Abstract (Last updated July 1, 2019)

Abstract: In this talk, Michael Shah (“Mike”) will be presenting a light introduction to Program Analysis. Terms, definitions, and the general practice of program analysis will be demonstrated. I will be then showing one simple application, for how to obtain a Control Flow Graph using LLVM.

Materials:

- Please bring a laptop with Clang/LLVM 8.0 setup if you want to follow along
- Otherwise materials will be posted to www.mshah.io

Resources:

- Downloading and setting up LLVM (make sure to include Clang!): http://llvm.org/docs/GettingStarted.html#checkout

Contact: mshah.475@gmail.com
Twitter: @MichaelShah

Terminology (Open in a second browser if you like)

- Static Analysis - [wiki]
- Dynamic Analysis - [wiki]
- Control Flow Graph (CFG) - [wiki]
- Call Graph (CG) - [wiki]

Introduction to Program Analysis using LLVM (Tutorial)

Mike Shah, Ph.D.
@MichaelShah | mshah.io

July 1, 2019

30-45 Minutes for talk (plenty of time for questions)

Demo Time! Right from the start!

- So you know what to pay attention to!
  - In case you (or maybe I) walked into the wrong room by accident!
  - (Or if you are deciding to commit to an hour long talk online in the *distant future*)
- For those attending this talk live
  - Take a moment to introduce yourself to someone next to you .

- demo5.sh - Control Flow Graph

Who Am I?

by Mike Shah

- Currently an assistant teaching professor at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- www.mshah.io

This is an introduction to Program Analysis

We have some specific goals

1. An introduction to what ‘program analysis’ is
2. An example of starting a program analysis in LLVM

Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future

Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future
- **As always--**Be able to run through these slides again with confidence and excitement!

Slides and code are at the following location

www.mshah.io

Introduction to Program Analysis

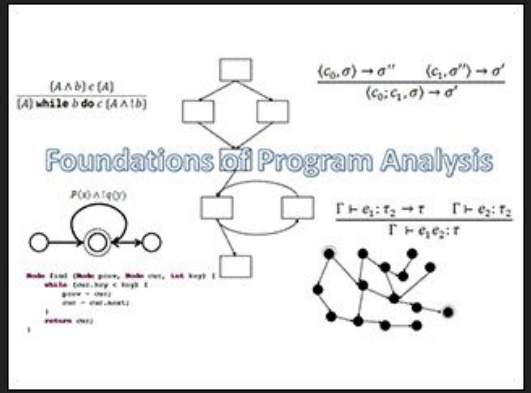


Image Source: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-820-fundamentals-of-program-analysis-fall-2015/6-820f15.jpg

The field of Program Analysis (1/2)

- Simply put--program analysis is the study and practice of building automated tools to understand and measure specific aspects of software.
  - Some categories include:
    - Finding where correctness bugs exist
    - Finding where security loopholes exist
    - Finding performance problems
    - Finding stylistic changes that need to be made to code
    - Potentially visualizing software
      - i.e. using mediums other than text like graphs or charts

The field of Program Analysis (2/2)

- Simply put--program analysis is the study and practice of building automated tools to understand and measure specific aspects of software.
  - Some categories include:
    - Finding where correctness bugs exist
    - Finding where security loopholes exist
    - Finding performance problems
    - Finding stylistic changes that need to be made to code
    - Potentially visualizing software
      - i.e. using mediums other than text like graphs or charts

So what are our tools for figuring out these properties?

Tools for Program Analysis

The two most common tools for program analysis are:

1. Static analysis
2. Dynamic analysis

*There are also hybrid analysis of static and dynamic analysis

Tools for Program Analysis

The two most common tools for program analysis are:

1. Static analysis
2. Dynamic analysis

1. Static Analysis

Static Analysis (1/2)

- Static analysis is looking at a programs *source code* to learn facts about the program.

```
int main(){
    int x;
    cin >> x;

    std::cout << "10 divided by x = " << 10/x << "\n";

}
```

Fact Check: Could this program ever divide by 0?

Static Analysis (2/2)

- Static analysis is looking at a programs *source code* to learn facts about the program.
- This analysis happens before you run a program
  - Notice the phrasing "*could this program*"
  - (Perhaps even before compile-time!)
    - (Though running an analysis on a program with syntax errors is probably not too useful...)

```
int main(){
    int x;
    cin >> x;

    std::cout << "10 divided by x = " << 10/x << "\n";

}
```

Fact Check: Could this program ever divide by 0?

Example #1 Static Analysis in the Compiler

- Dead code elimination
  - We can search the whole program and see this variable is never assigned or re-assigned a value--and thus get rid of it
    - i.e. there is no ‘use’ of a ‘defined’ variable named ‘unusedVariable’.

```
67
68  void blink(uint8_t pin, uint8_t times, uint16_t ms) {
69
70      uint16_t unusedVariable;                    ⚠ Unused variable 'unusedVariable'
71
72      for (uint8_t i=0; i<times; i++) {
73          digitalWrite(pin, HIGH);
74          delay(ms >> 1);
75          digitalWrite(pin, LOW);
76          delay(ms >> 1);
77      }
78  }
79
```

Example #2 Static Analysis in the Compiler

- Compilers parse our code for errors during compilation and can provide some analysis (i.e. using clang analyze).
  - User friendly compilers even output and point to the error they have found!

```
example.cc:31:5: error: Expected a Q_PROPERTY keyword
    Q_PROPERTY(int x READ getX WIRTE setX)
    ^
<scratch space>:116:19: note: expanded from here
"int x READ getX WIRTE setX"
                  ^
example.cc:32:5: warning: READ function 'heigth' not found; did you mean 'height'
    Q_PROPERTY(int height READ heigth)
    ^
<scratch space>:118:18: note: expanded from here
"int height READ heigth"
                 ^
example.cc:68:9: note: 'height' declared here
    int height() { return m_height; }
        ^
1 warning and 1 error generated.
```

Program Correctness and Program Optimization

- The two previous use cases are where we would use program analysis
  a. Find Optimizations
  b. Enforce program correctness

Example #1 Static Analysis in the Compiler

- Dead code elimination
  - We can search the whole program and see this variable is never assigned or re-assigned a value--and thus get rid of it
    - i.e. there is no 'use' of a 'defined' variable named 'unusedVariable'.

```
67
68  void blink(uint8_t pin, uint8_t times, uint16_t ms) {
69
70      uint16_t unusedVariable;                    ⚠ Unused variable 'unusedVariable'
71
72      for (uint8_t i=0; i<times; i++) {
73          digitalWrite(pin, HIGH);
74          delay(ms >> 1);
75          digitalWrite(pin, LOW);
76          delay(ms >> 1);
77      }
78  }
79
```

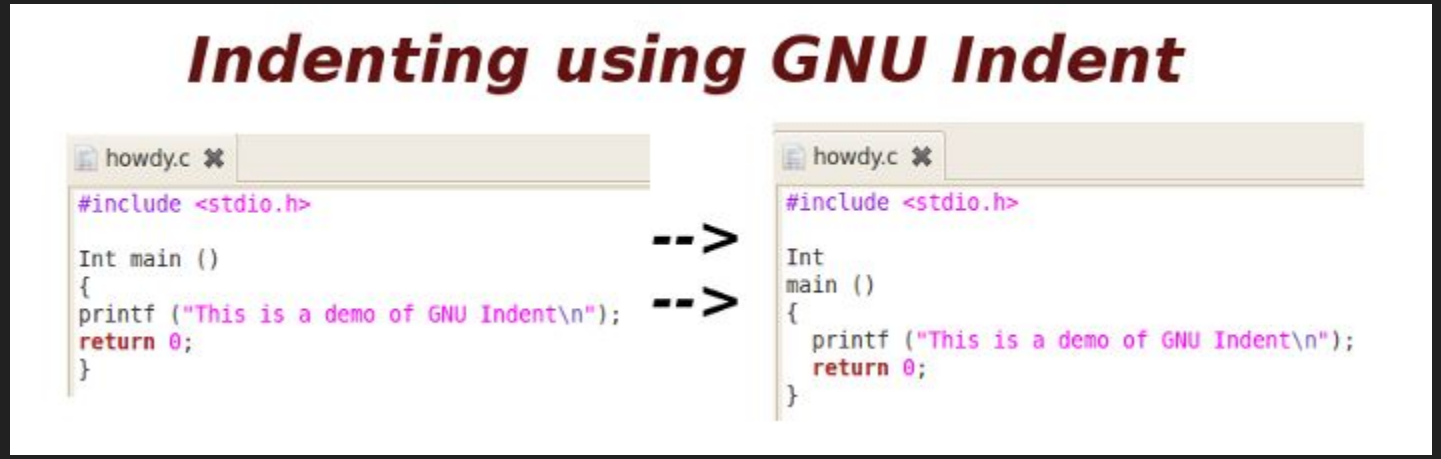
Example #2 Static Analysis in the Compiler

- Compilers parse our code for errors during compilation and can provide some analysis (i.e. using clang analyze).
  - User friendly compilers even output and point to the error they have found!

```
example.cc:31:5: error: Expected a Q_PROPERTY keyword
    Q_PROPERTY(int x READ getX WIRTE setX)
    ^
<scratch space>:116:19: note: expanded from here
"int x READ getX WIRTE setX"
                  ^
example.cc:32:5: warning: READ function 'heigth' not found; did you mean 'height'
    Q_PROPERTY(int height READ heigth)
    ^
<scratch space>:118:18: note: expanded from here
"int height READ heigth"
                 ^
example.cc:68:9: note: 'height' declared here
    int height() { return m_height; }
        ^
1 warning and 1 error generated.
```

Another Static Analysis (sort of) -- Indent tool (1/3)

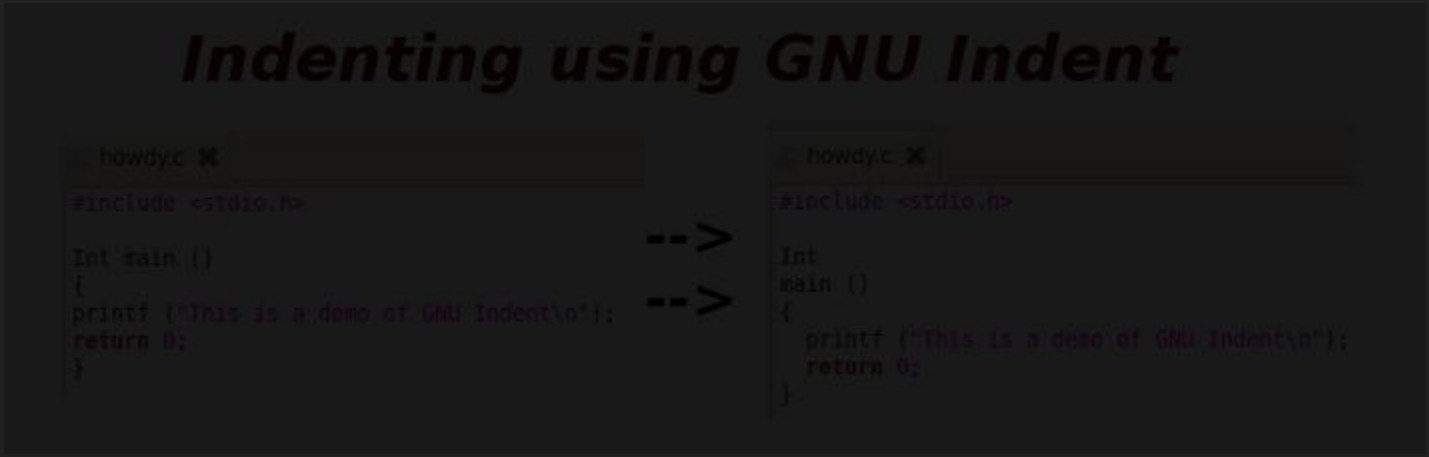
- A tool may looks at how our source code is formatted
  - This happens before run-time, and is an analysis of our code.
- Here's an example with the `indent` tool on unix
  - Code on the left has been analyzed (checking how nested code is)
  - Code on the right appropriately adjusted



Question to the audience: Is this a static analysis?

Another Static Analysis (sort of) -- Indent tool (2/3)

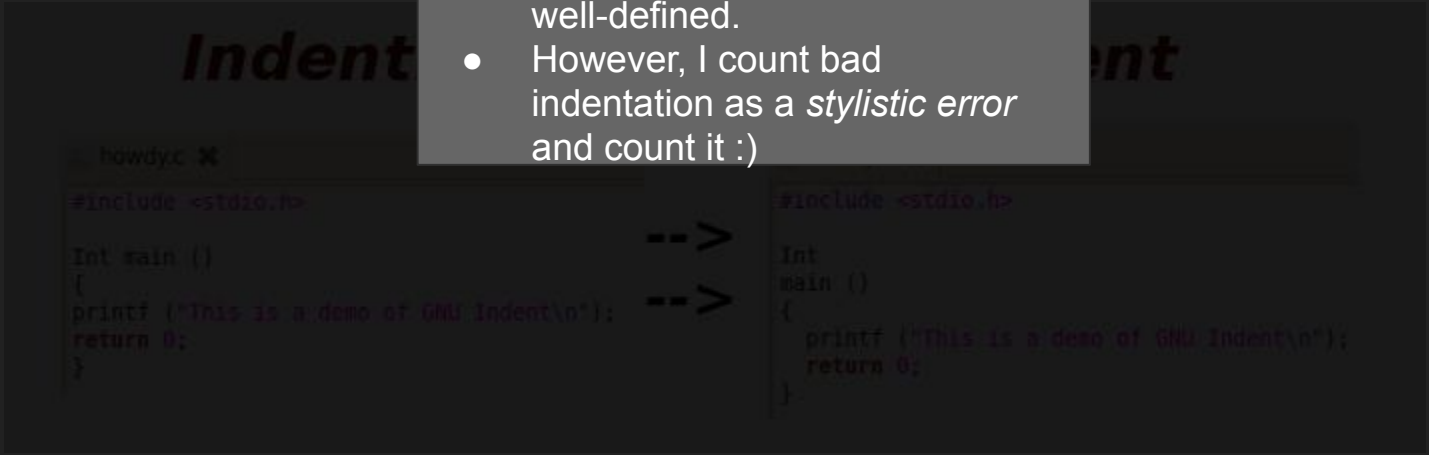
- A tool may looks at how our source code is formatted
  - This happens before run-time, and is an analysis of our code.
- Here's an example with the `indent` tool on unix
  - Code on the left has been analyzed (checking how nested code is)
  - Code on the right appropriately adjusted



Another Static Analysis (sort of) -- Indent tool (3/3)

- A tool may looks at how our so matted
  - This happens before run
- Here's an examp
  - Code o
  - Code on the right appro

- The line is a little gray here on if this is a static analysis.
- Indentation may give us some understanding of developer intent, but is not very well-defined.
- However, I count bad indentation as a *stylistic error* and count it :)



C++ Static Analysis Tools (1/2)

- For languages like C++ (and really any language) static analysis tools have great value
  - A listing of some common ones are found here: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C,_C++

C, C++ [edit]

Tool	Latest release	Free software	Duplicate code	Notes
Astrée		No; Proprietary		finds all potential runtime errors and data races by abstract interpretatio can prove their absence, and can prove functional assertions; tailored towards safety-critical C code (e.g. avionics and automotive). Includes MISRA checker.
Axivion Bauhaus Suite		No; Proprietary		A static code analysis tool suite for Ada, C, C++, C#, and Java code tha performs various analyses such as architecture checking, interface analyses, MISRA checking, and clone detection.
BLAST – (Berkeley Lazy Abstraction Software verification Tool)	2.7.2	Yes		An open-source software model checker for C programs based on lazy abstraction (follow-on project is CPAchecker.[5]).
Clang	8.0.0	Yes		An open-source compiler that includes a static analyzer.
CLion	2019.1	No; Proprietary		An IDE with a built-in source code analysis.
Coccinelle	1.0.7	Yes		An open-source source code pattern matching and transformation.

C++ Static Analysis Tools (2/2)

- For languages like C++ (and really any language) static analysis tools have great value
  - A listing of some common ones are found here: https://en.wi

Alright--on to dynamic analysis!

Axivion Bauhaus Suite		No; Proprietary		performs various analyses such as architecture checking, interface analyses, MISRA checking, and clone detection.
BLAST – (Berkeley Lazy Abstraction Software verification Tool)	2.7.2	Yes		An open-source software model checker for C programs based on lazy abstraction (follow-on project is CPAchecker.[5]).
Clang	8.0.0	Yes		An open-source compiler that includes a static analyzer.
CLion	2019.1	No; Proprietary		An IDE with a built-in source code analysis.
Coccinelle	1.0.7	Yes		An open-source source code pattern matching and transformation.

Tools for Program Analysis

The two most common tools for program analysis are:

1. Static analysis
2. Dynamic analysis

2. Dynamic Analysis

Dynamic Analysis

- Gathering facts about a program that is running
  - Typically this is done by logging and storing information in an internal data structure.

Fact Check: What are all of the values of ‘x’ a user input in this instance of the program

```
int main(){
    int x;

    while(1){
        cin >> x;
        LOG(x); // Some 'special' function to record values of 'x'
        std::cout << "10 divided by x = " << 10/x << "\n";
    }

}
```

Example #1 of Dynamic Analysis - Profilers (man perf)

- Profiling tools with our programs gathering performance information about a program.
  - Typically gathering execution time at different granularities (e.g. function) within the program
  - A log of the linux ‘perf’ tool is shown below.

Entries Sorted By Time Spent

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Call Tree

Example #2 of Dynamic Analysis - Valgrind (pronounced val--grinn)

- e.g.
  - Valgrind - a tool which monitors memory leaks (amongst other things)
  - http://www.valgrind.org/
  - (Research papers on implementation) http://www.valgrind.org/docs/pubs.html

Learn C The Hard Way — bash — 80×24

```
==70845== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==70845== Using Valgrind-3.11.0.SVN and LibVEX; rerun with -h for copyright in
==70845== Command: ./ex5
==70845==
You are 100 miles away.
==70845==
==70845== HEAP SUMMARY:
==70845==     in use at exit: 38,730 bytes in 415 blocks
==70845==   total heap usage: 515 allocs, 100 frees, 45,706 bytes allocated
==70845==
==70845== LEAK SUMMARY:
==70845==    definitely lost: 0 bytes in 0 blocks
==70845==    indirectly lost: 0 bytes in 0 blocks
==70845==      possibly lost: 0 bytes in 0 blocks
==70845==    still reachable: 0 bytes in 0 blocks
==70845==         suppressed: 38,730 bytes in 415 blocks
==70845==
==70845== For counts of detected and suppressed errors, rerun with: -v
==70845== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
acomputer:Learn C The Hard Way      ; sw_vers
ProductName:    Mac OS X
ProductVersion: 10.10.4
BuildVersion:   14E46
```

(Repeat) Program Correctness and Program Optimization

- The two previous use cases are where we would use program analysis
  a. Find Optimizations
  b. Enforce program correctness

Dynamic Analysis - Profilers (man perf)

- Profiling tools with our programs gathering performance information about a program.
  - Typically gathering execution time at different granularities (e.g. function) within the program
  - A log of the linux 'perf' tool is shown below.

Entries Sorted By Time Spent

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                                                 start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01    0.02       8+472     <cycle 2 as a whole> [4]
                0.01    0.02     244+260         offtime <cycle 2> [7]
                0.00    0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Call Tree

www.mshah.io

Dynamic Analysis - Valgrind (pronounced val--grinn)

- e.g.
  - Valgrind - a tool which monitors memory leaks (amongst other things)
  - http://www.valgrind.org/
  - (Research papers on implementation) http://www.valgrind.org/docs/pubs.html

```
Learn C The Hard Way — bash — 80×24
==70845== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==70845== Using Valgrind-3.11.0.SVN and LibVEX; rerun with -h for copyright ir
==70845== Command: ./ex5
==70845==
You are 100 miles away.
==70845==
==70845== HEAP SUMMARY:
==70845==     in use at exit: 38,730 bytes in 415 blocks
==70845==   total heap usage: 515 allocs, 100 frees, 45,706 bytes allocated
==70845==
==70845== LEAK SUMMARY:
==70845==    definitely lost: 0 bytes in 0 blocks
==70845==    indirectly lost: 0 bytes in 0 blocks
==70845==      possibly lost: 0 bytes in 0 blocks
==70845==    still reachable: 0 bytes in 0 blocks
==70845==         suppressed: 38,730 bytes in 415 blocks
==70845==
==70845== For counts of detected and suppressed errors, rerun with: -v
==70845== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
acomputer:Learn C The Hard Way      , sw_vers
ProductName:    Mac OS X
ProductVersion: 10.10.4
BuildVersion:   14E46
```

28

Static and Dynamic Analysis

(Digging a little deeper)

Static Analysis and Dynamic Analysis (1/3)

- Both analysis we are asking or interested in some property of a program.
- That means we are asking some question:
  - Static Analysis
    - “Given my program *P*, I am interested in knowing property *A*. Does *P* exhibit *A* in any possible execution?”
  - Dynamic Analysis
    - “Given my program *P*, I want to monitor property *A*. Log all occurrences of *A* in a single execution of *P*”

Static Analysis and Dynamic Analysis (2/3)

- Both analysis we are asking or interested in some property of a program.
- That means we are asking some question:
  - Static Analysis
    - "Given my program *P*, I am interested in kn[illegible]hit *A* in any possible execution?"
  - Dynamic Analysis
    - "Given my program *P*, I want to monitor property [illegible]es of *A* in a single execution of *P*"

- Correctness
- Performance
- Maybe Style
- etc.

Static Analysis and Dynamic Analysis (3/3)

- Both analysis we are asking or interested in some property of a program.
- That means we are asking some question:
  - Static Analysis
    - "Given my program *P*, I am interested in knowing property *A*. Does P exhibit A in any possible execution?"
  - Dynamic Analysis
    - "Given my program *P*, I want to monitor property A. Log all occurrences of A in a single execution of P"

Notice the key difference here between static and dynamic analysis?

Static Analysis - A little more rigorous (1/2)

- Static analysis however can be a bit more rigorous than just looking at the code however.
  - We can actually build up some sort of *model* of the source code.
  - Generally this allows us to ask questions about program behavior
    - i.e. Something more complex than indentation of source code, but about the execution of the software

Static Analysis - A little more rigorous (2/2)

- Static analysis however can be a bit more rigorous than just looking at the code however.
  - We can actually build up some sort of *model* of the source code.
  - Generally this allows us to ask questions about program behavior
    - i.e. Something more complex than indentation of source code, but about the execution of the software

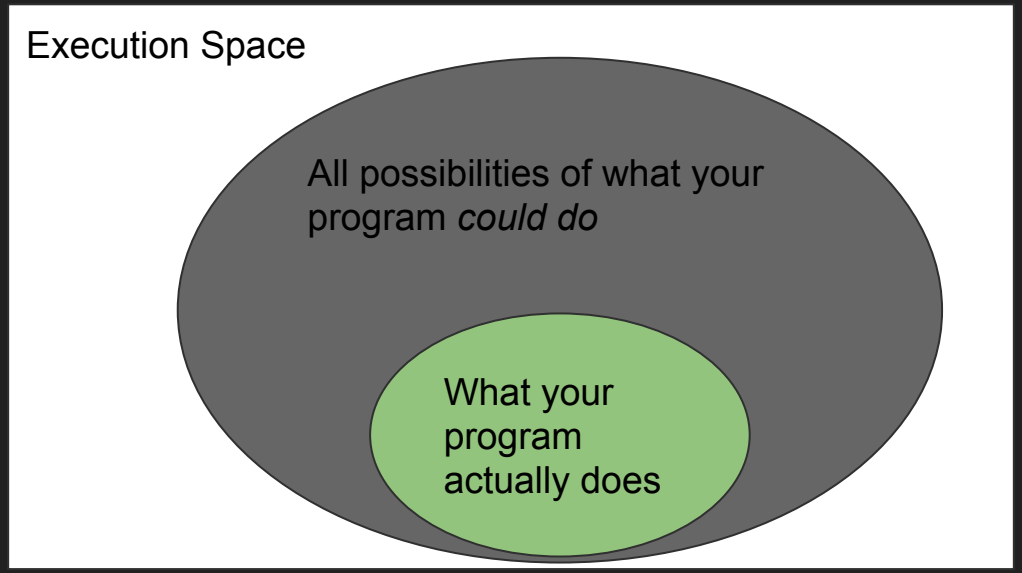
Static program analysis allows us to ask and answer questions about program behavior

Static Analysis

- "Given my program *P*, I am interested in knowing property *A*. Does P exhibit A in any possible execution?"

Static Analysis

- Because a static analysis looks at any possible execution, it is an over approximation of program behavior.
  - i.e. You are being very conservative in thinking about what can happen.

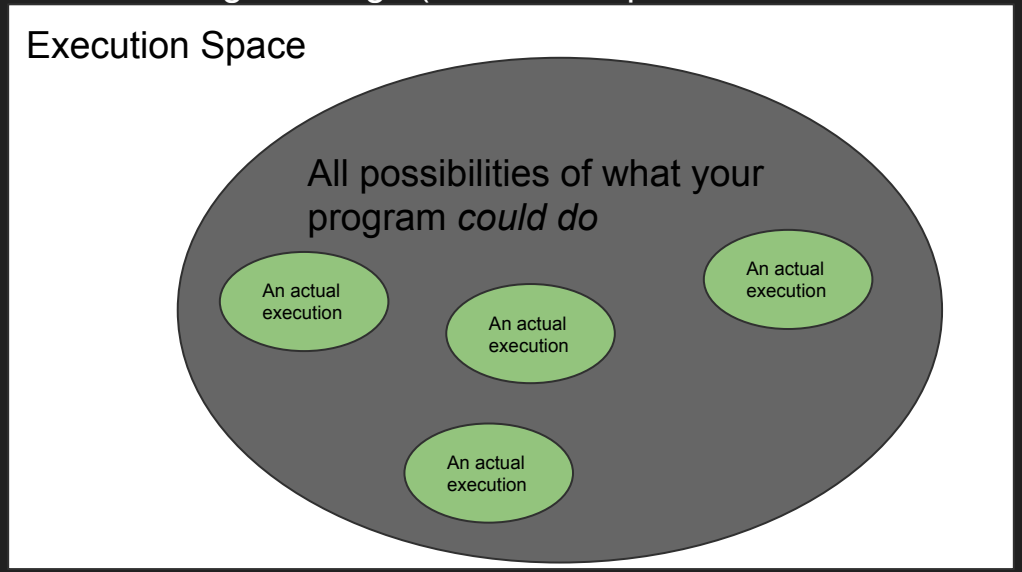


Dynamic Analysis

- "Given my program *P*, I want to monitor property A. Log all occurrences of A in a single execution of P"

Dynamic Analysis

- With dynamic analysis you are looking at one possible execution at a time each time you run your software.
  - Thus test your program with many inputs, to push it down several different execution paths in order to get better testing coverage (whether for performance or correctness).



Which type of analysis is better?

- One is not *better* per say--but can uncover bugs in different areas.

	Dynamic	Static
Effectiveness	Unsound (may miss errors)	Incomplete (may report spurious errors)
Cost	Proportional to program's execution	Proportional to program's size

Using LLVM

Whether for a static or a dynamic analysis...

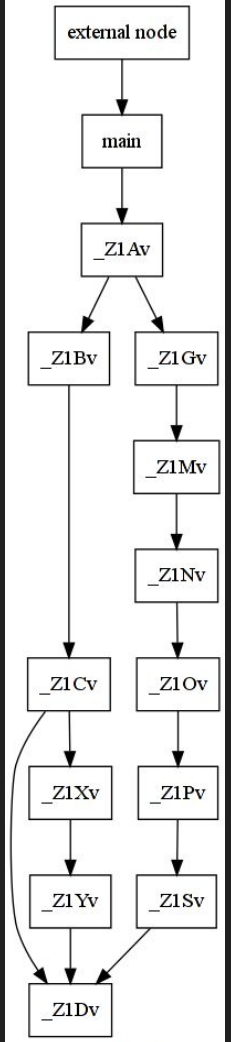
- We have two tools I generally use:
  - A call graph
  - A control flow graph
- Both are graph data structures for modeling our code.
- I will show two quick examples in LLVM for how to obtain them.

Dynamic Analysis Application in LLVM

(Call Graph)

Call Graph [CG]

- A call graph represents the calling relationships between procedures (i.e. functions in C++).
  - i.e. `foo()` was called from `main()`
- We have actually previously implemented a simple call graph in LLVM.
  - (See next slide)



Call Graph

```
#include "llvm/IR/CallSite.h"
namespace {
  // Hello3 - The third part of our tutorial
  struct Hello3 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello3() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of our instruction
                CallSite cs(&i);
```

```
    bool runOnFunction(Function &F) override {
        for(BasicBlock &bb: F){
            for(Instruction &i: bb){
                // Find where callsite is of our instruction
                CallSite cs(&i);
                if(!cs.getInstruction()){
                    continue;
                }
                Value *called = cs.getCalledValue()->stripPointerCasts();
                if(Function* f = dyn_cast<Function>(called)){
                    errs()  << "\tDirect call to function:" << f->getName()
                            << " from "                     << F.getName() << "\n";
                }
            }
        }

        return false;
    }
```

```
static RegisterPass<Hello3>
Z("hello3", "Hello World Pass (Get direct calls)");
```

Dynamic Analysis (1/5)

- Now we want to update our pass to make it a dynamic analysis
- This way we can actually figure out what paths a program is taking when executing
  - Helpful for program understanding
  - i.e.
    - We could recover the call stack and arguments.
    - See how often we traverse part of the call graph

```c
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

Dynamic Analysis (2/5)

- So let's say we are given the code to the right
  - (Our little countdown program)
- For our dynamic analysis, we want to collect some information such as how many times a function is called.
  - Typically we would 'instrument' the bitcode of our program, so we do not modify the original source.

```c
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

Dynamic Analysis (3/5)

- Question:
  a. How can I modify this program to count the number of calls to each function?
- Which option is better?
  a. Instrument within each function?
  b. Either add counters around each function?

a.

```
void countDown(){
    INC_TALLY_FUNCTION_CALLED("countdown")
    // body of countdown
}

vs

int main(){
...
    INC_TALLY_FUNCTION_CALLED("countdown")
    countDown();
...
}
```

b.

```
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

Dynamic Analysis (4/5)

- Question:
  a. How can I modify this program to count the number of calls to each function?
- Which option is better?
  a. Instrument within each function?
     - Probably fine if we have no library calls
  b. Either add counters around each function?
     - Likely the better option
     - (`printf(...,addFunc(5,2))` may need to be rewritten!)

a.

```
void countDown(){
    INC_TALLY_FUNCTION_CALLED("countdown")
    // body of countdown
}

vs

int main(){
...
    INC_TALLY_FUNCTION_CALLED("countdown")
    countDown();
...
}
```

b.

```
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

Dynamic Analysis (5/5)

- Question:
  a. How can I modify this program to count the number of calls to each function?
- Which option is better?
  a. Instrument within each function?
     - Probably fine if we have no library calls
  b. Either add counters around each function?
     - Likely the better option
     - (`printf(...,addFunc(5,2))` may need to be rewritten!)
  c. (Some more problems remain like with function pointers or perhaps polymorphic functions--to be continued as a research project)

a.

```
void countDown(){
    INC_TALLY_FUNCTION_CALLED("countdown")
    // body of countdown
}

vs

int main(){
...
    INC_TALLY_FUNCTION_CALLED("countdown")
    countDown();
...
}
```

b.

```
#include <stdio.h>

void countDown(){
    int x = 0;
    while(x < 10){
        ++x;
    }
}

int addFunc(int a, int b){
    return a+b;
}

int main(){
    printf("5+2=%d\n",addFunc(5,2));
    countDown();

    return 0;
}
```

(Aside)

- In previous talk we saw an example of how to add little ‘hooks’ to our code.
- This is how our counters would be inserted

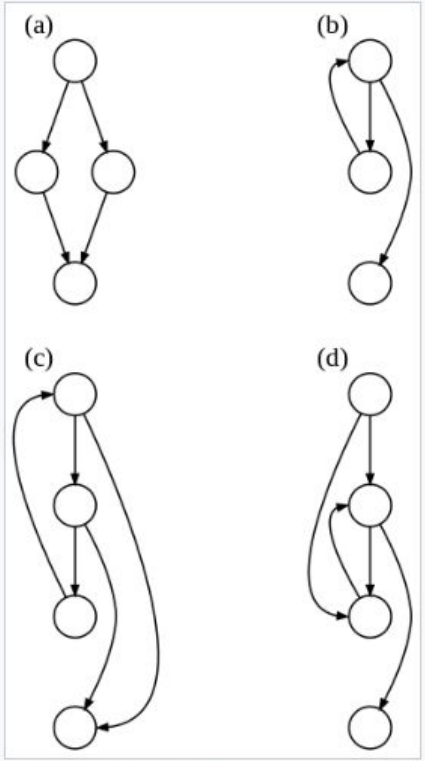
```
void InstrumentEnterFunction(StringRef InstrumentingFunctionName,Function& FunctionToInstrument, Module& M){
    // Create the actual function
    // If we have a function already, then the below is very useful
    //
    // FunctionType* funcTy = M.getFunction(InstrumentingFunctionName)->getFunctionType();
    //
    // However, we are hooking into a function that we will merge later, so we instead build our function type
    // Both methods will allow us to then modify the function arguments.
    //
    // Build out the function type
    auto &Context = M.getContext();
    // The functions return type
    Type* voidTy = Type::getVoidTy(Context);
    // The start of our parameters
    Type* intTy = Type::getInt32Ty(Context);
    // push back all of the parameters
    std::vector<llvm::Type*> params;
    params.push_back(intTy);
    // Specify the return value, arguments, and if there are variable numbers of arguments.
    FunctionType* funcTy = FunctionType::get(voidTy, params, false);
    // Create a Constant that grabs our function
    Constant* hook = M.getOrInsertFunction(InstrumentingFunctionName, funcTy);

    // We determine where we want to add our instrumentation.
    // In this instance, we want to instrument the first basic block, and
    // put the instruction at the front. Every function has at least an entry:
    // block in the LLVM IR, so this should be valid.
    BasicBlock *BB = &FunctionToInstrument.front();
    Instruction *I = &BB->front();

    // In order to set the arguments of the instrumenting function, we are going to
    // get all of our instrumenting functions arguments, and then modify them.
    std::vector<Value*> args;
    for(unsigned int i=0; i< funcTy->getNumParams(); ++i){
        Type* t = funcTy->getParamType(i);
        // We get the argument, and then we can re-assign its value
        // In this case, we are looking at our obController to see the function name in the hashmap, and then its value
        //
        // TODO: For now I know this is a constant, but perhaps this could change in the future.
        llvm::Value* foo = 0;
        //Value *newValue = dyn_cast<llvm::ConstantInt>(foo);
        Value *newValue = ConstantInt::get(t,0x1234);
        args.push_back(newValue);
        errs() << "getNumParams()" << i << "\n";
    }

    // Create our function call
    CallInst::Create(hook,args)->insertBefore(I);
}
```

Application in LLVM

(Control Flow Graph)

Control Flow Graph [CFG]

- Control Flow Graphs are graphical representations of paths that may be traversed during program execution.
  - Typically, it is the *right* tool for performing a static analysis.
  - I am working at a ‘basic block’ granularity

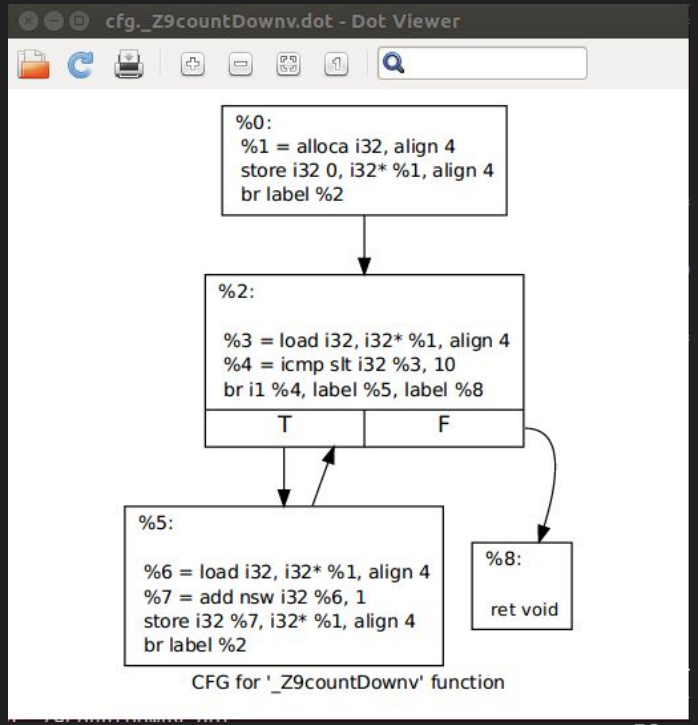


Some CFG examples:
(a) an if-then-else
(b) a while loop
(c) a natural loop with two exits, e.g. while with an if...break in the middle; non-structured but reducible
(d) an irreducible CFG: a loop with two entry points, e.g. goto into a while or for loop

Good news! Get the CFG for free!

```
void countDown(){
    int x = 0;
    while(x<10){
        ++x;
    }
}
```

- In order to generate a control flow graph using opt
  - `./../opt -dot-cfg < loops.ll > /dev/null`
- However, we may want to do this ourselves, such that we can annotate the control flow graph with our own analysis.
- Let's take a first look at how to build our own Control Flow Graph using LLVM.



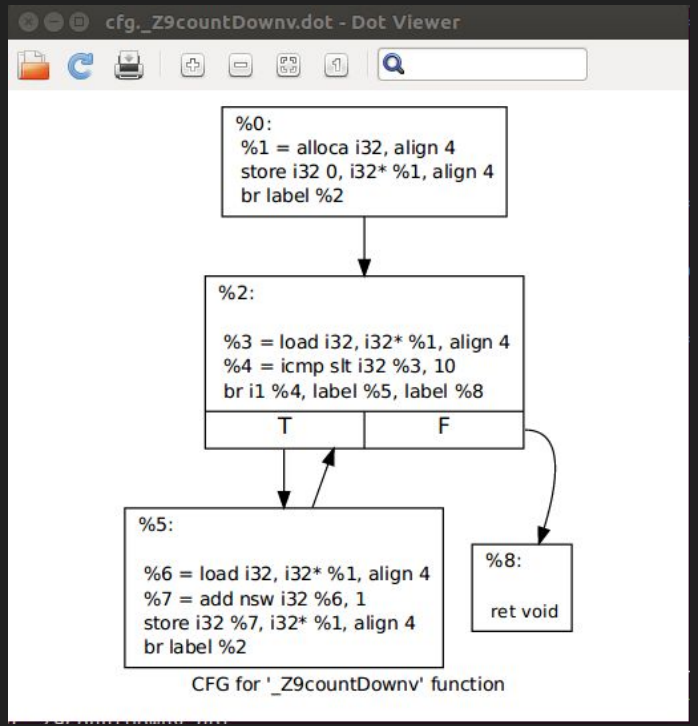
The Pass

- Here is the full pass in a slide for reference
- However, let's take a closer look at some of the components.

```
#include "llvm/IR/BasicBlock.h"
#include "llvm/ADT/SCCIterator.h"
#include "llvm/IR/CFG.h"
// See: source/llvm/tools/opt/PrintSCC.cpp for more
namespace {
  // Hello5 - Example extracting a CFG
  struct Hello5 : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    Hello5() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        errs() << "\nSCC for " << F.getName() << " in post-order:";
        unsigned sccNum = 0;

        for(scc_iterator<Function*> SCCI = scc_begin(&F); !SCCI.isAtEnd(); ++SCCI){
            const std::vector<BasicBlock*> &nextSCC = *SCCI;
            errs() << "\nSCC #" << ++sccNum << " : ";
            for(std::vector<BasicBlock*>::const_iterator I = nextSCC.begin(), E = nextSCC.end(); I!=E; ++I)
{
                // Retrieve bb name (i.e. automatically generated label
                StringRef bbName((*I)->getName().str());
                errs() << getSimpleNodeLabel((*I),&F) << ", ";
                if(nextSCC.size()==1 && SCCI.hasLoop()){
                    errs() << " (Has self-loop).";
                }
            }
        }
        errs() << "\n";

        return false;
    }

    static std::string getSimpleNodeLabel(const BasicBlock* bb, const Function* F){
        if(!bb->getName().empty()){
            return bb->getName().str();
        }
        std::string Str;
        raw_string_ostream OS(Str);
        bb->printAsOperand(OS,false);
        return OS.str();

    }

  };
}

char Hello5::ID = 0;
static RegisterPass<Hello5> ZZZ("hello5", "Extract CFG");
```

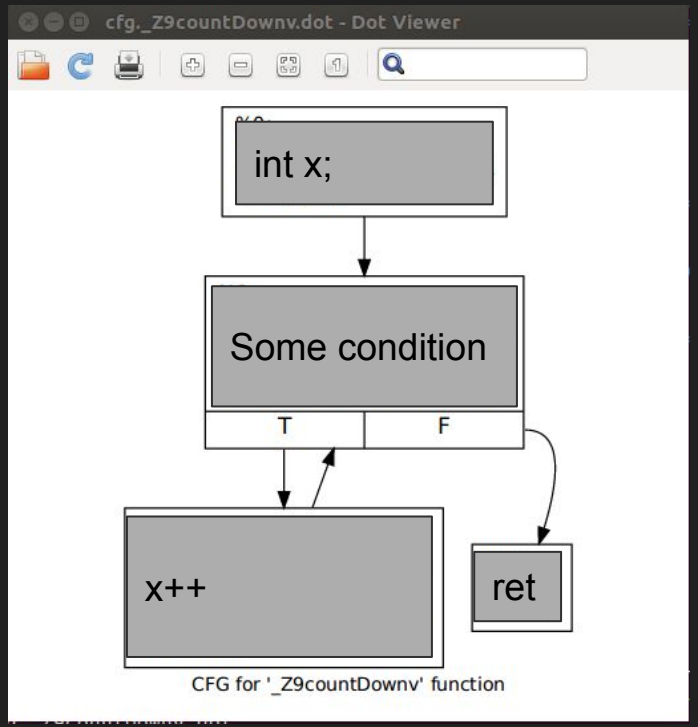
What I really want in my static analysis

- What I really want is the picture to the right
- Then I can start recording some information and answer a question I want to ask



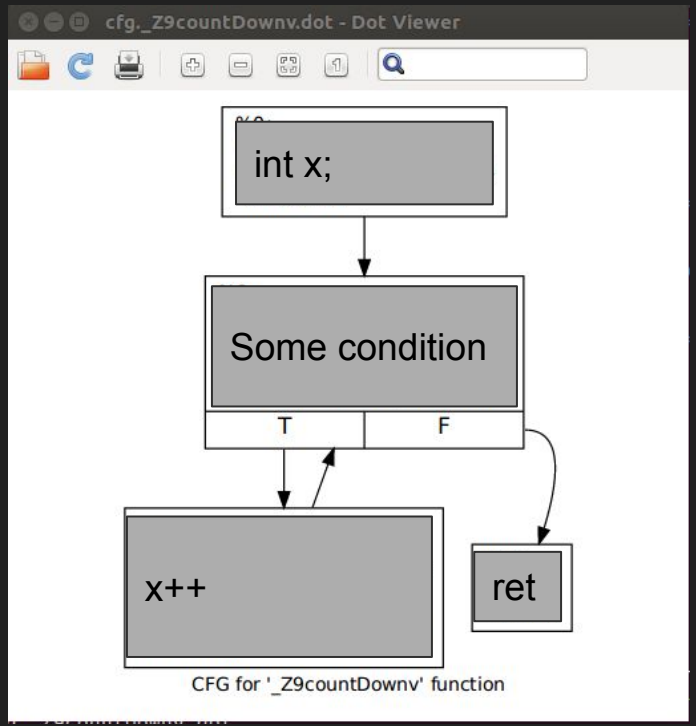
A problem (1/6)

- Let's forgot the code and pretend we have this problem
- We want to ask-- "Is 'x' assigned a value"
  - We see in one branch it is
  - In the other branch it is not
- [See related use-def and def-use chain documentation]



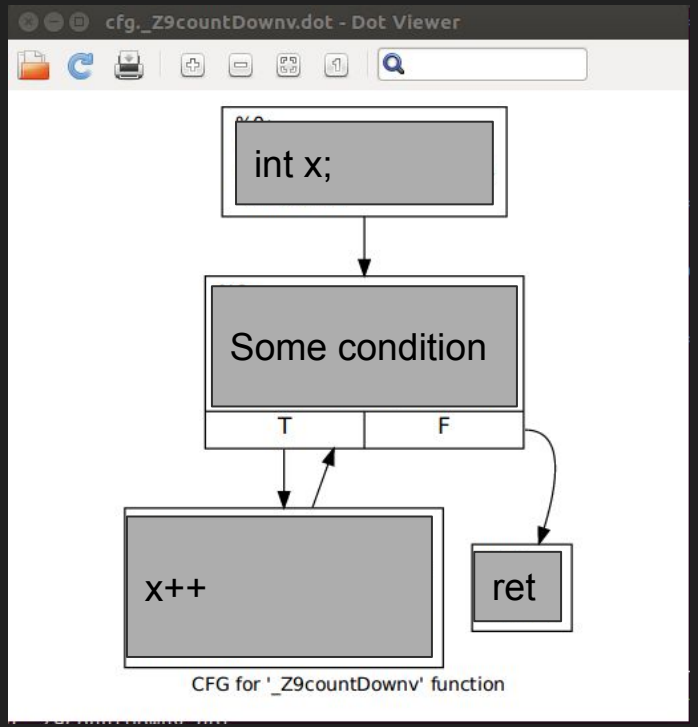
A problem (2/6)

- Remember, we're doing this statically, so we don't know what the condition will evaluate as
  - But we want to know if *anywhere* in our control flow graph '*x could be, or could not be assigned a value*'



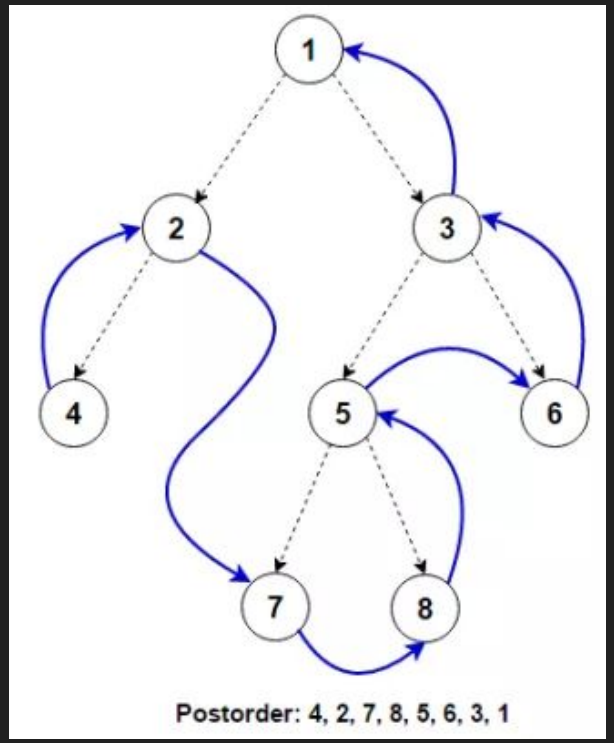
A problem (3/6)

- If I am doing this from scratch, here's what I want to do:
  - Figure out all of my 'values' (definitions)
  - See if they are 'used' anywhere
- The good news is, llvm provides some traversals of a functions basic blocks.
  - A post-order traversal can be one way to traverse a graph (i.e. a CFG).



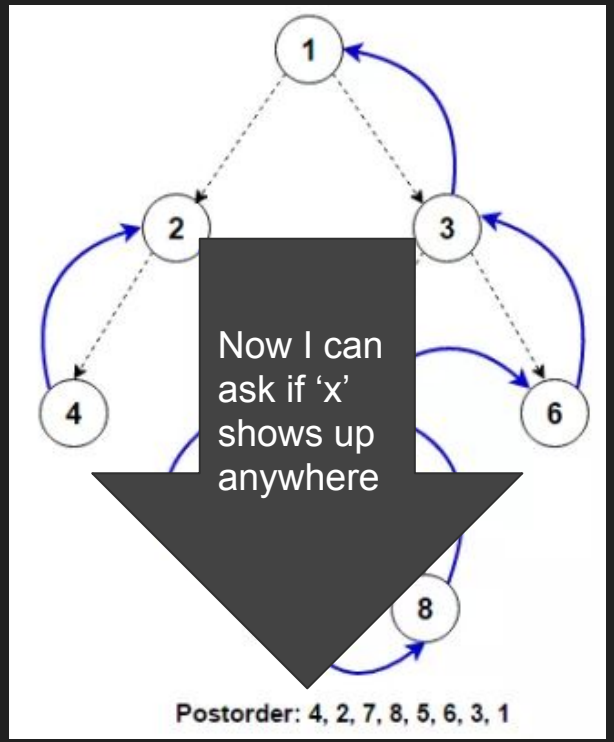
A problem (4/6)

- Here is a reminder of what the traversal would look like:



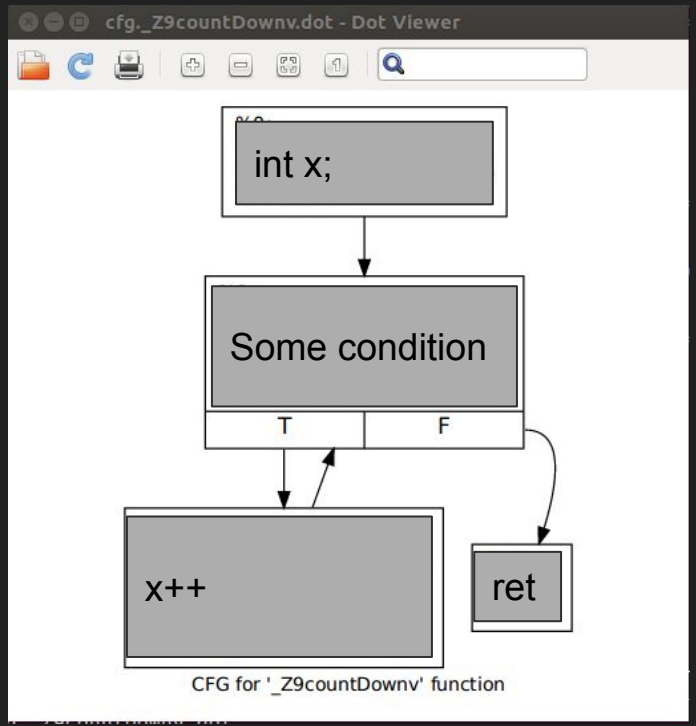
A problem (5/6)

- Here is a reminder of what the traversal would look like:



A problem (6/6)

- LLVM Provides a few options which I have in our pass
  - Simply travers our Basic blocks in a loop
    - (This is okay, but may not be correct)
  - post-order traversal
    - `po_iterator<BasicBlock>`
  - scc_iterator
    - Traverse the strongly connected components
    - Strongly connected components essentially allow us to collapse ‘cycles’ into a big node.
      - Often this makes the analysis more feasible



Function Pass (1/2)

- Our pass here is nothing more complicated than iterating through basic blocks (strongly connected where applicable)

```
bool runOnFunction(Function &F) override {
    errs() << "\nSCC for " << F.getName() << " in post-order:";
    unsigned sccNum = 0;

    for(scc_iterator<Function*> SCCI = scc_begin(&F); !SCCI.isAtEnd(); ++SCCI){
        const std::vector<BasicBlock*> &nextSCC = *SCCI;
        errs() << "\nSCC #" << ++sccNum << " : ";
        for(std::vector<BasicBlock*>::const_iterator I = nextSCC.begin(), E = nextSCC.end(); I!=E; ++I)

            // Retrieve bb name (i.e. automatically generated label
            StringRef bbName((*I)->getName().str());
            errs() << getSimpleNodeLabel((*I),&F) << ", ";
            if(nextSCC.size()==1 && SCCI.hasLoop()){
                errs() << " (Has self-loop).";
            }
        }
    }
    errs() << "\n";

    return false;
}
```

```
static std::string getSimpleNodeLabel(const BasicBlock* bb, const Function* F){
    if(!bb->getName().empty()){
        return bb->getName().str();
    }
    std::string Str;
    raw_string_ostream OS(Str);
    bb->printAsOperand(OS,false);
    return OS.str();

}
```

Function Pass (2/2)

- We can then look at each instruction in a basicblock and grab the value (or range of values if we like).

```
bool runOnFunction(Function &F) override {
    errs() << "\nSCC for " << F.getName() << " in post-order:";
    unsigned sccNum = 0;

    for(scc_iterator<Function*> SCCI = scc_begin(&F); !SCCI.isAtEnd(); ++SCCI){
        const std::vector<BasicBlock*> &nextSCC = *SCCI;
        errs() << "\nSCC #" << ++sccNum << " : ";
        for(std::vector<BasicBlock*>::const_iterator I = nextSCC.begin(), E = nextSCC.end(); I!=E; ++I)

            // Retrieve bb name (i.e. automatically generated label
            StringRef bbName((*I)->getName().str());
            errs() << getSimpleNodeLabel((*I),&F) << ", ";
            if(nextSCC.size()==1 && SCCI.hasLoop()){
                errs() << " (Has self-loop).";
            }
        }
    }
    errs() << "\n";

    return false;
}
```

```
static std::string getSimpleNodeLabel(const BasicBlock* bb, const Function* F){
    if(!bb->getName().empty()){
        return bb->getName().str();
    }
    std::string Str;
    raw_string_ostream OS(Str);
    bb->printAsOperand(OS,false);
    return OS.str();

}
```

Going Further (Challenges/Project Ideas)

Time permitting:

- Easy
  - Investigate PrintSCC.cpp and write a module pass to print out the Strongly Connected Components of a Call Graph
- Medium
  - Investigate CFGPrinter. See if you can reproduce in your own pass what the ‘dot-cfg’ analysis pass gives you.
    - Now try adding source information next to each basic block
      - *try* small code samples
  - Write an approximate static analysis to determine if a variable is positive, negative, or undetermined.
- Hard/Interesting?
  - Write an approximate static analysis to determine if the index of an array will be out of bounds
  - Write a static analysis to determine a set of functions a function pointer may point to.
    - If you can **always** determine what function it will point to--rewrite the source to that function pointer

Going Further (Challenges/Project Ideas)

Time permitting:

- For each of the homeworks
  - Try looking at the passes found in https://llvm.org/docs/Passes.html
  - The source code gives an introduction to some of the analysis you may want to perform.

LLVM's Analysis and Transform Passes

- Introduction
- Analysis Passes
  - **-aa-eval**: Exhaustive Alias Analysis Precision Evaluator
  - **-basicaa**: Basic Alias Analysis (stateless AA impl)
  - **-basiccg**: Basic CallGraph Construction
  - **-count-aa**: Count Alias Analysis Query Responses
  - **-da**: Dependence Analysis
  - **-debug-aa**: AA use debugger
  - **-domfrontier**: Dominance Frontier Construction
  - **-domtree**: Dominator Tree Construction
  - **-dot-callgraph**: Print Call Graph to "dot" file
  - **-dot-cfg**: Print CFG of function to "dot" file
  - **-dot-cfg-only**: Print CFG of function to "dot" file (with no function bodies)
  - **-dot-dom**: Print dominance tree of function to "dot" file
  - **-dot-dom-only**: Print dominance tree of function to "dot" file (with no function bodies)
  - **-dot-postdom**: Print postdominance tree of function to "dot" file
  - **-dot-postdom-only**: Print postdominance tree of function to "dot" file (with no function bodies)
  - **-globalsmodref-aa**: Simple mod/ref analysis for globals
  - **-instcount**: Counts the various types of **Instruction**s
  - **-intervals**: Interval Partition Construction
  - **-iv-users**: Induction Variable Users
  - **-lazy-value-info**: Lazy Value Information Analysis
  - **-libcall-aa**: LibCall Alias Analysis
  - **-lint**: Statically lint-checks LLVM IR
  - **-loops**: Natural Loop Information
  - **-memdep**: Memory Dependence Analysis
  - **-module-debuginfo**: Decodes module-level debug info
  - **-postdomfrontier**: Post-Dominance Frontier Construction

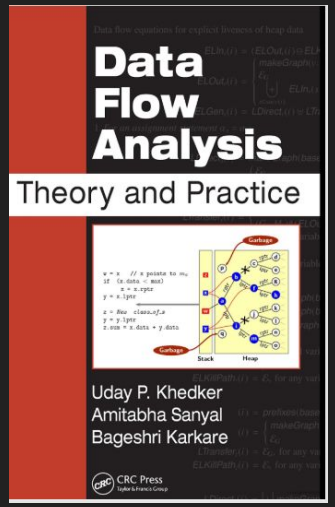
Resources

Useful resources

- What is static code analysis
  - https://www.perforce.com/blog/qac/what-static-code-analysis
- A Gentle Introduction to Program Analysis by Isıl Dillig (University of Texas)
  - https://www.cis.upenn.edu/~alur/CIS673/isil-plmw.pdf
- Analyzing function CFGs [of a function] with LLVM by Eli Bendersky
  - https://eli.thegreenplace.net/2013/09/16/analyzing-function-cfgs-with-llvm

Some things we still need to talk about in the future

- There is a semester's (or careers worth) of material to continue exploring
  - More practice and frameworks
    - Dataflow analysis [more]
    - Abstract Interpretation
  - Theory
    - Halting Problem
      - "Does my program terminate" -- undedicable!
    - A bit more graph theory
      - i.e.
        - Intuition for: *Why collapse things to a Strongly connected components?*



Wrapping up Program analysis

- Why do it?
  - We all write code--it helps us understand what we are coding
    - Whether the domain is performance, correctness bugs, security, etc.
  - We can visualize our results
    - i.e. as a call graph
  - We can have someone else debug our code for us
    - (Potentially automatically if we have a good enough tool!)

Conclusion

- Program analysis is process of [mostly] automatically analyzing the behavior of programs, either statically or dynamically.
- Carefully choosing techniques can be helpful in improving specific properties of your programs behavior.
  - i.e. program correctness or program performance.

Thank You!

@MichaelShah | www.mshah.io

Feedback Form TBD
(Whether you watched this talk now or in the future!)