A Brief Introduction to Using LLVM

Nick Sumner
What is LLVM?

- A compiler? (clang)
What is LLVM?

- A compiler? (clang)
- A set of formats, libraries, and tools.
What is LLVM?

- A compiler? (clang)
- A set of formats, libraries, and tools.
  - A simple, typed IR (*bitcode*)
  - Program analysis / optimization libraries
  - Machine code generation libraries
  - Tools that compose the libraries to perform task
What is LLVM?

• A compiler? (clang)

• A set of formats, libraries, and tools.
  – A simple, typed IR (*bitcode*)
  – Program analysis / optimization libraries
  – Machine code generation libraries
  – Tools that compose the libraries to perform task
What is LLVM?

- A compiler? (clang)
- A set of formats, libraries, and tools.
  - A simple, typed IR (*bitcode*)
  - Program analysis / optimization libraries
  - Machine code generation libraries
  - Tools that compose the libraries to perform task
What is LLVM?

• A compiler? (clang)

• A set of formats, libraries, and tools.
  – A simple, typed IR (*bitcode*)
  – Program analysis / optimization libraries
  – Machine code generation libraries
  – Tools that compose the libraries to perform task
What is LLVM?

• A compiler? (clang)

• A set of formats, libraries, and tools.
  – A simple, typed IR (*bitcode*)
  – Program analysis / optimization libraries
  – Machine code generation libraries
  – Tools that compose the libraries to perform tasks

• Easy to add / remove / change functionality
How will you be using it?

- Compiling programs to bitcode:
  ```
  clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
  ```
How will you be using it?

- Compiling programs to bitcode:
  ```
  clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
  ```
- Analyzing the bitcode:
  ```
  opt -load <plugin>.so --<plugin> -analyze <bitcode>.bc
  ```
How will you be using it?

- Compiling programs to bitcode:
  
  \[ \text{clang} \ -g \ -c \ -\text{emit-llvm} \ <\text{sourcefile}> \ -o \ <\text{bitcode}>.bc \]

- Analyzing the bitcode:
  
  \[ \text{opt} \ -\text{load} \ <\text{plugin}>.so \ --<\text{plugin}> \ -\text{analyze} \ <\text{bitcode}>.bc \]

- Writing your own tools:
  
  \[ ./\text{callcounter} \ -\text{static} \ \text{test}.bc \]
How will you be using it?

- Compiling programs to bitcode:
  
  ```
  clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
  ```

- Analyzing the bitcode:

  ```
  opt -load <plugin>.so --<plugin> -analyze <bitcode>.bc
  ```

- Writing your own tools:

  ```
  ./callcounter -static test.bc
  ```

- Reporting properties of the program:

  ```
  Function Counts
  ===============
  b : 2
  a : 1
  printf : 3
  ```
What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void
foo(unsigned e) {
    for (unsigned i = 0; i < e; ++i) {
        printf("Hello\n");
    }
}

int
main(int argc, char **argv) {
    foo(argc);
    return 0;
}
```

```
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
    %2 = icmp eq i32 %0, 0
    br i1 %2, label %3, label %4

; <label>:3:                 ; preds = %4, %1
    ret void

; <label>:4:                 ; preds = %1, %4
    %5 = phi i32 [ %7, %1 ], [ %0, %1 ]
    %6 = tail call i32 @str(i8* getelementptr
        ([6 x i8], [6 x i8]@str, i64 0, i64 0))
    %7 = add nuw i32 %5, 1
    %8 = icmp eq i32 %7, %0
    br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
    tail call void @foo(i32 %0)
    ret i32 0
}
```
What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>
void
foo(unsigned e) {  
    for (unsigned i = 0; i < e; ++i) {
        printf("Hello\n");
    }
}

int
main(int argc, char **argv) {
    foo(argc);
    return 0;
}
```

```mlir
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
    %2 = icmp eq i32 %0, 0
    br i1 %2, label %3, label %4
    %5 = phi i32 [%7, %4], [ %0, %1 ]
    %6 = tail call i32 @puts([i8* getelementptr ([6 x i8], [6 x i8]* @str, i64 0, i64 0)])
    %7 = add nuw i32 %5, 1
    %8 = icmp eq i32 %7, %0
    br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readonly) {
    tail call void @foo(i32 %0)
    ret i32 0
}
```

`clang -c -emit-llvm (and llvm-dis)`
What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void foo(unsigned e) {
    for (unsigned i = 0; i < e; ++i) {
        printf("Hello\n");
    }
}

int main(int argc, char **argv) {
    foo(argc);
    return 0;
}
```

```assembly
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
    %2 = icmp eq i32 %0, 0
    br i1 %2, label %3, label %4

    ; <label>:3:                 ; preds = %4, %1
        ret void

    ; <label>:4:                 ; preds = %1, %4
        %5 = phi i32 [ %7, %4 ], [ %0, %1 ]
        %6 = tail call i32 @puts(i8* getelementptr
                                   ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
        %7 = add nuw i32 %5, 1
        %8 = icmp eq i32 %7, %0
        br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
    tail call void @foo(i32 %0)
    ret i32 0
}
```
What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void foo(unsigned e) {
    for (unsigned i = 0; i < e; ++i) {
        printf("Hello\n");
    }
}

int main(int argc, char **argv) {
    foo(argc);
    return 0;
}
```

```c
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
    %2 = icmp eq i32 %0, 0
    br i1 %2, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
    tail call void @foo(i32 %0)
    ret i32 0
}
```
What is LLVM Bitcode?

• A (Relatively) Simple IR

```
#include<stdio.h>

void foo(unsigned e) {
    for (unsigned i = 0; i < e; ++i) {
        printf("Hello\n");
    }
}

int main(int argc, char **argv) {
    foo(argc);
    return 0;
}
```

Basic Blocks

```c
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
    %2 = icmp eq i32 %0, 0
    br i1 %2, label %3, label %4
}

; <label>:3:                 ; preds = %4, %1
    ret void
    ; <label>:4:                 ; preds = %1, %4
    %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
    %6 = tail call i32 @puts(i8* getelementptr i8, [6 x i8]* @str, i64 0, i64 0))
    %7 = add nuw i32 %5, 1
    %8 = icmp eq i32 %7, %0
    br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
    tail call void @foo(i32 %0)
    ret i32 0
}
```
What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                 ; preds = %4, %1
  ret void

; <label>:4:                 ; preds = %1, %4
  %5 = phi i32 [
    %7, %4 ], [
    0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
    ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, 0
  br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
  tail call void @foo(i32 %0)
  ret i32 0
}
```
What is LLVM Bitcode?

- A (Relatively) Simple IR
Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate

```c
Module& module = ...;
for (Function& fun : module) {
    for (BasicBlock& bb : fun) {
        for (Instruction& i : bb) {

        Iterate over the:
        • Functions in a Module
        • BasicBlocks in a Function
        • Instructions in a BasicBlock

...```
Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite,)

```
Module& module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }
    }
  }
...
```
Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite, outs(), ...)

```cpp
Module &module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }
      outs() << "Found a function call: " << i << "\n";
    }
  }
}
```

...
Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite, outs(), dyn_cast)

```cpp
Module &module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }
      outs() << "Found a function call: " << i << "\n";
      Value* called = cs.getCalledValue()->stripPointerCasts();
      if (Function* f = dyn_cast<Function>(called)) {
        outs() << "Direct call to function: " << f->getName() << "\n";
        ... 
      }
    }
  }
}
```

dyn_cast() efficiently checks the runtime types of LLVM IR components.
Dealing with SSA

- You may ask where certain values came from
  - Useful for tracking dependencies (PDG)
  - “Where was this variable defined?”
Dealing with SSA

- You may ask where certain values came from
- LLVM IR provides this through SSA form
Dealing with SSA

- You may ask where certain values came from
- LLVM IR provides this through SSA form

```c
void foo()
{
    unsigned i = 0;
    while (i < 10) {
        i = i + 1;
    }
}
```
Dealing with SSA

- You may ask where certain values came from
- LLVM IR provides this through SSA form

```c
void foo()
{
    unsigned i = 0;
    while (i < 10) {
        i = i + 1;
    }
}
```

What is the single definition of `i` at this point?
Dealing with SSA

- Thus the phi (φ) instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block
Dealing with SSA

- Thus the phi (φ) instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block

```c
void foo() {
  unsigned i = 0;
  while (i < 10) {
    i = i + 1;
  }
}
```

```c
define void @foo() {
  br label %1
  ; <label>:1 ; preds = %1, %0
  %i.phi = phi i32 [ 0, %0 ], [ %2, %1 ]
  %2 = add i32 %i.phi, 1
  %exitcond = icmp eq i32 %2, 10
  br i1 %exitcond, label %3, label %1
  ; <label>:3 ; preds = %1
  ret void
}
```
Dealing with SSA

- Thus the phi ($\varphi$) instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block

```c
void foo() {
    unsigned i = 0;
    while (i < 10) {
        i = i + 1;
    }
}
```
Dependencies in General

• You can loop over the values an instruction uses

```cpp
for (Use& u : inst->operands()) {
    // inst uses the Value* u
}
```
Dependencies in General

- You can loop over the values an instruction uses

```cpp
for (Use& u : inst->operands()) {
    // inst uses the Value* u
}
```

```cpp
for %a = %b + %c:
    [%b, %c]
```
### Dependencies in General

- You can loop over the values an instruction uses

```cpp
for (Use& u : inst->operands()) {
    // inst uses the Value* u
}
```

- You can loop over the instructions that use a particular value

```cpp
Instruction* inst = ...;
for (User* user : inst->users())
    if (auto* i = dyn_cast<Instruction>(user)) {
        // inst is used by Instruction i
    }
```
Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()
Dealing with Types

- LLVM IR is strongly typed
  - Every value has a type → getType()
- A value must be explicitly cast to a new type

```c
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```
Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()
- A value must be explicitly cast to a new type

```c
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```
Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → `get_type()`
- A value must be explicitly cast to a new type

```c
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```

- Also types for pointers, arrays, structs, etc.
  - Strong typing means they take a bit more work
Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)
Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/ structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

```
struct rec {
    int x;
    int y;
};

struct rec *buf;

void foo() {
    buf[5].y = 7;
}
```
Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

```assembly
%struct.rec = type { i32, i32 }
@buf = global %struct.rec* null

define void @foo() {
  %1 = load %struct.rec*, %struct.rec** @buf
  %2 = getelementptr %struct.rec, %struct.rec* %1, i64 5, i32 1
  store i32 7, i32* %2
  ret void
}
```
Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

```c
struct rec {
    int x;
    int y;
};

struct rec *buf;

void foo() {
    buf[5].y = 7;
}
```
Where Can You Get Info?

• The online documentation is extensive:
  – LLVM Programmer’s Manual
Where Can You Get Info?

- The online documentation is extensive:
  - LLVM Programmer’s Manual
  - LLVM Language Reference Manual
- The header files!
  - All in llvm-3.x.src/include/llvm/

BasicBlock.h  InstrTypes.h
CallSite.h    IRBuilder.h
DerivedTypes.h Support/InstVisitor.h
Function.h    Type.h
Instructions.h
Creating a Static Analysis
Making a New Analysis

- Analyses are organized into individual *passes*
  - ModulePass
  - FunctionPass
  - LoopPass
  - ...

Derive from the appropriate base class to make a Pass.
Making a New Analysis

- Analyses are organized into individual passes
  - ModulePass
  - FunctionPass
  - LoopPass
  - ...

3 Steps

1) Declare your pass
2) Register your pass
3) Define your pass
Making a New Analysis

• Analyses are organized into individual *passes*
  – ModulePass
  – FunctionPass
  – LoopPass
  – ...

3 Steps
1) Declare your pass
2) Register your pass
3) Define your pass

Let's count the number of *static direct calls* to each function.
Making a ModulePass (1)

- Declare your ModulePass

```cpp
struct StaticCallCounter : public llvm::ModulePass {

    static char ID;

    DenseMap<Function*, uint64_t> counts;

    StaticCallCounter()
        : ModulePass(ID)
    {
    }

    bool runOnModule(Module& m) override;
    void print(raw_ostream& out, const Module* m) const override;
    void handleInstruction(CallSite cs);
};
```
Making a ModulePass (1)

- Declare your ModulePass

```cpp
struct StaticCallCounter : public llvm::ModulePass {

    static char ID;

    DenseMap<Function*, uint64_t> counts;

    StaticCallCounter()
        : ModulePass(ID)
    {
    }

    bool runOnModule(Module& m) override;

    void print(raw_ostream& out, const Module* m) const override;

    void handleInstruction(CallSite cs);
};
```
Making a ModulePass (1)

- Declare your ModulePass

```cpp
struct StaticCallCounter : public llvm::ModulePass {

    static char ID;

    DenseMap<Function*, uint64_t> counts;

    StaticCallCounter() : ModulePass(ID) {
    }

    bool runOnModule(Module& m) override;
    void print(raw_ostream& out, const Module* m) const override;
    void handleInstruction(CallSite cs);
};
```
Register your ModulePass

- This allows it to be dynamically loaded as a plugin

```cpp
char StaticCallCounter::ID = 0;

RegisterPass<StaticCallCounter> SCCReg("callcounter",
    "Print the static count of direct calls");
```
Making a ModulePass (3)

- Define your ModulePass
  - Need to override `runOnModule()` and `print()`

```cpp
bool StaticCallCounter::runOnModule(Module& m) {
  for (auto& f : m)
    for (auto& bb : f)
      for (auto& i : bb)
        handleInstruction(CallSite(&i));
  return false; // False because we didn't change the Module
}
```
void StaticCallCounter::handleInstruction(CallSite cs) {
    // Check whether the instruction is actually a call
    if (!cs.getInstruction()) { return; }

    // Check whether the called function is directly invoked
    auto called = cs.getCalledValue()->stripPointerCasts();
    auto fun = dyn_cast<Function>(called);
    if (!fun) { return; }

    // Update the count for the particular call
    auto count = counts.find(fun);
    if (counts.end() == count) {
        count = counts.insert(std::make_pair(fun, 0)).first;
    }
    ++count->second;
}
void StaticCallCounter::handleInstruction(CallSite cs) {
  // Check whether the instruction is actually a call
  if (!cs.getInstruction()) { return; }

  // Check whether the called function is directly invoked
  auto called = cs.getCalledValue()->stripPointerCasts();
  auto fun = dyn_cast<Function>(called);
  if (!fun) { return; }

  // Update the count for the particular call
  auto count = counts.find(fun);
  if (counts.end() == count) {
    count = counts.insert(std::make_pair(fun, 0)).first;
  }
  ++count->second;
}
void StaticCallCounter::handleInstruction(CallSite cs) {
    // Check whether the instruction is actually a call
    if (!cs.getInstruction()) {
        return;
    }

    // Check whether the called function is directly invoked
    auto called = cs.getCalledValue()->stripPointerCasts();
    auto fun = dyn_cast<Function>(called);
    if (!fun) {
        return;
    }

    // Update the count for the particular call
    auto count = counts.find(fun);
    if (counts.end() == count) {
        count = counts.insert(std::make_pair(fun, 0)).first;
    }
    ++count->second;
}
• Printing out the results

```cpp
void CallCounterPass::print(raw_ostream& out, const Module* m) const {
    out << "Function Counts\n"
        << "===============\n"
        << "\n";
    for (auto& kvPair : counts) {
        auto* function = kvPair.first;
        uint64_t count = kvPair.second;
        out << function->getName() << " : " << count << "\n";
    }
}
```
Creating a Dynamic Analysis
**Making a Dynamic Analysis**

- We've counted the static direct calls to each function.
- How might we compute the *dynamic calls* to each function?
Making a Dynamic Analysis

- We've counted the static direct calls to each function.
- How might we compute the *dynamic calls* to each function?
- Need to *modify* the original program!
Making a Dynamic Analysis

- We've counted the static direct calls to each function.
- How might we compute the *dynamic calls* to each function?
- Need to *modify* the original program!
- Steps:
  1. *Modify* the program using passes
  2. *Compile* the modified version
  3. *Run* the new program
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

```c
void foo()
    bar();
}
```
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

```c
void foo()
{
    bar();
}
```

Keep a counter for each function!
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

Keep a counter for each function!

2 Choices:
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

```c
void foo()
    bar();
}
```

Keep a counter for each function!

**2 Choices:**
1) increment count for each function *as it starts*
2) increment count for each function *at its call site*
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

```c
void foo()
    bar();
}
```

Keep a counter for each function!

2 Choices:
1) increment count for each function *as it starts*
2) increment count for each function *at its call site*

Does that even matter? Are there trade offs?
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

We'll increment at the function entry.

(The demo code has both)
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

```
void foo()
bar();
}
```

We'll increment at the function entry

- *Using numeric IDs* for functions is sometimes easier
Modifying the Original Program

**Goal:** Count the dynamic calls to each function in an execution.

So how do we want to modify the program?

We'll increment at the function entry

- Using numeric IDs for functions is sometimes easier
- Inserting function calls is sometimes easier

```c
void foo()
    bar();
}
```

```
void foo()
countCall(1);
    bar();
}
```
Modifying the Original Program

What might adding this call look like?

```cpp
void DynamicCallCounter::handleInstruction(CallSite cs, Value* counter) {
    // Check whether the instruction is actually a call
    if (!cs.getInstruction()) {
        return;
    }

    // Check whether the called function is directly invoked
    auto calledValue = cs.getCalledValue()->stripPointerCasts();
    auto calledFunction = dyn_cast<Function>(calledValue);
    if (!calledFunction) {
        return;
    }

    // Insert a call to the counting function.
    IRBuilder<> builder(cs.getInstruction());
    builder.CreateCall(counter, builder.getInt64(ids[calledFunction]));
}
```
Modifying the Original Program

What might adding this call look like?

```c++
void DynamicCallCounter::handleInstruction(CallSite cs, Value* counter) {
    // Check whether the instruction is actually a call
    if (!cs.getInstruction()) {
        return;
    }

    // Check whether the called function is directly invoked
    auto calledValue = cs.getCalledValue()->stripPointerCasts();
    auto calledFunction = dyn_cast<Function>(calledValue);
    if (!calledFunction) {
        return;
    }

    // Insert a call to the counting function.
    IRBuilder<> builder(cs.getInstruction());
    builder.CreateCall(counter, builder.getInt64(ids[calledFunction]));
}
```
Modifying the Original Program

What might adding this call look like?

```c++
void DynamicCallCounter::handleInstruction(CallSite cs, Value* counter) {
    // Check whether the instruction is actually a call
    if (!cs.getInstruction()) {
        return;
    }

    // Check whether the called function is directly invoked
    auto calledValue = cs.getCalledValue()->stripPointerCasts();
    auto calledFunction = dyn_cast<Function>(calledValue);
    if (!calledFunction) {
        return;
    }

    // Insert a call to the counting function.
    IRBuilder<> builder(cs.getInstruction());
    builder.CreateCall(counter, builder.getInt64(ids[calledFunction]));
}
```

In practice, it's more complex. You can find details in the demo code.
Using a Runtime Library

Don't forget that we need to put `countCall()` somewhere!

- Placed in a library linked with the main executable

```c
void countCalled(uint64_t id) {
    ++functionInfo[id];
}
```
Dynamic Analysis Big Picture

Program/Module

Analysis Tool
- Instrumentation Pass
- Compilation
- Modified Program

Runtime Library

Input

Results!
Dynamic Analysis Big Picture

Program/Module

Analysis Tool

Instrumentation Pass

Compilation

Runtime Library

Input

Modified Program

Results!

Step 1: Insert useful calls to a runtime library
Dynamic Analysis Big Picture

**Program/Module**

**Analysis Tool**
- **Instrumentation Pass**
- **Compilation**

**Runtime Library**

**Step 2:** Compile & link against runtime library.

**Input**

**Modified Program**

**Results!**
Dynamic Analysis Big Picture

Step 3: Run the new program to produce your results

Program/Module

Instrumentation Pass

Compilation

Analysis Tool

Runtime Library

Input

Modified Program

Results!
Bringing It All Together
LLVM Projects

- LLVM organizes groups of passes and tools into *projects*
LLVM Projects

- LLVM organizes groups of passes and tools into projects

- Easiest way to start is by using the demo on the course page
LLVM Projects

- LLVM organizes groups of passes and tools into projects
- Easiest way to start is by using the demo on the course page
- For the most part, you can follow the directions online & in project description
Extra Tips

• I have a pointer to something. What is it?
  – The getName() method works on most things.
  – You can usually: outs() << x
Extra Tips

- I have a pointer to something. What is it?
  - The getName() method works on most things.
  - You can usually: outs() << x

- Sadly no longer true:

  How do I see the C++ API calls for constructing a module?
  - llc -march=cpp <bitcode>.bc -o <cppapi>.cpp