分类号 ________________ 密级 ________________

UDC ________________ 编号 ________________

中国科学院研究生院
博士学位论文

C程序的静态分析

许中兴

指导教师 张健

中国科学院软件研究所

申请学位级别 博士 学科专业名称 计算机软件与理论

论文提交日期 2009.4.24 论文答辩日期 2009.6.5

培养单位 中国科学院软件研究所

学位授予单位 中国科学院研究生院

答辩委员会主席 裘宗燕

Typeset and powered by:

$\LaTeX\,2_\varepsilon$ with `CASthesis` package from CTEX.org

GNU/Linux

Static Analysis of C Programs

Zhongxing Xu

Supervisor:

Jian Zhang

Institute of Software, Chinese Academy of Sciences

June, 2009

Submitted in total fulfilment of the requirements for the degree of Ph.D.
in Computer Science

摘　要

近年来，随着软件在社会生活中的作用越来越重要，软件的正确性也越来越受到人们的重视。然而如何保证软件的正确性却是一个一直都没有得到很好解决的问题。在上世纪六七十年代人们就提出了程序验证的概念，但是程序验证一直不能很好的处理用实用的程序设计语言编写的程序，并且自动化程度也不高，现代软件的规模往往有数百万行代码，导致程序验证无法得以应用。另一方面，面向大规模实际程序的传统数据流分析主要的应用是编译优化，它们的目标是获取保守的、粗略的程序信息，从而也不适合于程序正确性的检测。

随着计算性能的提高以及自动定理证明技术的成熟，另一大类基于符号执行的精确的程序分析技术逐渐被人们所重视。符号执行是一种假设输入变量为符号值，精确的模拟程序执行的路径敏感的静态分析方法。符号执行在上世纪七十年代被提出，但是由于无法处理程序中的指针、数组等语言成分，一直无法得以广泛应用。本文针对使用极为广泛的C语言，设计了一套完整精确模拟C 语言语义的算法和技术，包括内存建模，对各种表达式语义的模拟，函数调用的处理等。

作为符号执行技术的两种应用，本文对自动测试数据生成和内存泄漏检测分别进行了研究，提出了完整的算法，并实现了相应的工具。本文描述的符号执行算法目前已经在开源编译器Clang中得以实现，并还在不断改进之中。

Abstract

Recently, as software plays more and more important role in social life, the correctness of software receives more attention. How to guarantee the correctness of software, however, has not been solved well. In 1960s, program verification is proposed. But program verification has not been able to handle programs written in real programming languages, and the verification process cannot be done automatically. Modern software with millions of line of code prevents the program verification from being practical. On the other hand, traditional data flow analysis serves large programs well. But they aim to obtain conservative information from programs, which makes them not suitable for correctness checking.

Due to improvements in computational power and automatic theorem proving, another program analysis technique, symbolic execution, receives more attention. Symbolic execution tries to record as copious (symbolic) program execution state information as possible. Put it simply, symbolic execution assumes all input data be symbolic and simulates the program execution with these symbolic values. Symbolic execution was proposed in 1970s. But it could not handle arrays and pointers in real programs, which prevented it from being widely used. We designed algorithms and techniques for precisely simulating the full semantics of C programs, including memory modelling, handling of expressions, loops and function calls.

We applied the symbolic execution to two case studies: automatic test data generation for unit C programs and memory leak detection. Algorithms are designed, and tools are implemented. The symbolic execution algorithm proposed in this dissertation has been implemented in the Clang open source compiler, and being improved continually.

目　录

摘要 i

Abstract iii

目录 v

第一章　程序的正确性分析 1

1.1 程序的正确性分析 2

1.2 本文的贡献 5

第二章　数据流分析 7

2.1 基本的定义 8

2.2 数据流问题的分类和求解方法 9

2.3 迭代分析 11

2.4 总结 12

第三章　程序分析的模型 15

3.1 名字-值模型 16

3.2 数组模型 16

3.2.1 大数组模型 16

3.2.2 独立数组模型 17

3.3 基于区域的三元模型 18

3.3.1 初步介绍 18

3.3.2 精确定义 20

3.4 总结 22

第四章 C语言语义模拟 **23**
4.1 存储的建模 23
4.2 声明的处理 23
4.3 表达式求值 24
4.3.1 表达式求值的算法化 24
4.3.2 Primary expressions 25
4.3.3 Postfix operators 25
4.3.4 Unary operators 25
4.3.5 Binary operators 26
4.3.6 例子 26
4.4 总结 27

第五章 过程内分析 **33**
5.1 扩展图 33
5.2 总体分析框架 34
5.3 初始状态 34
5.4 程序点 34
5.5 对各种扩展图节点的处理 36
5.5.1 Block Edge – 对循环的处理 36
5.5.2 Block Entrance 37
5.5.3 PostStmt 37
5.6 状态的分裂 39
5.7 对数组的处理 40
5.8 符号值具体化 41
5.9 C的类型系统 42
5.10 总结 44

第六章 过程间分析 **45**
6.1 过程间的控制流分析：函数调用图 46

6.2 有限的函数内嵌法 46
6.3 基于函数总结的方法 47
6.3.1 函数总结(Function Summary) 48
6.3.2 部分函数总结 48
6.4 完整的算法 49
6.4.1 性质状态 49
6.4.2 路径条件 50
6.4.3 完整算法 50
6.5 总结 52

第七章 系统的实现 53

7.1 分析的对象 53
7.2 分析框架的结构 54
7.2.1 核心引擎 54
7.2.2 状态 55
7.2.3 各种管理器 56
7.3 总结 56

第八章 自动测试数据生成 57

8.1 引言 57
8.2 路径可行性分析 58
8.3 自动测试数据生成 58
8.3.1 概述 58
8.3.2 数组和指针 60
8.3.3 函数调用 64
8.3.4 路径生成策略 65
8.4 实验 65
8.4.1 例子：GNU coreutils中的remove_suffix() 66
8.4.2 例子：GNU coreutils中的strtol() 67

8.4.3 例子：getop() 67
8.4.4 例子：InsertionSort() 68
8.4.5 例子：GNU make中的dosify() 69
8.5 讨论 70
8.6 总结 70

第九章 内存泄漏的自动检测 73
9.1 介绍 73
9.2 一个例子 74
9.3 方法概述 74
9.3.1 内存对象的建模 77
9.3.2 逃逸模型 78
9.4 数据流信息格 78
9.5 函数建模 80
9.5.1 库函数建模 83
9.5.2 路径生成 84
9.5.3 路径可行性分析 84
9.5.4 函数建模 85
9.6 内存泄漏检查器 85
9.7 实现和实验 86
9.8 总结 86

第十章 相关工作 89
10.1 错误检测 89
10.2 符号执行 91
10.3 测试数据生成 92
10.4 内存泄漏 93
10.5 过程间分析 93

第十一章　结语 **95**

11.1 本文的贡献 95

11.2 进一步的工作 95

参考文献 **97**

发表文章目录 **105**

简历 **107**

致谢 **109**

表　　格

4.1　处理过声明语句后的程序模拟存储状态 27

4.2　处理过malloc语句后的程序模拟存储状态 28

4.3　处理过$sp->p=\&data$ 语句后的程序模拟存储状态 29

4.4　处理过$sp->p->d=3$语句后的程序模拟存储状态 29

4.5　处理过a[1] = data.d语句后的程序模拟存储状态 31

5.1　常见的代码缺陷类型 38

8.1　GNU coreutils 5.2.1软件包中用于实验的代码 66

9.1　在函数建模阶段完成后得到的函数总结 74

9.2　一些库函数建模的例子 83

插　　图

2.1 求解数据流问题的迭代算法 13

4.1 用于符号分析的例子：这个例子程序中有较为复杂的指针，结构，数组操作，并且内存块之间的关系种类也很多，使用简单的名字-值模型和数组模型都无法精确的模拟该程序的操作。基于区域的三元模型则可以精确的模拟该程序的操作，并将内存块之间的关系表示出来。 30

4.2 内存区域的层次关系：箭头指向的为父区域，其中MallocReg表示的是malloc()分配的区域 31

5.1 控制流图，带状态的控制流图和扩展图的对比 35

5.2 符号执行算法 36

5.3 处理结束语句的算法框架 39

5.4 直接以buf指向的MallocRegion为父区域生成ElementRegion，导致p[0]和q[0] 对应同一块区域，造成模拟的不精确。 43

5.5 Anonymous typed region 示意图：region_buf是buf指向的一块无类型内存，region_p是p指向的同一块加上了int类型信息的内存，同样，region_q是加上了char类型信息的内存，它们指向同一块父区域，表示它们本质上是同一块内存。p[0], q[0]分别是region_p 和region_q上的ElementRegion。 43

6.1 有限蛮力搜索算法：对每个被调用的函数，如果调用栈的深度小于最大深度，就分析被调用的函数，否则不分析该函数。 47

6.2 过程间分析算法 51

7.1 分析引擎组件示意图 55

8.1 一个简单的例子 59

8.2 使用单个数组模拟内存，指针也分配空间 62
8.3 函数参数指针p被用作一个具体指针。 63
8.4 函数参数指针p被用作一个符号指针。 63
8.5 函数参数指针s既被用作一个具体指针，又被用作一个符号指针。 63

9.1 一个从实际代码中简化出来的例子，在函数malloc_arg1中，返回值1/0指示着内存分配的成功或失败。在函数malloc_arg2中，返回值与内存分配的成功与否没关系。函数在L2处返回时，出现内存泄漏。 75
9.2 Analysis system overview 76
9.3 The escape model used by the analysis. 79
9.4 The lattice used in data flow analysis. 80
9.5 函数返回值与malloc有关 81
9.6 malloc行为的前置条件是g == 0 82
9.7 函数返回值与malloc无关 82
9.8 库函数建模语言 83
9.9 which 2.16中的内存泄漏：result从find_command_in_path()的调用中得到一块动态分配的内存，如果程序执行到代码L行处，free(result)就被跳过了，开始新的循环，result得到一块新分配的内存，旧的内存就被泄漏了。 87
9.10 wget 1.10.2中的内存泄漏：ftp_response()分配一块内存并将其地址存入它的第二个参数指向的变量。因此respline在程序M行处获得一块动态分配的内存，如果程序执行到L行处，内存没有被释放，函数就返回了，造成内存泄漏。 88

第一章　程序的正确性分析

如何保证程序的正确性是软件工程中的一个重要问题，也是一个一直都没有得到很好解决的问题。美国国家标准和技术委员会(NIST)在2002年的报告中指出，软件缺陷对美国经济每年造成的损失达595亿美元。

本文试图通过对程序进行自动化的数学分析，来帮助程序员提高程序的正确性。在开始描述如何对程序进行数学分析之前，先来看看其他领域的工程师是如何用数学来帮助自己保证设计的正确性的。

设想一个土木工程师设计了一座桥。在让工人开始建造之前，他一定要对他的设计进行计算。他根据设计需求列出一系列微分方程，然后求解，看是否会有异常情况出现。等他确信它的设计万无一失之后，他才会交付给施工人员，由他们建造出最终的产品——桥梁。

在软件工程中，上面所说的设计，建造，产品又分别对应什么呢？人们很容易认为软件工程师编写出来的程序代码就是最终的产品，而用自然语言书写的设计文档则是设计，软件工程师既是设计者又是建造者。事实上，还可以有一种更合适的看法。

在软件的生产过程中，软件工程师和土木工程师一样，只是设计者，只不过软件工程师的设计不是一张设计图纸，而是对软件产品的使用编程语言的描述，也就是程序的源代码。源代码是对软件的设计的最精确的描述。而可在机器上执行的二进制程序才是最终的产品。那么从设计（源代码）到产品（可执行代码）的建造过程是由谁完成的呢？是编译器。也就是说，在软件的生产过程中，从设计到产品的建造过程被自动化了。工程师只需要完成产品的设计，最终的产品就可以被自动建造出来。

但是，也正是因为有了这么容易的建造过程，软件工程师才忽略了对于其他工程师来说最重要的一步：对设计正确性的检验。软件工程师设计完了产品，直接建造出来，就开始使用，或者说测试。如果在产品使用过程中发现了问题，再回头去修改设计。然后再根据新的设计建造出新的产品，重复前面的过程。这就是现在人们使用的编码，编译，测试的流程。

这种生产方式对土木工程师来说是不可想象的。试想土木工程师在设计完

一座桥之后，直接交给工人建造。造好之后首先看桥塌不塌，如果塌了，回去修改设计；如果不塌，跑辆卡车看塌不塌......没有人能容忍这样的土木工程师。

所以，软件工程师应该借鉴其他工程师的做法，在编码完成之后增加一步对代码的数学分析，就可以提高设计出的程序代码的正确性。

1.1 程序的正确性分析

使用数学对程序进行验证或者说分析，分析的是什么呢？程序的二进制表示是最终的产品，程序的源代码是这个产品的设计。使用数学来分析的应该是软件的设计，也就是程序的源代码。本文假设从源代码到机器代码的翻译过程是可以保证正确的，现在也有很多工作在做编译器本身正确性的验证，这部分内容并不在本文讨论的范围内。

为了对程序进行推理分析，需要三个要素：

- 程序操作的模型。程序固有的操作模型是一台真实的计算机，但是这种程序固有的操作模型是不利于对程序的正确性进行分析的，所以需要建立一个新的模型来对程序进行分析。建模有多种方法，例如可以使用数学中的格，自动机，或者是完整的程序状态模型。建模方式的不同决定了分析方法的不同，前面这三种建模方法对应的分析方法分别是数据流分析，模型检测和符号执行。

- 程序正确性的表述。程序正确性的表述是一个很困难的问题，甚至无法完全地描述一个程序的正确性。在本文里，除了使用一些众所周知的正确性描述（比如被解引用的指针不能为NULL）以外，还用程序中的断言(assertion)来描述程序正确性的要求。使用一些类似于自动机描述语言的方法也在考虑范围之内，但并不是本文讨论的重点。

- 程序的语义，也就是程序是如何操作模型的。程序设计语言标准对语言的语义作了精确的定义，但是这种定义是针对语言如何操作一台真实的物理计算机的，本质上规定了语言应如何被翻译成机器语言。而我们定义的模型并不是真实的物理机器，所以我们需要根据语言的语义重新定义语言是如何操作我们设计的模型的。这往往很困难，因为计算机语言

被设计为操作计算机，要重新定义它对模型的操作，必然会带来不精确。在下面的论述中会看到这一点。

程序的源代码描述的是对内存状态的操作。程序有一个输入状态，经过操作之后，得到一个输出状态。那么程序正确性，应该是通过对程序运行过程中的或者是最终的内存状态的描述来确定的。

对内存状态的正确性描述，有一类是对所有程序都适用的，这一类的描述是一些普适的规则，比如：被dereference的指针不能为NULL，除数不能为0等等。

另一类的描述只适用于一个具体的程序，需由人工给出。人们为此也想了不少的方法，比如发明了很多spec语言。在本文里，使用C语言本身来描述在某一个程序点状态应满足的条件，也就是断言(assertion)。这对程序员来说是非常方便的。尽管对程序在某个点处要满足的全部正确性条件进行描述是困难的，但是程序员在编写程序的时候，一定知道程序要满足的一部分正确性条件。而这些要满足的条件并不会在程序本身的谓词中得到完全的体现。或者有些是程序员假设默认成立的，但是往往这些条件不会在所有情况下都成立。这时如果程序员能够用断言将这些部分正确性条件顺手描述在程序中，会对要进行的静态分析有莫大的帮助。并且书写这些部分正确性断言对于程序员来说不会成为很大的负担，相反，还会在某种程度上帮助他们对程序进行思考。

在程序的执行过程中，程序在每一点的状态都是具体可知的，这时对断言或者其他正确性条件的检查就变得十分平凡，只需要读取程序的内存状态，检查它是否满足正确性条件的描述。这样的做法，正是现在使用最多的，也就是软件的(动态)测试。给一个输入，运行程序，看结果是否满足正确性要求。软件测试的不足之处在于它只能检查在一种输入下的正确性，换一种输入，则需要重新执行一遍程序。然而程序的输入是无穷多的，在一般情况下不可能穷举所有的可能性进行测试。这也就是测试不能保证程序正确的原因。

既然输入的情况是无穷多的，那么不妨对它不作任何假设，就像数学中解方程一样只假设它是一个未知量，用一个符号表示。输入是未知的，程序不能真正运行，只能以静态的方式对程序语义进行模拟执行，称为符号执行。在执行的过程中，程序的状态，简单的说就是变量的值，是以符号表示的。可以根据所执行的程序路径，得到一组路径条件。路径条件是对符号值的约束，输入变量只有满足这样的约束，才能够是程序执行这条路径。如果约束不能够被满

足，则这条路径是不可行的。

有了路径条件P，碰到正确性条件C，只需要检查一下$P \Rightarrow C$是不是有效的（valid，意思是在任何情况下都成立，有效性的精确定义可参考数理逻辑教材[27]）。如果不是有效的，那么$P \wedge \neg C$就是可满足的，意思是在P成立的条件下，C有可能不成立，也就是说程序有可能到达错误的状态，那么就可以报一个错误警告，达到了检查程序错误的目的。

以上就说明了对程序进行数学分析的基本思路。目标明确了，但是要达到这个目标还有许多的困难。

一个困难就是软件产品的设计一般来说都很大。一个产品有上百万行的源代码是非常普遍的。Dijkstra, Hoare等计算机科学的先驱早就提出了用数理逻辑对程序进行手工分析，并且提供了所需要的理论工具，比如Hoare逻辑，但是一直没有得以广泛应用。一个重要的原因就是程序规模太大，没有公司或个人能够负担得起手工对程序进行严格的分析。

所以本文关注的一个技术重点就是自动化的分析，在分析过程中不需要人工干预。但是自动化就会带来不精确，主要体现在如下几个方面中：

1. 在有非线性运算，指针，数组，结构和变量是符号值情况下对程序操作语义的模拟；

2. 对循环的展开不能够穷尽，循环不变式无法自动得到，无法穷举状态空间；

3. 对函数调用的效果不能够忠实的模拟。

上面的原因会导致对程序的分析有近似的地方，从而出现漏查错误或者误报假错。本文后面几章针对这些自动分析必然存在的缺陷，提出了种种方法来弥补，目的是尽量减少由近似导致的虚假报错。本文各章内容如下：

第二章介绍一些背景知识以及对传统的数据流分析进行概述。

第三和四章针对C语言的所有语言特性，包括指针，数组，结构，建立内存模型，从而做到在有符号值的情况下对语句操作语义的精确模拟。

第五章针对过程内分析描述一套完整的算法。为解决无法穷尽遍历程序状态空间的问题，用路径可行性判定来削减不可达的状态空间。

第六章针对函数调用语句，讨论对函数的行为做自动总结的方法，得到近似的模拟函数效果的传输函数。

第七章描述整个系统在开源编译器Clang上的实现，说明重要的模块结构。

第八章描述了一种利用符号执行进行自动测试数据生成的方法。

第九章描述一种跨过程的自动内存泄漏检测算法。

第十章讨论相关工作。

第十一章讨论尚未解决的问题以及未来的工作。

1.2 本文的贡献

本文对C程序的静态分析有如下的一些贡献：

- 根据程序精确符号分析的要求和C语言的语义设计了一种新的内存模型，使得对程序进行精确的符号分析变得可行（第三章）。
- 对如何进行C程序的符号分析给出了详细的算法描述，用于C程序完整执行状态的模拟（第四章）。
- 设计了针对单元C程序的自动测试数据生成算法，并实现了相应的工具（第八章）。
- 设计了过程间的自动内存泄漏检测算法，并实现了相应的工具，检测出了真实程序中的内存泄漏缺陷（第九章）。
- 根据上述的模型和算法，在开源编译器Clang [17]上实现了一个完整的静态分析框架。

第二章 数据流分析

在编译器发展的早期，由于生成优化代码的需要，人们开始对被编译的程序进行分析，试图得到关于程序的数据流信息。作为一个简单的例子，考虑下面的代码片段，

```
...
x = 3;
*p = 4;
y = x;
...
```

如果要对x做常数传播，需要知道p是否有可能指向x,只有在p不会指向x的情况下，才能把程序变换成

```
...
x = 3;
*p = 4;
y = 3;
...
```

精确的说，数据流分析是一组在不运行程序的条件下，从程序中获取数据流信息的技术。由于不实际运行程序，所以也叫静态分析，这是相对于通过实际运行程序来获取数据流信息的动态分析来说的。人们常常关心的数据流信息有：可到达的变量定义，可用的表达式，别名信息等等。

从分析的精度来分类，数据流分析可以分为流不敏感，流敏感和路径敏感3种。

1. 流不敏感的分析一般给出的是一个函数整体的数据流信息，比如一个函数有可能修改哪些变量。

2. 流敏感的分析给出一个函数的控制流图(control flow graph, CFG)上每一点对应的信息。

3. 路径敏感的分析对函数CFG上每个点可能会给出多个信息。沿着不同的路径到达同一个程序点很可能会产生不同的状态信息，路径敏感的分析保留这些不同的信息，而流敏感的分析在控制流汇聚的程序点处会将不同分支传进来的状态汇聚成一个状态。

在传统的编译优化领域，人们主要关心流不敏感和流敏感的数据流分析，因为在编译时进行的分析需要很快的完成，而路径敏感的分析由于代价比较高，基本不被使用。另一方面，由于编译优化需要的信息是保守的，也就是在任何情况下都要成立的信息，而路径敏感的信息不具有这一特点，故也不被使用。然而在以错误检测为目标的静态分析中，需要的是精确的信息，路径敏感的分析方法就变得非常的适合。

这一章介绍传统的编译领域研究和使用的流不敏感和流敏感的数据流分析，为以后章节中的路径敏感的分析打下一些基础。

2.1 基本的定义

注意，这里定义的概念都是在源语言层次上的。在中间表示这个层次上也有相应的极为类似的概念。关于为何要在源代码层次上做分析在第七章中有详细的论述。

对于程序语句的序列，可以把它们划分成的最小的单位是基本块，基本块是满足以下二个条件的最长的语句序列：

1. 控制流只能从基本块的第一条语句进入。

2. 控制流只能从基本块的最后一条语句出去。

当程序的语句被划分成基本块之后，可以用一个图来表示基本块之间的控制关系，称为控制流图(CFG). CFG的节点就是基本块。CFG的边由以下规则添加：从基本块B到基本块C有一条边当且仅当程序的控制流可以从B出来进入C.

一个程序点定义为CFG上的一点，每2条语句之间有一个程序点，每个语句前后各有一个程序点。在C语言标准中对程序点有着更精确的定义，称为sequence point，可参考C语言的国际标准[43]。

一个程序点处的状态定义为程序运行到该点时的所有变量的值。

通常的数据流分析不要求知道每个变量的具体的值，而是把状态映射到一个抽象的域里，称为数据流值域。例如，我们关心变量在程序点处是否有定义，那么变量的值就被映射为$true$或者$false$. 状态就变成了一个位向量，每一个位对应于一个变量，该位是1表示该变量有定义，是0表示该变量没有定义。

通常我们把状态映射到的数据流值域是一个格(lattice)，数据流分析操作的就是格里的元素。格是一种代数结构，包括一个集合L，以及二个操作，$meet$操作，记为$\sqcap$, $join$操作，记为$\sqcup$,满足以下几个性质：

1. $\forall x, y \in L$, 都存在着唯一的z和$w \in L$ 使得$x \sqcap y = z$和$x \sqcup y = w$.(封闭性).
2. $\forall x, y \in L, x \sqcap y = y \sqcap x$ 以及$x \sqcup y = y \sqcup x$. (交换性)
3. $\forall x, y, z \in L, (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ 以及$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$. (结合性)
4. L中存在二个唯一的元素：底，记为$\perp$, 顶，记为$\top$，使得$\forall x \in L, x \sqcap \perp = \perp$, $x \sqcup \top = \top$. (存在唯一的顶和底元素)

在数据流分析中用到的大多数的格都以位向量作为它们的元素，$meet$和$join$分别为按位与和按位或。

每条语句都会对状态产生一定的影响，把在它之前的状态映射到它之后的一个新的状态。于是每条语句就对应着一个传输函数(transfer function)，这个函数将一个数据流值映射到另一个数据流值。因此传输函数是一个从格到它本身的函数，$f_S : L \to L$。它刻画着程序语句对状态的影响。对于同一条语句来说，根据不同的数据流分析问题，它会对应于不同的传输函数。

一个函数在格中的不动点是一个元素$z \in L$, 使得$f(z) = z$.

2.2 数据流问题的分类和求解方法

数据流分析的问题可以从几个方面进行分类：

1. 数据流分析所提供的信息。

2. 所使用的格，格中元素的含义，以及定义在格上的函数。

3. 信息流的方向：前向问题，信息流的方向与程序执行的方向相同，后向问题，信息流的方向与程序执行的方向相反。

下面描述编译优化里面用到的最重要的几类数据流分析问题。需要指出的是，进行数据流分析的对象程序的存在形式有很多种，比较常见的程序的表达形式有接近于源代码层次的AST，以及基于SSA的接近于汇编的三地址码中间形式。在不同的表示层次上进行数据流分析的方法和难度是不一样的。对于不同的表示形式来说，有些数据流分析问题容易解决，有些问题则难一些。所以在实际应用中应该根据需要选择不同的程序表示形式来分析不同的数据流问题。

- 到达定义。到达定义确定哪一个变量的定义(对变量的赋值)可以到达变量的使用地点。这是一个前向问题。使用位向量的格，每一个位对应于一个变量定义。

- 可用表达式。可用表达式确定在程序的每一点处哪些表达式是可用的。一个表达式在一个程序点处是可用的，意味着在表达式被求值的地点和该程序点之间的所有路径上没有对表达式中的变量的赋值。这是一个前向问题。位向量中的每个位对应于一个表达式的定义。

- 活变量。活变量确定在某个程序点处某个变量是否会在该程序点到程序出口之间被用到。这是一个反向问题。可以让每个变量使用对应于一个位，也可以让每个变量对应于一个位。

- 前向暴露的使用。前向暴露的使用确定哪些变量使用能够被一个特殊的变量赋值到达。这个问题和到达定义是对偶问题。也是前向问题，每个位对应一个变量的使用。

- 拷贝传播分析。拷贝传播分析确定在一个拷贝赋值$x \leftarrow y$和一个对x的使用之间是否有对y的赋值。这是一个前向问题，每个位对应于一个拷贝赋值。如果没有对y的赋值，就可以把对x的使用直接替换成y.

- 常数传播分析。常数传播分析确定在一个常数赋值$x \leftarrow c$和一个对x的使用之间是否有其他的对x的赋值。这是一个前向问题，每个位对应于一个常数赋值。
- 部分冗余性分析。部分冗余性分析确定哪个计算被重复执行了两次。这是一个双向问题。每个位对应于一个表达式计算。

上述的几种数据流分析问题并不是唯一的，但是它们是最重要的几种，也是编译优化里用得最多的。

解决数据流分析的方法有很多种，包括：

- Allen的强连通区域方法。
- Kildall的迭代法。
- Ullman的T1-T2分析。
- Kennedy的node-listing算法。
- Farrow, Kennedy, Zucconi的图语法方法。
- 消去法，比如区间分析。
- Rosen的语法导向方法。
- 结构分析。
- slotwise分析。

这里面使用最多的是迭代分析。

2.3 迭代分析

这一节对迭代分析作一个简单的介绍，更多的信息请参考[52] [1]。这里以前向分析为例给出迭代分析的实现框架。

假定有一个控制流图$G = \langle N, E, entry, exit \rangle$以及一个数据流信息格$L$.要计算出$in(B), out(B) \in L, \forall B \in N$，也就是说，我们要得到进入和流出每一个基

本块的数据流信息in(B)和out(B).可以根据要解决的数据流问题列出如下的数据流方程：

$$in(B) = \begin{cases} init & \text{for B = entry} \\ \sqcap_{P \in Pred(B)} out(P) & otherwise \end{cases} \quad (2.1)$$

$$out(B) = F_B(in(B)) \quad (2.2)$$

这里方程2.1中的init表示函数入口处的合适的数据流信息初始值。$\sqcap_{P \in Pred(B)} out(P)$表示B的入口信息由所有的前驱基本块的流出信息的$\sqcap$运算确定（根据不同的问题也有可能是$\sqcup$运算）。F(B)表示基本块B对应的传输函数。

计算数据流信息的算法Worklist_Iterate见图2.1.

图2.1中的算法只是迭代算法的一种实现，在实际应用中，迭代算法可能有很多种优化的形式。迭代算法的主要步骤是：初始化数据流信息状态以及工作列表，然后对工作列表中的节点，更新它的信息，同时把被它影响的节点加入工作列表中。由于传输函数在数据流信息的格中有固定点，所以迭代算法是可以终止的。

2.4 总结

本章概述了传统数据流分析的方法。给出了问题的基本定义，常用的算法，并详细描述了使用最多的迭代算法。本章描述的数据流问题基本都可以用布尔变量，也就是位向量来描述。如果要了解传统数据流分析请参考[1][52].本章为后面几章描述更复杂的路径敏感的符号分析奠定了基础。

```
procedure Worklist_Iterate(N, entry, F, dfin, Init)
  N: set of nodes of CFG.
  entry: entry node of CFG.
  F: transfer functions, (node, state) -> new state
  dfin: when applied to node, get its in state.
  Init: initial state in L.

begin
  B, P: node
  Worklist: set of nodes
  effect, totaleffect: state in the lattice L
  dfin(entry) := Init
  Worklist = N - {entry}
  for each B in N do
    dfin(B) := T

  repeat

    B := get node from Worklist
    Worklist -= {B}
    totaleffect := T

    // Calculate the input information of B.
    for each P in Pred(B) do
      effect := F(P, dfin(P))
      totaleffect /\= effect

    // Update the input information of B.
    if (dfin(B) != totaleffect)
      dfin(B) := totaleffect
      Worklist += Succ(B)

  until Worklist is empty
end
```

图 2.1: 求解数据流问题的迭代算法

第三章　程序分析的模型

本文所进行的程序分析是针对源代码的静态分析。静态分析是在不运行程序的条件下，通过分析程序的源代码或者可执行代码来获得程序在运行时的信息的技术。由于不能真正执行程序，要获得程序在运行时的状态信息，必须对程序在运行时的状态建立一个模型，通过模拟程序的语义，对这个模型进行操作，从而获得运行时状态的一种近似表示。

通过前面的背景介绍可以看出，传统的数据流分析主要从程序中提取布尔信息，比如变量是否被某个函数修改了，一个表达式的值在某个程序点是否可用，表达式之间是否具有别名关系等等。获取这样的布尔信息只需要对程序建立一个比较粗略的模型即可。比如可以建立一个表示布尔信息的位向量的格，然后把程序的操作语义进行简化，变成对格的操作。

由于传统的数据流分析所服务的目标是编译优化，这样的布尔信息也就够用了。但是本文的目标是对程序进行自动的错误检查和测试数据的生成，仅仅知道布尔信息就不够用了，必须对程序的运行时状态建立新的模型，获取更丰富的信息，才能够进行错误检查或是服务于其他的目的。

为此，我们试图对程序状态保存最精确的信息。在函数开始时，假设所有的输入变量是未知的，给它一个符号值。这个符号值具有和变量相同的类型。随着程序操作的进行，这些符号值会在程序操作的作用下衍生出新的符号值。我们也会通过程序路径中的条件表达式得到关于这些符号值的约束。有了在程序中每一点的状态的符号描述和关于符号的约束，把要检查的错误也表示成一个关于变量的值的条件表达式，再通过检查这个条件表达式在当前约束下是否可满足，就可以判定错误是否会发生了。

以上是对本文中符号分析整体思想的描述。事实上这种思想在七十年代就已经提出来了[46]，称为符号执行。但是由于种种技术上的困难，一直没有得以广泛应用。下面将逐步的推导出一系列的程序状态建模方法，最终得到一个较为完善的能够进行精确符号分析的模型。

3.1 名字-值模型

最基本的对程序状态的表示方法是建立一个从变量名到变量对应的值的映射。这个模型也是最初人们提出符号执行时所使用的模型[46]。每个变量都对应一个整数值（或是符号值）。它简单，直观。程序运行的状态表示为程序中所有变量的值。输入变量包括函数参数和全局变量。输入变量的值在函数开始时为符号值。例如：

```
void f(int x) {
  int y;
  y = x + 3;
}
```

这个模型的局限性在于它无法处理C语言中的指针，数组，结构等成分。考虑下面的代码：

```
void f(void) {
  int a;
  int *p = &a;
  *p = 3;
}
```

这里p的值是什么呢？在程序的语义中，p是一个指针变量，它的值是a的地址。但是在这一节的模型里，并没有地址这样一个概念，从而无法表示p的值。

3.2 数组模型

3.2.1 大数组模型

在[65]中，我们提出了一种用数组来模拟内存的符号执行方法。在这个模型中，计算机内存被一个大数组来表示。程序中所有的变量都分配在这个大数组上。对变量的操作也就变成了对数组单元的操作。变量的地址就可以用它所在的数组单元的下标来表示。为了实现这样的模型，需要在符号表中记录每个变量在大数组中的下标。例子：

```
void f(void) {
  int a[10];
  int *p = a;
  *p = 3;
}
```

这个模型用数组来模拟内存，是符合物理内存模型的。但是它有一些缺陷，使得在这个模型上进行符号执行有很大的限制。这个模型本质上是把内存地址抽象成为一个具体的整数，对于程序分析来说，这是个好的抽象，因为程序一般不关心地址的真实值是多少，进行的操作大多是相对的位移操作和比较，如p++, p != NULL.这样的操作用整数来模拟是完美的。但是这种建模方法，要求对每一个内存对象都知道它的具体长度，否则我们便不能在大数组上为其分配空间。如果程序中有如下的语句：int *p = malloc(n);其中n是一个变量，其值是符号值。如果使用这个程序状态模型，便无法模拟这个语句的操作。

3.2.2 独立数组模型

我们可以对上面的模型进行一点改进，不用一个大数组来模拟整个内存，而对每个内存对象单独分配一个数组。那么对变量到值的映射的相应的改动就是：对指针变量，不能仅仅记录一个整数来表示它指向的地址了，而要用一个(变量，位移)对来表示它的值。因为不仅要知道一个指针变量的相对地址，还得知道它指向的是哪个变量。

但是这样又带来了新的问题：如何表示没有显式变量名的内存对象。考虑下面的代码：

```
struct A {
  int d1;
  int d2;
};

struct B {
  struct A *f;
```

```
  int d;
};

void foo(struct B *p) {
  struct A *q;
  q = p->f;
  p->f = (struct A*) malloc(sizeof(struct A));
}
```

在这里又如何表示q和`p->f`的值呢？B::f这个域在本模型中并没有独立的实体存在，而仅仅作为一个数组单元存在于运算中。既无法表示它指向的对象，因为它指向的对象是malloc()分配的堆对象，没有独立变量名，也无法表示它本身的值，因为它本身是个指针，它所对应的数组单元无法承载一个(变量，位移)对。

3.3 基于区域的三元模型

3.3.1 初步介绍

试图建立变量到值的直接映射这种建模思路困难重重，我们重新考虑程序分析要解决的本质问题是什么。

第一章已经提出：用数学对程序进行分析是提高程序正确性的很好的途径。但是计算机程序和数学这两个看似相近的实体是否真的可以直接进行转化呢？它们两个之间是否存在着什么本质区别阻碍着人们使用数学对程序进行分析呢？经过思考，我们发现了程序和数学的本质区别：存储。

事实上存储这个概念在一开始学编程的时候所有人就已经知道了。程序就是一串操作内存的指令，程序的输入就是内存状态，输入也是内存状态。计算就是一个执行状态的序列。

人们看到的程序和看到的数学的样子非常相似。例如在数学中有

$$x = 3$$

在程序中也有

$$x = 3;$$

很容易把这两个表达式等价起来，但它们并不一样。

数学表达式表示的是一组静态的量之间的关系。这种关系是不会变的。x在任何地方都等于3。而程序表示的是一组动态的操作。x在这一点上等于3，但是到了以后，x可能被改变为5。这种静态关系和动态操作的区别是比较明显的。解决从动态到静态的转化所使用的方法就是通过对程序操作语义的模拟所进行的符号执行。

而程序分析所要解决的本质问题就是要从程序代码中提取出一组关于程序中的值之间的静态关系。

第二个区别则不是那么的明显了。请看下面的表达式

$$x = 5$$

如果这是个数学表达式，可以用上面的3直接替换x，因为在数学中，上面的式子表示x和3是等价的。替换完了之后得到3 = 5，这个关系是不成立的，因为3不等于5。

如果这是个程序语句，那么就不能用3替换x，替换进去得到3 = 5是没有意义的。因为=左边的x实际上表示的是x的存储位置。而y = x中，用3替换x又是完全合理的。

这说明程序中的表达式，或者叫名字，在不同的情况下具有不同的意义。

要搞清楚这种区别，得区分三个概念：名字，存储，值。一个名字是出现在程序中的表达式，如：x, &x, 5。有些名字有相应的存储空间。比如int x;表示在栈上分配4个字节的存储空间，并将其绑定(bind)到x这个名字上。存储空间上的比特位以某种方式解释成为一个值，这个值才等价于数学里的值的概念。

现在有了名字，存储，值这三个概念，就可以定义一个名字的值了。这里我们说的名字和表达式是一个东西，只是用名字更能体现概念上的区别。

和数学里不一样，程序中的名字有左值和右值之分。名字的右值就是通常意义上对表达式求值所得到的值。名字的左值是这个名字所代表的存储空间的地址。所有的名字都有右值，但不是所有的名字都有左值。下面这个表格给出了一些在例子程序结束时，表达式的左值和右值。

```
int x, *p;
```

```
x = 3;
p = &x;
x = *p;
```

名字	左值	右值
x	x的地址	3
&x	无	x的地址
p	p的地址	x的地址
*p	x的地址	3

3.3.2 精确定义

在分析程序时，我们看到的是程序中的表达式，我们要提取的数学关系却是关于内存中的值的关系，而不是关于程序中的名字的关系。所以程序分析要做的另一件事就是建立从程序中的表达式到内存位置的映射，再通过内存位置到值的映射得到程序语句实际操作的值，从而得到最终的关于值的数学关系。

这件事情对于C语言来说尤其复杂，因为C语言中的表达式十分复杂，别名关系也是能通过很多方式建立。

从编译器的角度来看，C语言中的表达式分为2类：左值表达式和右值表达式，简称为左值和右值。注意这里的左值和右值指的是表达式，请不要与上面讲的表达式的左值和右值混淆。在上下文比较清楚的地方，就用左值和右值来简称左值表达式和右值表达式。

指代内存中的对象的表达式称为左值表达式，其余的表达式都是右值表达式。

但是在静态分析中，仅用表达式及其对应的左值右值这些概念是不够的。因为C语言中有指针这样一个概念，仅用表达式到值的对应这样的2元模型很快就会陷入混乱之中。

需要对程序进行解释，并且使用一些数学对程序进行推理。为此，我们设计了程序的解释模型。

把程序中出现的概念分为三个大类：语言的，存储的，数学的。对于一个表达式，用一个三元组来解释它：

$$\text{expression} - \text{memory object} - \text{value}$$

expression表示的是编程语言中的概念。它是对表达式的一种文本上的表示。在编译器实现中，它对应于Expr数据结构。

接下来，表达式有可能对应于内存中的一个对象(称之为左值表达式，或者直接称为左值)。在静态分析中，我们设计了“区域(region)”这一概念来表示内存对象。

一个区域是对一个内存对象的一种抽象表示，因为一个内存对象总是占据地址空间中的一段连续的区域，所以用区域这个名词来表示对内存的一种抽象。

在C语言中，一个内存对象可以是一个变量，一个数组，一个数组元素，一个结构，一个结构的域，一个动态分配的内存块等等。对于一个正在运行的程序来说，它的地址空间是线性的。但是在进行静态分析的时候，我们关心的信息超出一个线性的模型。不仅需要知道表达式到区域的对应关系，还需要知道区域之间的结构关系，或者叫层次关系。比如，一个数组的区域本来是由它的所有的元素的区域组成的。但是对数组本身，和它的所有的元素本身都用不同的区域表示。同时由于一个数组元素的区域是从属于数组的区域的，在区域之间建立父子关系，数组元素的父区域就是数组区域。对于程序的栈，堆，静态存储，也用三个区域(stack region, heap region, static region)表示。这样，如果数组是局部数组，那么它的父区域就是stack region。

有了区域之间的层次关系之后，很多问题都可以通过查询区域之间的层次关系来解决。比如区域的区分问题。不同的struct的同一个域对应的区域，尽管它们的名字都一样(都是那个域的名字)，可以根据它们的父区域(从属的struct)的不同来进行区分。

第三个概念，value，是数学上的概念。这个值是内存中存储的值：比特位以某种方式进行解释得到的。把这些值分为两类：一类是描述内存地址的，称为位置值(location)，其余的都是描述普通值的，称为非位置值(non-location)。

这里重新更精确地定义一下表达式的左值和右值。表达式的左值是表达式指代的内存对象的指针，所以只有左值表达式才有左值。表达式的右值定义为，如果是左值表达式，则是其指代的内存对象的值，如果是右值表达式，则就是该表达式的值。在程序中，表达式的左值都是指针，右值可能是非指针，也可能是指针。比如int型变量x的左值就是一个指向x的指针，右值是这个指针指向的内存位置上存储的int型整数。表达式&x没有左值，因为它不对应一个

内存对象，它的右值是一个指向x的指针。

指针不是位置，或者说不仅仅是位置。指针是位置加上一个类型。类型有大小，所以指针指向的对象也有大小，位置仅仅表示出了指针指向的对象的起始位置。

那么既然表达式求值得到的值是在上面模型中的第3个域，也就是数学值中，而数学值只有位置和非位置两种，怎么来表示表达式求值得到的指针呢？答案是我们用上面定义过的区域来表示指针。区域不仅有起始位置的概念，它自己还有一个范围和类型的概念。比如一个数组的区域虽然和它的第一个元素的区域概念上对应的起始位置是一样的，但是它们的类型和范围是不一样的。从而就可以区别出指向数组的指针和指向数组元素的指针。

总结来说，表达式的左值是指针，右值是指针或者非指针。指针值用一个区域来表示，非指针值则用其本身来表示。

下一章将会详细描述如何从表达式得到值。

3.4 总结

本章分析了3种内存建模方法各自的优缺点。通过对程序中3种成份：表达式，内存，值的分析，建立了基于区域的三元内存模型。同时对一些易混淆的概念作了精确的定义和分析，为以后的论述打下了基础。

第四章　C语言语义模拟

4.1　存储的建模

我们把存储(Store)建模为一个从区域到值的映射。区域是第三章定义的对内存对象的抽象表示。值也是第三章定义的数学意义上的值。值分为二种：位置值和非位置值。位置值用区域来表示，非位置值则按照常规表示。

先定义一些记号。如果Var表示变量，Region(Var)返回该变量对应的区域。如果R表示一个区域，Store(R)返回该区域对应的值。lvalue(expr)表示表达式expr的左值，这是个指针值，在我们的分析系统中，用区域来表示。如果expr是个变量，那么就直接返回该变量对应的区域Region(expr)。
rvalue(expr) 表示表达式expr的右值。如果expr是个左值表达式，先求得expr的左值，再通过查询Store得到该区域对应的值，记为Store(lvalue(expr))。如果expr是个右值表达式，则直接根据程序语义求得该值。

4.2　声明的处理

一个声明告诉编译器如何解释一个标识符，规定了标识符的属性。一个声明如果满足以下的条件，则是一个标识符的定义：

- 对一个对象，使得该对象的存储空间被分配。
- 对一个函数，包括了该函数的函数体。
- 对一个枚举常量或者typedef的名字，是其唯一的声明。

在分析程序时，一般只需要处理作为定义的声明。对枚举常量和typedef的名字的定义的处理不需要多讲。对函数的定义的处理分解为对它内部语句的处理。

处理变量定义时，为其分配一个区域，用来表示它对应的内存对象。如果是简单变量，一个区域就够了。如果是数组或结构变量，不仅要给变量本身分配区域，对每个数组元素或结构域都需要分配区域，并将其父区域设置为变量的区域。

在声明的修饰符中，storage class specifier有五种:typedef, extern, static, auto, register. 这里面，typedef只是为了语法上的方便而被放入此类的。extern表示这是个外部声明，它的定义在外部，我们也为其分配存储区域。static表示这是静态存储，我们在处理它的赋值的时候需要特殊处理。auto和register与静态分析无关。

在为变量声明分配了内存区域之后，在Store中建立从该声明到分配的内存区域的映射。

4.3 表达式求值

一个表达式是一个操作符和操作数的序列，规定了对一个值的计算。表达式的值有左值和右值之分。表达式的左值是指向它代表的内存对象的指针，记为lvalue(expr)。表达式的右值是通常意义下对它求值所得到的值。如果是左值表达式，则先求得它的左值，在通过查询Store 得到该区域对应的值。如果是右值表达式，则直接根据程序语义求得。

4.3.1 表达式求值的算法化

需要注意的是，不仅对不同的表达式的计算有先后顺序（一般来说按照它们在程序中出现的顺序计算），在一个单独的表达式的求值过程中，对它的不同的部分的计算也是先后完成的。所以在对一个表达式的求值过程中，会得到许多个中间状态。一个简单的例子：

$$x = 3; \tag{4.1}$$

输入状态是在赋值表达式4.1之前的一个点的状态。输出状态是在赋值表达式4.1之后的一个点的状态。在这中间，还有二个状态。首先要对4.1的右手边进行求值，得到一个中间状态，在这个状态中，表达式“3”的值被计算出来，是一个非位置值3。然后对4.1 的左手边进行求值，得到第二个中间状态，在这个状态中，表达式“x”的左值lvalue(x)被求出，是x对应的区域(region)。最后更新Store中lvalue(x)对应的值为3。

如果在表达式的子表达式求值过程中，状态产生了分裂，那么需要对分裂出来的所有状态分别继续进行运算。这样一个简单的赋值就有可能分裂出多个

状态。比如，数组的下标是符号值时，需要对其具体化，就要对每种可能的下标值生成一个新的状态。第5.6节对此有详细的描述。

4.3.2 Primary expressions

Primary expression有3种：identifier, constant, string literal. 需要说明的是identifier的求值。

	左值	右值
常量n	无	n
简单变量x	Region(x)	Store(lvalue(x))
数组变量a	Region(a)	无

对于标识符来说，如果是简单变量，则左值是它对应的内存区域，右值是它的区域对应的值。如果是数组变量，右值没有定义，左值是它对应的区域，也就是一个指向数组的指针。注意，数组变量表达式是唯一没有右值只有左值的表达式。其他的表达式都有右值，有的没有左值。

4.3.3 Postfix operators

后缀表达式里面要说明的是如何对数组表达式和结构成员表达式求值。

	左值	右值
a[expr]	ElementRegion(lvalue(a), rvalue(expr))	Store(lvalue(a[expr]))
s.d	FieldRegion(lvalue(s), d)	Store(lvalue(s.d))
`p->d`	FieldRegion(rvalue(p), d)	Store(lvalue(`p->d`))

ElementRegion(base, index)根据base表示的数组区域和index表示的指标得到一个数组元素区域。这个数组元素区域的父区域是base表示的数组区域，相应的指标是index。FieldRegion(base, field)根据base表示的结构区域和域的声明field得到一个结构域区域。

4.3.4 Unary operators

这里面需要说明的是取地址操作符和解引用操作符。

	左值	右值
`&expr`	无	lvalue(expr)
`*expr`	rvalue(expr)	Store(rvalue(expr))

&expr没有左值，因为它并不代表一个内存对象，右值是epxr的左值。*expr的左值是expr的右值，右值是expr的右值代表的内存对象对应的值。

4.3.5 Binary operators

一般求值规则

1. 如果所有的操作数都是具体值，则按照具体运算进行
2. 如果至少一个操作数是符号值，则按照符号运算进行
3. 如果至少一个操作数是未知的，则结果是未知的
4. 如果至少一个操作数是未定义的，则结果是未定义的
5. 如果既有未知值，又有未定义值，则结果是未定义的
6. 条件表达式的值是一个约束，需要结合其他的约束在一起求解，得到它的真假值。

$==, !=, <, >, <=, >=$ 这些操作符表示一个条件表达式。条件表达式一般出现在基本块的结束语句中。它们的值应该是布尔型的。但是在碰到这些表达式的时候只是先把它们的符号表示形式先构造出来，也就是先对表达式的两边分别求值，然后构造相应的符号关系表达式。等到处理结束语句的时候，再对这些表达式是否可满足进行求解，得到相应的布尔值。这种处理叫作lazy evaluation，好处是可以节省计算，仅当不得不计算的时候再进行计算。

&&, || 对于复合条件表达式来说，它们的两边是被分别求值的。先对左边进行求值。对于&&来说，如果左边是假，那么就没有必要再对右边进行求值了。对于||来说，如果左边是真，就没有必要对右边再求值了。

4.3.6 例子

考虑图4.1中的代码片段，用区域模型对该代码进行符号模拟。

第一步，在处理完函数foo中的3个变量声明语句之后，Store如表4.1所示：

在处理malloc语句时，malloc会建立一个新的region，sp的region对应的值更新为该region，Store更新为表4.2。

表 4.1: 处理过声明语句后的程序模拟存储状态

表达式	区域	值
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	undefined
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined

接下来处理语句`sp->p = &data;`。`&data`要求我们求出data的左值，为region 1。接着根据postfix operator的求值规则，先求出sp的右值region 7，然后用FiledRegion(region 7, p)得到`sp->p`的左值region 8，更新后的Store如表4.3所示。

下面处理语句`sp->p->d = 3;`。同上一步，求出`sp->p`的右值region 1，用FieldRegion(region 1, d) 得到`sp->p->d`的左值region 2，更新后的Store如表4.4所示。

下面处理语句`a[1] = data.d;`。首先得到data的左值region 1，用FieldRegion(region 1, d)得到`data.d`的左值region 2，从而得到右值3。a的左值为region 4，用ElementRegion(region 4, 1)得到`a[1]`的左值region 6，更新后的Store如表4.5所示。

这些内存区域之间的层次关系见图4.2。

图4.1中的代码中有较为复杂的指针，结构，数组操作，并且内存块之间的关系种类也很多，使用简单的名字-值模型和数组模型都无法精确的模拟该程序的操作。基于区域的三元模型则可以精确的模拟该程序的操作，并将内存块之间的关系表示出来。

4.4 总结

本章对基于区域的内存模型做了进一步的细化，将其应用到C的各种表达式的求值上，得到了一套可以完整模拟C语言语义的内存建模方法和表达式求

表 4.2: 处理过malloc语句后的程序模拟存储状态

表达式	区域	值
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	**region 7**
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
unnamed	**region 7**	**N/A**
unnamed.p	**region 8**	**undefined**

值算法。

重要的是，区域内存模型可以在不做指针分析的情况下，仅使用求值算法就自然的得到表达式到内存对象的正确对应，相互为别名的表达式会被映射到相同的内存对象上。这样就省去了别名分析的过程。这样的改进一方面得益于区域内存模型的先进性，另一方面也是由于我们进行的分析是路径敏感的分析，每条特定的执行路径上的信息是确定的。

表 4.3: 处理过$sp->p=\&data$ 语句后的程序模拟存储状态

表达式	区域	值
data	region 1	N/A
data.d	region 2	undefined
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
unnamed	region 7	N/A
unnamed.p	region 8	**region 1**

表 4.4: 处理过$sp->p->d=3$语句后的程序模拟存储状态

表达式	区域	值
data	region 1	N/A
data.d	region 2	**3**
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	undefined
unnamed	region 7	N/A
unnamed.p	region 8	region 1

```
struct s1 {
  int d;
};

struct s2 {
  struct s1 *p;
};

void foo(void) {
  struct s1 data;
  struct s2 *sp;
  int a[2];

  sp = malloc(sizeof(struct s2));
  sp->p = &data;
  sp->p->d = 3;
  a[1] = data.d;
}
```

图 4.1: 用于符号分析的例子：这个例子程序中有较为复杂的指针，结构，数组操作，并且内存块之间的关系种类也很多，使用简单的名字-值模型和数组模型都无法精确的模拟该程序的操作。基于区域的三元模型则可以精确的模拟该程序的操作，并将内存块之间的关系表示出来。

表 4.5: 处理过a[1] = data.d语句后的程序模拟存储状态

表达式	区域	值
data	region 1	N/A
data.d	region 2	3
sp	region 3	region 7
a	region 4	N/A
a[0]	region 5	undefined
a[1]	region 6	**3**
unnamed	region 7	N/A
unnamed.p	region 8	region 1

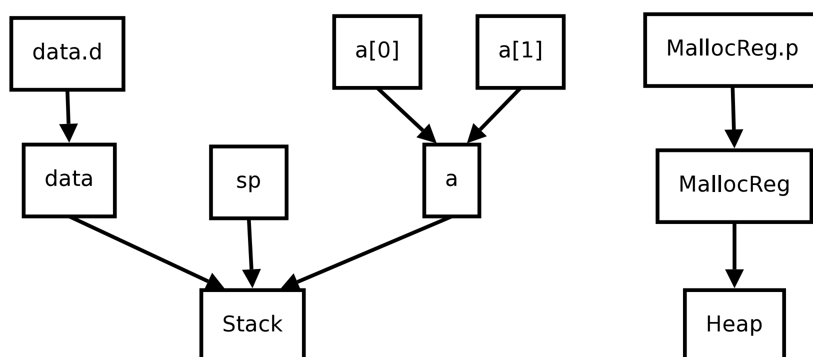

图 4.2: 内存区域的层次关系：箭头指向的为父区域，其中MallocReg表示的是malloc()分配的区域

第五章　过程内分析

上一章讨论了如何对程序中的表达式进行求值，包括求表达式的左值和右值。这一章描述如何进行单个函数内的符号分析。符号执行与数据流分析主要不同点在于，数据流分析以格为模型，只记录程序的部分数据信息，而符号执行以程序的完整执行状态为模型，记录程序的完整数据信息。分析的模型不同，也就决定了对程序的语义模拟不同，符号执行对程序语义的模拟更加完整，也更为复杂。

符号执行的目标是把程序转化成约束公式，约束中既包含程序中的路径条件，也包含要求程序满足的正确性条件或者程序员给出的断言。

函数是C程序的最基本的组织单元。C程序就是由一个个函数组成的。所以分析也以函数为单元进行。

在第二章已经知道流敏感的数据流分析给每一个程序点关联上一个状态。如果是前向分析，一个语句后面的状态由语句前面的状态经过语句所对应的传输函数所确定。在CFG上分支合并的地方，为了保证一个程序点只有一个状态与之相关联，不同分支传过来的状态要根据数据流分析的类型应用格中的meet或join操作合并成一个状态。

在面向程序正确性的分析中，如果也使用流敏感的分析，在控制流汇聚的节点合并前面的状态，则会造成信息的过分丢失，从而无法得到想要的结果。

因此在控制流汇聚的节点不合并状态，保留所有的状态，也就是对每个CFG的节点关联上不止一个状态。不同路径传播下来的状态在控制流汇聚的地方不会合并，而是保持原样继续传播下去。这样每个节点的前驱和后继形成了一条单独的程序执行路径。

路径敏感的分析中每个程序点处的状态是一条真实的程序执行路径到达该处的状态。而流敏感的分析中程序点的状态是对所有可能到达该点的程序路径的状态的总结，总体的近似。

5.1　扩展图

在流敏感的分析中，分析完之后产生的包含状态的CFG和原先的CFG结构

是一样的，只是在每个程序点处关联上了一个状态。

在路径敏感的分析中，如果把不同的(程序点，状态)对看成不同的节点，也就是说即便程序点相同，状态不同的节点也看成不同的节点，那么分析完之后产生的图的节点比原先的CFG的节点要多。把这个新的图称为扩展图(exploded graph)。

原始的CFG定义为(N, E, entry, exit), 其中N是所有顶点的集合，E是所有边的集合，entry 为唯一起始节点，exit为唯一结束节点。

定义一个扩展图(exploded graph)为(N', E', (entry,init_state), EOP), 其中N'是生成的(程序点，状态)的集合，E' = (a,b) 如果a的程序点到b的程序点有一条边在E中, (entry, init_state)是起始节点和初始状态，EOP是(exit,结束状态)的集合。因为不同的路径到达exit的状态不同，所以EOP中也包含了多个节点。

扩展图的节点是(程序点，状态)的集合，边则是根据原始CFG的控制结构得到的边，起始节点还是一个，结束节点则变成了多个。控制流图，带状态的控制流图和扩展图的对比见图5.1。

5.2 总体分析框架

分析的目标是为函数建立扩展图。这里使用的分析方法是前向分析。后向分析不利于状态在分支处的分裂，也不符合程序执行的直观感觉。最基本的，我们无法知道程序在结束时的状态信息，所以无法进行后向分析。

我们采用work list的算法框架，见图5.2.

5.3 初始状态

在函数的入口处，初始化状态。对每一个函数参数和全局变量，我们为其建立所对应的region,并将region对应的值初始化为符号值，代表一个未知量。对每个局部变量，为其建立所对应的region,并将region对应的值初始化为未定义值。

5.4 程序点

基本块包含常规语句和结束语句，结束语句主要包括：复合条件语

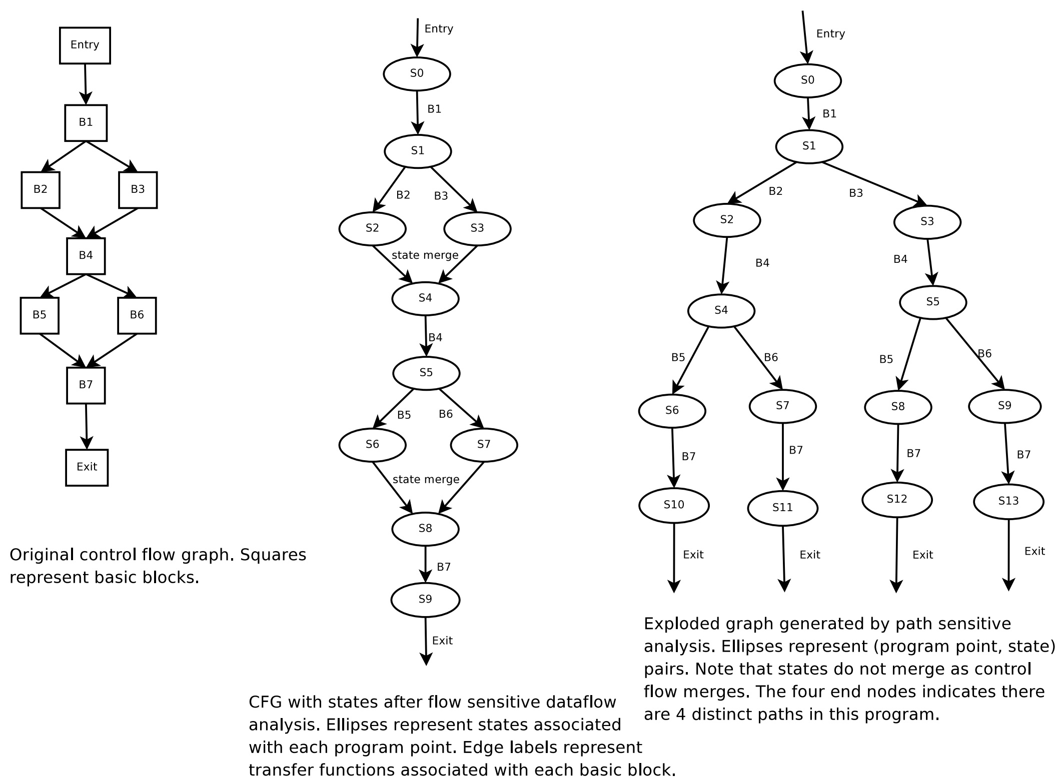

图 5.1: 控制流图，带状态的控制流图和扩展图的对比

句(&&, ||)，选择语句，if语句，while语句，do-while语句，for语句等等。一般来说基本块会包含一个结束语句，但是也有可能不包含结束语句，即所有语句都是常规语句。

在设计中区分3种程序点:

1. BlockEdge, 这是在二个block之间的一个点，记录了刚刚处理完的一个block和将要进入的一个block.

2. BlockEntrance, 这是在刚刚进入一个block的点，还没有处理该block的第一条语句。

3. PostStmt, 这是最多的一种程序点，在基本块的每一条非结束语句的后面都会有这样一个点。

```
WL: work list of exploded graph nodes.

Get the initial state and construct the first exploeded node, put it
into WL;

Initialize MaxSteps to a reasonable value, e.g. 30000;

while (MaxSteps > 0 && WL has nodes) {
  MaxSteps--;

  N = get node from WL;

  Process the node according to the kind of the program point;
  apply the transfer function to generate new nodes;
  put the new nodes into the WL;
}
```

图 5.2: 符号执行算法

5.5 对各种扩展图节点的处理

当从work list中取出一个扩展图节点之后，根据节点的程序点类型分别进行处理。

5.5.1 Block Edge – 对循环的处理

进行分析的初始节点就是从Entry基本块到第一个基本块的BlockEdge节点。之后，在每二个基本块之间都会碰到一个BlockEdge节点。

对BlockEdge节点的处理很简单，因为没有状态的变化。最简单的处理就是取出目的基本块B，生成进入B的BlockEntrance节点。

不过，为了防止路径出现无限循环，对每个基本块进入的次数做了记录。在生成BlockEntrance节点之前，检查基本块的访问次数，如果过多，则不生成相应的BlockEntrance节点。这相当于人为的对状态空间进行了削减。

对于while和do-while循环，一般来说很难静态的得知它的确切的循环上界，只能用人为规定最大循环次数的办法。对于for循环，很多时候它的循环上界是已知的，这时可以从它的常数循环上界中得到一些提示，来帮助确定循环次数。但是这里面也有很多复杂性，如果迭代变量在循环体中被改变了，则不容易得到循环的次数。

事实上，对程序中循环的处理一直是程序分析领域的难题。最好的办法是能够得到一个合适的循环不变量。但是对循环不变量的自动提取在一般情况下是不可计算的。现在一般的做法是对规定最大的循环次数，如果超出，就停止对循环体的执行。这种策略对于检查source-sink一类的错误来说是相当有效的。但是对于buffer overflow这类错误则变得效果不好，尤其是当数组下标由循环控制的时候，很难预先知道循环多少次能够形成数组访问越界，尤其是当循环嵌套出现的时候，对循环次数的估计更困难。

我们曾经在早期的试验中对几个已发现的著名的buffer overflow代码进行了自动分析，包括sendmail, wuftp, bind等程序。尽管我们想了不少的循环探测的策略，比如给基本块加权重，增加包含++操作的基本块权重，但是仍然无法有效的探测到导致越界的状态空间。主要的原因就是由于循环嵌套，导致潜在的状态空间爆炸，无法在有效的时间内探测到导致越界的状态空间。所以在处理循环的策略方面还有待于进一步的研究。

5.5.2 Block Entrance

对BlockEntrance节点的处理是非常简单的，只需要做一些机械的计数器增加工作，并处理一些特殊情况（比如空的基本块）即可。通常情况下，会访问基本块的第一条语句，生成这个基本块的第一个PostStmt节点。

5.5.3 PostStmt

在此会访问这个程序点处的下一条语句。下一条语句有两种情况：普通语句和结束语句，需要分别处理。

5.5.3.1 普通语句

对于普通语句的访问是整个分析中最繁琐的一块，也是真正复杂的一块。关键的步骤是根据第四 章的模型计算出每个表达式的值。得到了表达式的值，就可以根据程序的语义规则以及数学规则对程序进行符号计算了。

表 5.1: 常见的代码缺陷类型

内存泄漏
在free之后使用动态分配的内存
在free之后再次free
文件指针泄漏
未初始化变量的使用
未初始化内存的使用
空指针引用
返回栈地址
整数溢出
数组访问越界
由逻辑错误而无法到达的代码

在处理语句的同时，可以检查一些不需要人工描述的程序错误，常见的错误类型见表5.1。

也可以检查程序员加入的assert()语句。

5.5.3.2 结束语句

结束语句是一个分支语句，有二个或者更多的后继基本块。结束语句一般会包含一个条件，条件的真或假决定真实的程序执行会选择哪个分支。称这个条件为路径条件。

在静态分析中，由于状态里记录的变量的值是符号值，导致路径条件有可能都能够被满足。在这种情况下，需要生成二个后继的BlockEdge节点，表示状态在此分裂了，将沿着2条路径分别执行下去。算法框架请看图5.3

在图5.3的算法中，对路径条件做了可满足性判断：

$$if(PC \wedge C \text{ is satisfiable})...$$

对路径条件的可满足性判断不是一件容易的事情。在最一般的情况下，这个问题是不可判定的。随着研究的不断深入，人们发现了越来越多的可判定的

```
WL: the work list containing all non-ending nodes.
B: current block.
Term: the terminator statement of the block.
PC: the set of path condition already collected along the
    execution path.
TrueBlock: the block to be taken if the condition in Term is true.
FalseBlock: the block to be taken if the condition in Term is false.

C = get condition from Term;
if (PC && C is satisfiable)
  add C to PC;
  generate BlockEdge(B, TrueBlock) and put it into WL;

if (PC && (!C) is satisfiable)
  add (!C) to PC;
  generate BlockEdge(B, FalseBlock) and put it into WL;
```

图 5.3: 处理结束语句的算法框架

理论，并为之设计了判定过程。这方面的工作成果被SMT汇总到一个统一的框架下[58]。

5.6 状态的分裂

在分析的过程中，状态的数量会增加。由一个状态可能繁衍出多个新的状态。上面已经看到了这样的例子：在处理基本块结束语句的时候，根据结束语句所包含的路径条件的真假值不同，会得到二个新的状态，一个对应于该结束语句包含的条件为真的情况，一个对应于该结束语句包含的条件为假的情况。

在分析的过程中，并不是只有在控制流出现分支的地方状态才会出现分裂。事实上在任何程序点处都可能出现状态的分裂。为什么会这样呢？因为所做的是符号分析，变量的值是以未知量的形式出现的。那么，在需要知道符号具体值才能继续分析下去的地方，必须对符号的每一种可能的取值都进行分

析，才不会漏掉对可能的状态空间的探索。一个典型的例子出现在下面一节中。

5.7 对数组的处理

第四章已经从原理上解决了对数组变量的处理，包括求它的左值和右值。但是在实际分析的时候有很多种设计选择。考虑下面这个代码片段：

```
...
char s[10];
... // some unclear processing of s.
char *p = strchr(s, '.');
*p = 3;
```

strchr(s,c)函数的行为是返回s中第一次出现c的位置，如果c没在s中出现，则返回0.在不知道s的每一个元素是什么的情况下，不可能知道p的具体值。能对p做的最好的近似就是给p一个符号值。这种情况下，在后面要读取p指向的值或者给p指向的位置赋值的时候，就有了两种选择。

一种办法是不对p做任何处理，如果有给它指向的位置赋值，就直接在store中建立一个由p的符号左值到被赋值的映射。之后如果有对该符号左值对应内存位置的读取，就正好可以利用这个建立好的映射，把刚才赋的值读取出来。这样处理简单有效，很巧妙。不管p的符号左值到底是什么，就把它看成一个抽象的左值。利用符号表示本身的唯一性，保证了对符号地址存取的正确性。但是这样简化在某些情况下会跟实际程序的操作有一些出入。比如程序有可能对s的某个具体元素进行了操作，而p也是有可能指向该元素的，但是假设p的符号左值不同于任何一个具体左值，从而造成了对程序语句模拟的误差。

如果符号左值出现在路径条件中，在求解路径条件的时候，直接把这个符号值交给SMT求解器。同时SMT求解器一般也要求用户给出数组的大小。然后SMT求解器就会求解出一个具体的数组下标来。SMT求解的方法实际上是个枚举的办法，对符号下标的每种可能的取值都进行尝试。也可以把这个过程搬到程序分析当中来。这就产生了程序中的状态分裂。具体如下。

在处理符号左值的时候，如果数组长度不大，可以对每一种可能的取值分别进行分析。如果数组的长度是5，那么从当前状态就会衍生出5个新状态，在新状态中，该符号左值分别被一种可能的具体值所替代。这样做看似比上面保留符号值的方法繁琐，但事实上，它可以提高以后的分析精度，因为减少了状态中的一个符号值。状态中的符号值越少，符号分析的精确度就越高。

5.8 符号值具体化

上一节提出，在程序分析过程中数组下标需要一个具体的值才能继续往下分析的时候，就将数组下标具体化。为了不遗漏可能的取值，将状态分裂，对每种可能的取值都生成一个新的状态，继续分析。

这实际上是一种符号值具体化思想应用到数组下标的特殊情况。在对程序进行符号分析时，有时会碰到无法进行符号执行的情形，上面的数组变量就是其中一种。其他的情形还包括：

- 调用一个行为复杂的库函数，比如随机数生成函数。
- 非线性运算，即使以符号的形式执行下来了，交给路径条件求解器也无法求解出来。
- 某些不想符号执行的运算，比如位操作。
- 循环的边界值是符号值。
- 数组的大小是符号值。

在这些情形下，为了保证符号执行的效果，都需要把符号值具体化。在具体化的时候，要考察该符号值的取值范围。取值范围包含多种因素，一种是该值固有的范围，如果是char型，则在-128到127之间，如果是枚举值，则在枚举值定义的范围内。另一种因素是先前的路径条件对该符号值的约束，如果将该符号值具体化为一个不符合之前的路径条件约束的值，则直接造成路径的不可行，也就不用继续分析了。

由于有了这些复杂因素，在将符号值进行具体化的时候也就有了多种方案。

- 最简单的，可以随机生成一个具体值。
- 在取值可能有限的情况下，比如数组下标或者是枚举值，可以穷举所有的可能，将状态分裂。
- 也可以根据当前已有的路径条件，求解出所有符号值的一组解，用这个解将被关心的符号值具体化。这样做的好处是可以保证得到一个合法的值，保证继续探索的路径的可行性。缺点也非常明显，需要大量的计算，降低分析的效率。

上面几种做法很难简单的判断孰优孰劣，只能根据需要进行选择，比如想进行详尽而昂贵的分析，还是进行快速粗略的分析等等。

5.9 C的类型系统

C是一种弱类型的语言，各种类型（特别是指针）之间可以进行强制转换。这个特性对静态分析造成了极大的困难。考虑下面这个代码片段：

```
void* buf = malloc(10);
int* p = (int*) buf;
p[0] = 4;
char* q = (char*) buf;
q[0] = 'a';
```

这段代码是完全合法的：buf从malloc调用得到一块无类型的内存，然后p和q分别将该块内存作为两种类型的缓冲区进行使用。在系统程序中，这种类型的用法是非常普遍的：一块泛型内存，在不同的情况下通过强制类型转换作不同的用处。

现在考虑如何来精确的模拟这段操作。首先malloc调用返回一块内存，可以用MallocRegion来表示。一种简单的做法就是直接用该MallocRegion作为父区域生成ElementRegion，再进行值的绑定操作，如图5.4。但是注意p[0]和q[0]都对应以0为index的ElementRegion，用这种方法是无法区分的（ElementRegion有两部分信息：父区域和index）。而实际上p[0]和q[0]代表的是两个不同的对象，它们的大小不一样，表示的概念也不一样。

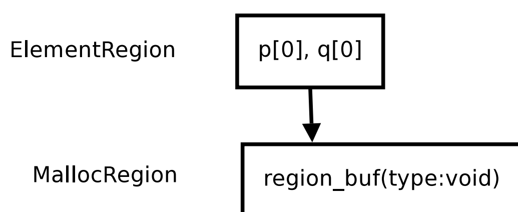

图 5.4: 直接以buf指向的MallocRegion为父区域生成ElementRegion，导致p[0]和q[0] 对应同一块区域，造成模拟的不精确。

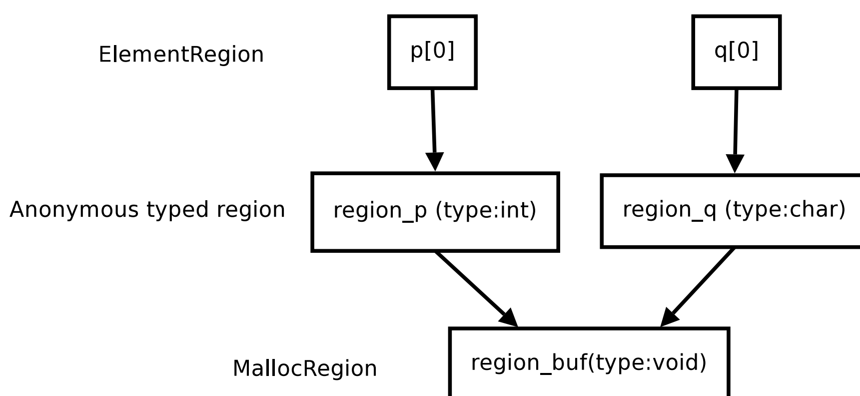

图 5.5: Anonymous typed region 示意图：region_buf是buf指向的一块无类型内存，region_p是p指向的同一块加上了int类型信息的内存，同样，region_q是加上了char类型信息的内存，它们指向同一块父区域，表示它们本质上是同一块内存。p[0], q[0]分别是region_p 和region_q上的ElementRegion。

为此，我们设计了AnonymousTypedRegion，专门用来处理强制类型转换。C中的强制类型转换，本质上是对一块内存加上了一种类型的概念。所以就在原来的无类型区域上再加一层间接，生成一个AnonTypedRegion用来携带附加的类型信息。用图来表示这一概念较为清晰，见图5.5。

region_buf是buf指向的一块无类型内存，region_p是p指向的同一块加上了int类型信息的内存，同样，region_q是加上了char类型信息的内存，它们指向同一块父区域，表示它们本质上是同一块内存。p[0], q[0]分别是region_p和region_q上的ElementRegion。

有了AnonymousTypedRegion这样一个间接层之后，对强制类型转换带来的概念和操作上的复杂性进行清晰的表示。

5.10 总结

本章对如何对单个函数进行分析作了详尽的描述。从总体框架开始，到程序中各种语句的分析，再到符号分析的状态分裂，给出了完整的符号分析方法。这些方法已经在工具中得以实现，有着很好的实验基础，并还在不断的完善之中。

第六章　过程间分析

前面几章已经对如何对一个函数里的各种语句进行符号分析作了详尽的描述和讨论，但是对函数调用语句的分析还没有解决。

在现代软件工程中，模块化被认为是解决软件复杂性的一个重要手段。但是模块化也给程序分析带来了非常大的挑战。

一个函数调用语句的行为是它调用的函数决定的，所以无法从函数调用语句本身推断出它的语义来。只有在分析过被调用的函数之后，才能知道该函数到底干了什么。

如果每次碰到一个函数调用语句就进入该函数进行分析，很快就会陷入状态爆炸之中。这比由函数内的分支引起的状态增加要快很多。所以碰到一个函数就跟进去分析的蛮力搜索办法对于一般大小的程序就已经行不通了。必须想别的办法减少需要探索的状态空间。

如果对被调用的函数一无所知，在分析的时候碰到函数调用就必须作最保守的估计。比如，如果一个指针作为参数传给了一个函数，就必须假设所有该指针可能指向的对象都被修改了。在编译优化中大都是这么做的，因为优化必须要保证结果的正确性。保守带来的不利之处最多就是失去优化机会，而没有其他的问题。但是保守的结果会给静态分析带来非常大的不精确性，大大的增加误报率。这是我们不愿意看到的。

要想得到比最保守的估计更精确的结果，就必须做过程间的分析。也就是不把函数看成黑盒，而是通过某种方法得到它的行为信息，进行相应的分析。

在工业界的编译器里，过程间分析的使用并不像过程内分析的那么普遍。主要的原因有以下几点：

- 进行过程间的分析需要编译器能“看见”全部的程序。而现在许多编译器在编译时是以文件为单位进行的，一次只能看见一个翻译单元。在链接时只是进行一些名字的解引用和地址重定位等机械性的操作。

- 过程间优化的代价往往很大，导致编译时间变长，而带来的好处又不能抵消增加的代价。

- 过程间优化难度很大，导致效果不明显，有时甚至会使优化后的代码变慢。

一般来说，工业界编译器主要用如下两种机制进行过程间分析：

- 在链接时进行全程序的分析和优化。这种方法的好处是不用改变现有的编程模型。缺点是每次改变了一个程序，就得重新做所有的分析。链接时是编译过程中第一个能够看到全程序的阶段。在这里做过程间的分析和优化是最自然的。
- 用编程环境，类似于R^n[20]。缺点是和现有的编程模型不一样，复杂性高。优点是效率高，不用每次都重新编译所有的模块。

6.1 过程间的控制流分析：函数调用图

要进行过程间的数据流分析，需要对函数进行一定顺序的访问。所以要先构造函数调用图，来获得函数之间的调用关系，指导对函数进行分析。

给定一个程序P，由函数$p_1, p_2, \ldots, p_n$组成。P的调用图$G = \langle N, S, E, r \rangle$，其中$N = \{p_1, \ldots, p_n\}$, S是调用地点集合，$E \subseteq N \times S \times N$ 是边的集合，边$e = \langle p_i, s_k, p_j \rangle$ 表示p_i中的调用地点s_k调用了函数p_j。所以调用图实际上是个多图(multigraph)，节点之间可能有多条边，每条边上的标签(call site)不同。

在没有函数指针的情况下，构造函数调用图是一件比较容易的事。一个基本的迭代算法就可以满足要求。但是C语言中有函数指针，这就给调用图的创建带来了麻烦。一般情况下只能建立保守的调用图，即对于被取地址的函数，假设它能够被任何函数调用。

6.2 有限的函数内嵌法

过程间分析对于静态代码查错具有非常大的重要性。下面讨论过程间数据流分析的一些方法。

可以直接把函数内联嵌入到调用地点，然后用过程内分析的方法进行分析。但是这样会导致状态爆炸。对它的一点改进是，对一个函数，可以给它一个参数，规定它调用的函数最多的展开层数。再规定一个作为分析入口的函数集合，作为分析的起始点。之后就可以按照过程内的分析方式来进行分析。

```
Analyze(f, d)
{
  f: function
  d: depth

  if (d == 0)
    return;

  Analyze f as usual.
  When call(g) is visited, Analyze(g, d-1).
}

AnalyzeAll(EntryFunctions, MaxDepth)
{
  for each f in EntryFunctions
    Analyze(f, MaxDepth);
}
```

图 6.1: 有限蛮力搜索算法：对每个被调用的函数，如果调用栈的深度小于最大深度，就分析被调用的函数，否则不分析该函数。

首先人为指定一组入口函数，分析入口函数集合中的每个函数。碰到函数调用，则进入该函数进行分析，同时把分析深度减1，如果分析深度等于0，则不分析该函数。这样就保证了不会无限制的展开函数进行分析。这个算法见图6.1。

6.3 基于函数总结的方法

一种对上面的方法的改进是不对已经分析过的函数进行重复的分析。这就要求对函数分析得到的信息进行缓存。

6.3.1 函数总结(Function Summary)

一种最简单的办法就是直接记录函数的进入和退出时的状态对。在进入函数调用之前保存状态$S1$，然后进入函数，保存出函数时的状态$S2$，建立$S1$到$S2$的映射。下次再碰到调用该函数，直接查找跟进入时的状态$S1$匹配的退出状态$S2$。

进入函数调用时的状态只包括参数和全局变量。出函数的状态删除掉函数内部的状态。根据具体分析的内容还可以做进一步的抽象。

这种做法从原理上讲非常简单，但是它的一个最大的缺陷就在于非常占用空间。因为在进入函数之前的状态是多种多样的，如果每个都进行单独记录的话，很快就会陷入状态爆炸的困境之中。

6.3.2 部分函数总结

既然无法全面地获得一个函数行为的所有信息，所以，一个有效的办法就是针对每种错误检查，设计需要得到的信息种类。我们不求得到关于一个函数的所有信息，而只根据需要得到函数的某些信息。这在实际中是可行的，也是有效的。下面举一些针对某种错误类型的函数总结信息的例子。

6.3.2.1 内存泄漏

关于内存泄漏的详细描述，请参见第九章，这里只是对其作简要描述。如果一块动态分配的内存既没有被逃逸，也没有被释放，就说它被泄漏了。对于内存泄漏的检查，需要知道一个函数如下的行为信息：

- $None$: 函数没有跟堆对象有关的行为。
- $ReturnHeapObj$: 函数分配了堆对象，并返回指向它的指针。
- $MallocGlobal$: 函数分配了堆对象，并把它的地址存入一个全局变量。
- $MallocArg$: 函数分配了堆对象，并把它的地址存入一个参数指向的变量。
- $FreeGlobal$: 函数释放了一个全局变量指向的堆对象。
- $FreeArg$: 函数释放了一个参数指向的堆对象。

6.3.2.2 未初始化的变量

如果一个变量，或者更一般的，一个存储单元，在被赋予有效值之前就被使用了，就说这是一种使用未初始化变量的错误。对于这种错误的检查，需要知道一个函数是否写了全局对象或者参数对象。

6.3.2.3 空指针的解引用

这个类型的错误检查并不需要太多的过程间信息。最有用的过程间信息就是函数的返回值。但是为了更精确的检查，不仅要获得函数的返回值信息，最好还要得到函数在什么路径条件下返回某个值，也就是说需要得到一个特定返回值关联的路径条件。

6.4 完整的算法

6.4.1 性质状态

前面几节论述了过程间分析的种种困难，有理论上的，包括对函数的行为无法完整地刻画，也有工程上的，比如要改变编程的模式以便在链接时进行全程序的分析。在这一节中，我们描述一种较为完整的过程间分析算法。这个算法在已有工作的基础上[22]进行了改进，加入路径条件的约束，提高分析的精度。

我们无法完全地描述函数的所有行为，但是可以描述函数关于我们关注的bug的行为，比如对于内存泄漏，关注函数是否有分配内存的行为并将新分配的内存指针传到函数之外。

对于这种跟bug相关的状态特征，称之为性质状态(type state)。我们关注函数是否改变了程序的性质状态。比如，对于内存泄漏检查来说，考虑下面的程序：

```
int *p;
...
some_malloc(&p);
...
```

如果在调用函数some_malloc()之后，p的值被改变了，变成指向一个堆对象，就说some_malloc()改变了程序的性质状态。

对于每种要检查的bug，都定义出所关注的性质状态。这样一来我们关心的函数的行为就是函数如何改变程序的性质状态。函数的总结信息就是从一个性质状态到另一个性质状态的映射。

$$summary_f : D \rightarrow D' \tag{6.1}$$

6.4.2 路径条件

实验证明，上面的将函数summary表达成从一种性质状态到另一种性质状态的映射是有效的。但是仍然有改进的空间。进一步可以发现，这种映射其实是有条件的。函数内部的路径条件决定了这种映射成立的条件。所以，为了提高精确度，在函数summary中再加上一个成份，就是每个映射相关联的路径条件C。函数summary变为：

$$summary_f : D \rightarrow (D', C) \tag{6.2}$$

路径条件包括路径的和输入变量相关的约束，以及函数的返回值。约束当中暂时不包括不可见变量相关的约束。不可见变量指函数中从外部输入获得的变量。如果考虑最新的一些技术[24]，也可以适当的包含不可见变量相关的约束。

如果把一般的语句看成是在程序执行状态空间的一种变换的话，函数调用语句就是在程序性质状态空间的一种变换，辅以变换的条件（路径条件）。

6.4.3 完整算法

有了上面对函数summary的定义，可以大致描述出算法的思想：对每个函数，仍然像第五章中描述的一样进行过程内分析。碰到函数调用，则检查被调用的函数是否有可以用的summary信息。如果没有，则以调用前的性质状态为输入参数分析该函数，得到对性质状态的改变的行为信息，并记录在函数summary中。然后回到上层函数继续进行分析。算法描述见图6.2。

图6.2中的算法与之前过程内的分析算法最大的不同就是对函数调用语句的处理。在以前过程内分析的时候，直接对函数调用的效果做最保守的估计。

```
procedure Solve()
  build program call graph
  entry = get entry function of the program
  Visit(entry, s0)

procedure Visit(f, s)
  f: function, s: initial state
  W: the work list set of all nodes.
  A node is the (program point, state) pair.

  Initialize worklist W with the node (entry, s).
  while (W is not empty)
    remove a node n from W
    switch (n)
      case n is a call node:
        let f' = callee(n), s = state(n)
        if s is in summary(f'), then
          get the after-call state s' from function summary.
          s' = summary(f', s)
          put (next_stmt, s') into W
        else
          (next_stmt, s') = Visit(f', s)
          put (next_stmt, s') into W
          extract new type state transitions from s',
          and put them into the summary of f'.

      case n is other kind of node:
        perform intra-procedural analysis as usual.
```

图 6.2: 过程间分析算法

而这里对函数调用的处理分成了两种情况。如果进入函数的状态在函数总结信息里存在，那么就直接使用函数的总结信息。如果不存在，则分析该函数，并保存进入和退出的状态信息到函数的总结信息里面。

6.5 总结

本章分析了从单个函数到过程间分析带来的根本性的困难，并给出了使用函数总结信息这个基本的解决问题的思路。在函数总结信息中记录完整的输入输出状态对有可能陷入状态爆炸的困境。所以我们提出了只记录性质状态改变的状态对，并辅之以相关的路径条件。

过程间的分析一直都是程序分析中最困难的问题，至今没有一个完整的数学模型。

第七章　系统的实现

这一章描述基于前述算法和设计实现的静态分析工具。静态分析实际上是对编译器的一种扩展，前面的词法分析，语法分析部分都与编译器相同，到了代码生成的地方，静态分析就和编译器分道扬镳了。静态分析接过编译器前端生成的对程序的数据结构表示（往往是一个程序的控制流图，其中的节点是语句和表达式的抽象语法术表示），开始了自己的分析。

7.1　分析的对象

静态分析工具是建立在编译器的基础之上的。现代编译器一般有3个部分：前端，中端，和后端。前端负责分析源代码，生成抽象语法树AST, 做一定的语法和语义检查之后，最终生成中间表示IR. 每个编译器都会自己定义一种IR.定义IR并没有一定的要求，只需要它能够满足编译器的需求就可以了，一般来说，需要它能够完全表达前端所支持的语言的语义。有的中间表示接近于源语言，有的则更接近汇编语言。有的编译器则有多层中间表示。相对于编译器的其他部分来说，中间表示的设计更像是一门艺术。

前端将源程序变成IR之后，中端在IR上做各种分析和优化。最后，后端对IR进行汇编代码生成。

由于编译器里对程序有这么多层次的表示：AST, IR, assembly code。做静态分析也就可以在很多层次上做。

在中间表示上做分析的好处是：

1. 被分析的语言相对简单。一般来说只有有限的指令，如Add, Store, Call等等。
2. 分析代码独立于源语言，只要前端将源语言翻译成IR,就可以对其进行分析。

缺点是：

1. 由于编译器前端已经对源语言做了一次变换，导致一些源程序的信息丢失，比如一些高级的类型信息，使得分析变得相对困难，能够检查的元素也变少了。

2. 使和用户的交互变得困难，由于错误信息是针对IR的，而用户看到的源代码，使得我们在报错之前要将错误信息对应回源程序。如果没有debug信息的辅助，这是非常麻烦的一件事。

经过试验，我们决定采取在源代码AST上的静态分析。这样我们可以得到原原本本的程序信息，使得分析更加接近人对程序的理解，和用户的交互也变得方便。但是这样做的缺点也很明显，就是分析的对象层次很高，也更加复杂，需要做更多的建模。

在得到一个函数的AST之后，首先对其进行一遍结构分析，得到它的控制流结构，表示成控制流图(CFG)的形式。之后的分析都是在这个图结构之上进行的。CFG的每个节点仍然是相应元素的AST。CFG的建立并没有改变原先AST的任何部分，只是在其上加上了一层图结构。

分析的过程在CFG的基础上生成一个扩展图(exploded graph)，用于模拟程序的执行状态空间。扩展图的每个节点包含一个程序点(program point)和一个状态，表示程序模拟执行到该程序点时具有什么样的状态。

扩展图保存所有的节点，并记录每条路径的最终节点。这样，有了最终节点，根据每个节点的前驱后继就可以得到所有的程序路径了。

7.2 分析框架的结构

分析框架的结构见图7.1.在这个框架中，程序经由前端生成AST，交给分析引擎。分析引擎在AST上建立CFG，并初始化各模块的状态，从函数入口处开始以深度优先的方式探索函数的CFG.下面分别对各个模块进行简要的介绍。

7.2.1 核心引擎

核心分析引擎CoreEngine实现exploded graph的可达性的分析。它遍历CFG，生成扩展图。遍历的方式可以是深度优先搜索(DFS)，也可以是宽度优先搜索(BFS)，对于循环的处理交给分析引擎。分析引擎决定处理哪个分支，它就

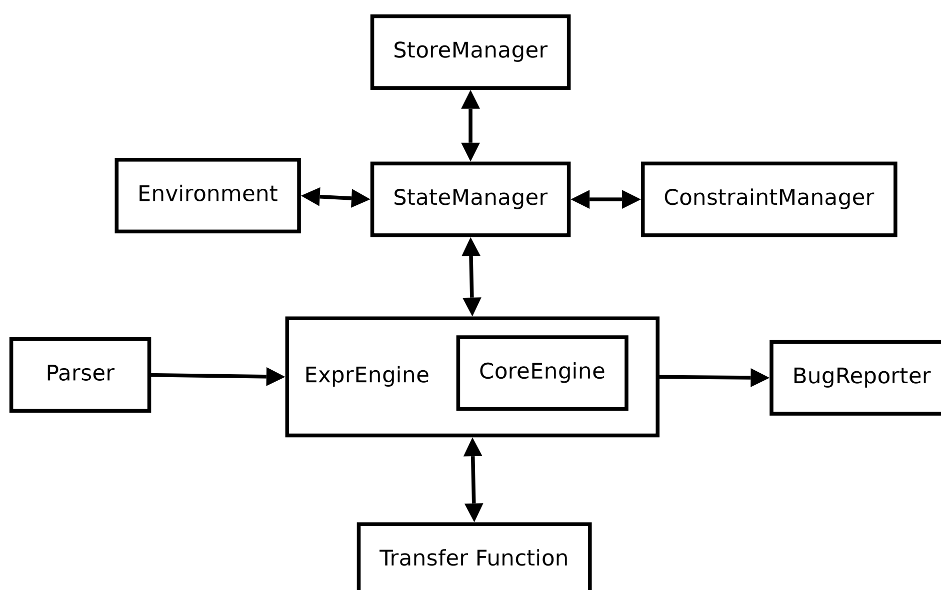

图 7.1: 分析引擎组件示意图

处理哪个分支。分析引擎需要对所有的语句和表达式种类进行语义的模拟执行，根据每个程序点处的输入状态，模拟执行语义，生成新的状态。必要的情况下对状态进行分裂。

分析引擎以插件的形式定义了各种各样的传输函数，这些传输函数在可配置的情况下按需调用。有的传输函数完成程序语义的模拟，有些传输函数对某个特定的错误类型进行检查。大部分的传输函数都会对状态进行改变。由于状态的改变有可能分裂成多个状态，所以状态的传递是以集合的形式进行的。

7.2.2 状态

状态(state)是描述程序模拟执行过程中的状态的对象。一个状态中包含以下的信息：

- 程序点，该状态所处的程序CFG上的位置。
- 环境，记录着程序中每个表达式到值的映射。
- 存储，记录着程序中变量和动态分配的存储单元到值的映射。
- 约束，记录着从函数入口处到该程序点处的路径上所有的路径条件信息。

由于在分析过程中状态非常多，避免状态爆炸引起的内存使用过量是设计中重点考虑的问题。为此，使用了immutable数据结构来存储状态。每一个状态在生成之后就不变了，如果要生成新的状态，让新的状态与旧的状态共享相同的部分，而只为新的状态生成改变的部分。这样就大大减少了存储状态所需要的内存。

为了进一步减少内存使用，我们还在预处理阶段进行活动变量分析。在每次分析一条新的语句之前，根据活动变量的信息，对状态中死变量的信息进行删除。这样不仅可以减少信息的传播量，还可以增加状态间共享信息的部分。

由于以上两个优化的使用，在测试过程中还没有发现内存不够的情况。

7.2.3 各种管理器

状态管理器，管理状态的生成和变化。环境管理器，管理一个状态中环境的变化。存储管理器，管理一个状态中存储的变化。约束管理器，管理状态中的约束，对于查询，给出当前的约束集合是否可满足的结论。

这些管理器模块都以插件的形式集成到分析引擎当中，也就是说它们都是可分别替换的。这样的设计大大方便了研究实验。因为静态分析工具中各个模块设计中的权衡非常多。以约束管理器为例，就有多种实现方法，可以基于简单的集合，也可以基于SAT求解器，还可以基于SMT求解器。可以在不改变其他模块的条件下替换进不同的约束管理器，进行实验。

7.3 总结

作者已经根据前文描述的符号分析算法实现了完整的C语言静态分析工具。该工具具有高度模块化、易扩展、高效率的特点，并且还在不断的改进之中。

第八章　自动测试数据生成

8.1　引言

测试是软件工程中用来发现bug的使用最多的方法。不过直到目前为止，测试人员进行测试的方法仍然以手工测试为主。测试工具大都只是对测试流程起一些辅助的作用，对测试中真正困难的问题——测试数据生成没有任何帮助。自动测试数据生成一直是人们渴望的。

这一章中我们利用前面描述的符号程序分析方法，对C的单元程序进行自动测试数据生成。一个单元程序，一般来讲是一个函数。

本章描述的自动测试数据生成的基本方法是对程序的每一条路径做符号执行，收集路径中分支处的路径条件，这样就可以得到使程序执行一条特定路径的关于输入变量值的一组约束。如果能求解出满足这组约束的变量的值，就可以得到一组可以驱使程序执行这条特定路径的测试数据。

在最一般的情况下，程序中可能出现的路径条件约束是不可判定的。但是，在一定的限制下，比如，忽略非线性约束，路径条件是可以求解的。在SMT的框架下，可以求解（判定）的约束类型是足够丰富的。

用符号执行的办法自动生成测试数据在七十年代就已有人提出[46][9]。但是由于种种困难，比如符号数组下标，指针等等，没有在实际中得到广泛的应用。这些困难的一般处理在前几章已经描述过了。应用那些方法，应该可以比较完美的解决这些困难。这一章针对测试数据生成再描述一下具体的做法。

主要的特点有：

1. 在测试数据生成这个具体应用中，使用的是内存的数组建模方法。因为这个工作是在早期做的，当时尚未发展出后来的三元建模法。同时也由于这里处理的程序相对简单，数组建模法也已够用了。地址空间被看成一个大数组，所有的指针算术操作用数组下标的操作来模拟。

2. 区分符号指针和具体指针。符号指针被当作一个未知量。具体指针代表一个具体的内存单元，用一个数组下标来代替。

3. 求解布尔和数值的混合约束。

我们实现了一个工具叫做SimC，能够接受标准C程序。我们用许多文献中的例子程序和一些来自GNU程序当中的代码例子做了实验。结果非常乐观，对一些很复杂的函数都能够自动生成覆盖所有控制流图分支的测试数据。

下一节先介绍一点基本概念。

8.2 路径可行性分析

如同前几章里一样，假设程序是以源代码级的控制流图方式给出。

一条程序路径定义为一个赋值语句和条件谓词的序列。PAT[72]是一个路径分析器，它接受一条路径，返回这条路径是否可行。一条路径可行是指这条路径中的路径条件可以被满足。PAT可以接收的变量类型包括整数变量、浮点变量、布尔变量和数组。PAT通过对路径的符号执行，收集路径中出现的条件谓词，然后求解这组谓词约束。如果有解，则给出这组解，相当于给出一组能够使程序执行这条路径的输入数据。SimC使用了EPAT，是PAT的一个改进版本。

8.3 自动测试数据生成

8.3.1 概述

这一部分先对SimC给出一个概述。测试数据生成过程由下面几个步骤构成：

1. 对C代码进行词法分析和语法分析生成AST并在其之上构造CFG.

2. 生成程序的路径并将它们翻译成EPAT可以接受的格式。

3. 对这些路径进行符号执行并求解路径条件。

4. 根据路径覆盖要求选择一组可行的路径。

最后得到一组能够驱使程序执行指定路径的测试数据。

SimC为用户指定的函数生成路径。为简单起见，假设每个函数只有一个出口。每条路径从函数入口开始，在函数出口结束。路径中包含两种语句：赋

```
void f(char *s)
{
  char *p;
  p = s;
  while (*p != 'a')
    p++;
}
```

图 8.1: 一个简单的例子

值语句和条件表达式。一个赋值语句把一个常数或者存储位置上的值存入另一个存储位置。一个条件表达式是在该点处变量必须满足的数学约束。存储位置由简单变量或数组变量表示。

使用数组来模拟内存。程序里所有的存储位置，包括简单变量，结构，数组和动态分配的内存都映射在用来模拟内存的数组上。在符号表中记录变量的映射之后的地址和指针指向的地址。通过这样的模拟所有对存储位置的引用都可以用数组变量的引用来表示。所有的指针操作都可以转化成为数组下标的操作。在约束求解器的帮助下，可以求解出生成的路径条件，得到测试数据。

看一个例子，见图8.1。其中一条路径可以被翻译成：

```
p = 1;
@ mem[p] != 'a';
p++;
@ mem[p] == 'a';
```

在这个例子中，mem是引入的用来模拟内存的辅助数组。语句`p = s;`被翻译成`p = 1;`，这里1是数组s的模拟起始地址。模拟起始地址从1开始是因为0被用于表示NULL指针。否则0成为一个有效的指针，会对C的语义模拟造成麻烦。

指针的解引用`*p`被翻译成`mem[p]`。指针的增加`p++`被翻译成数组指标的增加`p++`。头上的@标记表示这是一个路径条件而不是赋值语句。

然后这条路径被EPAT进行计算，得到一组解为`s[1] = 'b'` 和
`s[2] = 'a'`。

这一节集中关注如何生成路径。生成路径其实是一个翻译过程。SimC试图把各种复杂的C程序构造翻译成EPAT可以接受的一种简单规范的格式。下面讨论如何处理翻译过程中的各种问题。

8.3.2 数组和指针

C语言允许程序员对指针做各种各样的操作。为了模拟指针操作，使用数组来模拟内存。如果变量p被模拟内存的数组分配到第i个单元上，p就被mem[i]来表示。

指针操作大致可以被分为2类：指针算术和指针解引用。指针算术可以被数组下标算术来模拟。指针解引用被翻译成对数组元素的引用。

8.3.2.1 模拟方案

我们有两个设计元素需要决定。一个方面是用一个大数组来模拟整个内存还是用若干个小数组来模拟，给每个对象单独分配一个模拟数组。这其实是个如何重命名的问题。

如果使用一个大数组，那么变量名就不会出现在路径中，所有的变量名都变成了同一个数组名。每个变量被它所分配到的数组元素替代。如果给每个对象分配单独的数组，对象就会有不同的名字。

用一个单个数组模拟的好处是简化路径的表示。每个变量都在数组中有自己的位置。指针的指向地址就存在数组元素中。缺点是增加了路径条件求解器的计算量。如果有一个未知的指针，那么我们在求解的时候得尝试每一个可能的位置。数组越大，尝试的位置就越多。这是指数级复杂度增加。

另一方面，如果给每个内存对象单独分配一个数组的话，对于指针来说必须在符号表中记录它指向那个对象，从而才能正确地找到它指向的模拟数组。这割裂了指针运算和算术运算。指针运算交给了分析系统，算术运算交给了EPAT。有一些操作不能被模拟，考虑下面的代码：

```
struct node {
  int i;
  struct node *next;
};
```

```
struct node *p;

p = malloc(sizeof(struct node));
```

我们可以表示结构s的i数据域成p[0]。但是没法表示`p->next` 指向的对象。因为对于域next来说，在符号表中没有它的位置，它只是一个数组元素。

第二个需要确定的设计选择是是否需要把指针本身分配在模拟数组上。如果不分配，那么就得在符号表中记录关于指针指向对象的信息。

我们认为功能最强的模拟方案是使用单个数组来模拟整个内存，并且把指针分配在数组上。图8.2中的例子说明了这种方案。

假设使用数组mem来模拟内存。变量a,p,q的模拟地址分别为1, 2, 3.模拟地址0留给NULL指针。程序被翻译之后的路径在图8.2的下半部分。

模拟地址从1开始，因为指针经常和NULL比较。而在路径中指针是被他们的指向地址替代的。如果0是其实地址的话，0就成了一个合法的地址，这对语义的模拟造成麻烦。所以保留所有数组的0元素不用。

8.3.2.2 符号指针和具体指针

SimC是一个单元测试工具。当做单元测试的时候，不像真正运行程序的时候能够有完整的程序的信息，这种信息缺乏对于局部指针来说没什么问题，因为它们在被使用之前总是会被初始化。但是函数的参数却不是这样。

我们观察到函数参数中的指针有以下3种情况：

1. 指针是一个具体的指针，意思是它指向一个已存在的内存区域的开始。比如给一个函数传入一个缓冲区一般使用这种方法，比如图8.3里的例子。在这种情况下，应该给这个指针分配一块内存，并让它指向这快内存的起始位置。具体指针的值在符号执行的过程中是不变的。

2. 符号指针。符号指针不指向内存区域的起始位置。它们指向一个任意的位置。对于测试数据生成来说，它们的值应该由约束求解器求解出来。例如，在图8.4中的代码里，buf是一个具体指针，指向一个内存区域的起始地址。应该给buf分配一块内存。但是根据代码可以推断出p指向的不是一块新的内存区域，而是buf中的某个位置。p的值应该根据路径约束求解出来。

```
void f(void)
{
  int a;
  int *p;
  int **q;

  a = 1000;
  p = &a;
  q = &p;
  **q = 1000;
  p--;
}
```

被翻译成

```
mem[1] = 1000;
mem[2] = 1;
mem[3] = 2;
mem[mem[mem[3]]] = 1000;
mem[2] = mem[2] - 1;
```

图 8.2: 使用单个数组模拟内存，指针也分配空间

3. 还有一种情况是指针既是一个具体指针，又是一个符号指针。在这种情况中，指针先被用作传入一块缓冲区，然后它就被用来作为一个符号指针，指向任意的位置了。看图8.5 中的代码。s一开始被用来传一块内存给函数bar,然后它被赋予了新的值。

一般来说，没有自动的方法能够识别出一个指针是具体指针还是符号指针。因此，让用户指定一个函数参数指针的类别。同时让用户指定具体指针指向的内存区域的大小。注意这里的指针类别并不是C语言里面定义的，而是针对测试数据生成这个具体的应用定义出来的。

```
void foo(char *s)
{
        int i;
        for (i = 0; i < 10; i++)
                s[i] = i;
}
```

图 8.3: 函数参数指针p被用作一个具体指针。

```
void mem_free(char *buf, char *p)
{
        ...
        position = (p - buf) / 16;
        ...
}
```

图 8.4: 函数参数指针p被用作一个符号指针。

```
void bar(char *s)
{
        ...
        putchar(s[0]);
        ...
        s = strchr(s, 'a');
        ...
}
```

图 8.5: 函数参数指针s既被用作一个具体指针，又被用作一个符号指针。

一个具体指针指向一个已知大小的内存区域。从而它的模拟地址就是已知的，在生成的路径中，它被替换成它的模拟地址。

一个符号指针代表一个未知的地址，这个地址在程序中随着程序的执行会改变，也会被路径条件约束。它以一个未知量的形式出现在路径中，例如，图8.4 中的语句被翻译成：

```
position = (p - 1) / 16;
```

这里buf被它的模拟地址1替代了。

对符号指针的操作跟普通符号变量一样。因为它们指向的地址是未知的，无法想操作具体指针一样操作符号表中的信息。语句p++就被翻译成了路径中的p = p+1。解引用一个符号指针引入了下标为未知量的数组变量。路径求解器EPAT可以分析这样的路径。

8.3.3 函数调用

在单元测试中处理函数调用一般有两种方法。一种是把被调用的函数直接在调用处展开。另外一种是对函数进行建模。

把函数展开十分直接，用显式的赋值替代函数传递过程。然后我们把函数体嵌入进来，最后再用赋值语句替代函数返回值的过程。

如果对函数建模，用几个条件表达式和赋值语句来描述函数的行为。例如，当遇到一个输入函数，通常不关心它是如何从文件或者网络得到输入的。只需要知道它返回了一个整数值就足够了。这时就可以用一个刻画它的返回值的条件表达式来代替这个函数调用。况且，有些时候库函数的源代码是不可得到的，只能用建模的办法来处理相应的函数调用。

SimC同时采用了这两种方法。对于标准库函数，用一种简单的约束形式来建模。例如，对函数调用语句c = getchar();翻译成：

```
c = INPUT;
@ 0 <= c <= 255;
```

再看一个例子：函数调用p = strchr(s,c);被翻译成

```
(p == NULL) || (*p == c && s <= p && p <= s + lengthOf(s))
```

这些简单的建模在实际应用中很有用。

8.3.4 路径生成策略

从CFG生成路径的策略对于生成的测试数据的有效性非常重要。

深度优先搜索在有循环的程序中有时会不太有效，尽管可以用给循环限定次数的方式来避免陷入一个长长的循环中。

SimC使用了宽度优先的搜索策略，同时限制生成路径的最大长度。在将节点压入队列之前计算之前的部分路径长度，如果超过了限制的长度，则不继续探索这条路径。当遇到exit节点时，记录整个路径并交给EPAT处理。

在路径长度被限制的情况下，我们还是能够对函数生成几百条路径。我们使用了一种最优路径选择方法，大意是，把对CFG的边覆盖作为选择路径的条件。让CFG的每条边至少被覆盖1次这个目标可以用0-1整数规划来建模。

我们开发了一个单独的工具来进行路径选择。我们使用了lp_solve[51]作为整数规划的求解器。这个路径选择工具非常有效，一般来说10条左右的路径就可以完成对函数CFG的边覆盖。

8.4 实验

许多先前的测试数据生成论文都是在小的经过特殊改写的使用受限的语言编写的程序上进行实验[30][54]。在真实的C程序上试验我们的方法，这些程序包括GNU的coreutils和make. coreutils中用来实验的代码片段在表8.1中列出。

我们选择用来实验的函数的标准是：

1. 函数有比较多的指针操作，控制流中有较多的循环和分支。

2. 函数不调用没有建模的其它函数。这是因为我们的实现中还没有支持跨函数的路径生成。

由于我们实现的C语言前端只能接受部分标准C代码，所以用来实验的程序经过了轻微的修改，主要是去掉GCC的扩展，并不影响函数的主体功能。

实验是在一台装有Pentium 4 3.4G, 1G内存的PC上进行的。所有的计算时间都在2分钟之内。

表 8.1: GNU coreutils 5.2.1软件包中用于实验的代码

Function	File
remove_suffix()	basename.c
cat()	cat.c
cut_bytes()	cut.c
parse_line()	dircolors.c
set_prefix()	fmt.c
attach()	ls.c
bsd_split_3()	md5sum.c
hex_digit()	md5sum.c
isint()	test.c
make_printable_char()	tr.c
strtol()	strtol.c

8.4.1 例子：GNU coreutils中的remove_suffix()

这是SimC擅长处理的典型的代码例子。代码有非常多的指针运算和解引用。指针引起的别名关系也存在。SimC把变量都映射到模拟内存的数组上。

```
void remove_suffix(char *name,
            const char *suffix)
{ ...
  np = name + strlen(name);
  sp = suffix + strlen(suffix);
  while (np > name && sp > suffix)
    if (*--np != *--sp) return;
  if (np > name) *np = '\0';
}
```

这段代码是命令basename的实现的一部分。虽然它不大，但是它的复杂度已经超出了我们所知的其它测试生成工具的处理能力。SimC为其生成了5组测试数据，覆盖了所有的分支。

8.4.2 例子：GNU coreutils中的strtol()

这段代码更复杂一些，在下面列出相关的一部分。

```
int strtol(const STRING_TYPE *nptr,
   STRING_TYPE **endptr,
   int base,
   int group LOCALE_PARAM_PROTO)
{
  ...
  save = s = nptr;
  while (ISSPACE (*s))
    ++s;
  if (*s == L_('-')) {
      negative = 1;
      ++s;
    }
  ...
}
```

路径最大长度为20时，生成了10组测试数据完成对CFG的覆盖。

8.4.3 例子：getop()

这个例子来自K & R的C程序设计语言一书[45]。Bertolino和Marré [8]也用了它作为例子。

```
int getop(char *s, int lim)
{
  c = getchar();
  while (c == ' ' || c == '\t' || c == '\n')
    c = getchar();
  if (c != '.' && (c < '0' || c > '9'))
    return(c);
```

```
  ...
  for(i = 1; c >= '0' && c <= '9'; i++) {
   ...
  }
  if (c == '.') {
    if (i < lim) s[i] = c;
    c = getchar();
    for(i++; c >= '0' && c <= '9'; i++) {
...
    }
  }
  if (i < lim) {
...
  } else {
    while (c != '\n' && c != -1)
      ...
  }
  return;
}
```

这个程序有复杂的控制结构，手工进行测试非常困难。限制路径长度为20。SimC 生成了178组测试数据。应用路径选择工具，仅用4条路径就覆盖了函数。在[8]中，作者描述了如何生成可能可行的路径，他们生成了7条路径，并且没有自动工具的支持。

8.4.4 例子：InsertionSort()

这是一个插入排序算法的实现。

```
void sort(int *a, int n)
{
  int cur, j, low_ind, temp;
  for (cur = 0; cur < n-1; cur++) {
```

```
    low_ind = cur;
    for (j = cur + 1; j < n; j++) {
      if (a[j] < a[low_ind])
        low_ind = j;
    }
  ...
  return;
}
```

在这个例子中有循环的嵌套。手工生成能够遍历CFG的测试数据很困难。

把最大路径长度限制为20。EPAT生成了9组测试数据：{0}, {0,1} {1,0} {0,1,2} {1,2,0} {0,2,1} {1,0,2} {2,1,0} {2,0,1}。程序在这些输入下执行了不同的路径。

8.4.5 例子：GNU make中的dosify()

这里对GNU make中的dosify()函数进行实验。

```
static char *
dosify (char *filename)
{
  ...
  df = dos_filename;
  ...
  if (*filename != '\0') {
      *df++ = *filename++;
      for (i = 0; *filename != '\0' &&
            i < 3 && *filename != '.'; ++i)
      *df++ = ...;
    }
  while (*filename != '\0' &&
         *filename != '.')
    ++filename;
}
```

这个例子中的指针运算很多。参数filename用来传入一个字符数组。它被和0比较。指定它为具体指针。在长度小于20的总共420条路径中，EPAT在1分钟的运算时间内发现了50条可行路径。经过最优路径选择之后，3条路径就可以覆盖CFG所有的边。

8.5 讨论

在进行了广泛的实验之后，我们得到了以下的几条经验：

1. 并不是所有的函数都需要自动测试。许多函数的结构很简单。很容易保证它们的正确性。只有一些复杂的函数需要详尽的测试。
2. 我们的方法能够为拥有复杂控制结构的函数生成测试数据，并且生成的测试数据能够覆盖函数绝大部分的语句。
3. 对许多库函数建模比较容易。但是一些字符串处理函数建模较为困难。

方法的一些缺陷：

1. 函数指针在这种方法中没有被模拟。
2. 对于使用递归数据结构的程序，不能生成测试数据，比如链表，树等。
3. 没有处理递归函数调用。
4. EPAT不能处理非线性的约束。

8.6 总结

本章描述了针对单元程序的自动测试数据生成算法及实验。

在当前的算法中，对函数调用的处理依赖于手工对函数建模。这种办法费时费力，对于容易建模的函数还好，对于一些行为复杂的函数，就不怎么有效了。考虑如何自动为函数生成总结信息，以便做测试数据生成的时候使用。

一个函数的行为可能有很多种，用人对其进行描述甚至都不能描述完全，更别说用自动的办法来生成对它的描述了。所以对函数行为的描述一定是针对某项特定的应用，总结出来的部分行为信息。那么我们就思考对于测试数据生

成来说一个函数的什么信息是最重要的。一个最直接的答案就是函数的返回值。函数的其他行为对于我们来说要么不重要，例如是否分配了内存，要么过于复杂，无法精确刻画，例如是否通过指针改写了什么数据。只有函数的返回值是关系密切并且可以进行总结描述的。

对于函数返回值的描述，更精确的，可以把它跟输入参数关联起来，也就是说，我们给出的描述是：一个函数F，在输入参数是a，路径条件是P的情况下，返回R, 其中a是参数向量，P是关于a的约束，R是关于a的线性表达式。这里忽略输入参数和返回值之间的非线性关系。这样，函数的返回值就可以用下面这个公式来表达：

$$P \Rightarrow (RetVal_{F,a} = R)$$

这样对返回值的描述是否可行，是否有效，还需要实现之后在真实程序上实验才能得知。

除了这样的改进之外，本章中所使用的数组模拟内存的方法，和对程序语句的直接翻译方法有固有的局限性。

1. 把所有的指针操作都转化为显式的算术操作有局限性，它无法表示一些定性的指针操作。

2. 并不是所有的指针都可以具体化。如果有些未知长度的内存对象，就无法在模拟内存的数组上给它们分配空间。

3. 如果不把指针具体化，而是在metadata中表示，如果数组的域本身是一个指针，也无法表示它的值。

这些在第三章中有更详细的论述。换用三元建模法可以解决这些局限性。

事实上，在后来的工作中，我们已经在三元模型基础上重新实现了本章中的测试生成算法。由于采用了新的模型，现在可以处理任意嵌套的指针，数组，结构等语言成份。

第九章 内存泄漏的自动检测

9.1 介绍

C语言要求程序员手工分配和释放堆上的内存。内存泄漏成为一个普遍的问题。在长时间运行的程序中，比如一些服务程序，内存泄漏会造成性能的下降，甚至程序的崩溃。

内存泄漏问题很难被发现，因为它没有什么明显的特征，除了会造成应用程序占用的内存缓慢地增加。

我们设计了一种自动发现内存泄漏的方法，这种方法具有以下的特征：

- 使用了一种新的内存模型，以及逃逸分析。这种内存模型既适用于逃逸分析又适合符号执行。
- 所做的分析是路径敏感的，使用了约束求解器CVC3来判定路径的可行性。这样可以消除大量的假报错。同时，路径敏感的分析使得很容易得到导致错误发生的程序路径。
- 路径敏感的分析让我们不需要再做单独的保守的指针分析。指针分析是用来收集程序中出现的别名信息的。在一条单独的路径上进行分析，指针的指向关系是很清晰的。与传统的数据流分析相比，基于路径的分析是不在控制流汇聚的地方对数据流信息进行汇聚的。这种好处使得分析更精确，实现更简单。
- 分析是上下文敏感的。上下文敏感的意思是区分同一个函数在不同的调用地点的行为。具体到内存泄漏的分析中就是，对在不同调用地点分配的内存块进行单独的追踪。不同调用地点的函数参数也被区分。
- 分析是跨过程的。函数总结系统精确并且可扩展性好。

本章第9.2节给出一个简单的例子来说明方法的有效性。第9.3节对方法做出综述，描述对内存对象的建模方法。第9.4描述使用的表示数据流信息的格。第9.5和第9.6节详细描述分析方法。实验结果在第9.7节。

表 9.1: 在函数建模阶段完成后得到的函数总结

Function	Behavior	Return Value	Feasible Paths
malloc_arg1	MallocArg	1	1
malloc_arg2	MallocArg	Unknown	2
foo	None	void	2

9.2 一个例子

这一节给出一个例子程序来说明方法的有效性。图9.1中的代码是真实程序的简化版。

分析从编译代码，构造每个函数的控制流图和全程序的函数调用图开始。然后经过函数建模阶段和内存泄漏检测阶段。

在函数建模阶段，按照call graph中自底向上的顺序访问函数，即叶节点先于父节点被访问。这样做的好处是当碰到函数调用时，被调用的函数信息已经有了。对于一个函数，把循环的次数限制在小于2次，然后生成它的所有的可行路径。详细的路径生成和可行性判断方法在第9.5节描述。函数建模器访问函数的每条可行路径，提取出函数的行为信息。

在函数建模阶段之后，可以得到关于图9.1中程序的函数总结信息，见表9.1。

函数`malloc_arg1()`和`malloc_arg2()`都分配内存并且把地址存入参数指向的变量。但是`malloc_arg1()`把内存分配的行为和它的返回值1进行了关联。在分析中，总是假设malloc是成功的。所以，在`malloc_arg1()`中只有1条可行路径。

有了这样精确的总结信息以后，内存泄漏检查器就可以正确地检测到在L2行处的内存泄漏。由于对malloc_arg2()的返回值信息一无所知，在L2处返回的这条路径就是可行的。从而在malloc_arg2()中分配的内存就泄漏了。

另一方面，我们知道malloc_arg1()的返回值为1，这使得在L1处返回的路径不可行，从而不会报出此处有内存泄漏的错误。

9.3 方法概述

分析方法的框架在图9.2中显示。分析步骤如下：

```
int malloc_arg1(int **p) {
  int *t = malloc(8);
  if (!t) return 0;
  else {
    *p = t; return 1;
  }
}
int malloc_arg2(int **p) {
  int *t = malloc(8);
  if (!t) assert(0);
  *p = t;
  if (...)
    return 0;
  else
   return 1;
}
void foo(void) {
  int r, *p;
  r = malloc_arg1(&p);
  if (!r)
L1:     return;
  else free(p);
  int q = malloc_arg2(&p);
  if (!q)
L2:     return;
  else free(p);
}
```

图 9.1: 一个从实际代码中简化出来的例子，在函数malloc_arg1中，返回值1/0指示着内存分配的成功或失败。在函数malloc_arg2中，返回值与内存分配的成功与否没关系。函数在L2处返回时，出现内存泄漏。

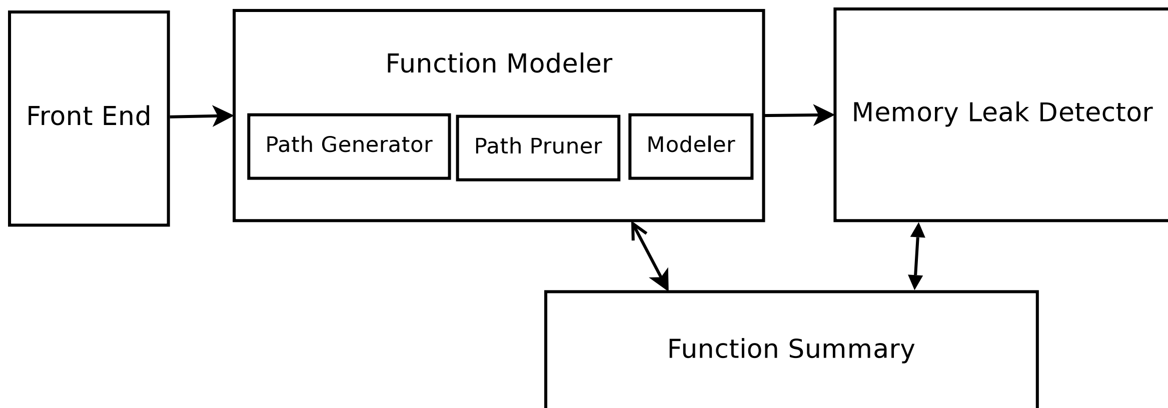

图 9.2: Analysis system overview

1. 编译器前端编译程序成中间表示，并进行链接，得到一个包含所有函数的中间表示形式的文件。然后构造整个程序的函数调用图。

2. 函数建模阶段。按照函数调用图中自底向上的顺序访问函数。对每个函数做如下事情：

   (a) 为函数生成路径

   (b) 检查路径的可行性，记录可行的路径

   (c) 分析每条可行的路径，记录函数总结信息。

3. 内存泄漏检查阶段。访问每个函数，对每个函数检查它的可行路径上是否有内存泄漏。

进行跨过程的分析的主要困难是如何得到精确的关于函数行为的总结信息，因为不想每次碰到调用函数时都重新分析一遍被调用的函数。所以我们的分析被分为了两个阶段。

在函数建模阶段，按照函数调用图中自底向上的顺序访问每个函数。对每个函数，生成它的所有的路径。循环的次数被限制在0次和1次。每条路径被路径可行性检查器分析。一条路径是可行的如果存在一组输入数据使得程序按照这条路径执行。路径可行性检查器不是sound的。但是它很快。它尽最大努力来判定路径的可行性。实验证实它能够消除掉大部分的不可行路径，从而减少了整体的分析时间。所有的可行路径交给函数建模器来进行建模分析。

函数建模阶段的产物是对每个函数行为的总结信息。函数总结信息包括下列信息：

- 函数的可行路径
- 对堆对象的操作行为
- 返回值

除了程序中定义的函数之外，也对库函数进行建模。我们使用了自己定义的一种简单的库函数建模语言来描述C标准库函数的行为。例如，函数malloc()可以用`malloc { return heapobj }`, 函数`printf`被描述成可忽略的函数：`printf { ignored }`.

当前对300多个函数进行了建模，并且函数模型库还在根据需要增加之中。

把路径可行性的分析限制在一个函数体之内，没有考虑跨函数的路径的可行性。这样做可以降低计算开销。在将来，考虑跨函数的路径的可行性是一个研究课题。

在内存泄漏检查阶段，检查每条可行路径。我们集中分析堆对象的状态，而不是指向它们的指针。对内存对象进行精确的建模。

9.3.1　内存对象的建模

在实验所用的编译器LLVM中，所有的内存对象都是显式分配的，也就是通过指令分配的。全局对象在模块这一级别上定义。栈对象用alloca指令分配。堆对象用malloc指令分配。

用下列信息描述一个内存对象：

- Kind: 一个内存对象可能是Global, Stack, Argument, Heap中的一种。在LLVM中，函数参数用虚拟寄存器传入函数，被store指令存在栈上。用Argument类型来表示一个参数指针指向的对象。
- State: 给每个堆对象关联一个状态：Malloced, Freed, Returned, EscapedByArg, EscapedByGlobal, EscapedByUnknown. 分析算法不是域敏感的，不能追踪结构和数组元素的信息。所以我们把地址存入结构或数组

元素的堆对象的状态标记为EscapedByUnknown.由于这里的近似，不能发现由递归数据结构引起的内存泄漏。

- Index: 用一个索引序号来抽象内存地址。每个内存对象有个编号，是个整数值。分析不支持跨对象的地址计算。
- EscapedBy: 记录堆对象是被谁逃逸的。
- PointedBy: 记录谁指向这个对象。这在泄漏分析模块中使用到。
- SValue: 记录数据流信息，是格上的一个值。格在第9.4 节描述。

9.3.2 逃逸模型

给定一条程序路径，内存泄漏分析器通过模拟指针操作来追踪记录沿着这条路径上的堆对象的状态信息。当分析完一条路径之后，检查堆对象的状态。如果堆对象既没有被释放，也没有被逃逸，就认为它泄漏了。

图9.3显示了逃逸模型：returned, escaped by global variable, escaped by argument, escaped by unknown.最后一个escaped by unknown需要说明一下：通常这种情况发生在程序员构建递归数据结构的时候。堆对象的指针被存入一个结构的域中，就认为该堆对象escaped by unknown.

把函数建模和内存泄漏检查分开使得系统模块化更好。

9.4 数据流信息格

为分析设计了一个数据流信息格，见图9.4。

格中有3中类型的值：布尔，整数，指针。

- Boolean:
  - TRUE.
  - FALSE.
  - SymbolicBool. 符号布尔指是程序中的条件谓词，比如`x > 3`.
- Integer:

```
void *foo()
{
  void *p = malloc(8);
  return p;
}
```

The heap object is *returned*.

```
void *p;
void foo()
{
  p = malloc(8);
}
```

The heap object is *escaped by global variable* `p`.

```
void foo(void **p)
{
  *p = malloc(10);
}
```

The heap object is *escaped by argument* `p`.

```
struct list *head;
void foo(void) {
  struct list *n =
         malloc(sizeof(struct list));
  head->next = n;
}
```

The heap object n is *escaped by unknown*.

图 9.3: The escape model used by the analysis.

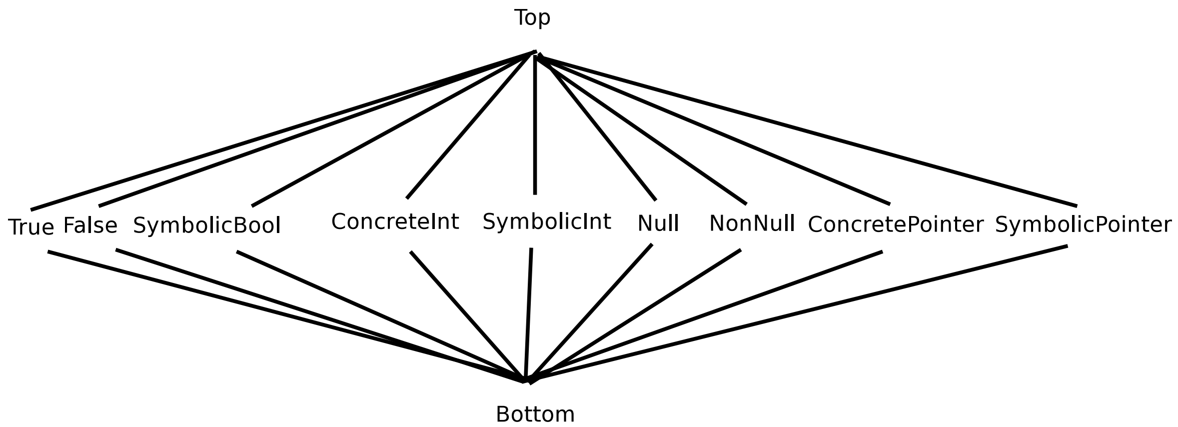

图 9.4: The lattice used in data flow analysis.

- Concrete Integer. 具体整数，例如3, 5, 8, ...
- Symbolic Integer. 符号整数是整数类型的表达式，它可能由语句`x += y`构造出来，其中x或y是符号值。

- Pointer:
  - Null.
  - NonNull. NonNull表示指针一定不是空。
  - ConcretePointer. 具体指针以它指向的内存对象的索引为值。
  - SymbolicPointer. 符号指针是未知的指针值，比如`p = strchr(s, '.');` `p` 得到一个指向字符串`s`中某个位置的指针。

这个格可以保留程序中的大部分状态信息。符号值未知量交给约束求解器来处理。把含有未知量的表达式交给约束求解器，然后它返回是否可满足的信息。

9.5 函数建模

函数建模的目的是得到关于函数的两类信息：一类是函数中的可行路径，它们将被内存泄漏检查器来进行分析；另一类是函数关于堆对象的行为信息。先讨论后者。

函数对堆对象的行为可以总结为下面几类：

```
int foo(int **p) {
  int *t = malloc(8);
  if (t) {
    *p = t; return 1;
  } else {
    return 0;
  }
}
```

图 9.5: 函数返回值与malloc有关

- *None*: 函数没有跟堆对象有关的行为。
- *ReturnHeapObj*: 函数分配了堆对象，并返回指向它的指针。
- *MallocGlobal*: 函数分配了堆对象，并把它的地址存入一个全局变量。
- *MallocArg*: 函数分配了堆对象，并把它的地址存入一个参数指向的变量。
- *FreeGlobal*: 函数释放了一个全局变量指向的堆对象。
- *FreeArg*: 函数释放了一个参数指向的堆对象。

在最初的实验之后，我们观察到一些经常出现的代码模式，发现有必要对上面的函数行为进行扩展，增加跟行为相关联的路径条件信息。下面是一个说明例子。

在图9.5中，函数的返回值和内存分配的结果相关。函数foo()返回1 意味着内存分配成功，返回0意味着内存分配失败。在图9.6中，全局变量g只有在它为NULL时才会被分配内存。

上面两个例子里的代码有一个共同的特征，就是它们关于堆对象的行为都关联着一个条件。如果不对这种现象进行建模，会得到很多假报错。但是并不是所有的内存分配行为都会关联一个特定的返回值。在图9.7中的代码里，返回值就和第3行的内存分配行为无关。

```
int *g;
void bar() {
  if (!g)
    g = malloc(8);
  else
    // do not allocate
}
```

图 9.6: malloc行为的前置条件是g == 0

```
int foo(int **p, int x) {
  *p = malloc(8);
  if (x > 3)
    return 1;
  else
    return 0;
}
```

图 9.7: 函数返回值与malloc无关

为了建立正确的内存分配行为和返回值之间的关系，我们收集所有有内存分配行为路径的返回值，并对它们作格9.4中的meet操作，以得到最终的内存分配行为的关联返回值。

经过扩展后的函数总结信息如下：

- $None$: 函数没有跟堆对象有关的行为。
- $ReturnHeapObj$: 函数分配了堆对象，并返回指向它的指针。
- $MallocGlobal$: 函数分配了堆对象，并把它的地址存入一个全局变量。
- $MallocGlobalCond$: 函数在某路径条件下分配了堆对象，并把它的地址存入一个全局变量，

```
Model -> Name { Behavior }
Name -> [A-Za-z0-9_]*
Behavior -> ignored
           | return Value
Value -> heapobj
        | int
        | pointer
        | @1
```

图 9.8: 库函数建模语言

表 9.2: 一些库函数建模的例子

模型	意义
`random { return int }`	返回符号整数
`strchr { return pointer }`	返回符号指针
`strcat { return @1 }`	返回第一个参数

- *MallocArg*: 函数分配了堆对象，并把它的地址存入一个参数指向的变量。
- *MallocArgRetVal*: 函数分配了堆对象，并把它的地址存入一个参数指向的变量，并有一个相关联的返回值。
- *FreeGlobal*: 函数释放了一个全局变量指向的堆对象。
- *FreeArg*: 函数释放了一个参数指向的堆对象。

9.5.1 库函数建模

C语言有很大的标准库。如果完全忽略这些库函数会造成很不精确的分析结果。我们设计了一种函数建模语言来帮助手工对函数建模。当前这种语言主要为内存泄漏服务，不过对其他行为的描述可以容易的加进去。

建模语言的语法见图9.8。表9.2显示了一些库函数建模的例子。

现在在库函数建模库里有大约300个函数。新的函数在有需要的时候被添加进去。

9.5.2 路径生成

函数的分析是路径敏感的。路径生成器从CFG中生成路径。循环被展开0次和1次，这对内存泄漏检查来说是有效的。生成函数体内的所有路径（循环展开次数有限）。在路径的基础上进行分析有以下的好处：

- 每次分析一条路径简化了指针分析。保守的指针分析不再需要了。在一条路径中一个指针最多能指向一个位置。通常可以精确地知道指针指向哪里。
- 基于路径的分析给程序员定位错误带来了方便。一旦内存泄漏错误被报告，造成该错误的路径就已经可以提供给用户了。这对我们在开发工具过程中进行调试也提供了极大的方便。
- 基于路径的分析能够降低误报率。在一个高精度的路径可行性分析器的帮助下消除了大部分的误报。

基于路径的分析的缺点在于代价相对较高。因为随着CFG中分支数的增加，路径数指数的增加。不过限制了循环的次数，最终得到的路径条数是可以承受的。

9.5.3 路径可行性分析

在图9.4的格上对路径进行符号执行，收集路径条件，求解路径条件来判断路径的可行性。在分析开始的时候，全局变量和函数参数被设置成符号值。然后根据程序的操作进行符号执行。

有一些操作现在的符号执行系统没有模拟，他们的结果被置成格中的$\perp$。这些操作包括：浮点运算，除法，位操作。函数调用根据函数总结信息来模拟。由于是自底向上的访问函数，通常被调用的函数的总结信息已经存在了。当递归调用存在时，简单的忽略函数的调用。这些近似使得路径分析器并不是sound的，会有一些不可行的路径被判断为可行的。但是由于只是要减少误报，所以并没有很大的影响。

当路径被分析完之后，路径条件交给约束求解器进行求解。理论上说SMT求解器的计算复杂度都是指数级的。但是实际观察到的结论是，路径中出现的约束大都十分简单，求解起来不需要指数级的复杂度。

可行路径被存入函数的总结信息，在以后的函数建模和内存泄漏检查阶段使用。

9.5.4 函数建模

函数建模器提取函数的对堆对象的操作行为信息，第9.5对这些行为已有描述。建模器分析第9.5.3节得到的每条可行路径。关注的焦点是指针类型的变量。

当遇到函数调用时，应用函数总结中该函数的信息，进行相应的模拟。

当堆对象的指针被存入一个变量时，根据该变量的类型改变堆对象的状态。如果该变量是全局变量，设置堆对象状态为EscapedByGlobal, 如果该变量是参数指向的，设置堆对象的状态为EscapedByArg. 如果是结构或者数组的域，我们设置堆对象的状态为EscapedByUnknown. 如果堆对象指针被返回，则设置堆对象的状态为Returned.

这里有一个要注意的是，除了要改变正在操作的堆对象的状态以外，还得把被赋值到的指针变量原先指向的堆对象的状态改变为Malloced. 举个例子：

```
void f(int **p) {
  *p = malloc(10);
  ...
  *p = malloc(10);
}
```

这里，当*p第二次被赋值时，第一次分配的内存实际上没有被逃逸。如果不改变第一次分配的堆对象的状态，就会造成*p逃逸了二个堆对象的错误。

分析完路径之后，检查路径中每个堆对象的状态。如果有EscapedByArg状态的对象，在函数总结中记录相应的参数和返回值的信息。如果有EscapedByGlobal 状态的对象，在函数总结中记录相应的全局变量信息。如果有Returned状态的对象，在函数总结中记录ReturnHeapObj信息。

9.6 内存泄漏检查器

当建模完所有的函数之后，就开始检查内存泄漏。

依次访问函数。分析每条存在函数总结中的可行路径的信息。分析的过程类似于符号执行和函数建模的过程。同样记录堆对象的状态。在最后，如果堆对象的状态是Malloced，则认为该对象被泄漏了。

9.7 实现和实验

我们在LLVM[50]编译器框架上实现了检测工具。LLVM是一种低级的中间代码指令集。它类似于RISC的指令，但是提供了丰富的语言独立的类型信息，是一种SSA表示。LLVM显式分配内存对象。所有的计算发生在虚拟寄存器当中。虚拟寄存器和内存位置是不同的。虚拟寄存器没有地址，只能表示标量。数据值在虚拟寄存器和内存之间通过load和store指令进行转移。

我们使用被实验的程序自身的build框架来构建程序，通常是GNU的autotools。程序被编译成LLVM的中间表示代码bitcode. 得到的bitcode文件包含程序所有的函数。然后就可以做完整的跨过程分析。

我们选择了一些人们经常使用的程序进行实验，包括make, wget, bzip2, gzip, adns, time, protpd, which等等。这些程序的大小在2000行到50000行代码之间。所有的实验在2G内存的笔记本上进行。检查程序的时间都在5分钟以内。

对这些常见的程序工具也发现了若干个内存泄漏，图9.9和9.10是我们挑选的二个具有代表性的错误。这些错误都得到了开发人员的证实，并且在新版本中得到了修复。这些错误都是跨过程的错误，没有函数的行为信息根本无法发现这类错误。

9.8 总结

本章描述了一种跨过程的基于函数总结和符号执行的内存泄漏检测算法。算法根据函数对动态内存分配的行为为函数建立总结信息，把内存泄漏分析扩展到了全程序，大大增强了内存泄漏的检测能力。经过对算法的实践检验，检测到真实程序中的内存泄漏错误多处，证明了算法的有效性。

```
// in file which.c:
int path_search(...) {
  if (...) {
    ...
    do {
M:    result = find_command_in_path(...);
      if (result) {
        ...
        if (...) {
        } else if (in_home) {
          if (skip_tilde) {
            next = 1;
L:          continue;
          }
          ...
        }
        ...
      }
      free(result);
    } while (...)
  }
}
```

图 9.9: which 2.16中的内存泄漏：result从find_command_in_path()的调用中得到一块动态分配的内存，如果程序执行到代码L行处，free(result)就被跳过了，开始新的循环，result得到一块新分配的内存，旧的内存就被泄漏了。

```
// in file ftp-basic.c:
uerr_t ftp_pasv(...) {
   ...
M: err = ftp_response (csock, &respline);
   ...
   s = respline;
   for (s += 4; *s && !ISDIGIT (*s); s++);
   if (!*s)
L:   return FTPINVPASV;
   ...
}
```

图 9.10: wget 1.10.2中的内存泄漏：ftp_response()分配一块内存并将其地址存入它的第二个参数指向的变量。因此respline在程序M行处获得一块动态分配的内存，如果程序执行到L行处，内存没有被释放，函数就返回了，造成内存泄漏。

第十章　相关工作

近年来，计算机安全成为研究领域的一个热点。源代码的分析是提高软件安全性的一个重要的方法。程序分析有两种方式，动态分析和静态分析。动态分析是将程序插桩后编译成可执行文件，实际运行程序得到相关的信息的分析方法。动态分析的优点在于实现相对简单，得到的信息较为精确，因为信息是从程序实际运行当中得到的，一般不会有虚假报错。但是动态分析也有它的缺点。动态分析要求分析的程序是完整的，可编译成可执行程序。同时，动态分析能够得到的信息仅限于被执行的路径。没有被执行到的代码的信息是得不到的。所以动态分析的效果强烈依赖于输入数据的质量。

本文所关注的主要是静态分析方法。静态分析相对于动态分析有以下的优点。

- 静态分析不受限于输入数据。静态分析可以分析程序的每一条路径。各种实际执行中难以执行到的代码都可以分析到。
- 静态分析不需要程序是完整的，只要求程序可以被编译，不需要程序被链接成可执行文件。

当然，静态分析的缺点也是明显的，比如精确性不如动态分析的高，并且由于探索的状态空间大，效率也低一些。

下面对相关工作的介绍集中于静态分析方面。

10.1　错误检测

自动程序错误检测方面的工作近年来非常多。尤其微软公司在这方面投入了很大的研究力量。

ESP[22]是微软具有代表性的工作之一。ESP提出了一种性质模拟的思想。由于程序的状态非常多，完全模拟不现实。所以ESP只关注跟bug相关的程序性质，比如文件的打开关闭状态。这样使得分析的效率得以大大的提高。ESP还提供了一个完整的过程间分析算法，具有很好的参考价值。我们提

出的过程间算法在ESP的基础上引入了路径条件和函数总结的关联，可大大提高分析的精确度。相关的工作还有[25].

Hampapuram等人[39] 也使用了一种基于region的内存模型。我们的模型在很多方面与它们的不同。我们模型的主要特点是引入了region层级结构，这种层级结构在处理内存对象之间的位置关系方面是至关重要的。而且，我们的内存模型可以精确的追踪经过类型强制转换的内存对象。

模型检测是微软在程序正确性检查方面的一个重要的研究方向。他们提出先将程序抽象成非常简单的布尔程序，再根据检测结果对程序的抽象逐步精化的研究思路，这方面的工作有：[6][5][4][7][3].

除了微软之外，Stanford大学Dawson Engler小组的工作也有非常大的影响力。MetaCompiling [28][38]中描述了一组编译器的扩展规则找到了500多个Linux, OpenBSD, exokernel中的bug。EXE [12] 让程序以符号值作为输入数据运行，使用的是标准的符号执行方法。所不同的是，EXE并不是对程序进行静态符号模拟，而是对程序进行插桩之后，编译成可执行程序来运行。这样做的好处是可以利用程序运行时的优势，对无法进行符号执行的部分进行具体执行。EXE使用的约束求解器是STP [32]，一个专门为其设计的基于SAT的求解器。EXE最初的思想出现在关于EGT [11]的技术报告中。

KLEE [10]是Engler小组最近的工作，也是他们认为最好的一个工作。Klee是一个在LLVM [50] 编译器框架上实现的符号执行虚拟机，所使用的技术就是符号执行。所做的贡献主要有3点：一是工程做的非常好，工具非常稳定，二是在环境模拟上做了一些工作，可以在人工指导下模拟命令行参数，三是使用了一些约束求解的优化技术，提高了效率。这些特点使得Klee可以完全符号执行许多系统级的代码，比如GNU coreutils [21]等等。

除了符号执行以外，Engler的小组在自动错误查找方面还使用其他方法进行了尝试，如模型检测[67][69][68]，使用策略进行估计[29]，对错误报告进行排序[48]等等。

Saturn [64] 是一个静态检测系统框架。它使用布尔可满足性(SAT)技术对程序操作进行建模，精确到位操作级别。在Saturn中，使用Guarded Location Set (GLS) 对指针进行建模。GLS方法实质上给每个内存位置起了一个名字，并记录一个指针变量可能指向的位置集合。这种内存建模方法跟我们的三元内存模型相比单薄许多。在我们的内存模型中，内存位置是作为一种值，可以进

行运算。Saturn 的基于名字的内存模型使得它不能对指针算术和数组进行推理，而在我们的内存模型中这两者都可以被很自然的处理。

Calysto[2]是又一个基于LLVM的静态分析工具。它号称对程序进行path-sensitive, context-sensitive, bit-precise的分析。但是仅从论文来看，看不出太多的技术细节。它提出的structural abstraction，本质上是一种逐步求精的方法。对函数调用先用比较粗略的约束代替，在不满足要求的情况下，在对函数进行进一步更精确的分析。

10.2 符号执行

符号执行在七十年代被提出[9][46]。基本的想法是对每条程序路径生成路径条件，求解路径条件得到能驱使程序执行该路径的输入数据。但是很少有系统完整的实现了这个想法。而能够在真实C程序上运行的工具则更少。

近年来基于符号执行的程序测试和错误查找方面的研究有复兴的趋势。

DART[35]项目和CUTE[56]项目都把具体执行和符号执行混合在一起。DART用一个具体的输入运行被测试的单元程序。同时收集路径上的路径条件。然后把其中一个路径条件取反，进行约束求解。这样得到的输入数据就可以保证使程序沿着和上一次不同的路径执行。DART只处理关于整数值的约束。当符号执行进行不下去的时候，DART采用随机测试继续。

CUTE扩展了DART的工作，能够处理如下形式的简单的指针约束：p == NULL, p != NULL. 但是不能处理符号数组下标。相对于DART和CUTE，SimC对符号指针和数组下标也进行了建模，从而能够生成更精确的测试数据。

CBMC[18]是针对C语言的有限模型检测工具。它的设计目的是检测程序中的指针安全，数组限界安全以及用户提供的assertion。像SimC一样，CBMC完全符号地执行代码。它把代码翻译成布尔公式。如果有错误发生，则相应的布尔公式是可满足的。翻译得到的布尔公式被SAT求解器求解。CBMC对循环和递归函数调用进行展开。这有时会使检测陷入无限循环当中。同时CBMC缺少对库函数的支持。这些缺点使得CBMC无法检查真实的代码。

Xie等人[62]描述了如何使用符号执行对面向对象的程序生成测试数据。但是他们的工作没有考虑数组表达式。

[24]提出一种sound complete的路径敏感程序分析方法。它对于函数总结作了简化，记录的约束是一个函数f在输入为α的情况下返回特殊值c的路径

条件。其实也就是输入参数和返回值之间的约束。没有说怎么处理循环。只能回答一个函数是否可能或者一定返回某个特殊的值。使用的约束求解工具是SAT.由于只记录变量和常数之间的等于或者不等关系，所以SAT够用了。

10.3 测试数据生成

第八章中的工作前身是[72]。[73] 描述了对[72]的一点扩展。[71]对结构和指针的处理方法提出了最初的想法。第八章的工作在这些工作的基础上进行了扩展和实现。对指针和结构的扩展更加详细。实现了原型工具，并在真实程序上进行了实验。严俊的基于basis path的测试生成的工作[66]也是基于本文的工作所做的。

先前的测试数据生成工作基本上都在作者自己定义的语言上实验[54][30]。程序里面没有指针和数组。

Gotlieb等人[37]提出了基于SSA和约束求解的自动测试生成技术。他们的系统接受C语言的一个子集，不包括指针和动态内存分配。他们文章中实验的最大的程序在20行左右。

Korel[47]使用动态方法，在程序分支处进行回溯搜索。他把测试数据生成问题规约成函数最小化问题，使用近似搜索的办法求解。这种方法并不能保证生成正确的测试数据。

Demillo等人[23]描述了一种用符号执行来生成测试数据的方法。他们的方法基于已发现的错误。他们的约束求解方法是近似的，对路径的遍历也不完整。

Ferguson等人[30]提出一种使用程序的实际执行来生成测试数据的方法。[31] 也使用了程序插桩的办法。这些办法每次只考虑程序的一个分支和一个输入变量，同时使用回溯。所以它们的方法需要多次的迭代才能覆盖所有的分支，效率相当低。如果几个路径条件依赖同一个输入变量，这种方法会在回溯上浪费大量的时间。

近年来，微软在测试生成方面也做了大量的研究工作，其基本思想也是利用符号执行的方法收集路径约束，再利用SMT求解器生成测试数据，代表工作有[55][34][36][33]

10.4　内存泄漏

使用动态分析的内存泄漏检查已经进入程序员的工具箱很多年了。Purify[40]和Valgrind[60]是两个代表性的动态检测工具。动态内存泄漏检测受到测试数据的限制。如果测试数据不能够触发内存泄漏，它就检测不到。

因为我们研究的是静态分析技术，所以这里对相关工作的介绍也集中于静态方面。

Saturn [63]是和我们的工作最接近的。它把内存泄漏问题规约成布尔可满足性问题，然后用SAT求解器求解来发现错误。他们的逃逸分析和我们的类似：任何没有释放也没有被逃逸的内存被认为泄漏了。但是我们使用了不同的函数总结系统和路径可行性分析系统。具体来说，我们的函数总结系统刻画了函数参数导致的逃逸出函数的行为。而Saturn的函数总结刻画了函数参数导致的传递进函数的逃逸行为。所以Saturn 不能检测出图9.1和图9.10中的错误。并且，我们采用的程序分析方法与Saturn不同。我们使用数据流分析和符号执行，得到路径条件后用SMT求解。这比全部使用SAT来求解要快和自然一些。

Clouseau[41]使用了一种拥有模型来追踪对象的引用。在这种模型中，每个对象只能被一个指针拥有。它们对程序行为作了大量的假设，这种假设适合于设计良好的C++程序。对于C程序来说就不怎么实用了。

LC[53]使用向后的堆分析来反证内存泄漏的不存在。为了确定内存泄漏是否会在一处发生，分析使用反向的跨过程流分析来反证它的不存在。

Fastcheck[15]通过guarded value flows来检测内存泄漏。他用稀疏的程序value flow graph来追踪从内存分配地点到释放地点的值传播情况。LC和Fastcheck 都没有像我们的方法一样对函数进行详细的建模。他们对路径可行性的考虑也是用SAT来求解。

我们把LC应用于图9.1中的例子上，LC报了二个警报。它没能消除L1行处的假错误。而我们的工具则只报了L2行处的真错误。

10.5　过程间分析

过程间分析一直是程序分析领域的难题，虽然有很多的工作，但是没有特别有效的方法。老的研究大多集中于mod/ref, aliasing分析和常数传播。一般的方法是将要得到的信息分解成几个部分，然后通过分别分析被调用的函数和

本函数得到相关的信息，这其中的不同之处主要是分析顺序和近似算法的不同。代表工作有：[49][16][14][57][61][19][44][13][59][42][26]。

新的工作致力于对过程生成比较精确的summary，能够表示的信息用一种较为抽象的框架来描述，但实际上能够表示的信息还是像以前一样那么几种，代表工作有[70]。

第十一章　结语

本文所关注的方面是对程序正确性的自动检查。计算机程序的复杂性和规模一直阻碍着人们对程序正确性作出严格的证明。一直以来，人们除了用测试和人工代码审查以外，没有什么好的办法来提高程序的正确性。

对程序正确性的自动检查是一个跨学科的领域。在这里，编译原理、数理逻辑、定理证明、程序设计语言，软件工程等领域交织在一起。

本文沿着Floyd, Hoare, Dijkstra等先驱者的道路，把程序正确性方面的理论成果向着实用方面推进了一步。针对实用化中的困难进行了尝试，取得了一些成果。

11.1　本文的贡献

本文对C程序的静态分析有如下的贡献：

- 根据程序精确符号分析的要求和C语言的语义设计了一种新的内存模型，使得对程序进行精确的符号分析变得可行（第三章）。
- 对如何进行C程序的符号分析给出了详细的算法描述，用于对C程序进行模拟执行（第四章）。
- 设计了针对单元C程序的自动测试数据生成算法，并实现了相应的工具（第八章）。
- 设计了过程间的自动内存泄漏检测算法，并实现了相应的工具。应用该算法检测出了真实程序中的内存泄漏缺陷（第九章）。
- 根据上述的模型和算法，在开源编译器Clang [17]上实现了一个完整的静态分析框架。

11.2　进一步的工作

总体的目标是尽量多的把人脑对程序进行的推理算法化。本文已经提出了不少的算法，将来要针对还没有很好解决的问题再进行研究和实验。在过程内

分析方面，目前对于循环的处理还不是很完善，将来需要针对某些特殊的错误类型，比如数组越界，对循环作有针对性的分析，比如自动识别迭代变量等。对某些操作，如位操作，非线性运算，主要的处理方法是把它们交给约束求解器来处理，这需要对SMT求解器技术进行研究。在过程间分析方面，对函数总结信息的提取仍然没有得到完善的解决，存在的主要问题是总结信息精度不够，导致误报率较高。将来还要针对具体的错误类型对函数总结信息的提取作进一步的改进。

未来另一个方向的工作是把现有的语义模型扩展到C++。C++是一种比C复杂得多的语言。完整地处理C++的语义绝非易事。如何模拟C++的对象模型是首先要解决的问题。在这方面的工作几乎是空白。

最宏伟的目标是自动验证程序的正确性。我们所能做的，是在向着这个目标的道路上努力。就像人工智能研究一样，虽然至今没有实现真正的人工智能，但是研究过程中产生的新技术已经使人们受益匪浅。

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[2] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of ICSE 2008*, 2008.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob, Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[4] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *ACM SIGPLAN Notices*, 36(5), 2001.

[5] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(2):314–343, 2005.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[8] Antonia Bertolino and Martina Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, 1994.

[9] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT — a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, 2008.

[11] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. Technical Report CSTR 2005-04 3, Stanford University, 2005.

[12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), December 2008.

[13] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of PLDI*, 1988.

[14] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *Proceedings of PLDI*, 1995.

[15] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proc. of PLDI-2007*, 2007.

[16] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of POPL*, 1993.

[17] Clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[18] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the ASP-DAC 2003*, pages 308–311, 2003.

[19] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of PLDI*, 1988.

[20] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[21] GNU core utitlities. http://www.gnu.org/software/coreutils/.

[22] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[23] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[24] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, 2008.

[25] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of ISSTA 2004*, 2004.

[26] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of PLDI*, 1994.

[27] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2000.

[28] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, 2000.

[29] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[30] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. on Software Engineering and Methodology*, 5(1):63–86, 1996.

[31] Matthew J. Gallagher and V. Lakshmi Narasimhan. ADTEST: A test data generation suite for ada software systems. *IEEE Trans. on Software Engineering*, 23(8), 1997.

[32] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of Computer Aided Verification 2007*, 2007.

[33] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007.

[34] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[36] Patrice Godefroid, Michael Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS) 2008*, 2008.

[37] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the 1998*

ACM SIGSOFT international symposium on Software testing and analysis, pages 53–62, 1998.

[38] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[39] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 52–58, 2005.

[40] R Hastings and B Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[41] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of ACM PLDI 2003*, 2003.

[42] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[43] WG14 ISO/IEC 9899:201x, editor. *Programming Languages - C*. ISO, 1999. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1336.pdf.

[44] Neil D. Jones and Stevens S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of POPL*, 1982.

[45] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall PTR, 1978.

[46] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[47] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[48] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2004.

[49] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of PLDI*, 1993.

[50] The LLVM compiler infrastructure. http://llvm.org/.

[51] lp_solve. http://groups.yahoo.com/group/lp_solve/.

[52] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Academic Press, 1997.

[53] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradiction. In *Proceedings of the International Static Analysis Symposium (SAS '06)*, 2006.

[54] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.

[55] Pex - automated white box testing for .NET. http://research.microsoft.com/en-us/projects/pex/.

[56] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[57] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of POPL*, 1997.

[58] SMT-LIB, the satisfiability modulo theories library. http://combination.cs.uiowa.edu/smtlib/.

[59] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of POPL*, 1996.

[60] Valgrind. http://www.valgrind.org/.

[61] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of POPL*, 1980.

[62] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of TACAS 2005*, LNCS 3440, pages 365–380.

[63] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proc. of ECSE/FSE 2005*, 2005.

[64] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3), 2007.

[65] Zhongxing Xu and Jian Zhang. A test data generation tool for unit testing of C programs. In *Proceedings of the International Conference on Quality Software*, 2006.

[66] Jun Yan and Jian Zhang. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 107(3-4), 2008.

[67] Junfeng Yang, Can Sar, and Dawson Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, 2006.

[68] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '06)*, 2006.

[69] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.

[70] Greta Yorsh, Eran Yahav, and Satish Chandra. Generaing precise and concise procedure summaries. In *POPL 2008*, 2008.

[71] Jian Zhang. Symbolic execution of program paths involving pointers and structure variables. In *Proceedings of the Fourth International Conference on Quality Software*, pages 87–92, 2004.

[72] Jian Zhang and Xiaoxu Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.

[73] Jian Zhang, Chen Xu, and Xiaoliang Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004.

发表文章目录

[1] Zhongxing Xu and Jian Zhang, Path Context Sensitive Memory Leak Detection, Proceedings of the Eighth International Conference on Quality Software (QSIC 2008)

[2] Zhongxing Xu and Jian Zhang, SimC: Automatic Test Data Generation for Unit C Programs, Proceedings of the Sixth International Conference on Quality Software (QSIC 2006)

[3] Zhongxing Xu, Ted Kremenek, and Jian Zhang, A Memory Model for Static Analysis of C Programs, FSE 2009 (Submitted)

[4] J. Yan, J. Zhang, and Z. Xu, Finding Relations Among Linear Constraints, Proceedings of the 8th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2006)

简　　历

基本情况

许中兴，男，江苏省无锡市人，1981年6月出生，已婚，中国科学院软件研究所博士研究生。

教育状况

1999 年 9 月至 2003 年 7 月，清华大学数学系，本科，专业：数学与应用数学。

2004 年 9 月至 2009 年 7 月，中国科学院软件研究所，硕博连读研究生，专业：计算机软件与理论。

工作经历

2003年 10月 2004年 8月，中国科学院软件研究所计算机科学实验室实习

研究兴趣

程序分析，编译器

联系方式

通讯地址：北京市海淀区中关村南四街4号，中国科学院软件研究所计算机科学实验室

邮编：100190

E-mail: xuzhongxing@gmail.com

致　谢

首先要感谢的是我的导师张健研究员，我还记得五年前在我前途迷茫之际您只跟我通了个电话就接收我到您身边学习。此后一步步带我走上科学研究之路，在我遇到困难的时候总是能给我很大的帮助，并让我自由地探索感兴趣的领域。

感谢严俊学长这五年来从各个方面给我的帮助，在我感到困惑时用自己的经历给我答疑。也感谢陈伟、刘生、贾祥雪，阮辉，陈百强，马菲菲同学，大家一起愉快地度过了研究生的时光。

感谢实验室的系统管理员庄老师，秘书郭老师，张丽，图书馆的黄老师等为我的工作和生活提供了种种便利。

感谢LLVM/Clang开发团队的工程师们，特别是Ted Kremenek，从与你的讨论和合作中我学到了很多东西。

感谢我的父母和妻子，你们一直是我生活中最大的支持者。