Advanced Topics in Compilers

Call Graph

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Call graph (summary from 323)

• Call graph in NOELLE

• Other abstractions generated from call graph in NOELLE
Call graph

• First problem: how do we know what procedures are called from where?
  • Especially difficult in higher-order languages, languages where functions are values
  • What about C programs?
  • We’ll ignore this for now

• Let’s assume we have a (static) call graph
  • Indicates which procedures can call which other procedures, and from which program points

```c
void foo (int a, int (*p_to_f)(int v)){
  int l = (*p_to_f)(5);
  a = l + 1;
  return a;
}
```
Call graph example

```plaintext
f() {
    1:   g();
    2:   g();
    3:   h();
}

g() {
    4:   h();
}

h() {
    5:   f();
    6:   i();
}

i() { ... }
```

From now on we assume we have a static call graph
Using CallGraphWrappingPass

• Declaring your pass dependence

```cpp
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired< CallGraphWrapperPass >();
}
```

• Fetching the call graph

```cpp
bool runOnModule(Module &M) override {
    errs() << "Module " << M.getName() << "\n";
    CallGraph &CG = getAnalysis<CallGraphWrapperPass>().getCallGraph();
```
Using CallGraphWrappingPass

• From a Function to a node of the call graph

```cpp
errs() << " Function " << F.getName() << "\n";
CallGraphNode *n = CG[&F];
```

• From node to callees

```cpp
for (auto callee : *n){
    auto calleeNode = callee.second;
    auto callInst = callee.first;
}
```

• From node to Function

```cpp
auto calleeF = calleeNode->getFunction();
errs() << " \"" << calleeF->getName() << "\n";
```
Call graph

• how do we know what procedures are called from where?
  • Especially difficult in higher-order languages, languages where functions are values
  • What about C programs?

• Call graph generated by LLVM:
  • If the callee is unknown: no edge is generated
  • If there are N possible callees (N > 1): no edge is generated
  • In other words: the call graph of LLVM is not complete

```c
void foo (int a, int (*p_to_f)(int v)){
  int l = (*p_to_f)(5);
  a = l + 1;
  return a;
}
```
Outline

• Call graph (summary from 323)

• Call graph in NOELLE

• Other abstractions generated from call graph in NOELLE
Call graph in NOELLE

• Called “Program Call Graph (PCG)”

• PCG is complete (and conservative)

• If there are N possible callees (N > 1): there are N outgoing edges

• It is a hierarchical graph
Let’s compute the PCG
Normalize the code

Code must be normalized before you use NOELLE

• noelle-norm MYIR.bc –o IR.bc
  or
• noelle-simplification MYIR.bc –o IR.bc
Fetching the program call graph (PCG)

```cpp
/*
 * Fetch NOELLE
 */
auto& noelle = getAnalysis<Noelle>();

llvm::noelle::Noelle

auto fm = noelle.getFunctionsManager();

llvm::noelle::FunctionsManager *

auto pcf = fm->getProgramCallGraph();

llvm::noelle::CallGraph *
Using the PCG

```cpp
/* * Fetch the next program's function. */
auto f = node->getFunction();
if (f->empty()){
    continue ;
}

for (auto node : pcf->getFunctionNodes()){
    // lines: Fetch the next program's function.
}

/* * Fetch the outgoing edges. */
auto outEdges = node->getOutgoingEdges();
if (outEdges.size() == 0){
    errs() << " The function " << f->getName() << " has no calls\n";
    continue ;
}

errs() << " The function " << f->getName() << "\";
errs() << " invokes the following functions:\n";
for (auto callEdge : outEdges){
    auto calleeNode = callEdge->getCallee();
    auto calleeF = calleeNode->getFunction();
    lines: errs() << " [" ;
```
PCG: from function to node

```cpp
auto mainNode = pcf->getFunctionNode(mainF);
```
Edges in the PCG

• Two type of edges: may and must
  • May: when the related call executes, the destination of the edge might be called
  • Must: when the related call executes, the destination of the edge will always execute

```
if (callEdge->isAMustCall()){
  errs() << "must";
} else {
  errs() << "may";
}
```
PCG of NOELLE is hierarchical

• If a function F invokes G N times, the PCG includes only one edge \( e \) from F to G
  • Source of \( e \): F
  • Destination of \( e \): G

• That edge includes N sub-edges
  • Source of a sub-edge: the specific call instruction of F
  • Destination of all sub-edges: function G
PCG of NOELLE is hierarchical

```cpp
earns() << " The function " << f->getName() << "\n";
earns() << " invokes the following functions:\n";
for (auto calleeEdge : outEdges){
    auto calleeNode = calleeEdge->getCallee();
    auto calleeF = calleeNode->getFunction();
lines. earn() << "\n",--------------------------------------------
}
```
Outline

• Call graph (summary from 323)

• Call graph in NOELLE

• Other abstractions generated from call graph in NOELLE
Islands

• Island: disconnected sub-graph of a graph

• Island in the PCG:
  set of functions that **cannot** reach
  from any other function of another island

```cpp
auto islands = pcf->getIslands();

auto islandOfMain = islands[mainF];

for (auto& F : M){
    auto islandOfF = islands[&F];
    if (islandOfF != islandOfMain){
        errs() << " Function " << F.getName() << " is not in the same island of main\n";
    }
}
```
Strongly Connected Component Call Acyclic Graph (SCCCAG)

```cpp
auto sccCAG = pcf->getSCCCAG();

auto mainNode = pcf->getFunctionNode(mainF);

auto scCAG = sccCAG->getNode(mainNode);
```