第 33 卷 第 9 期 计 算 机 工 程 2007 年 5 月
Vol.33 No.9 Computer Engineering May 2007
·软件技术与数据库· 文章编号：1000—3428(2007)09—0066—03 文献标识码：A 中图分类号：TP311

基于中间表示的可复用程序分析系统

郑瑶海，陈 伟，赵 琛
(中国科学院软件研究所互联网技术实验室，北京 100080)

摘 要：采用一种语言独立 AST 中间格式以便把程序的分析算法同编译器内部表示分离，利用这种格式设计与实现了 Compiler Auxiliary Toolkit(COA)系统框架。COA 可同时应用于静态分析算法和动态分析算法。在介绍了 COA 框架之后，用实例说明了如何利用 COA 框架进行函数调用图生成算法的静态分析和循环不变量的动态分析。

关键词：静态分析；动态分析；中间表示

Retargetable Program Analysis System Based on Intermediate Representation

ZHENG Yaohai, CHEN Wei, ZHAO Chen
(Lab for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing 100080)

【**Abstract**】This paper presents a compiler auxiliary toolkit (COA) using analysis-specific IR. It separates the analysis implementation from development of compiler infrastructure. This paper illustrates its capability through its usage in generation of function call graph and loop invariant detection.

【**Key words**】Static analysis; Dynamic analysis; Intermediate representation

1 概述

随着程序设计技术的发展，程序分析在软件工具和软件工程中得到逐步应用与推广，对软件开发的智能化和自动化起着重要作用。在当前存在的程序分析技术和程序分析算法中，许多是同编译器内部框架系统耦合的，这种情况下，导致了两方面的不足：

(1)各种程序分析算法在不同编译器内部重复设计和实现。虽然对编译内部框架结构表示进行了大量研究[1]，但没有一种体系框架被广泛应用；另一方面，针对不同的程序分析技术的研究，需要采取其最适合与有效的中间格式。这种情况下，为了实现程序分析和转换算法，就需要深入相应语言的编译器框架实现内部，了解编译器使用的中间代码格式等。由于编译器的实现复杂度高，使这一任务时间消耗量大且效果低下，针对某一语言实现的程序分析算法代码难以移植到另一语言进行重用。

(2)编译器框架缺少同时支持静态分析和动态分析的能力。程序分析算法是在编译器框架基础之上利用编译器所获取的信息进行分析的，而编译器的内部框架通过解析程序文本，获取的只是程序的静态信息。这种情况下，进行静态程序分析过程，就需要绕开一些动态信息如变量值，采取近似方法或抽象方法[2]。这些方法获取的信息一般是非精确值，从而阻碍编译优化算法的进一步提高。因此，如果需要得到精确的程序信息，就需要动态运行程序，分析运行结果。另外的一些应用场景，例如程序理解技术、探测循环不变量，都需要一个动态获取程序信息的过程。但是，现有的编译体系框架引起动态信息获取需要重新实现，系统组件重复开发而效率低下。

针对以上提到的两个问题，一个行之有效的解决方案是在语言的编译前端和后端分析算法之间引入中间抽象表示，以使程序分析算法同编译器内部表示分离开来，达到两个目的：(1)分析算法语言独立性；(2)平台同时支持静态分析和动态分析技术。根据这一思想，本文设计并实现了编译优化辅助工具 COA(Compiler Optimization Auxiliary)，其系统总体框架如图 1 所示。

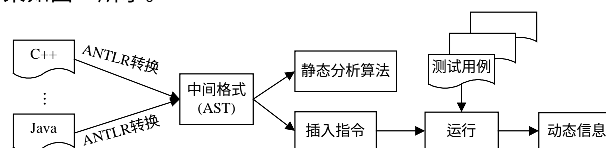

图 1 COA 程序分析总体框架

COA平台提供两方面的程序信息：(1)静态信息。用于刻画程序的控制流；以便编译器能够删除任何无用的生成代码，并优化操作。(2)动态信息。包括循环不变量的探测，用于进行循环优化；程序实际执行路径，用于确定程序执行的流程以及解决测试用例的路径覆盖问题等。COA是一个语言独立的可复用的程序分析系统。它的实现独立于各语言的编译器，不依赖具体的编译器框架。因此，利用COA系统，可以独立于语言进行通用的静态分析算法或者动态分析算法，也可在此基础上进行协同的程序分析技术[3]。

2 系统框架的设计

本节解析 COA 系统设计是如何达到上文所提到的两个

基金项目：国家“863”计划基金资助项目“基于国产操作系统的领域应用支持平台的研制及示范应用”(2004AA1Z2100)
作者简介：郑瑶海(1980－)，男，硕士生，主研方向：编译相关技术，自动测试系统；陈 伟，博士生；赵 琛，研究员、博导
收稿日期：2006-06-14 **E-mail**：zhengyaohai@itehcs.iscas.ac.cn

目的，包括中间抽象层 AST 的设计和在 AST 基础构建的完整系统。

2.1 中间抽象层设计

COA 采用了同抽象语法树相关的 AST(Abstract Systax Tree)结构作为中间格式代码表示。AST 的设计是面向程序分析算法的，也是同一般的编译器中间所不同的地方。AST 接口如下所示。

```
struct    filePosition { int line, int column };
class AST    {
public:
/**  返回节点的不同类型  */
int getNodeType();
/**  相应 AST 树上的节点  */
AST getFirstChildNode();
AST getParentNode();
AST getLeftSibling();
AST getRightSibling();
/**中间格式 AST 和文本间的映射  */
AST fileASTNode(string pathname);
      AST findASTNode(   FilePosition start,
  FilePosition end);
String getASTText();
FilePosition    getASTBegin();
FilePosition    getASTEnd();
/**  其它接口部分  */
  ... ...
};
```

AST 类的中间格式保留了源程序的代码结构，可以清晰描述程序的流程，有利于进行控制流分析，而且在 AST 基础上进行的中间格式与源程序文本间的映射也较直接。在此基础上，增加同程序分析算法支持相关的接口函数，包括静态分析接口和动态分析接口。因此，它的设计符合了前面提到的两点目标。

下面对 AST 的节点类型体系、控制流支持以及代码插桩支持等 3 个方面分别进行介绍。

(1)节点类型体系。程序的分析算法是通过判断 AST 节点类型来进行流程操作的，通过当前的节点类型来判断如何解析相应的子节点，反映程序的控制结构。在 AST 中是通过 getNodeType 接口返回节点的类型。如图 2，当读到 AssignStmt 类型(赋值语句)的节点时，就可以相应地判断子节点中的 lhs 和 rhs 部分。

(2)控制流支持。让中间格式能够接受尽可能多的语言的语法树表示，这要求一个 AST 节点能够包含任意数目的子节点。而子节点间通过链表衔接可达到这一目的。AST 中间表示提供了同遍历相关的 4 个节点：A 为父节点，B 为第 1 个子节点，C 为前一个兄弟节点，D 为后一个兄弟节点。A，B 形成的双向链表用于纵向遍历。C，D 形成的双向链表用于横向遍历。这样的结构可以保证包含任意数目的子节点，同时也提供了逆向遍历的操作，为实现控制流的遍历提供支持。

(3)代码插桩支持。AST 结构需要在程序文本和中间格式间进行直接对应，支持动态分析需要的代码插桩。为了达到这一目的，函数 findASTNode 和 fileASTNode 实现从文件名或者选择部分文本到 AST 节点的映射。函数 getASTBegin 和 getASTEnd 分别得到 AST 到文本中的位置，getASTText 获取当前节点下的文本信息。这样，这些函数确保 AST 可以达到最细粒度符号(token)一级的代码插桩。

图 2 是示例程序 TestFunc.c 经过 COA 解析后得到的 AST 语法树结构，图中省略了与说明无关的节点。AST 树由各种不同的 AST 节点构成，在每个节点旁边标出了相对应的类型。每个节点对应了语法解析过程的非终结节点(Non-Terminal)或终结节点(Terminal)。图中，Function 和 AssignStmt 就是两种不同类型的节点。每个节点可以根据自身的节点类型，利用相应的语义来解析子节点。这里为了说明，把根节点分别指向全局变量和函数。实际中，这些节点以链表的形式连接。

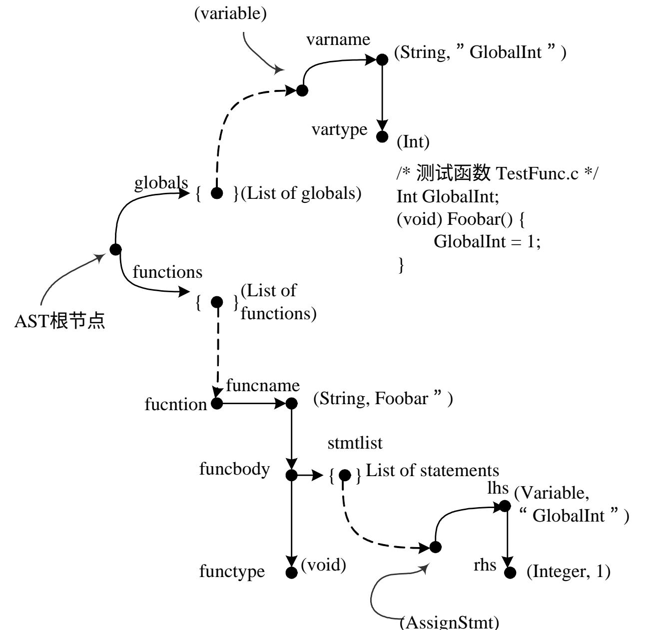

图 2 AST 中间结构表示的程序文本

2.2 框架体系结构

系统其它部分的设计与实现是围绕着 AST 中间抽象层表示进行的，包括从前端语言文本转换成中间 AST 格式，以及后端的进行程序分析算法的接口体系。

2.2.1 语言解析与转换

源语言的解析和转换是动静态分析技术都需要实现的部分，包括语言的预处理、词法分析、语法解析、符号表和类型系统等编译器前端模块[5]。这些模块同时也为动静态的分析技术提供了信息支持，如符号表内容、变量作用域等。我们采用ANTLR工具[4]来实现COA的前端。ANTLR把词法分析、语法分析等功能集成在同一个工具内部，并可在解析语法的同时指导中间格式代码以及语法树的生成。目前，已存在各种语言的ANTLR解析，如C、C++、Java、Python等。ANTLR可以将使用这些语言编写的程序快速转换成 2.1 节所定义的统一中间格式——抽象语法树(AST)，这样便可以在AST的基础上实现各种语言独立的程序分析算法。采用这种方法，开发人员可主要关注中间格式的设计和后端具体分析算法的实现，从而提高开发的效率。

2.2.2 后端接口体系

基于 AST 格式的后端系统包括 3 个不同功能部分：分析管理器(包括静态分析管理器和动态分析管理器)，AST 接口以及分析结果部分。系统设计的基本思想是采用接口的方式以到达底层的语言细节对程序分析算法隐藏的目标。程序分析算法只需要调用 AST 相应的接口，确保了语言独立的目

的；同时，客户端需要的分析结果保存在 Analysis Results 接口中，这样客户便无需关心分析算法的具体流程。

后端的接口关系如图 3，图中中间格式 AST 定义了同静态分析和动态分析相对应的接口，并在内部实现这两个接口。静态分析接口关注于控制流分析算法的产生和应用；而动态分析接口主要提供用于指令插桩(Instrumentation)方面的支持。后端的算法利用接口提供的功能进行程序分析。

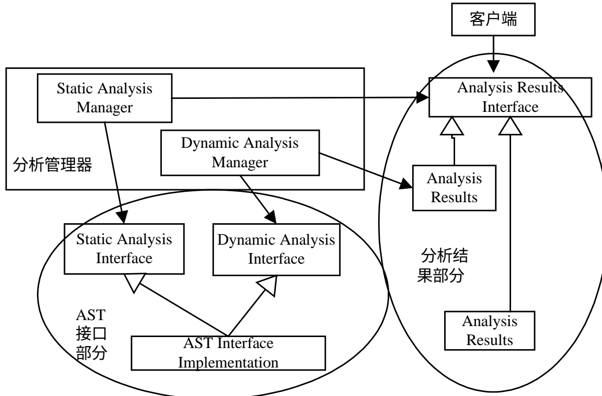

图 3 系统的接口关系

3 COA 中的静态分析

下面以函数调用图的生成算法来说明 COA 系统解决静态程序分析算法的功能。函数调用图的生成算法是典型的静态程序分析问题。它把一个程序中的各种函数调用关系按有向图的方式显示出来，以帮助用户对程序的理解。图 4 是函数调用在 AST 树中的表示。

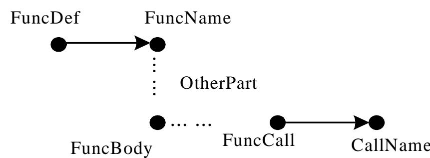

图 4 函数调用语句结构表示

函数调用图构造的程序分析算法是属于过程间的控制流分析[5]，需要处理相应的上下文程序信息。图 4 中，函数定义(FuncDef节点)和实际的函数调用(FuncCall节点)离散地分布在AST树不同部分的节点上，但FuncCall一般是包含在某个函数定义体(FcunDef)的内部。利用这一特点，在构造的过程中，首先从根节点开始，如果找到函数定义节点，则设置定义的函数为调用者(caller)，然后利用AST提供的遍历函数来搜索函数定义体，寻找函数调用点(callee)，找到后加入全局函数调用图中。算法的主体部分伪代码如下。由于AST提供的功能，使本来复杂的函数调用构造流程比较清晰。图的构造过程StoreEdgeToCallGraph非本文重点，在此省略。

算法 函数调用图构造

```
/** 输入程序根节点为参数 */
Procedure: BuildCallGraph(AST root)
/* 搜索函数定义节点，currentNode 表示当前节点 */
if currentNode type is FuncDef
set caller to funcName
  /* 遍历查找函数体内的调用语句(FuncCall) */
  Traversefunctionbody(callername)
Prodedure: Traversefunctionbody(String caller)
forall ASTNode.type is FucnCall
set callee funcCallName
/*把调用者和被被调用者作为一条边加入调用图中*/
StoreEdgeToCallGraph(caller, callee);
```

4 COA 中的动态分析

COA 的中间格式 AST 实现了中间格式和源程序文本间的一一映射，因此接近源代码格式，所以有利于实现源代码级的指令插桩。下面将说明如何利用指令插桩技术来进行循环不变量的探测分析。循环不变量的分析是典型动态分析问题，它是指在循环执行过程中，确定哪些关系量保持不变。这里只对如下 2 种不变量进行探测。

(1)常量值：x = a 说明变量 x 是一个常量值；x= unin 说明变量 x 没有被初始化；变量的集合值 $x \in \{a, b, c\}$；

(2)区间值：变量 a 处于某个区间，x>=a，x<=b，或 a<=x<=b 等。

为了探测这些循环不变量，相应的动态分析的过程分为如下 4 步：

(1)寻找指令插桩点。指令的插桩是基于语法结构的分析。因此，首先通过 AST 查找循环 ForStmt 语句(见图 5)，通过 ForStmt 的结构解析确定到指令插桩点。同时，在语法的解析过程中，利用 COA 提供的静态信息获取哪些变量需要进行循环不变量的探测。

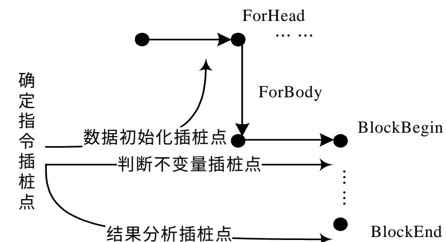

图 5 动态指令插入点确定

(2)指令的插入。对 ForSmst 语句来说，在循环前需插入初始化指令，以初始化循环不变量关系与设置先决条件(pre-condition)。循环体中插入的是每次都需要执行的代码，它主要对上文提及的各种循环不变量关系进行数值计算与检测，看是否保持某种不变量的关系。在循环末尾插入对结果数据进行分析统计的代码，输出结果。

(3)产生测试用例。COA 中主要采用随机测试技术或者用户指定的方式产生测试用例。

(4)动态运行。编译运行插桩后的代码，在运行过程中收集所需要的信息并进行动态的分析，从而计算出循环不变量。

利用循环不变量检测得到的结果，可以进行一些相应的循环编译优化。除了用于循环不变量检测之外，COA 还支持测试技术中路径分析、def-use 的动态分析过程等。

5 总结

编译器系统复杂度高，并且不能同时支持静态程序分析和动态程序分析。从而在此基础上实现的程序分析工具一般都是针对特定语言和特定的程序分析算法，难以移植，需要重复开发系统组件而导致效率低下。本文通过在编译器前端的语言解析和后端的程序分析算法之间引入中间抽象层表示，从而提供了一个语言独立的面向程序分析技术的可复用系统 COA，解决了上述问题。实例证明了系统设计思想的可行性。利用 COA 系统，开发者可以关注于后端具体程序分析算法的设计与使用，减少了分析算法和系统的重复实现，有效地提高了开发效率。 (下转第 71 页)