第 42 卷 第 6A 期
2015 年 6 月

计 算 机 科 学
Computer Science

Vol. 42 No. 6A
June 2015

简化 GCC 抽象语法树的新型算法

田冰川 孙 珂 巢汉青

(南京航空航天大学计算机科学与技术学院 南京 211100)

摘 要 抽象语法树是程序源代码的树状表现形式,在代码分析与特征提取过程中发挥着重要作用。GCC 可以导出 C 语言源程序的抽象语法树文件,但其中包含大量冗余信息与无关信息,不利于上述工作的展开。针对此问题,提出一种简化 GCC 抽象语法树的算法,在保持语法树基本结构完整的前提下,移除其中与源程序无直接关联的节点,以线性时间复杂度重建语法树文件,达到简化的目的。

关键词 GCC,抽象语法树,简化,算法,C 语言

中图法分类号 TP311.56 文献标识码 A

New Algorithm of Simplifying GCC Syntax Tree

TIAN Bing-chuan SUN Ke CHAO Han-qing

(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211100, China)

Abstract Abstract syntax tree(AST) is tree representation of program source code, which plays a significant role in code analysis and feature extraction process. GCC can help to export the AST file in the source program of C language, which however contains a large amount of redundant information and irrelevant information that will exert a negative influence on the accomplishment of relevant tasks. To address this problem, a simplified GCC AST algorithm was presented to remove the nodes irrelevant to source program and rebuild AST file by means of linear time complexity based on a complete basic structure of AST for the sake of simplification.

Keywords GCC, Syntax tree, Simplification, Algorithm, C language

1 引言

1.1 研究背景

抽象语法树在缓冲区溢出漏洞的检测[1]、代码抄袭检测[2]、提取程序特征[3]等代码分析领域发挥着重要的作用,其构建可以使用已有的开源工具,也可以根据需求自行实现[4]。GCC 是构建抽象语法树常用的开源工具,然而其生成的语法树文件包含大量与源代码无直接关联的信息,这些信息不仅不利于提高代码分析的准确度,还会严重影响到代码分析的效率。因此,对 GCC 抽象语法树的简化变得尤为重要。国内外对这一课题有一些研究,但在诸多方面仍存在着问题。

1.2 GCC 抽象语法树

定义 1(抽象语法树文件) C 语言源文件经过 GCC 编译后形成的以文本形式存储的语法树文件

定义 2(抽象语法树) 通过解析抽象语法树文件得到的树形结构。

定义 3(返祖边) GCC 抽象语法树文件中,节点指向其某一级祖先的边。

在源代码的编译过程中,GCC 对每个源文件生成一个抽象语法树文件。使用-fdump-tree-all 参数或者-fdump-translation-unit 参数[5]均能得到以“源代码文件名 001t.tu”命名的抽象语法树文件。抽象语法树文件是一种以行为单位存储的文本文件,每一行或若干行描述了一个节点。

图 1 描述了抽象语法树的存储结构。图中包含抽象语法树文件中的 8 个节点,每个节点均由 3 部分组成:节点编号、节点类型、子节点信息。节点编号是一个节点的唯一标识符,GCC 抽象语法树文件保证每一个节点按照编号连续升序排列。

```
@1    type_decl         name: @2     type: @3       chan: @4
@2    identifier_node   strg: int    lngt: 3
@3    integer_type      name: @1     size: @5       align: 32
                        prec: 32     sign: signed   min : @6
                        max : @7
@4    type_decl         name: @8     type: @9       chan: @10
@5    integer_cst       type: @11    low : 32
@6    integer_cst       type: @3     high: -1       low : -2147483648
@7    integer_cst       type: @3     low : 2147483647
@8    identifier_node   strg: char   lngt: 4
```

图 1 GCC 抽象语法树的存储结构

抽象语法树的节点可分为 8 种类型,如表 1 所列。其中,仅有某一类特定的表达式节点与简化算法相关,对于其他节点,不加区分地将其作为普通节点处理。

图 2 描述了抽象语法树的逻辑结构,其结构与图 1 相对应。由于返祖边的存在,抽象语法树的存储结构与逻辑结构并不完全对等。

本文受中央高校基本科研业务费专项资金资助。

田冰川(1993—),男,主要研究方向为机器学习,E-mail:tbc@nuaa.edu.cn;孙 珂(1994—),男,主要研究方向为机器学习;巢汉青(1994—),男,主要研究方向为机器学习。

表 1　GCC 抽象语法树节点类型[6]

节点标识	节点含义
* –expr	表达式节点
* –type	类型节点
* –ref	引用节点
* –decl	声明节点
identifier–node	标识符节点
* –est	常量节点
* –list	列表节点
(else)	其他辅助节点

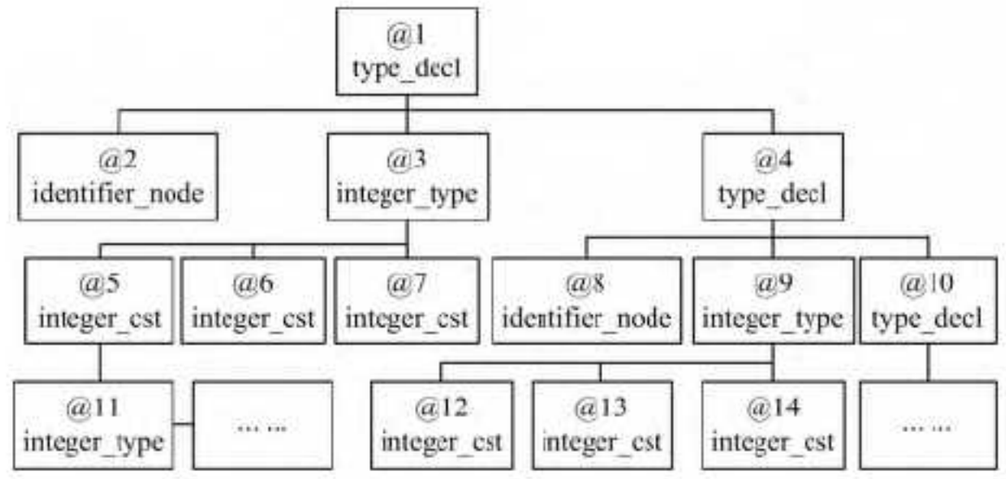

图 2　GCC 抽象语法树的逻辑结构

1.3　研究现状及存在问题

定义 4(冗余信息)　源代码中的系统头文件引入 GCC 抽象语法树中的无关信息，编译器为保障程序正常编译与执行而引入的无关信息。

GCC 生成的抽象语法树包含了许多有助于编译的细节信息[7]，例如由预处理指令 #include 所引入的头文件中的全部函数、常量、变量、结构体等，但这些信息不利于对源代码的分析。其次，抽象语法树文件体积庞大，约为源代码长度的 50～500 倍，增加了对源代码分析时的内存消耗，降低了分析效率。

通过对抽象语法树文件的分析可以发现，部分语法树的叶节点存在返祖边，对于代码分析而言，返祖边是冗余信息。由于其加大了代码分析的难度，构建语法树时应予以去除。有国内文献给出了一种基于宽度优先遍历的化简方法[7]，并取得了一定的效果，但新版本(如 4.8.2 及以上)GCC 生成的抽象语法树文件结构更为复杂，导致该算法的化简效果欠佳。

针对上述问题，本文提出一种简化 GCC 抽象语法树的新型算法，在不影响代码分析的基础上，以线性时间复杂度完成对 GCC 抽象语法树的化简，同时完成对抽象语法树文件的重建。

2　算法描述

2.1　算法设计目标

对 GCC 抽象语法树文件进行化简，在不影响代码分析的前提下，去除抽象语法树中与源程序无直接关系的部分，消除抽象语法树中的冗余关系。

2.2　算法的基本思想

2.2.1　节点分类

根据 scrp 字段，将节点分为已知节点和未知节点 2 类，已知节点包含有用节点和无用节点。包含 scrp 字段的节点为已知节点，若 scrp 字段的值为源文件文件名，则该节点为有用节点；scrp 字段的值不为源文件文件名，说明该节点来自系统头文件，与源文件没有直接关系，极有可能为无用节点。不包含 scrp 字段的节点为未知节点。

2.2.2　抽象建模

通过对 GCC 抽象语法树的分析可知，G 在逻辑结构上为树，而在存储结构上为有向图。将 GCC 抽象语法树抽象为图结构。令图 V 为点集，表示抽象语法树的节点；E 为边集，表示抽象语法树中的每一条有向边。

$$G=\langle V,E\rangle \tag{1}$$

2.2.3　求解过程

从每一个已知节点出发，对图 G 做深度优先遍历，找出返祖边。定义 E_{back} 为返祖边边集，令图

$$G'=G-E_{back} \tag{2}$$

易知 G' 为有向无环图。

对 G' 进行拓扑排序，得到 G' 的一个拓扑序 S。之所以对 G' 进行拓扑排序，是为了保证后续步骤中递推的正确与高效。

在拓扑序 S 上进行递推，对于每一个未知节点 V_i，其为有用节点当且仅当存在有用节点 V_s，且

$$\langle V_s,V_i\rangle\in E(G') \tag{3}$$

2.2.4　修正过程

节点分类方法将少部分有用节点误判为无用节点，需要在算法的最后一步将其找回。事实上，节点分类过程中的误判客观上保证了求解过程的正确性。遍历每一个类型为 call–expr(函数调用语句)的节点，其 fn 字段指向的节点 V_{fn} 是其调用函数的声明节点，该节点是有用节点。进一步，V_{fn} 的 $op0$ 字段指向的节点 V_{name} 是该函数的函数名称，其是否算作有用节点视代码分析的具体需求而定。

2.3　核心算法描述

算法 1　删除返祖边

输入：包含返祖边的 GCC 抽象语法树 G=〈V,E〉

输出：不含返祖边的抽象语法树 G′=〈V,E′〉

过程：

```
1. visit[] ← 0
2. for each v_i in V
3.    if v_i is not determined
4.       dfs(v_i)
5.    end if
6. end for
```

子过程：dfs(v_i)

```
1. visit[v_i] ← 1
2. for each 〈v_i, v_j〉 in E
3.    if visit[v_j]=0 then
4.       dfs(v_j)
5.    else
6.       if visit[v_j]=1 then
7.          remove 〈v_i, v_j〉 from E
8.       end if
9.    end if
10. end for
11. visit[v_i] ← 2
```

算法 2　求解未知节点

输入：不含返祖边的抽象语法树 G=〈V,E〉

G 的拓扑序 S[]

节点可用性列表 useful[]

其中，0 表示未知，1 表示有用，−1 表示无用

输出：只含 −1 和 1 的节点可用性列表 useful[]

过程：

```
1. for each v_i in V
2.    if useful[S[v_i]]=0 then
```

```
3.      useful[S[v_i]]=-1
4.    else
5.      for each ⟨v_i, v_j⟩ in E
6.        if not useful[v_j]=1 then
7.          useful[v_j] ← useful[v_i]
8.        end if
9.      end for
10.   end if
11. end for
```

算法 3　找回误消除的节点

输入：不含返祖边的抽象语法树 $G=\langle V,E\rangle$

按行存储的 GCC 抽象语法树文件文本 NodeText[]

节点可用性列表 useful[]

输出：经过修正的节点可用性列表 useful[]

过程：

```
1. for each i in NodeText
2.   if type of node[i] is "call_expr" then
3.     t1 ← node[i].fn
4.     useful[t1] ← 1
5.     t2 ← node[t1].op0
6.     useful[t2] ← 1
7.   end if
8. end for
```

2.4　算法复杂性分析

由于 GCC 抽象语法树文件的特殊性，文件中每行的长度有限，每个节点在抽象语法树中占据的行数有限，每个节点的子节点数目有限，因此，所有与单行文本有关的操作均能在常数时间 C 内完成。不妨设每个节点最多占据 L 行，每个节点最多有 S 个子节点。若令抽象语法树文件的行数为 n，则有

$$n/L\leqslant|V|\leqslant n/1 \tag{4}$$

且

$$n/L*1\leqslant|E|\leqslant n/1*S \tag{5}$$

进而有

$$|V|=\Theta(n) \tag{6}$$

且

$$|E|=\Theta(n) \tag{7}$$

分类操作与找回操作只需将 GCC 抽象语法树文件按行扫描一遍，并逐行处理，时间复杂度为

$$\Theta(C*n) \tag{8}$$

建模操作需要按行扫描抽象语法树文件，同时要遍历图的每一个点和每一条边，时间复杂度为

$$\Theta(C*n+2*n) \tag{9}$$

删边操作、排序操作及求解操作均需遍历图的每一个点和每一条边，时间复杂度为

$$\Theta(2*n) \tag{10}$$

因此，整个算法的时间复杂度为

$$T(n)=\Theta(n) \tag{11}$$

已达到理论下界。

算法的执行需要若干个长为 n、$|V|$ 和 $|E|$ 的辅助存储空间，递归调用所消耗的系统堆栈空间不超过 $|V|$。因此整个算法的空间复杂度为

$$S(n)=\Theta(n) \tag{12}$$

3　实验结果

3.1　算法性能

为了评估算法的化简性能，选取 5 段长度不等的 C 语言源代码对算法进行测试，分别统计化简前后抽象语法树文件中的节点数目。测试结果如表 2 所列，化简后抽象语法树文件对比如图 3 所示。

表 2　算法性能

程序编号	源代码行数	化简前节点数目	化简后节点数目	化简率
1	12	4028	130	96.77%
2	23	4064	168	95.87%
3	42	4084	190	95.35%
4	64	4199	303	92.78%
5	99	4272	423	90.10%

以上程序摘自文献[8]，均在 Linux 下使用 GCC 4.8.2 导出抽象语法树文件。可以看出，5 个抽象语法树文件的化简率均达到 90%以上。其中：

程序 1 选自 22 页《使用多条 printf 语句输出字符》；

程序 2 选自 23 页《加法程序》；

程序 3 选自 59 页《考试结果问题》；

程序 4 选自 209 页《使用"按引用调用"的冒泡排序》；

程序 5 选自 336 页《信用查询程序》。

简化前：

```
@80   integer_cst    type: @77    high: -1    low : -1
@81   identifier_node  strg: float  lngt: 5
@82   real_type      name: @78    size: @5    algn: 32
          prec: 32
@83   type_decl      name: @84    type: @85    chan: @86
@84   identifier_node  strg: double  lngt: 6
@85   real_type      name: @83    size: @19    algn: 64
          prec: 64
```

简化后：

```
@80   integer_cst    type: @77    high: -1    low : -1
@81   identifier_node  strg: float  lngt: 5
@82   real_type      name: @78    size: @5    algn: 32
          prec: 32
@107  void_type      name: @104   algn: 8
@464  pointer_type   size: @5     algn: 32    ptd : @3
@820  pointer_type   size: @5     algn: 32    ptd : @825
```

图 3　化简前后对比(局部)

3.2　算法效率

选取 20 组长度不等的 C 源文件，使用 GCC 生成抽象语法树文件，对每一个抽象语法树文件运行本算法 100 次，求得平均耗时如表 3 所列。使用最小二乘法对源文件代码长度与平均耗时进行线性拟合，得图 4 所示回归直线，方程为

$$\hat{y}=0.003347x+12.39 \tag{13}$$

相关系数为

$$\hat{r}=0.9981 \tag{14}$$

呈高度的线性正相关关系。

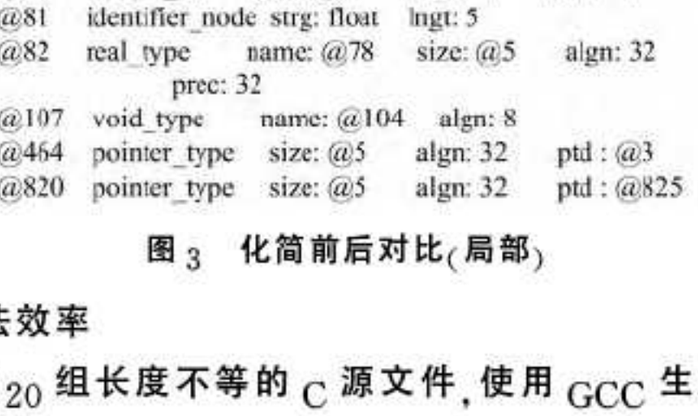

图 4　算法执行效率

(下转第 530 页)

和表 3 的实验结果可以看出，本文提出的检测模型对人工注入的缺陷具有较低的漏检率和误检率，尤其是漏检率全部为 0，说明该方法特别适用于对软件可靠性要求较高的安全关键的软件系统的缺陷检测，由于编程的灵活性和复杂性，检测到的疑似缺陷均需开发人员进一步确认是开发人员有意而为之（即误检）还是真正的缺陷。

结束语　本文在检测克隆代码的基础上进行上下文不一致性缺陷的检测，通过改进上下文不一致性缺陷检测的过滤规则和对建立的表达式抽象语法树进行标准化处理，能够在更广的范围上检测出上下文结构不一致性和上下文条件谓词不一致性两种缺陷。实验表明，该模型对这两种缺陷有较低的漏检率和误检率，能够有效地帮助开发人员尽早发现缺陷，提高软件质量。

参 考 文 献

[1] Baxter I D, Yahin A, Moura L, et al. Clone Detection Using Syntax Trees[C] // Proc. Int'l Conf. Software Maintenance. 1998: 368-377

[2] Prechelt L, Malpohl G, Philippsen M. Finding Plagiarisms among a Set of Programs with JPlag[J]. J. Universal Computer Science, 2002: 1016-1038

[3] Schleimer S, Wilkerson D S, Aiken A. Winnowing: Local Algorithms for Document Fingerprinting[C] // Proc. ACM SIGMOD Int'l Conf. Management of Data. 2003: 76-85

[4] Manber U. Finding Similar Files in a Large File System[C] // Proc. USENIX Winter 1994 Technical Conf. 1994: 1-10

[5] Jiang L, Misherghi G, Su Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C] // ICSE. 2007: 96-105

[6] Komondoor R, Horwitz S. Using Slicing to Identify Duplication in Source Code[C] // Proc. Eighth Int'l Symp, Static Analysis. 2001

[7] 李建忠，刘建宾，余楚迎. 基于过程蓝图的参数化重复代码检测技术研究[J]. 汕头大学学报：自然科学版，2007，22(1)：54-59

[8] Ducasse S, Rieger M, Demeyer S. A Language Independent Approach for Detecting Duplicated Code[C] // Proc. Int'l Conf. Software Maintenance. 1999: 109-118

[9] Van Rysselberghe F. Vingerafdrukken van code om duplicatie op te sporen[D]. Antwerper: Master Thesis at Universiteit Antwerpen, 2001-2002

[10] Baker B. A Program for Identifying Duplicated Code[C] // 24th Symposium on the Interface Proceedings of Computing Science and Statistics. 1992: 132-141

[11] Baker B. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenanc[C] // 25th Annual ACM Symposium on Theory of Computing. 1993: 75-96

[12] Baker B S. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance[J]. SIAM Journal on Computing, 1997, 26(5): 1343-1362

[13] Church K W, Helfman J I. Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code[J]. J. Computational and Graphical Statistics, 1993: 2(2): 153-174

[14] Kamiya T, Kusumoto S, Inoue K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code[J]. IEEE Trans. on Software Eng, 2002, 7(28): 654-670

[15] Basit H A, Jarzabek P S. Efficient token based clone detection with flexible tokenization[C] // ESEC/FSE'07. ACM Press, 2007: 513-516

[16] Wise M J. YAP3: improved detection of similarities in computer program and other texts[C] // Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education. 1996: 130-134

[17] L Jiang, Z Dong, E Chiu. Context-Based Detection of Clone-Related Bugs[M]. ACM Press, 2007: 55-64

[18] 王伟，苏小红，马培军，等. 标识符重命名不一致性缺陷的检测[J]. 哈尔滨工业大学学报，2011，43(1)：89-98

[19] Wang Tian-tian, Su Xiao-hong, Wang Yu-ying, et al. Semantic similarity-based grading of student programs[J]. Information and Software Technology, 2007, 49(2): 99-107

[20] 马培军，王甜甜，苏小红. 基于程序理解的编程题自动评分方法[J]. 计算机研究与发展，2009，46(7)：1136-1142

（上接第 518 页）

表 3　算法效率

编号	代码长度(Byte)	平均执行时间(ms)
1	197	13.45
2	1137	14.66
3	2127	18.95
4	2614	21.35
5	2960	21.55
6	3711	25.69
7	4088	27.21
8	5029	28.86
9	6017	29.81
10	6996	36.64
11	8395	40.31
12	8672	44.47
13	10504	49.61
14	11924	51.82
15	13588	58.19
16	14393	58.77
17	15265	63.48
18	17309	69.96
19	19256	77.15
20	20332	79.92

结束语　理论分析与实验验证表明，本算法可以在不影响代码分析的前提下，对 GCC 抽象语法树进行很大程度的化简，具有较低的时间复杂度，准确性和效率都得到了保障。经过化简后的 GCC 抽象语法树文件，可以更为方便地应用于代码分析领域，并在一定程度上提高相应算法的执行效率。

参 考 文 献

[1] 吴楠. 基于 GCC 的缓冲区溢出检测研究[D]. 哈尔滨：哈尔滨工程大学，2010

[2] 赵彦博. 基于抽象语法树的程序代码抄袭检测技术研究[D]. 呼和浩特：内蒙古师范大学，2010

[3] 张良德，赵彦博. 一种基于 GCC 抽象语法树的程序特征提取方法[J]. 电子技术与软件工程，2013(20)：269-270

[4] 江梦涛，荆琦. C 语言静态代码分析中的调用关系提取方法[J]. 计算机科学，2014，41(6A)：442-444

[5] Option Summary [EB/OL]. (2013-10-16) [2014-11-10]. http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Option-Summary.html

[6] 杨昌坤，许庆国. C 程序控制流程模型的提取技术与实现[J]. 计算机科学，2014，41(5)：208-214

[7] 李鑫，王甜甜，苏小红，等. 消除 GCC 抽象语法树文本中冗余信息的算法研究[J]. 计算机科学，2008，35(10)：170-172

[8] Deitel P, Deitel H. C 语言大学教程(第六版)[M]. 苏小红，等译. 北京：电子工业出版社，2012