据集生成模块的实现

训练数据集生成模块主要负责生成可供神经网络模型使用的训练数据集合，此训练数据实际包括两部分，一部分为源代码的二进制码，另一部分为能够触发浮点程序明显误差累积的数值用例，此部分需要验证。

图 [5.8](#_bookmark118)代码主要显示数值用例的验证，以函数 *f* (*x*) = √*x* + 1 − √*x* 为例。该

函数使用 iRRAM 包的REAL 类，通过编写原函数的误差函数，通过同样输入值时观察误差是否变大，得到一个使得输出误差最大的数值用例，与 Herbie 工具产生的数值用例作对比，验证其正确性。

REAL sqrt\_minus\_real(const REAL &x) {

return REAL(1.0) / (REAL(sqrt(x + 1)) + REAL(sqrt(x)));

}

//误差函数

图 5.8: 验证数值区间的代码

图 [5.9](#_bookmark119)代码主要是说明训练数据的生成和存储的过程，此过程主要需要复用浮点程序预处理模块的 getIR 和中间代码编译模块的 getByteCode 方法，来获取需要的 IR 对象和 ByteCode 对象，同时获得经过校验的浮点程序数值用例，使用这些信息创建 TrainData 对象并将其存入训练数据表中。

public static void storeTrainData(String source\_path, String error\_interval){ IR ir = proprecessService.getIR(source\_path);

ByteCode byteCode = compileService.getByteCode(ir.getIRPath()); TrainingData trainData = new TrainingData(source\_path, ir.getIRPath(),

byteCode.getByteCodePath(), error\_interval);

//将 TrainingData 对象存进训练数据表中

}

图 5.9: 生成训练数据的代码

## 神经网络训练模块的实现

神经网络模型训练模块主要展示模型的具体实现方式，具体代码如图 [5.10](#_bookmark121)所示。我们使用的模型为 CNN 模型，需要进行参数的设置，包括输入数据的格式、正则项系数以及模型相关量等；同时需要对模型进行具体的构建：首先需要对输入的数据进行数据重定型使其成为可用的向量形式，然后第一、第二层均需要对数据进行卷积操作，第一个卷积层中我们需要设置卷积核的尺寸大小、图形

通道数以及卷积核的数目，然后卷积核按照遍历步数进行遍历，进行最大值池化操作，最终将输入数据的数量缩小。第二个卷积层中我们则需要确定采用的通道数，并确定最终需卷积出的特征数量，并使用上一卷积层的池化结果进行进一步池化。第三层全连接，继续进行卷积和池化并使用dropout 减少过拟合。第四层输出层，进行矩阵卷积，最终输出结果与output\_size\*batch\_size。同时需要建立预测的损失函数、正则项、训练张量以及多标签结果。最后从数据库中读取训练数据集，投入到神经网络模型中进行训练。根据训练结果调整参数再训练，直至获得满意的训练结果。

class CNN:

def init (self):

self.batch\_size = 64 self.input\_x = 145 self.input\_y = 32 self.output\_size = 1 self.regular\_scale = 0.001 self.graph = tf.Graph()

//定义模型参数

def build\_model(self):

with self.graph.as\_default():

x\_image = tf.reshape(self.x, [-1, self.input\_x, self.input\_y, 1])

//第一二层卷积操作，第三层全连接操作，第四层进行输出操作 def build\_loss(self):

with self.graph.as\_default(): return loss

def build\_regular\_term(self): return reg\_term def build\_train\_op(self): return train\_op

def get\_multi\_result(self): return value, index train\_data\_x, train\_data\_y = generator.read\_data() valid\_data\_x, valid\_data\_y = generator.read\_data() test\_data\_x, test\_data\_y = generator.read\_data()

model = CNN() //最后启动模型进行训练

图 5.10: 神经网络模型的代码

## 数值用例生成模块的实现

数值用例生成模块主要面向用户提供用于浮点程序误差检测的数值用例生成的工作，如图 [5.11](#_bookmark123)所示。该模块前半部分的方法调用与训练数据集生成模块类似，主要区别在于最后一个方法需要使用到的是训练完成的神经网络模型，系统只需将待检测的浮点程序的二进制码输入进去，就可获得对应浮点程序的数值用例，同时在此功能中还需要通过获得系统时间的差值来获取任务整体的运行时间。

public static userData errorAnalysis(String source\_path, Model model){

long start = System.currentTimeMillis()/1000; //生成 IR 并编译

String error\_interval = getError(byteCode,model); long end = System.currentTimeMillis()/1000;

TrainingData userData = new UserData(source\_path, ir.getIRPath(),

byteCode.getByteCodePath(), error\_interval, end - start);

}

图 5.11: 数值用例生成功能的代码

## 本章小结

本章对本系统五个模块：程序预处理模块、中间代码转换模块、训练数据集生成模块、神经网络训练模块以及数值用例生成模块的实现代码进行了展示，并对每段代码进行了详细解释，并在本章最后展示了本系统主要实现的页面。

# 第六章 实验评估与测试

本章将依据第四章的系统设计和第五章的模块实现描述，首先给出对系统进行实验评估，并对系统各模块的功能进行测试。本章将首先描述实验目的，并给出实验环境，通过对实验的详细描述评估本系统效果；然后对系统的功能性需求与非功能性进行测试，并给出测试结果。

## 实验评估

本节将对用例生成系统进行实验评估，实验主要从实验环境、实验设计以及实验结果三个部分进行介绍。实验环境部分详细介绍实验过程中需要的硬件配置和软件依赖；实验设计部分主要介绍并详细解释实验过程；实验结果部分以表格的形式对实验结果进行展示并对结果进行评估。实验主要是在神经网络训练模块以及数值用例生成模块实现的基础上进行的，一方面使用神经网络模型获取用于浮点程序误差检测的数值用例，另一方面通过检验系统生成的数值用例是否能够触发浮点程序的明显误差累积来评估系统有效用例的生成效率。

* + 1. 实验环境

表 6.1: 实验环境

|  |  |
| --- | --- |
| 配置名 | 配置参数 |
| 客户端 | 操作系统：Windows 10；浏览器：Google Chrome，版本：  89.0.4389.90 |
| 后端服务器 | 操作系统：Ubuntu 18.04.3 LTS；CPU：8 核；内存：16G；  硬盘：500G；gcc：7.4.0；Java：1.8.0\_191；LLVM：9.0.0svn |
| 数据库服务器 | 操作系统：Ubuntu 18.04.3 LTS；CPU：8 核；内存：16G；  硬盘：500G；Mysql：8.0.16 |
| 前端服务器 | 操作系统：Ubuntu 18.04.4 LTS；CPU：2 核；内存：4G；  硬盘：50G；Vue：2.6.11 |

在测试时，本系统主要将前端页面、后端功能服务以及数据库服务部署在两台服务器上，服务器主要选取了Ubuntu 操作系统来作为评估面向浮点程序误差检测的用例生成系统的系统环境，同时客户端浏览器主要使用 Google Chrome浏览器，每个服务所在的服务器详细配置如表 [6.1](#_bookmark128)所示。

* + 1. 实验设计

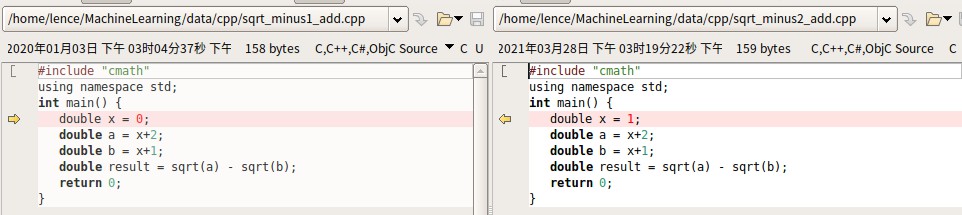
现有用例生成技术 [[44](#_bookmark198)] 主要采用表达式扰动方法，通过对单一函数表达式的转换，将函数表达式变换为语法不同但语义相同的等价表达式，并观察不同形式对于浮点运算误差的影响，并由此生成使得此函数表达式输出结果偏差较大的数值用例。此技术实际上仅仅对单一函数表达式进行处理，这就导致此系统只能够针对单一函数表达式进行用例生成工作，而无法针对浮点程序全局进行用例生成；同时由于此技术生成的数值用例是随机获取的，因此需要生成大量用例才能找到那些能够触发函数表达式明显误差累积的有效用例，有效用例的生成效率较低。

本实验主要针对现有的用例生成技术面向单一表达式生成用例、生成有效用例的效率低等问题，对系统提出的面向浮点程序误差检测的数值用例生成方法进行评估。实验刚开始，本系统所采用的数值用例生成方法对源浮点程序的处理主要采用 LLVM 编译器，使其转换为二进制码，来供神经网络模型进行训练。中途在 LLVM 编译器上出现了一些问题，下面详细讲述问题出现的原因、表现以及当前方法如何解决该问题，同时给出利用本系统的数值用例生成方法的实验过程，并给出判断本系统使用方法是否有效的指标。

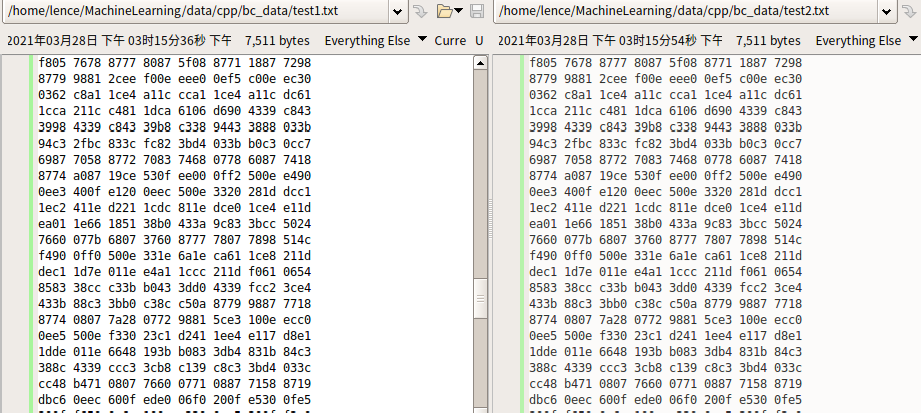
实验初始阶段，只使用 LLVM 编译器转源程序代码为二进制码，对于算法相同，但参数的值不同的浮点程序代码，如图 [6.1(a)](#_bookmark130)所示，我们得到的 bc 文件中数据完全相同，如图 [6.1(b)](#_bookmark131)所示，导致转换成的二进制码也完全相同。于是我们从源代码 bc 文件的生成开始向前追溯，找到了 LLVM 生成中间代码 IR 的部分，并将两个源程序的中间代码IR 存储到了本地两个不同文件中。通过对中间代码文件的进一步对比分析，我们发现了 bc 文件相同的原因：LLVM 生成的中间代码会在类似汇编部分的开头阶段生成一个栈空间来存储操作数变量，但由于下方使用操作数是实际使用的是操作数在栈中的位置信息，虽然程序中变量值发生了变化，但中间代码的部分不会因此变化，所以LLVM 生成的bc 文件对于不同源程序是相同的。同时 bc 文件的开头部分存在大量无用信息，因此使用 bc 文件进行训练是不准确的。

基于以上发现，我们对本系统所采用的数值用例生成方法进行了修改，改为只利用传统LLVM 编译器的中间代码IR 生成部分，而将中间代码转二进制码的部分交由笔者进行转换规则的重新设计，规则内容主要包括最终生成的二进制码行列数、操作数与操作符所对应的二进制码以及二进制码总位数不足用 0 补足等，从而对不同的浮点程序得到不同但有实际意义的二进制码，最终满足神经网络训练的要求。同时我们还需要利用 Herbie 工具来获取组成训练数据所需的具体数值用例，并利用 iRRAM 包中的REAL 类编写对应的误差函数来验证获

取到的具体数值用例是否准确。最终，通过将由二进制码与数值用例组成的训练数据投入模型进行训练，我们就能够获取到可用于数值用例生成的神经网络模型。



* + - 1. 浮点程序代码文件对比



* + - 1. 二进制码文件对比

图 6.1: 实验过程的问题

实验中，我们使用优化后的数值用例生成方法生成用于测试的浮点程序的二进制码。将此二进制码投入训练完成的神经网络模型中，我们就能获取到对应浮点程序的数值用例输入。通过 Herbie 工具以及 iRRAM 包，我们可以验证获取到的数值用例输入能够触发浮点程序的明显误差累积的比例。接着，我们还需要计算数值用例生成所耗费的具体时间。最终，通过对相同的函数表达式或浮点程序进行处理，并对比本文所采用的数值用例生成方法与现有数值用例生成技术的各项指标，包括有效数值用例生成效率(即有效用例数在生成用例总数中的占比)、用例生成耗时、方法适用范围等，对本文所采用的数值用例生成方法的有效性进行评估。

* + 1. 实验结果评估

实验中，我们分别使用现有的用例生成方法以及本系统采用的用例生成方法对 5 个常见函数进行处理，包括 *log*2(*x*)、*exp*2(*x*)、*sqrt*(*x*)、*sinpi*(*x*) 以及 *sinh*(*x*)，最终生成这些函数表达式的具体数值用例输入，其中两种方法的有效数值用例生成效率以及用例生成耗时对比如表 [6.2](#_bookmark134)所示。通过对实验结果的对比分析，我们可以发现，虽然对于函数 *exp*2(*x*) 和 *sinh*(*x*)，本系统所使用的用例生成方法生成的有效用例的总数占比低于现有的用例生成方法，但是对于函数 *log*2(*x*)、 *sqrt*(*x*) 以及 *sinpi*(*x*)，本系统使用的用例生成方法生成的有效用例的总数占比远高于现有的用例生成方法；同时，虽然对于其中三个函数，本系统采用的用例生成方法的用例生成耗时略高，但对于其他两个函数，本系统方法的用例生成耗时远低于现有的用例生成方法。综合 5 个常见函数来看，虽然本系统采用的用例生成方法生成数值用例总数相对偏少，但是本系统方法生成的有效数值用例的总数占比 74.17% 高于现有用例生成方法的 67.57%，且本系统方法的用例生成平均耗时远低于现有的用例生成方法，所以此结果表明本系统采用的用例生成方法相对而言是有效的。

表 6.2: 有效数值用例生成效率以及用例生成耗时对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试函数 | 生成数值用例  总数 | 度量指标  方法 | 有效数值用例  数 **(**总数占比**)** | 用例生成平均  耗时 **(s)** |
| *log*2(*x*) | 216 | 现有用例生成方法 | 32897(50.20%) | 8.65 |
| 212 | 本系统用例生成方法 | 2945(71.90%) | 2.52 |
| *exp*2(*x*) | 216 | 现有用例生成方法 | 61548(93.91%) | 0.89 |
| 212 | 本系统用例生成方法 | 3172(77.44%) | 2.38 |
| *sqrt*(*x*) | 216 | 现有用例生成方法 | 32897(50.20%) | 0.07 |
| 212 | 本系统用例生成方法 | 2959(72.80%) | 2.23 |
| *sinpi*(*x*) | 216 | 现有用例生成方法 | 30976(47.27%) | 32 |
| 212 | 本系统用例生成方法 | 2895(70.68%) | 2.70 |
| *sinh*(*x*) | 216 | 现有用例生成方法 | 63084(96.26%) | 0.27 |
| 212 | 本系统用例生成方法 | 3219(78.59%) | 2.31 |

同时，对于数值用例生成方法的适用范围指标，通过上述实验结果可以表明，现有的用例生成方法主要面向单一的函数表达式进行用例生成工作。而本系统使用的用例生成方法，不仅适用于单一的函数表达式，还可以面向浮点程序全局进行数值用例的生成。实验中，我们主要选取了四种常见的浮点程序，如图 [6.2](#_bookmark139)所示。其中，图 [6.2(a)](#_bookmark135)中的浮点程序代码主要进行简单库函数的调用，因此我们将其命名为“origin\_program.cpp”；图 [6.2(b)](#_bookmark136)中的浮点程序代码与图 [6.2(a)](#_bookmark135)相比多了

if 条件判断语句，因此我们将其命名为“if\_program.cpp”；图 [6.2(c)](#_bookmark137)中的浮点程序代码与图 [6.2(a)](#_bookmark135)相比多了 for 循环语句，因此我们将其命名为“for\_program.cpp”；图 [6.2(d)](#_bookmark138)中的浮点程序代码与图 [6.2(a)](#_bookmark135)相比多了 while 循环语句，因此我们将其命名为“while\_program.cpp”。

#include "camth" using namespace std; int main() {

double x = 0; double a = x + 2; double b = x + 1;

double result = sqrt(a) - sqrt(b); return result;

}

#include "camth" using namespace std; int main() {

double x = 0; double a = x + 2; double b = x + 1; if(x + 1 > 0){ a--; a++; }

else{ a += 1.0; }

double result = sqrt(a) - sqrt(b); return result;

}

(a) 简单库函数调用 (b) if 条件判断

#include "camth" using namespace std; int main() {

double x = 0; double a = x + 2; double b = x + 1; for(int i = 0; i < a; i++)

{ a += 1.0; a -= 1.0; }

double result = sqrt(a) - sqrt(b); return result;

}

#include "camth" using namespace std; int main() {

double x = 0; int i = 0;

double a = x + 2; double b = x + 1; while( i < a ){ a += 1.0; a -= 1.0; i++;} double result = sqrt(a) - sqrt(b); return result;

}

(c) for 循环 (d) while 循环

图 6.2: 实验过程使用的浮点程序

我们通过利用本系统中的用例生成方法，可以生成上述四个浮点程序的多组二进制码与数值用例，作为神经网络模型的训练数据集。通过此训练数据集对神经网络模型进行迭代训练，并根据训练结果调整模型参数，最终获取到可用于浮点程序数值用例生成的神经网络模型。在实验中，我们通过提供不同浮点程序的对应二进制码，利用训练完成的神经网络模型针对每个浮点程序文件生成了 4200 个数值用例，具体实验结果如表 [6.3](#_bookmark140)所示。

表 6.3: 浮点程序的部分数值用例评估

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 浮点程序 | 数值用例  总数 | 有效数值用例数 | 有效用例数占比 | 用例生成平均  耗时 |
| origin\_program | 4200 | 3304 | 78.67% | 2.31s |
| if\_program | 4200 | 3199 | 76.17% | 2.52s |
| for\_program | 4200 | 3031 | 72.17% | 2.73s |
| while\_program | 4200 | 2968 | 70.67% | 2.77s |

实验结果表中，有效数值用例数使用本系统中的用例生成方法生成，同时通过 iRRAM 包的REAL 类验证得到。有效数值用例数占比，即有效用例的生成效率指标。用例生成平均耗时则是通过系统时间的计算得到。我们发现，随着浮点程序内部逻辑的复杂程度逐步提高，有效用例的生成效率会逐渐降低，用例生成的平均耗时会逐步增加。但就整体而言，有效用例的生成效率约为 74.42%，用例生成的平均耗时约为 2.58s，相对而言效率较高且平均耗时较短。根据以上实验结果，我们发现本系统所用的用例生成方法能适用于浮点程序全局，同时其有效用例的生成效率较高，用例生成的平均耗时较短，因此表明此方法是有效的。

## 功能性测试

本节将根据功能性需求分析的结果以及各模块的用例描述，对系统的各项功能点进行测试，以保证系统的功能满足需求的规范要求与预期结果。该功能测试过程主要为黑盒测试，用户无需关心程序结果，只需要根据需求设计测试用例，并检查系统各项功能是否按照设计规范准确执行。

表 6.4: 中间代码生成的测试用例

|  |  |
| --- | --- |
| 测试编号 | TC1 |
| 测试名称 | 中间代码生成测试 |
| 测试目标 | 用户通过系统能够正确的进行词法分析，抽象语法树生成以  及中间代码生成 |
| 测试流程 | 1. 用户点击进入 IR 生成界面 2. 用户提供源代码本地路径 3. 用户点击词法分析按钮 4. 用户点击语法树生成按钮 5. 用户点击 IR 生成按钮 |
| 预期结果 | 1. 界面显示词法分析已完成，并在本地生成 token 文件 2. 界面显示抽象语法树已生成，并在本地生成语法树文件 3. 界面显示中间代码已生成，并在本地生成中间代码文件 4. 界面显示中间代码信息已保存到数据库，数据库 IR 表中出现中间代码信息 |
| 测试结果 | 与预期结果相符合 |

表 [6.4](#_bookmark142)为中间代码生成功能的详细测试用例设计。该用例主要为测试浮点程序预处理模块的中间代码生成功能，其中包含了词法分析、语法树生成、中间代码 IR 的生成与存储等功能。该测试用例旨在测试系统能否正确生成 token，能

否正确生成抽象语法树，能否正确生成中间代码IR 以及能否将中间代码信息存储到数据库中。

表 [6.5](#_bookmark143)为中间代码转换功能的详细测试用例设计。该用例主要为测试中间代码转换模块的二进制码生成功能，其中包含了转换规则展示以及二进制码的生成与存储等功能。该测试用例旨在测试系统能否正确展示转换规则，能否正确生成浮点程序的二进制码以及能否将二进制码的信息准确存储到数据库中。

表 6.5: 中间代码转换的测试用例

|  |  |
| --- | --- |
| 测试编号 | TC2 |
| 测试名称 | 中间代码转换测试 |
| 测试目标 | 用户通过系统能够正确的将浮点程序的中间代码转换为二  进制码 |
| 测试流程 | 1. 用户点击进入 ByteCode 生成界面 2. 用户提供待处理的中间代码的本地路径 3. 用户点击转换按钮 |
| 预期结果 | 1. 进入界面首先显示部分转换规则中的对应关系 2. 界面显示浮点程序二进制码已生成，并在本地生成二进制码文件 3. 界面显示二进制码文件信息已保存到数据库，数据库的 bytecode 表中出现二进制码的相关信息 |
| 测试结果 | 与预期结果相符合 |

表 [6.6](#_bookmark144)为生成训练数据功能的详细测试用例设计。该用例主要为测试训练数据集生成模块的训练数据生成功能，其中包含了中间代码生成、二进制码生成、训练数据生成、训练验证以及训练数据存储等功能。该测试用例旨在测试系统能否正确生成浮点程序的中间代码，能否正确生成浮点程序二进制码，能否正确生成包含触发浮点程序明显误差累积的数值用例的训练数据，能否对生成的训练数据进行验证以及能否将中间代码、二进制码以及训练数据的相关信息准确存储到数据库的对应表中。用户根据本测试用例步骤对本系统进行实际测试，测试结果显示，系统能够生成训练数据中所需的二进制码与数值用例，并将其存储进数据库表中，与预期结果相符合。

表 [6.7](#_bookmark145)为神经网络模型生成功能的详细测试用例设计。该用例主要为测试神经网络模型训练模块的神经网络生成功能，其中包含了神经网络模型训练、神经网络模型训练效果评估与参数调整以及神经网络模型生成等功能。该测试用例旨在测试系统能否正确迭代训练神经网络模型，能否正确使用测试数据评估神经网络模型的优劣并修改对应的模型参数以及能否正确生成可用的神经网络

表 6.6: 训练数据生成的测试用例

|  |  |
| --- | --- |
| 测试编号 | TC3 |
| 测试名称 | 训练数据生成测试 |
| 测试目标 | 用户通过系统能够正确的生成训练数据的相关内容，并将训  练数据存储到对应数据库表中 |
| 测试流程 | 1. 用户点击进入训练数据生成界面，提供路径 2. 用户点击 IR 生成按钮 3. 用户点击 ByteCode 生成按钮 4. 用户点击训练数据生成与检验按钮 |
| 预期结果 | 1. 界面显示中间代码已生成，数据库的 IR 表出现中间代码信息 2. 界面显示二进制码已生成，数据库的 byteCode 表出现二进制码信息 3. 界面显示训练数据已生成，数据库的 trainData 表中出现训练数据的相关信息 |
| 测试结果 | 与预期结果相符合 |

模型。用户根据本测试用例步骤对本系统进行实际测试，测试结果显示，系统能够进行正常的模型迭代训练过程，并可以通过测试数据对模型进行评估与模型参数调整，最终系统能够生成用于用例生成任务的神经网络模型，与预期结果相符合。

表 6.7: 神经网络模型生成的测试用例

|  |  |
| --- | --- |
| 测试编号 | TC4 |
| 测试名称 | 神经网络模型生成测试 |
| 测试目标 | 用户通过系统能够正确的生成可用的神经网络模型 |
| 测试流程 | 1. 用户点击进入神经网络模型生成界面, 点击模型训练按钮 2. 用户填写测试路径，点击测试按钮 3. 用户点击模型生成按钮 |
| 预期结果 | 1. 经过一段时间，界面显示已完成模型训练 2. 界面显示用于该程序误差检测的数值用例，同时数据库的对应表中出现对应信息 3. 界面显示可用的神经网络模型训练已生成，并在本地存在可用的模型文件 |
| 测试结果 | 与预期结果相符合 |

表 [6.8](#_bookmark146)为数值用例生成和运行时间生成功能的详细测试用例设计。该用例主

要为测试浮点程序误差检测的数值用例生成和运行时间生成功能，其中包含了数值用例生成、任务运行耗时计算等功能。该测试用例旨在测试系统能否正确显示用于源程序误差检测的数值用例以及能否保证任务耗时小于 5 秒。用户根据本测试用例步骤对本系统进行实际测试，测试结果与预期结果相符。

表 6.8: 数值用例和运行时间生成的测试用例

|  |  |
| --- | --- |
| 测试编号 | TC5 |
| 测试名称 | 数值用例和运行时间生成测试 |
| 测试目标 | 用户通过系统能够正确的生成数值用例以及任务耗时 |
| 测试流程 | 1. 用户进入数值用例生成界面 2. 用户填写源程序路径，点击生成按钮 |
| 预期结果 | 1. 界面显示用于该程序误差检测的数值用例 2. 界面显示该任务所耗费的时间 3. 数据库的用户数据表中出现对应信息 |
| 测试结果 | 与预期结果相符合 |

## 非功能性测试

非功能性测试用于测试面向浮点程序误差检测的用例生成系统的实现是否满足本文的非功能性需求。本小节将从易用性与性能方面对系统进行测试，保证系统可以满足本文的非功能性需求。

* + 1. 易用性

面向浮点程序误差检测的用例生成系统的易用性主要包括使用体验，该部分的测试主要针对可度量的用户操作流程进行。传统用例生成系统使用的步骤大多复杂且难于操作，因此本系统为了提高用户的使用体验，保证新用户能够在 15 分钟内熟练使用本系统所有功能，且每个功能步骤不超过 5 步。

经过 20 位新用户对系统的使用后，测试结果表明用户熟练使用本系统各功能最少需要 10 分钟，最多需要 18 分钟，平均每位用户需 14 分钟就能熟练使用本系统。同时，所有功能的操作步骤均不超过 5 步，因此满足非功能性测试中对系统易用性的要求。

* + 1. 性能

性能测试主要对面向浮点程序误差检测的用例生成系统的任务运行时长进行测试。为了减少用户的等待时间，提高其使用体验，对本系统主要功能的响应时间要求不超过 3s。如表 [6.9](#_bookmark150)所示，该表为用例生成系统的功能耗时测试表。

表 6.9: 用例生成系统任务耗时性能测试

|  |  |  |
| --- | --- | --- |
| 轮次 | 任务总耗时 | 平均任务耗时 |
| 1 | 2.808s | 2.808s |
| 10 | 27.329s | 2.733s |
| 50 | 133.350s | 2.667s |
| 100 | 263.128s | 2.631s |

其中分别进行了 1 次、10 次、50 次、100 次输入区间生成任务，总体而言,平均每次输入区间任务耗时约为 2.7s。且随着任务次数的增加，任务耗时逐步稳定在 2.6s 左右，满足本系统主要功能的响应时间要求不超过 3s 的性能需求。

## 本章小结

本章从实验评估、功能测试与非功能测试这三个方面对系统的测试工作进行了详细的介绍。通过以上三个小节，可以得知系统所涉及实验的具体过程以及出现的问题和解决方案，并对新的解决方案效果作出评估，极大程度的保证系统各模块的功能能正确运行，同时保证了系统的可靠性。

# 第七章 总结与展望

## 总结

浮点计算被广泛应用于各种对可靠性要求极高的关键工业软件中，因此我们需要通过技术手段来保障浮点程序计算结果的准确性。目前，华为公司采用了自动化的误差检测技术来控制其工业软件中的浮点误差，从而防止浮点运算的表示误差累积造成严重后果。对于误差检测技术来说，能够触发误差累积的数值用例生成至关重要，现有的用例生成系统存在两点不足：一是生成用例的适应性不足，系统针对单一浮点运算表达式来生成局部用例，无法结合软件前后的浮点语句，综合生成面向全局的检测用例。二是有效用例的生成效率较低，由于有效数值用例输入通常分散在一些范围较小的输入区间内，现有用例生成技术需要生成大量用例才能命中这些小区间。针对上述问题，本文面向华为工业软件中浮点计算程序的相关需求，设计并实现了一个面向浮点程序误差检测的用例生成系统。该系统对浮点程序进行全局分析，同时通过训练神经网络模型来提高区间命中率，最终实现了浮点程序误差检测的有效数值用例生成，并提高了系统效率。

本系统在设计与实现上分为五个模块：程序预处理模块、中间代码转换模块、训练数据集生成模块、神经网络模型训练模块和数值用例生成模块。程序预处理模块基于传统的LLVM 编译器，截取其词法、语法分析来生成中间代码。中间代码转换模块实现了一种新的转换算法将中间代码转换成可用于神经网络模型的二进制码。训练数据集生成模块为后续预测模型构造训练数据集，并将训练数据输出到神经网络模型训练模块进行迭代训练，通过对模型参数的调整和训练结果的评估而获得满意的训练结果。最终数值用例生成模块综合各模块的结果，向用户输出生成的有效浮点数值用例。

本系统目前已经在华为内部上线。我们针对一系列浮点软件，共生成了 16800 个数值用例来进行实验评估。实验评估结果表明本系统能够针对浮点程序全局进行用例生成，生成用例中能触发浮点程序误差累积的有效用例约占总数的 74.42%，且生成用例的平均耗时约为 2.58s，符合工业软件的用例生成要求。同时我们对系统进行了完整的功能性与非功能性测试，验证了系统正确性。

## 展望

本文设计并实现了一个面向浮点程序误差检测的用例生成系统。未来还可以从以下几个方面来进一步对系统实施改进：一是可以尝试采用最新的深度学习模型算法来提高神经网络模型的预测效果，这依赖于收集更多训练数据。二是可以考虑对当前系统做高性能分解，以进一步提高系统的吞吐量。

# 参考文献

1. S. Yin, Q. Shi, Y. Wang, C. Chen, Summary of software reliability research, IOP Conference Series: Materials Science and Engineering 1043 (5) (2021) 39–52.
2. IEEE Standard for Binary Floating-Point Arithmetic: ANSI/IEEE Std 754-1985, 1985.
3. F. Jézéquel, J. M. Chesneaux, CADNA: a library for estimating round-off error propagation, Comput. Phys. Commun. 178 (12) (2008) 933–955.
4. E. Tang, E. T. Barr, X. Li, Z. Su, Program instability detection based on system- atically optimized numerical perturbation, Scientia Sinica Informationis 44 (11) (2014) 45–46.
5. E. Goubault, S. Putot, Static analysis of finite precision computations, in: R. Jhala,

D. A. Schmidt (Eds.), Verification, Model Checking, and Abstract Interpretation

- 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings, Vol. 6538 of Lecture Notes in Computer Science, Springer, 2011, pp. 232–247.

1. F. Ivancic, M. K. Ganai, S. Sankaranarayanan, A. Gupta, Numerical stability analysis of floating-point computations using software model checking, in: 8th ACM/IEEE International Conference on Formal Methods and Models for Code- sign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010, IEEE Computer Society, 2010, pp. 49–58.
2. D. Kopec, S. Tamang, Failures in complex systems: case studies, causes, and possible remedies, ACM SIGCSE Bull. 39 (2) (2007) 180–184.
3. J. H. Wilkinson, Rounding errors in algebraic processes, in: Information Process- ing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris, June 15-20 1959, UNESCO (Paris), 1959, pp. 44–53.
4. N. J. Higham, Accuracy and stability of numerical algorithms, SIAM, 1996.
5. D. E. Knuth, The art of computer programming, Volume II: Seminumerical Algo- rithms, 3rd Edition, Addison-Wesley, 1998.
6. 周毓麟, 袁国兴, 关于科学计算用数字电子计算机字长问题, 计算机工程与科学 (10) (2005) 1–2.
7. P. R. Eggert, D. S. Parker, Perturbing and evaluating numerical programs without recompilation - the wonglediff way, Softw. Pract. Exp. 35 (4) (2005) 313–322.
8. M. Martel, Semantics-based transformation of arithmetic expressions, in: H. R. Nielson, G. Filé (Eds.), Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings, Vol. 4634 of Lec- ture Notes in Computer Science, Springer, 2007, pp. 298–314.
9. M. Martel, Program transformation for numerical precision, in: G. Puebla, G. Vi- dal (Eds.), Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Eval- uation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009, ACM, 2009, pp. 101–110.
10. E. Tang, E. T. Barr, X. Li, Z. Su, Perturbing numerical calculations for statistical analysis of floating-point program (in)stability, in: P. Tonella, A. Orso (Eds.), Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010, ACM, 2010, pp. 131–142.
11. J. Muller, On the error of computing *ab*+*cd* using cornea, harrison and tang’s method, ACM Trans. Math. Softw. 41 (2) (2015) 1–8.
12. C. Jeannerod, P. Kornerup, N. Louvet, J. Muller, Error bounds on complex floating-point multiplication with an FMA, Math. Comput. 86 (304) (2017) 881–

898.

1. C. Jeannerod, N. Louvet, J. Muller, A. Plet, Sharp error bounds for complex floating-point inversion, Numer. Algorithms 73 (3) (2016) 735–760.
2. J. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, et al., Handbook of Floating- Point Arithmetic (2nd Ed.), Springer, 2018.
3. H. Thapliyal, S. Kotiyal, M. B. Srinivas, Novel BCD adders and their reversible logic implementation for IEEE 754r format, in: 19th International Conference on

VLSI Design (VLSI Design 2006), 3-7 January 2006, Hyderabad, India, IEEE Computer Society, 2006, pp. 387–392.

1. M. DeLorimier, A. DeHon, Floating-point sparse matrix-vector multiply for fp- gas, in: H. Schmit, S. J. Wilton (Eds.), Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, FPGA 2005, Mon- terey, California, USA, February 20-22, 2005, ACM, 2005, pp. 75–85.
2. G. Even, W. J. Paul, On the design of IEEE compliant floating point units, IEEE Trans. Computers 49 (5) (2000) 398–413.
3. D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys 23 (1) (1991) 5–48.
4. J. Harrison, A machine-checked theory of floating point arithmetic, in: Y. Bertot,

G. Dowek, A. Hirschowitz, C. Paulin-Mohring, et al. (Eds.), Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs’99, Nice, France, September, 1999, Proceedings, Vol. 1690 of Lecture Notes in Computer Science, Springer, 1999, pp. 113–130.

1. 唐珍, 舍入误差分析引论, 上海科学技术出版社, 1987.
2. P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, Z. Tatlock, Automatically improv- ing accuracy for floating point expressions, in: D. Grove, S. M. Blackburn (Eds.), Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, ACM, 2015,

pp. 1–11.

1. N. T. Müller, The irram: Exact arithmetic in C++, in: J. Blanck, V. Brattka,

P. Hertling (Eds.), Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Vol. 2064 of Lecture Notes in Computer Science, Springer, 2000, pp. 222–252.

1. A. Edalat, P. J. Potts, A new representation for exact real numbers, in: S. D. Brookes, M. W. Mislove (Eds.), Thirteenth Annual Conference on Mathemati- cal Foundations of Progamming Semantics, MFPS 1997, Carnegie Mellon Uni- versity, Pittsburgh, PA, USA, March 23-26, 1997, Vol. 6 of Electronic Notes in Theoretical Computer Science, Elsevier, 1997, pp. 119–132.
2. V. Brattka, P. Hertling, Feasible real random access machines, J. Complex. 14 (4) (1998) 490–526.
3. N. T. Müller, Towards a real real RAM: a prototype using C++, in: K. Ko, N. T. Müller, K. Weihrauch (Eds.), Second Workshop on Computability and Complex- ity in Analysis, CCA 1996, August 22-23, 1996, Trier, Germany, Vol. TR 96-44 of Technical Report, University of Trier, 1996.
4. C. Lattner, V. S. Adve, The LLVM compiler framework and infrastructure tuto- rial, in: R. Eigenmann, Z. Li, S. P. Midkiff (Eds.), Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers, Vol. 3602 of Lecture Notes in Computer Science, Springer, 2004, pp. 15–16.
5. C. Cadar, K. Sen, Symbolic execution for software testing: Three decades later, Commun. ACM 56 (2) (2013) 82–90.
6. C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI’08, USENIX Association, Berkeley, CA, USA, 2008, pp. 209–224.
7. S. Anand, C. S. Pǎsǎreanu, W. Visser, JPF-SE: a symbolic execution extension to java PathFinder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, No. 4424 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 134–138.
8. Y. LeCun, Y. Bengio, G. Hinton, Deep learning, nature 521 (7553) (2015) 436–

444.

1. S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, et al., A survey on deep learning: Algo- rithms, techniques, and applications, ACM Comput. Surv. 51 (5) (2019) 1–36.
2. G. E. Dahl, D. Yu, L. Deng, A. Acero, Context-dependent pre-trained deep neu- ral networks for large-vocabulary speech recognition, IEEE Trans. Speech Audio Process. 20 (1) (2012) 30–42.
3. O. Russakovsky, J. Deng, H. Su, J. Krause, et al., Imagenet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252.
4. 陈先昌, 基于卷积神经网络的深度学习算法与应用研究, Ph.D. thesis, 浙江工商大学 (2014).
5. 杜选, 高明峰, 人工神经网络在数字识别中的应用, 计算机系统应用 (2)

(2007) 21–22.

1. 周开利, 康耀红, 神经网络模型及其 MATLAB 仿真程序设计, 清华大学出版社, 2005.
2. R. Pascanu, Ç. Gülçehre, K. Cho, Y. Bengio, How to construct deep recurrent neural networks, in: Y. Bengio, Y. LeCun (Eds.), 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, 2014.
3. C. J. Kuo, Understanding convolutional neural networks with a mathematical model, J. Vis. Commun. Image Represent. 41 (2016) 406–413.
4. J. P. Lim, M. Aanjaneya, J. Gustafson, S. Nagarakatte, An approach to gener- ate correctly rounded math libraries for new floating point variants, Proc. ACM Program. Lang. 5 (POPL) (2021) 1–30.

# 致 谢

因匿名稿要求隐去

因匿名稿要求隐去

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：

日期： 年 月 日