A Brief Introduction to LLVM

Nick Sumner
wsumner@sfu.ca

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 tasks

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

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

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

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:

```
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
```

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
  intermediate representation (IR)
  - It captures the
    program dependence graph

What is LLVM Bitcode?

- A (relatively) simple intermediate representation (IR)
  - It captures the program dependence graph

Code

```
#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;
}
```

```
clang -c -S -emit-llvm -O1 -g0
```

IR

```
@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 intermediate representation (IR)
  - It captures the program dependence graph

```
#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;
}
```

```
clang -c -S -emit-llvm -O1 -g0
```

```
@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 intermediate representation (IR)
  - It captures the program dependence graph

```
#include<stdio.h>

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

int
main
  fo
  re
}
```

Functions

```
clang -c -S -emit-llvm -O1 -g0
```

```
@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 intermediate representation (IR)
  - It captures the program dependence graph

```
#include<stdio.h>

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

int
main
  fo
  re
}
```

Basic Blocks

```
clang -c -S -emit-llvm -O1 -g0
```

```
@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 intermediate representation (IR)
  - It captures the program dependence graph

```
#include<stdio.h>

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

int
main
  fo
  re
}
```

Basic Blocks

clang -c -S -emit-llvm -O1 -g0

```
@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
                                   [ 0, %1 ]
                                   (i8* getelementptr
                                   @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
}
```

labels & predecessors

What is LLVM Bitcode?

- A (relatively) simple intermediate representation (IR)
  - It captures the program dependence graph

```
#include<stdio.h>

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

int
main
  fo
  re
}
```

Basic Blocks

`clang -c -S -emit-llvm -O1 -g0`

```
@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 @m                              ) {
  tail call void @foo(i32 %0)
  ret i32 0
}
```

branches & successors

What is LLVM Bitcode?

- A (relatively) simple intermediate representation (IR)
  - It captures the program dependence graph

```c
#include<stdio.h>

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

int
main
  fo
  re
}
```

Instructions

`clang -c -S -emit-llvm -O1 -g0`

```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
}
```

Inspecting Bitcode

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

Inspecting Bitcode

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

```
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. CallBase, outs(), dyn_cast)

```
Module& module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallBase* cb = dyn_cast<CallBase>(&i);
      if (!cb) {
        continue;
      }

...
```

`dyn_cast()` efficiently checks the runtime types of LLVM IR components.

Inspecting Bitcode

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

```
Module& module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallBase* cb = dyn_cast<CallBase>(&i);
      if (!cb) {
        continue;
      }
...
```

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

CallBase provides a common interface for different type of function calls

Inspecting Bitcode

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

```
Module& module = ...;
for (Function& fun : module) {
  for (BasicBlock& bb : fun) {
    for (Instruction& i : bb) {
      CallBase* cb = dyn_cast<CallBase>(&i);
      if (!cb) {
        continue;
      }
      outs() << "Found a function call: " << i << "\n";
...
```

`outs()` and other printing functions make inspecting components easy

Inspecting Bitcode

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

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

Working within the API allows you to ask questions about code.

Inspecting Bitcode

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

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

Working within the API allows you to ask questions about code.

Inspecting Bitcode

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

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

Working within the API allows you to ask questions about code.

Static Single Assignment (SSA)

- Program dependence graphs help answer questions like:
  - Where was a variable defined?
  - Where is a particular value used?

Static Single Assignment (SSA)

- Program dependence graphs help answer questions like:
  - Where was a variable defined?
  - Where is a particular value used?
- Compilers today help provide this using SSA form
  - Each variable has a single definition,
    so resolving dependencies is easier

Static Single Assignment (SSA)

- Program dependence graphs help answer questions like:
  - Where was a variable defined?
  - Where is a particular value used?
- Compilers today help provide this using SSA form
  - Each variable has a single definition,
    so resolving dependencies is easier

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

Static Single Assignment (SSA)

- Program dependence graphs help answer questions like:
  - Where was a variable defined?
  - Where is a particular value used?
- Compilers today help provide this using SSA form
  - Each variable has a single definition, so resolving dependencies is easier

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

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

Static Single Assignment (SSA)

- Program dependence graphs help answer questions like:
  - Where was a variable defined?
  - Where is a particular value used?
- Compilers today help provide this using SSA form
  - Each variable has a single definition,
    so resolving dependencies is easier
- Phi instructions select which incoming value to use among options
  - Phi nodes must occur at the *beginning* of a basic block

Static Single Assignment (SSA)

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

```llvm
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
}
```

- Phi instructions select which incoming value to use among options
  - Phi nodes must occur at the *beginning* of a basic block

Static Single Assignment (SSA)

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

```
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
}
```

- Phi instructions select which incoming value to use among options
  - Phi nodes must occur at the *beginning* of a basic block

Static Single Assignment (SSA)

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

```
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
}
```

- Phi instructions select which incoming value to use among options
  - Phi nodes must occur at the *beginning* of a basic block

Dependencies in General

- You can loop over the values an instruction uses

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

Dependencies in General

- You can loop over the values an instruction uses

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

Given %a = %b + %c:
[%b, %c]

Dependencies in General

- You can loop over the values an instruction uses

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

Given %a = %b + %c:
[%b, %c]

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

```
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()
- A value must be explicitly cast to a new type

```
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

```
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

```
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)

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

struct rec *buf;

void foo() {
  buf[5].y = 7;
}
```

```
%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)

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

struct rec *buf;

void foo() {
  buf[5].y = 7;
}
```

```
%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)

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

struct rec *buf;

void foo() {
  buf[5].y = 7;
}
```

```
%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)

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

struct rec *buf;

void foo() {
  buf[5].y = 7;
}
```

```
%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
}
```

Where can you get more information?

- The online documentation is extensive:
  - LLVM Programmer's Manual
  - LLVM Language Reference Manual

Where can you get more information?

- The online documentation is extensive:
  - LLVM Programmer’s Manual
  - LLVM Language Reference Manual
- The header files!
  - All in llvm-12.x.src/include/llvm/

```
BasicBlock.h      InstrTypes.h
DerivedTypes.h    IRBuilder.h
Function.h        Support/InstVisitor.h
Instructions.h    Type.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
  - …

  } Derive from the appropriate base class to make a Pass

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
  - ...

  } Derive from the appropriate base class to make a Pass

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

```
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(CallBase& cb);
};
```

Making a ModulePass (1)

- Declare your ModulePass

```
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(CallBase& cb);
};
```

Making a ModulePass (1)

- Declare your ModulePass

```
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(CallBase& cb);
};
```

Making a ModulePass (2)

- Register your ModulePass
  - This allows it to even be dynamically loaded as a plugin
  - Depending on your use cases, it may not be necessary

```
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()`**

```
bool
StaticCallCounter::runOnModule(Module& m) {
  for (auto& f : m)
    for (auto& bb : f)
      for (auto& i : bb)
        if (CallBase *cb = dyn_cast<CallBase>(&i)) {
          handleInstruction(CallSite{&i});
        }
  return false; // False because we didn't change the Module
}
```

Making a ModulePass (3)

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

```
bool
StaticCallCounter::runOnModule(Module& m) {
  for (auto& f : m)
    for (auto& bb : f)
      for (auto& i : bb)
        if (CallBase *cb = dyn_cast<CallBase>(&i)) {
          handleInstruction(CallSite{&i});
        }
  return false; // False because we didn't change the Module
}
```

Making a ModulePass (3)

- Analysis continued…

```
void
StaticCallCounter::handleInstruction(CallBase* cb) {
  // Check whether the called function is directly invoked
  auto called = cb.getCalledOperand()->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;
}
```

Making a ModulePass (3)

- Analysis continued...

```
void
StaticCallCounter::handleInstruction(CallBase* cb) {
  // Check whether the called function is directly invoked
  auto called = cb.getCalledOperand()->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;
}
```

Making a ModulePass (3)

- Analysis continued…

```
void
StaticCallCounter::handleInstruction(CallBase* cb) {
  // Check whether the called function is directly invoked
  auto called = cb.getCalledOperand()->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;
}
```

Making a ModulePass (3)

- Printing out the results

```
void
CallCounterPass::print(raw_ostream& out, const Module* m) const {
  out << "Function Counts\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 have counted the *static* direct calls to each function.
- How might we count all ***dynamic calls*** to each function?

Making a Dynamic Analysis

- We have counted the *static* direct calls to each function.
- How might we count all ***dynamic calls*** to each function?
- Need to modify the original program!

Making a Dynamic Analysis

- We have counted the *static* direct calls to each function.
- How might we count all ***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?

```
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?

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

```
void foo()
  count[foo]++;
  bar();
}
```

- We'll increment at the function entry.

  (the demo code shows both options)

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();
}
```

⇒

```
void foo()
  count[1]++;
  bar();
}
```

```
0 ↦ main
1 ↦ foo
2 ↦ 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?

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

⇒

```
void foo()
  countCall(1);
  bar();
}
```

```
void
countCall(id)
  count[id]++;
}
```

- We'll increment at the function entry.
  - *Using numeric IDs* for functions is sometimes easier
  - *Inserting function calls* is easier than adding raw instructions

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();
}
```

⇒

```
void foo()
  countCall(1);
  bar();
}
```

```
void countCall(id)
  count[id]++;
}
```

- We'll increment at the function entry.
  - *Using numeric IDs* for functions is sometimes easier
  - ***Inserting function calls*** is easier than adding raw instructions
    - Add new definitions to the original code
    - Link against an ***instrumentation library***

Modifying the Original Program

- What might adding this call look like?

```
void
DynamicCallCounter::handleInstruction(CallBase& cb, Value* counter) {
  // Check whether the called function is directly invoked
  auto calledValue    = cb.getCalledOperation()->stripPointerCasts();
  auto calledFunction = dyn_cast<Function>(calledValue);
  if (!calledFunction) {
    return;
  }

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

Modifying the Original Program

- What might adding this call look like?

```
void
DynamicCallCounter::handleInstruction(CallBase& cb, Value* counter) {
  // Check whether the called function is directly invoked
  auto calledValue    = cb.getCalledOperation()->stripPointerCasts();
  auto calledFunction = dyn_cast<Function>(calledValue);
  if (!calledFunction) {
    return;
  }

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

Modifying the Original Program

- What might adding this call look like?

```
void
DynamicCallCounter::handleInstruction(CallBase& cb, Value* counter) {
  // Check whether the called function is directly invoked
  auto calledValue    = cb.getCalledOperation()->stripPointerCasts();
  auto calledFunction = dyn_cast<Function>(calledValue);
  if (!calledFunction) {
    return;
  }

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

In practice, it is more complex.
You can find details in the demo code.

Using a Runtime Library

- Recall that the definition of **`countCall()`** needs to live somewhere
  1) Add directly to the modified code
  2) Implemented separately & linked in via a library

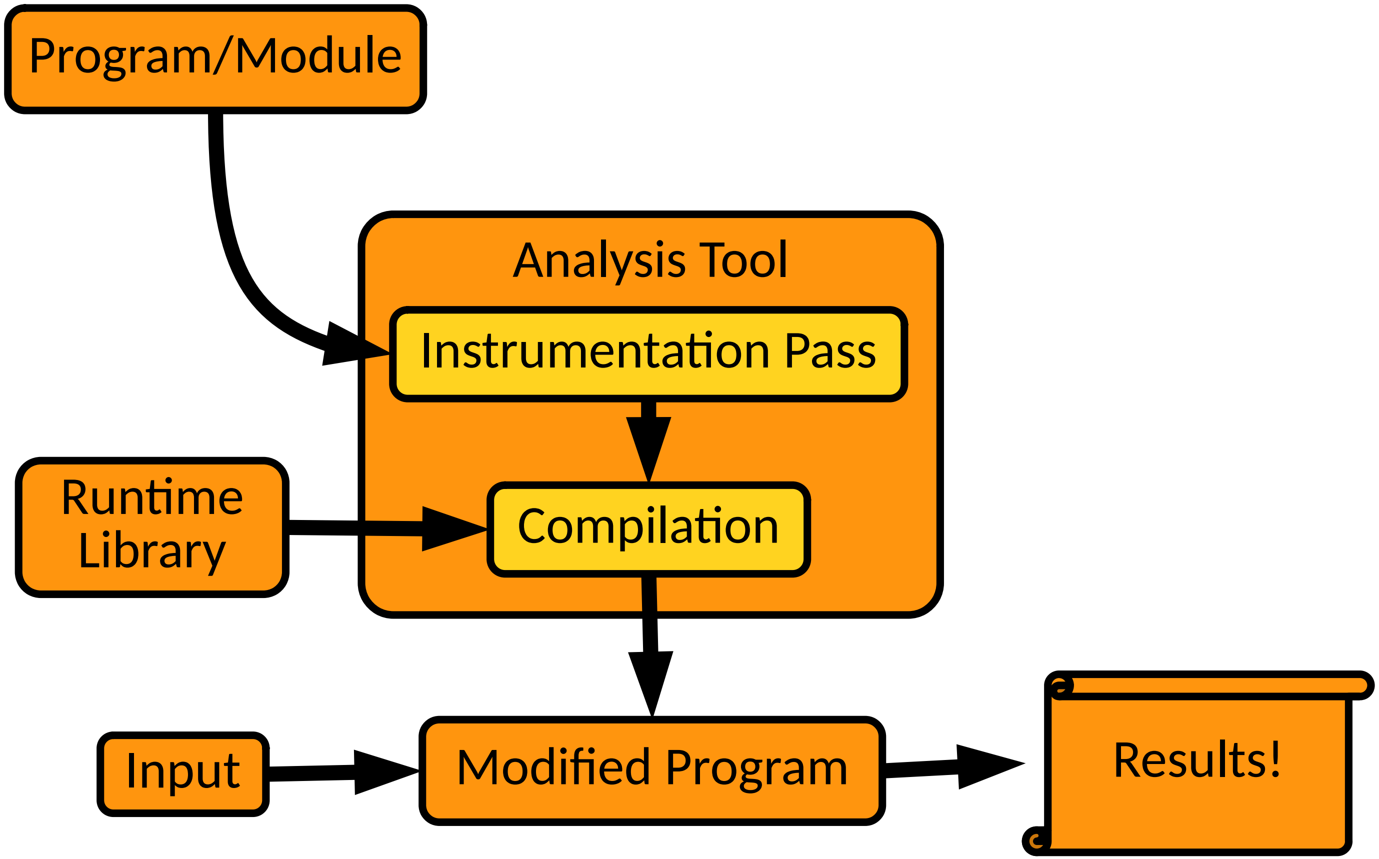
What trade offs do you see?

Using a Runtime Library

- Recall that the definition of `countCall()` needs to live somewhere
  1) Add directly to the modified code
  2) Implemented separately & linked in via a library
- In practice, linking against a library is common, easy, & powerful
  - Regardless of the language being analyzed

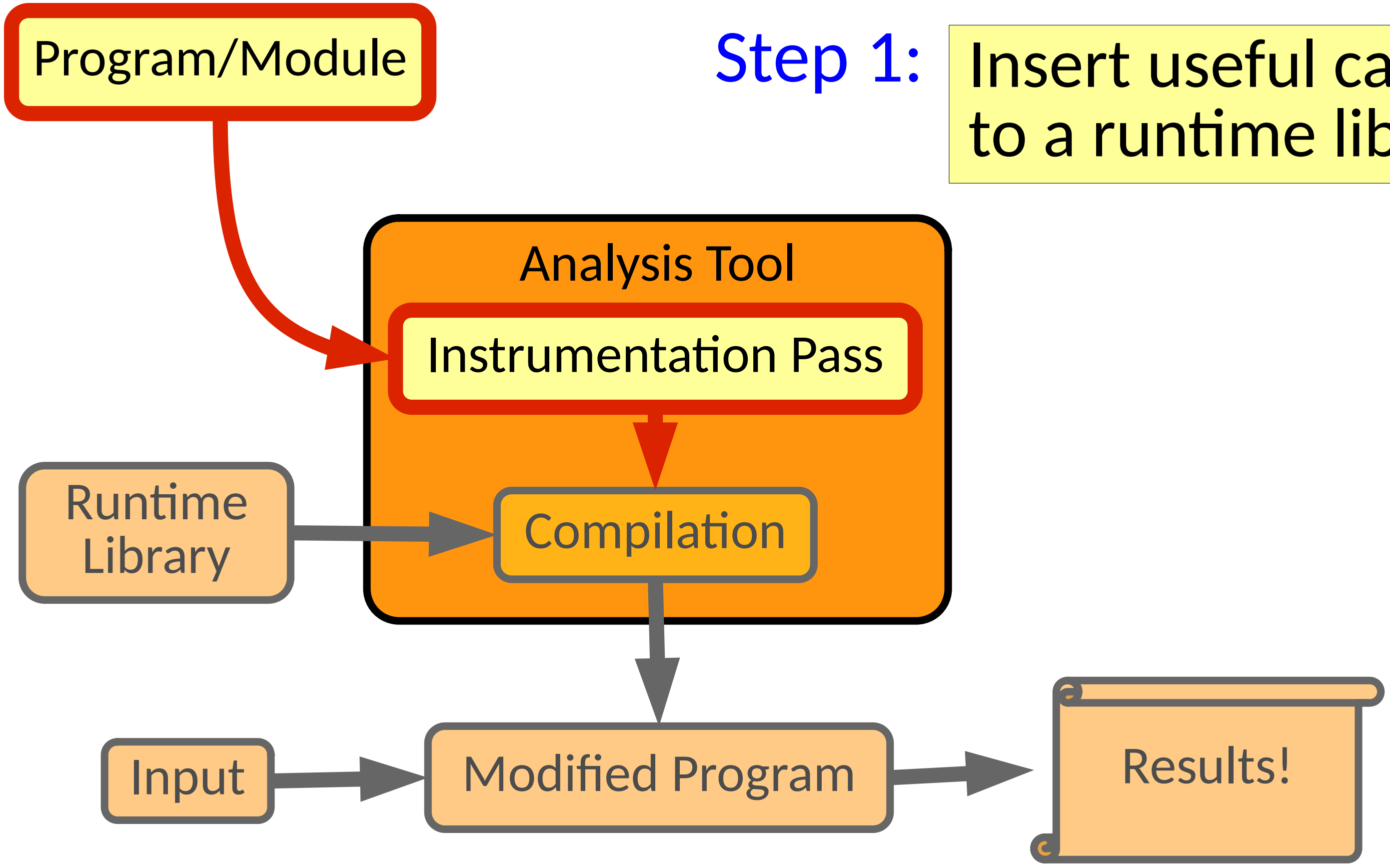
```
void
countCalled(uint64_t id) {
  ++functionInfo[id];
}
```

Revisiting the Big Picture of Dynamic Analysis



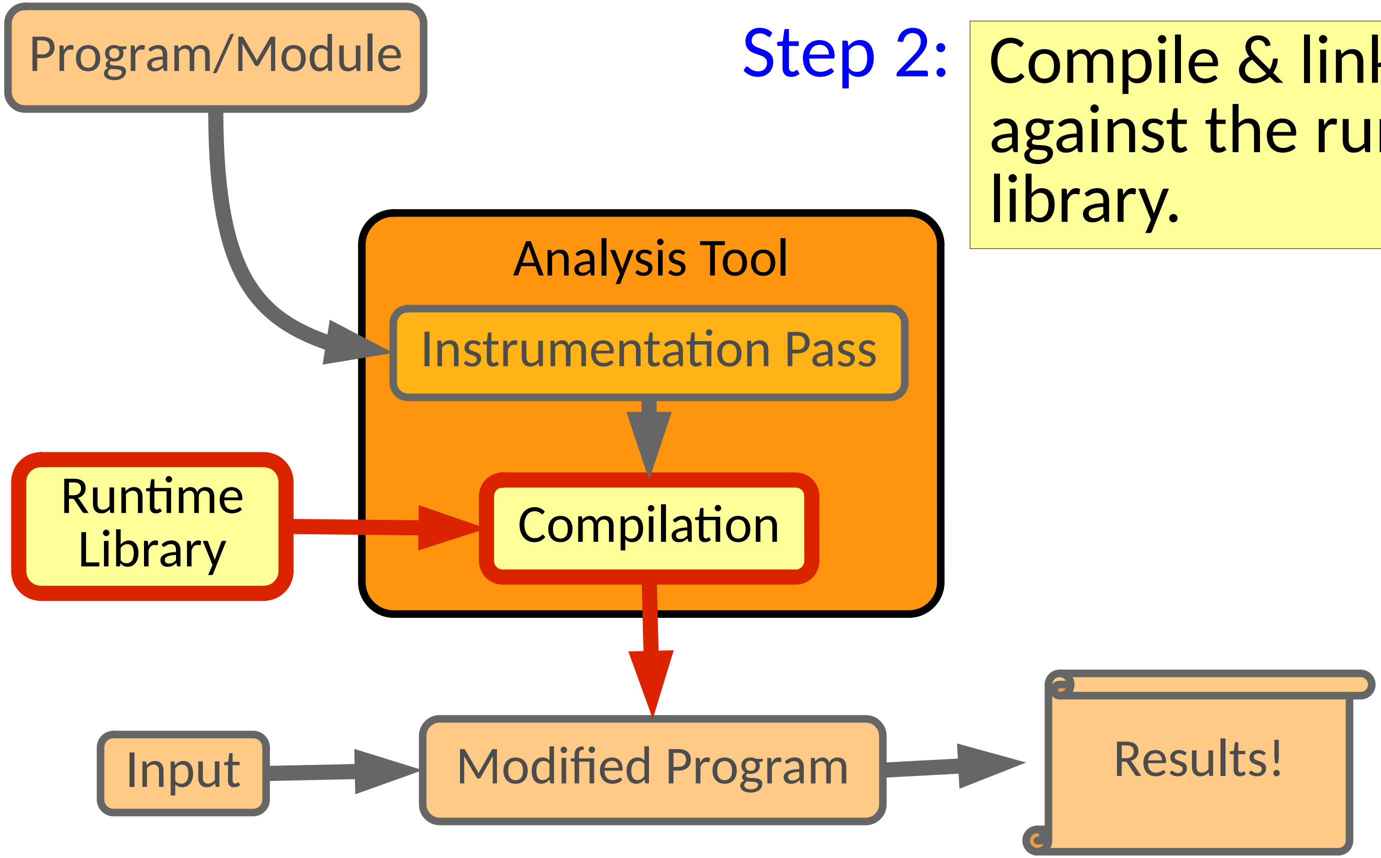
Revisiting the Big Picture of Dynamic Analysis

Step 1: Insert useful calls to a runtime library



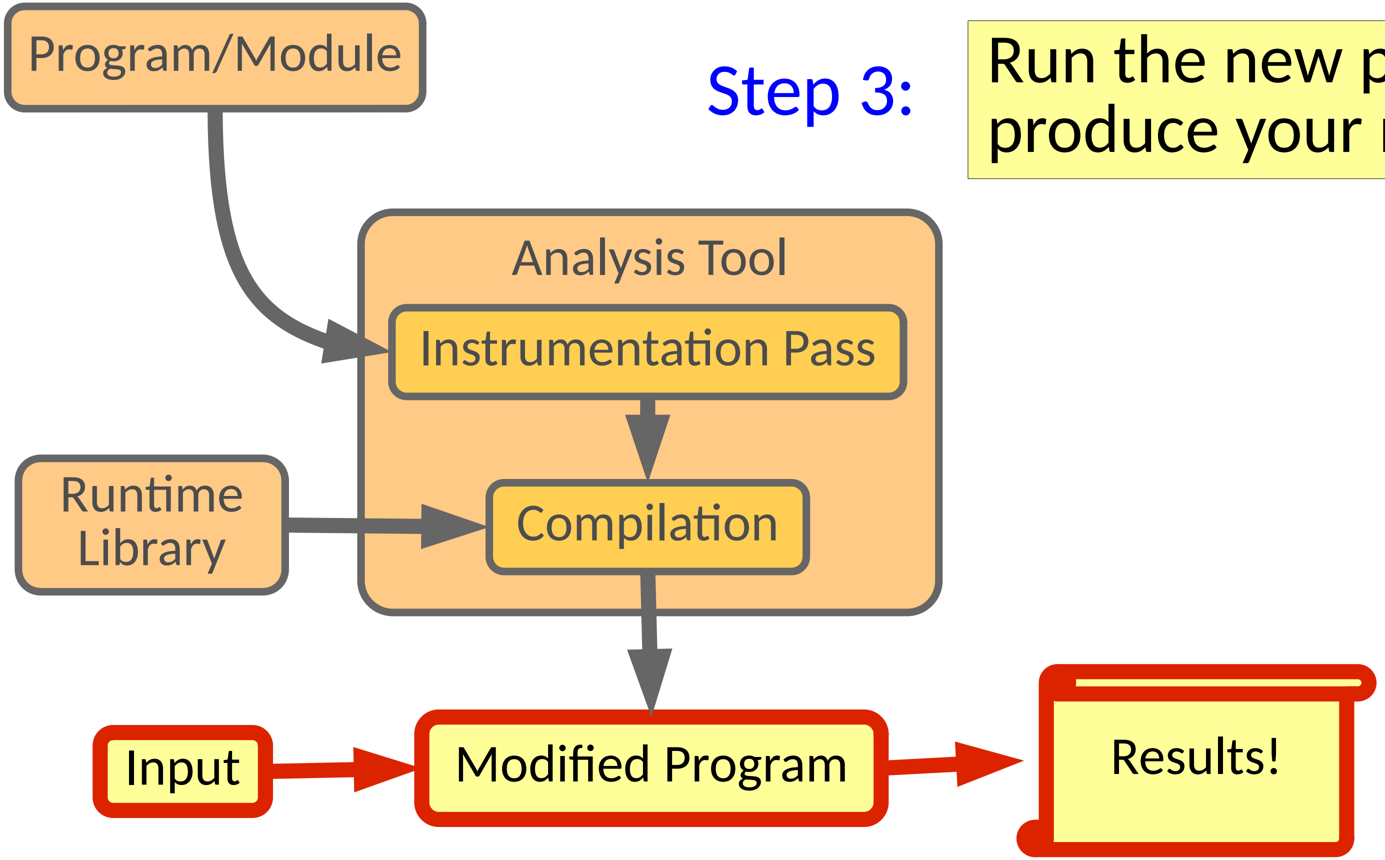
Revisiting the Big Picture of Dynamic Analysis

Step 2: Compile & link against the runtime library.



Revisiting the Big Picture of Dynamic Analysis

Step 3: Run the new program to produce your results



Summary

- 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 the project description