# MultiFileCallGraph调研报告

组员：张艺潇、王宇飞、赵宏祝

## 一、CallGraph理论调研

CallGraph（函数调用）是控制流图的一种，用来反映计算机程序中的函数或过程间的调用关系。在CallGraph中，每个结点代表一个函数或过程，一条从结点f到结点g的边表示在f中调用了g，而图中如果出现环路则表示程序中存在递归调用。

CallGraph可以动态地或者静态地生成。动态生成的CallGraph是程序的一次执行过程的记录，它可以精确地反映在程序的该次执行中的函数调用关系。而静态生成的CallGraph是从代码出发，试图对程序的所有可能执行过程进行模拟，因此代码中出现的所有调用都会被反映在生成的CallGraph中。要得到精确的静态CallGraph是一个不可判定问题，因此现有的算法通常只是做到一种近似（过近似）。

根据应用的不同需求，CallGraph可以有不同的精度。显然，精度越高，那么计算CallGraph的时间也越长，占用空间也越多。精度较高的CallGraph通常是语境敏感的，意思是说，对于每一个函数，它可能在活动记录栈处于不同状态时被调用，对于每个这样的状态，在图中都有一个结点与之对应。完全语境敏感的CallGraph被称为calling context tree，它可以容易地被动态生成。而精度较低的CallGraph是语境不敏感的，它不分析调用一个函数时的语境，或者说不区分在程序处于不同状态时对该函数的调用。在图中，每个函数只有唯一的结点与它对应。

CallGraph可以为一些应用提供有用的信息。例如，它可以帮助找到那些永远不会被调用的函数，检测程序执行中的异常情况，或检测代码注入攻击等。

## 二、需使用的llvm库的介绍

### 1 LLVM PASS

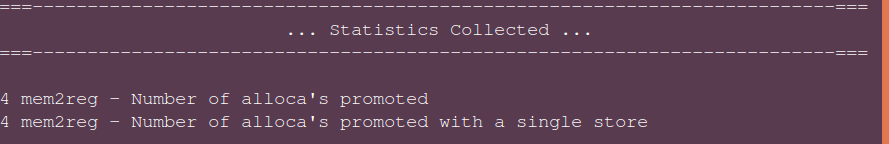
#### 1.1 什么是PASS

在LLVM编译器结构中，IR一层是大部分机器无关优化进行的地方。在IR上进行转换或优化的工具被称为pass。LLVM提供了一些pass，可以进行编译时的优化或者链接时的优化，它们可以由opt命令调用。

pass 举例：mem2reg是一个重要的pass，它实现的优化功能是将一些“内存变量”提升为“寄存器变量”。例如，借助于phi指令并运用一种标准的SSA构造算法，可以将对内存变量的访问改为对寄存器变量的访问，并生成符合SSA标准的代码。要用它对源IR代码进行优化时，可以使用如下指令

$ opt xxx.bc -mem2reg -o xxx.bc

如果加上打印信息的参数，还可以输出被提升的变量个数。如下图：



关于完整的LLVM提供的pass信息，可以参见http://llvm.org/docs/Passes.html。

pass分为两种类型，analysis pass和transform pass。analysis pass不改变程序，它们分析程序的特征，提取可以进行优化的点，并为transform pass提供信息与一些必要的数据结构，而transform pass通常依赖于一些analysis pass，并会以某种方式更改源程序。analysis pass和transform pass间的依赖关系可以更正式地表示如下：

（1）显式依赖：某个transform pass要求在运行之前先运行某个analysis pass。例如，为了使得SSA生成算法能够确定安放phi指令的位置，需要依赖 dominator tree（支配树）数据结构。因此，mem2reg pass 依赖于 domtree pass，并在它的实现代码中显式地调用 domtree pass。

（2）隐含依赖：通常IR上有多种方式表达相同的语义，但一些pass可能假定它们所分析的IR具有一些特定的特征，这些特征可以帮助它们识别所想要的模式。如果在IR中没有发现这类特征，这些pass就会直接跳过，不执行所期望的操作。而这些特征也许需要在IR上运行过某个transform pass后才能得到，这样就会产生隐含的依赖关系。

可以通过一个pass manager来管理各pass间的依赖关系。pass manager能够根据pass间的依赖关系，自动安排各个pass的执行顺序。

LLVM 提供了一个Pass类作为各种优化的实现机制，但大部分情况下并不直接使用该Pass类，而是根据具体的优化问题，选择Pass的合适的子类。例如：

（1）ModulePass：这是最一般的pass。通过这个子类，可以对整个模块进行分析，对模块内容做修改如删除函数等。

（2）FunctionPass：这一子类允许一次分析一个函数，但不保证函数的分析次序。它不允许对外部函数做修改，也不允许函数和全局变量的删除。

（3）BasicBlockPass：这一子类以一个基本块为粒度进行分析，它不允许对外部基本块的修改和删除。

此外，还有LoopPass、CallGraphSCCPass等。它们的分析粒度如名字所示。完整的Pass子类定义见http://llvm.org/docs/WritingAnLLVMPass.html。

#### 1.2如何写一个自己的PASS

LLVM提供了方便地接口让我们按照需求实现自己的pass。本实验中的CallGraph程序构造函数调用图就是以pass的形式实现的。

第一步：编写自己的pass，首先要根据待解决的问题选择具有合适粒度的pass子类。例如，在进行函数调用图的构造时，需要遍历模块中的函数并深入分析函数内出现的所有call指令，因此选择ModulePass是合适的。

第二步：编写一个类继承自该pass子类，在类中定义一个变量ID。LLVM通过ID来标识每个pass。当定义一个该类pass的对象时，ID将在构造函数中被设置一个全局唯一的值。示例如下：

class MyPass : public FunctionPass {

public:

static char ID;

MyPass() : FunctionPass(ID) {};

virtual bool runOnFunction(Function &F);

}

第三步：在各个pass子类中，有相应的entry point方法，即在运行该pass时由pass manager首先调用的方法。例如，在ModulePass类中的entry point为runOnModule()方法，在FunctionPass类中的entry point为runOnFunction()方法，等等。这些方法被定义为虚方法。在编写自己的pass时，需要重新定义来覆盖这些entry point方法，并在这些方法内部实现pass真正的功能。这些方法返回一个布尔量，以标识当前的pass是否对所分析的对象（module,function,basicblock等）进行了修改。

bool runOnFunction(){

// here put the code where the pass does its work

...

//return true if the pass modifies its object, false otherwise

}

#### 2 Alias Analysis and Mod/Ref Information

##### 2.2.1 Alias Analysis

Alias Analysis也被称为Pointer Analysis，用于分析两个指针是否可能指向内存中的同一个对象。它对代码进行静态分析，对程序的数据结构进行构建。Alias analysis可以回答以下形式的询问：

alias(p1, s1, p2, s2)

其中P1、P2为程序中的指针，S1、S2为整形常量，代表以字节为单位的对应指针指向区域的大小。

回答为下列三种形式之一：

MustAlias：P1和P2等同；

NoAlias：P1与P2指向的空间没有交集；

MayAlias：无法判断。

有了以上信息，编译器就可以准确地对代码进行更激进的优化。比如当对两块没有交集的内存区域进行memmove操作时，编译器可以将memmove优化为memcpy。

例如如下两个例子：

(1) %1=\*P1

\*P2=%2

%3=\*P1

如果alias analysis能够得出P1和P2为NoAlias的结论，编译器就可以将“t3=\*P1”优化为“%3=%1”，从而减少一次load操作。

(2) do {

t1=\*P1

…

\*P2=t2

}

同样，如果alias analysis能够得出P1和P2为NoAlias的结论，编译器就可以将“t1=\*P1”提升到循环外部。

LLVM Infrastructure中提供了Alias Analysis的工具，我们可以方便的调用这些分析结果，编写自己的Pass来实现各种功能和优化。

##### 2.2.2 Mod/Ref Information

Mod/ref analysis 也是属于静态分析。它可以对形式为modref(I, P, S)的询问给出回答。其中I是一条基本操作，P是一个指针，S是一个整形常量，代表区域的大小。回答有四种形式：

NoModRef: I没有接触过P指向的空间；

Ref: I可能读取过P指向空间的内容，但是未修改过；

Mod: I可能修改过P指向空间的内容，但是未读取过；

ModRef: I可能修改和读取过P指向空间的内容。

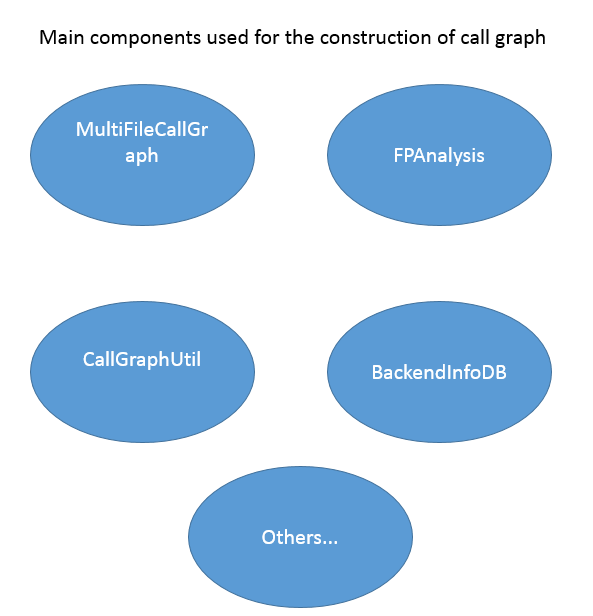
类似的，Mod/ref analysis也能给出所询问的两个基本指令之间的依赖关系。询问的形式为modref(I1, I2)，给出的回答与上面一种基本形式类似。

## 三、CallGraph程序

最初版本的CallGraph程序是由张维（2012级硕士研究生）在LLVM3.3上开发的跨文件的C语言函数调用图构造器。

该程序接受的输入是若干C语言源文件，并需要指定其中的一个的文件与该文件中的一个函数名；输出是从该函数出发能到达的所有函数及它们之间的调用关系，结果以图的形式呈现。

整个程序的结构大体是这样的：主要工作由两个pass完成，第一个pass名为MultiFileCallGraph，第二个pass名为FPAnalysis，它们都继承自ModulePass类。这两个pass的具体实现将在下文介绍。此外，还定义了一个CallGraphUtil类，其中定义了许多辅助的数据结构和函数。main函数负责整个程序的控制逻辑、用户交互与错误处理等任务，并调用这两个pass以实现它们的功能。



下面分几部分介绍该程序是如何实现的。

### 1、构建数据库

为了实现跨文件的函数调用图构造，需要维护三张表。这三张表存储在后端数据库BackendCollectedInfoDB类中，分别是FunctionList、CallInstList和FunctionPointerCall。

FunctionList：记录模块与函数的对应关系。其中的一个条目记录一个模块及其中定义或声明的所有函数。在这里，一个模块对应一个LLVM IR文件。

CallInstList：记录函数与函数调用指令的对应关系。其中的一个条目记录一个函数以及该函数体中出现的所有call指令。

FunctionPointerCall：记录函数指针与通过它所调用的函数的对应关系。由于在静态分析时有时难以确定通过一个函数指针究竟调用了哪个函数，因此为保守起见，将该函数指针可能指向的所有函数都记录下来。在这张表中，一个条目记录一个函数指针，以及一个该指针可能指向的函数的列表。

在进行函数调用的分析之前，先要建立前两张表，这项工作是在第一个pass中实现的。第三张表的建立在第二个pass进行指针分析时完成。

第一个pass名为MultiFileCallGraph，它在main函数中被显式调用，并且在每个待分析的模块上都运行一遍。该pass的算法也很简单，主体部分就是一个二重循环。外层循环中，遍历模块中的所有函数，并将它们添加到数据库的FunctionList表中。内层循环中，遍历一个函数的函数体并寻找其中出现的所有函数调用指令，将它们添加到数据库的CallInstList表中。

### 2、进行跨文件的函数调用分析

#### 2.1、如何做到跨文件的分析

完成了数据库的构建后，就可以开始进行分析了。为了实现跨文件分析的功能，首先要把所有的分析模块合并成一个模块。具体做法如下：

（1）在CallGraphUtil类中，定义两个数据结构，一个是集合类型的变量RML(Related Module List)，用来存储分析时涉及到的所有模块。另一个是llvm::Module类型的对象LinkedModule，用作合并后的总模块。

（2）首先，要找到所有需要合并的模块。由于程序的输入中给定了分析的起始函数，因此可以采用一个类似深度优先遍历的算法：先将该起始函数所在的模块加入集合RML中，然后查询数据库，得到该起始函数调用的所有函数，对每个这样的函数进行深度优先遍历，同时将这些函数所在的模块也加入集合RML。

（3）找到待连接的所有模块后，使用LLVM提供的Linker类，将这些模块依次连接到LinkedModule中。

（4）将所有模块连接完成后，就可以像分析单个模块那样分析其中的函数调用关系，这通过在总模块LinkedModule上运行第二个pass FPAnalysis来实现。

#### 2.2、指针分析：

（1）首先，为了方便的进行函数指针调用情况的分析，需要将所有的模块连接成成一个总模块，这在上一部分已经说明。

（2）然后则可以在合并好的总模块上运行第二个pass：FPAnalysis。这个pass会调用DSA分析模块，分析每个函数指针可能调用的函数的位置，并将这些可能会调用的位置保存在DSA分析模块的一个子类型DSCallGraph的一张表中。

（3）最后只需要从DSCallGraph中提取出每个函数指针对应的位置，在后端数据库中重新构建一个名为FunctionPointerCall的map保存每个函数指针对应的可能调用的函数位置即可。这个数据结构在构建函数调用图的时候可以较为方便地使用。

这里，有一点需要特别指出：由于在静态分析的时候构建函数调用图是一个不可判定问题，因此DSA（分析）返回的是每个函数指针对应的可能会调用的位置的表，在上文中也有提及这一问题。不仅如此，DSA分析可能还可能会出现失败的情况，在进行函数调用图构建的时候也需要注意这一点。

## 四、我们的工作：如何支持C++函数调用分析、如何实现针对某类函数的统计功能

C++由于具有面向对象的特性，在继承的时候会引起各种多态，这将导致静态分析的时候难以唯一确定某个实例方法实际在运行时调用的是哪个方法；除此之外，函数指针机制允许通过指针来完成过程调用，这种机制导致静态分析的时候无法唯一确定函数指针所指向的函数对象，这些都是在静态分析的时候构建C++函数调用图的难点。

但好消息是，目前函数调用图的构建是基于LLVM IR的，在IR层面上已经没有了面向对象的机制，函数调用的时候也没有多态的特性，每一个显式调用的函数都是可以唯一确定的，这样就解决了第一个问题；函数指针的处理可以参见上文Alias Analysis and Mod/Ref Information部分以及参考文献中给出的分析机制，DSA部分已经给出了关于指针分析的一个完整实现。因此，基于目前构建函数调用图的方案构建C++的函数调用图的主要工作，就是找到IR中函数名称同被分析源代码中函数名称的对应关系，这涉及到C++函数的name mangling机制。

name mangling是C++（C）编译器在编译时对函数名称的一种转化机制。具体而言，函数名称的转化机制大致如下：

类中的成员函数名的转换方式为：\_ZN/\_ZZ + (namespacelength + namespace)\* + methodnamelength + methodname + (C/D + number)? + E + v/i/c...

\_ZN是前缀，'N'用于成员函数，'\_ZZ'用于声明为const的函数；

namespacelength是函数所在命名空间的名字长度；

namespace是函数所在命名空间的名字；

namespacelength和namespace可以有多层；

methodnamelength是方法名长度；

methodname是方法名；

C/D + number C表示构造函数，D表示析构函数，number表示不同类型的构造或析构函数，如C1表示complete object constructor，C2表示base object constructor等；

E后面的符号表示函数的参数类型，如v表示void，ic表示第一个参数为int、第二个参数为char，等等。

举例而言的话，\_ZN1A6PrintAEv对应的函数名称就是A::PrintA()。这样，只要找到IR中当前调用函数的名称，根据翻译规则将当前的函数名称还原，就可以在构建函数调用图的时候清晰地显示出函数调用的关系。一般地，如果想要方便地根据IR中的函数名称还原原函数名，可以使用c++filt工具；在构建函数调用图的时候，则可以使用<cxxabi.h>所提供的函数\_\_cxa\_demangle，这个函数接受一个指向IR中函数名称的字符型指针，并返回一个指向翻译后函数名称的指针，这样就可以方便地完成函数名称的还原，省去了手工编写翻译函数的麻烦。

剩下的部分主要就是一些工程实现上的问题。在原来函数调用图的实现中，所有的函数名都用手工编写的翻译函数统一转化成标准的函数名称再进行比较和分析，但是在构建c++函数调用图的时候会遇到一些麻烦：因为类似\_ZN2B1C2Ev和\_ZN2B1C1Ev翻译得到的原函数名都是B1::B1()，但是比较的时候显然不能把他们当成同一个函数；另外，在生成graphviz结点的时候，翻译得到的函数名B1::B1()也不能够当作结点名称（结点名称不能含有括号）。这里，比较好的方式是在比较函数和生成结点的时候都采用IR中的函数名，只有生成结点标签信息的时候才使用原函数名，因此需要对原来的实现方案做相应的调整。在指定分析的根节点函数时，由于用户输入的是原函数名，因此也需要将这个函数名转化成IR中的函数名，再调用分析函数。

实现函数统计则相对要更为简单。由于在IR中，每一个函数所调用的函数经过一遍扫描都是可以唯一确定的，因此统计函数调用次数时只需要在分析的时候，每次遇到指定函数的调用的时候将计数变量加一，最后输出总的计数次数即可；在原来的实现中getCountofCallInst已经实现了一个类似的机制。这里有一个问题，如果遇到递归调用的情况，没有办法实现精确的统计，这一点在静态分析的时候也确实很难做到；在原来的实现中，也只是简单地将这种情况标注为有递归，而没有去具体地进行更细致的分析。

## 五、总结与展望

在本次调研实验中，我们学习了LLVM代码优化的机制，知道了怎么通过pass在IR层面做分析与优化，并且了解了指针分析的原理。通过对现有的跨文件C语言函数调用图构造器的调研，我们清楚了大致的流程和其中的算法，感觉还是学到了很多东西的～

以下是我们在调研与实现改进的过程中的一些感悟与思考：

（1）根据需求选取合适的工具很重要。在本实验中，只需要知道函数之间的可能的调用关系就可以了，因此选择在IR层面进行语境不敏感分析是比较合适的，而如果选择在AST一层做静态分析时构造函数调用图，无疑会大大增加工程的复杂度。

（2） 现有的程序的可扩展性不好，例如在进行函数调用分析前对源程序的处理是写死的，而基于IR的分析完全可以方便地适配不同的前端；而且最后的结果输出是与分析过程混在一起的，这导致扩展该程序的功能不是很方便，如果先构建分析树再进行输出，那么就可以很方便地实现函数调用图的剪枝功能（只输出含有指定函数的分支）。如果进行重构，将各个功能独立的部分分开，可以方便后续的改进。

（3）我们现在遇到的一个问题是，当在头文件中include某个C++库文件时，程序找不到相应的库文件。这不是函数调用分析部分的问题，而是预处理器的问题，因为原来的程序进行预处理时没有考虑支持C++。因此现有的对C++函数的扩展还是有局限的。要解决这个问题需要了解一下LLVM预处理器的接口，需要一定的工作量……

## 参考文献

[1] Lattner C A. Macroscopic Data Structure Analysis and Optimization[J]. 2005.

[2] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world[J]. Acm Sigplan Notices, 2007, 42(6):278-289.

[3] Hammer C, Snelting G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs[J]. International Journal of Information Security, 2009, 8(6):399-422.

[4] Lopes B C, Auler R. Getting Started with LLVM Core Libraries[M]. Packt Publishing, 2014.

[5] 宁宇, 张昱. LLVM 3.9 介绍[R]. 合肥:中国科学技术大学, 2016.

[6] LLVM Project. LLVM Alias Analysis Infrastructure[EB/OL]. llvm.org/docs/AliasAnalysis.html, 2016.

[7] LLVM Project. LLVM’s Analysis and Transform Passes[EB/OL]. llvm.org/docs/Passes.html, 2016.

[8] LLVM Project. Writing an LLVM Pass[EB/OL]. http://llvm.org/docs/WritingAnLLVMPass.html, 2016.

[9] Wikipedia. Name mangling[EB/OL]. https://en.wikipedia.org/wiki/Name\_mangling, 2016.

[10] Wikipedia. Call graph[EB/OL]. https://en.wikipedia.org/wiki/Call\_graph, 2016.