LLVM Passes

Nick Sumner

(see also https://github.com/nsumner/llvm-demo)

Matt Dwyer

(see also https://github.com/matthewbdwyer/tipc/)

Where Can You Get Info?

- The online documentation is extensive:
  - LLVM Programmer’s Manual
  - LLVM Language Reference 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-9.x.src/include/llvm/

BasicBlock.h
CallSite.h
DerivedTypes.h
Function.h
Instructions.h
InstrTypes.h
IRBuilder.h
Support/InstVisitor.h
Type.h

Where Can You Get Info?

The discussion in these slides is accompanied by:

- https://github.com/nsumner/llvm-demo

Another very good resource is:

- http://llvm.org/docs/WritingAnLLVMPass.html

An example of using passes is:

- https://github.com/matthewbdwyer/tipc/

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(CallSite cs);
};
```

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(CallSite cs);
};
```

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(CallSite cs);
};
```

Making a ModulePass (2)

- Register your ModulePass
  - This allows it to by dynamically loaded as a plugin

```
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)
        handleInstruction(CallSite{&i});
  return false; // False because we didn't change the Module
}
```

Making a ModulePass (3)

- analysis continued...

```
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;
}
```

Making a ModulePass (3)

- analysis continued...

```
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;
}
```

Making a ModulePass (3)

- analysis continued...

```
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;
}
```

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'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!

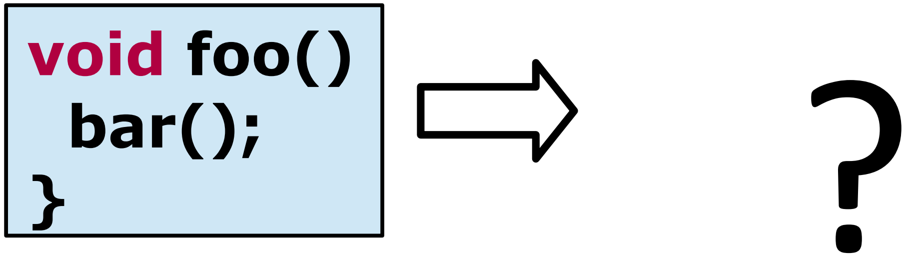
Making a Dynamic Analysis

- 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?



```
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?

```
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?

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

?

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?

```
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?

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

⇨ ?

Keep a counter for each function!

2 Choices:

1) increment count for each function *as it starts*

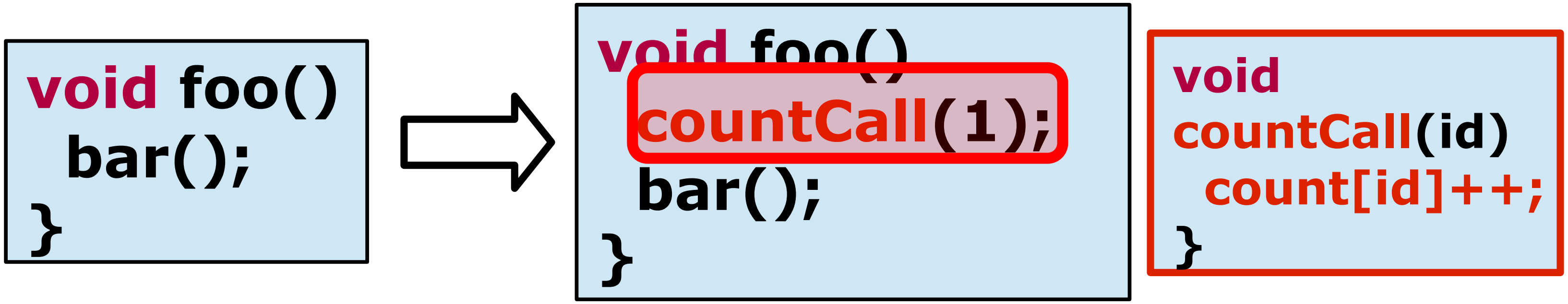
2) increment count for each function *at its call site*

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

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

We'll increment at the function entry with a call

Modifying the Original Program

What might adding this call look like?

```
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?

```
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?

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

  // Chec
  auto ca
  auto ca
  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 llvm-demo code.

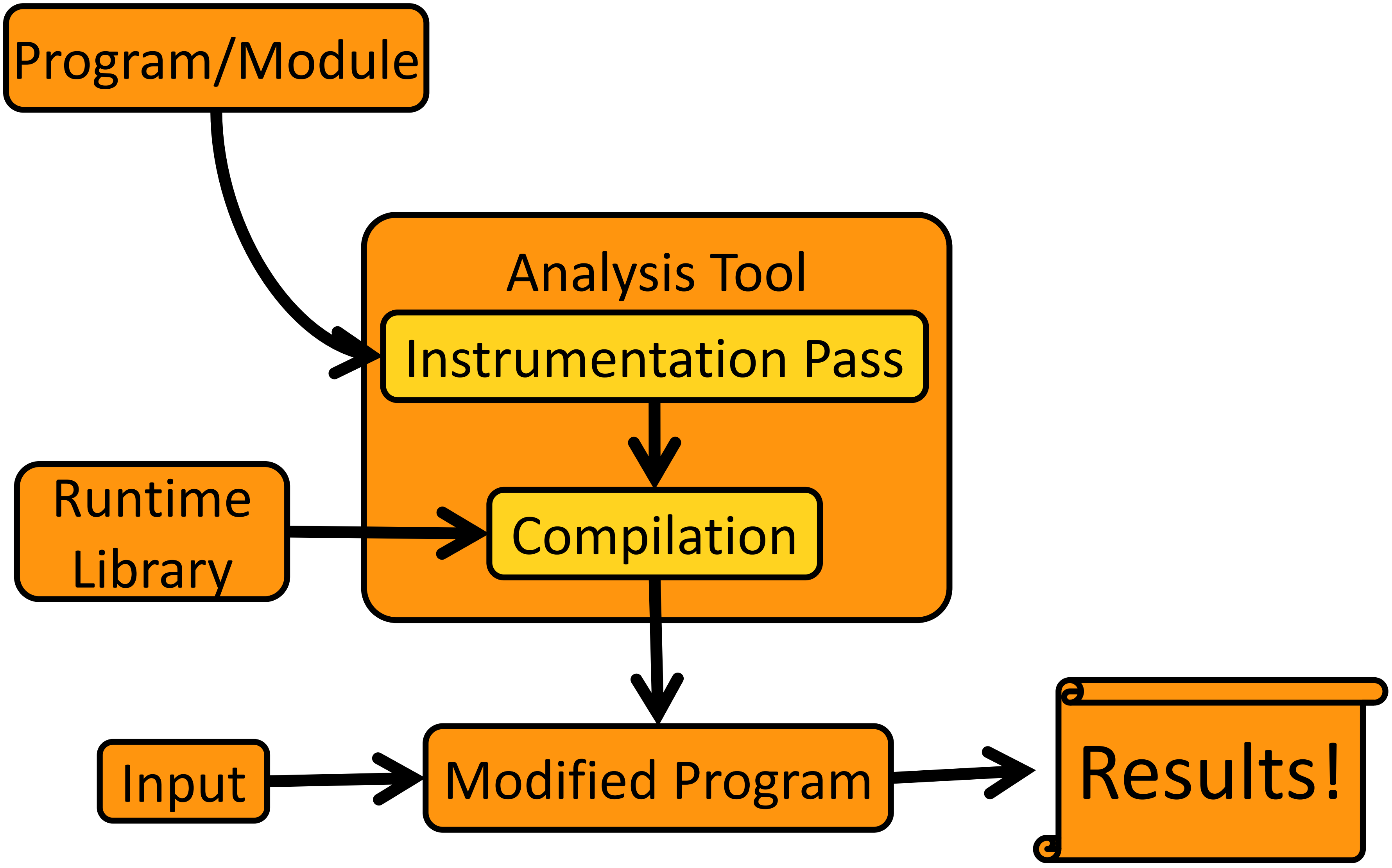
Using a Runtime Library

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

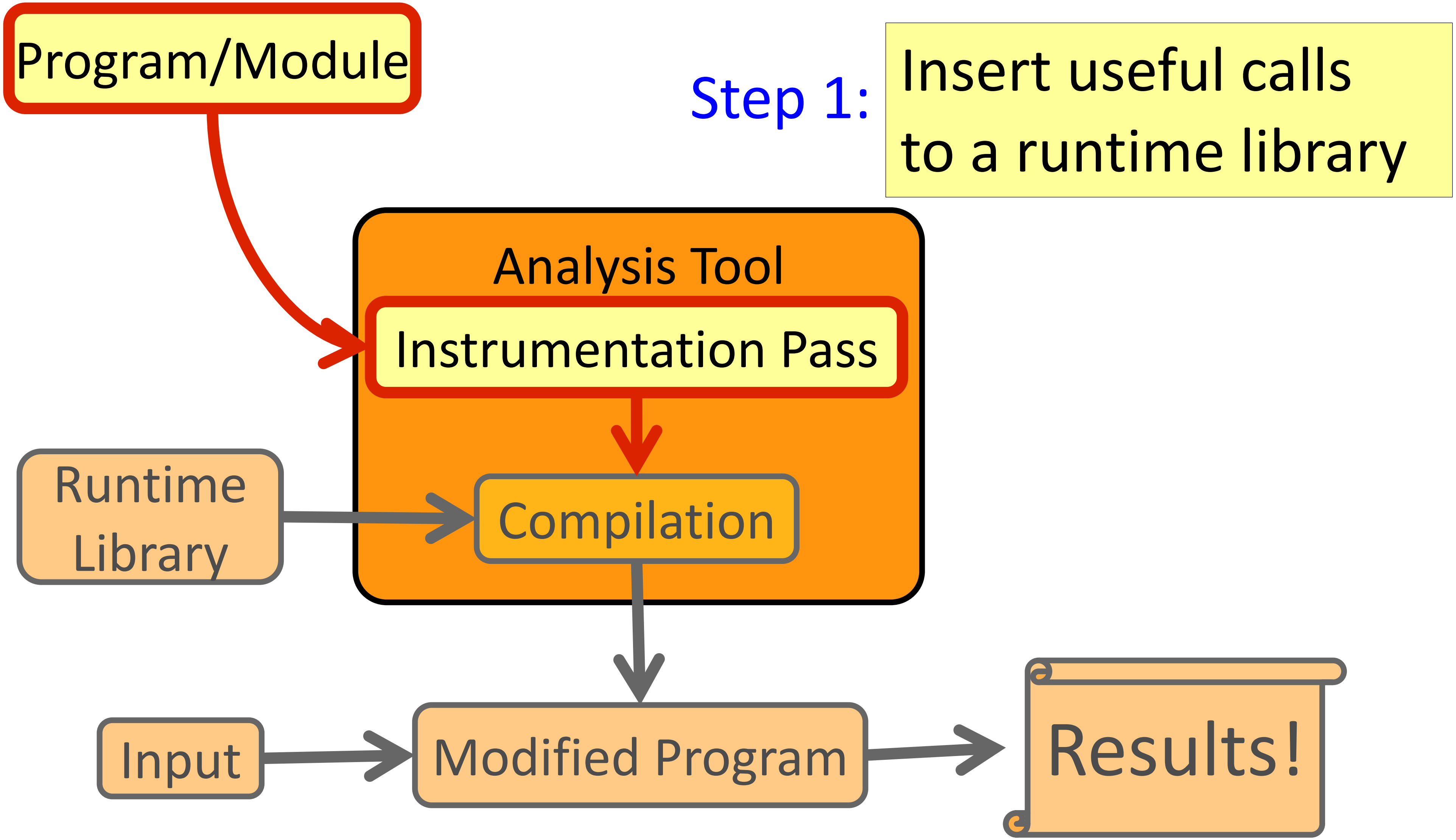
- Placed in a library linked with the main executable

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

Dynamic Analysis Big Picture

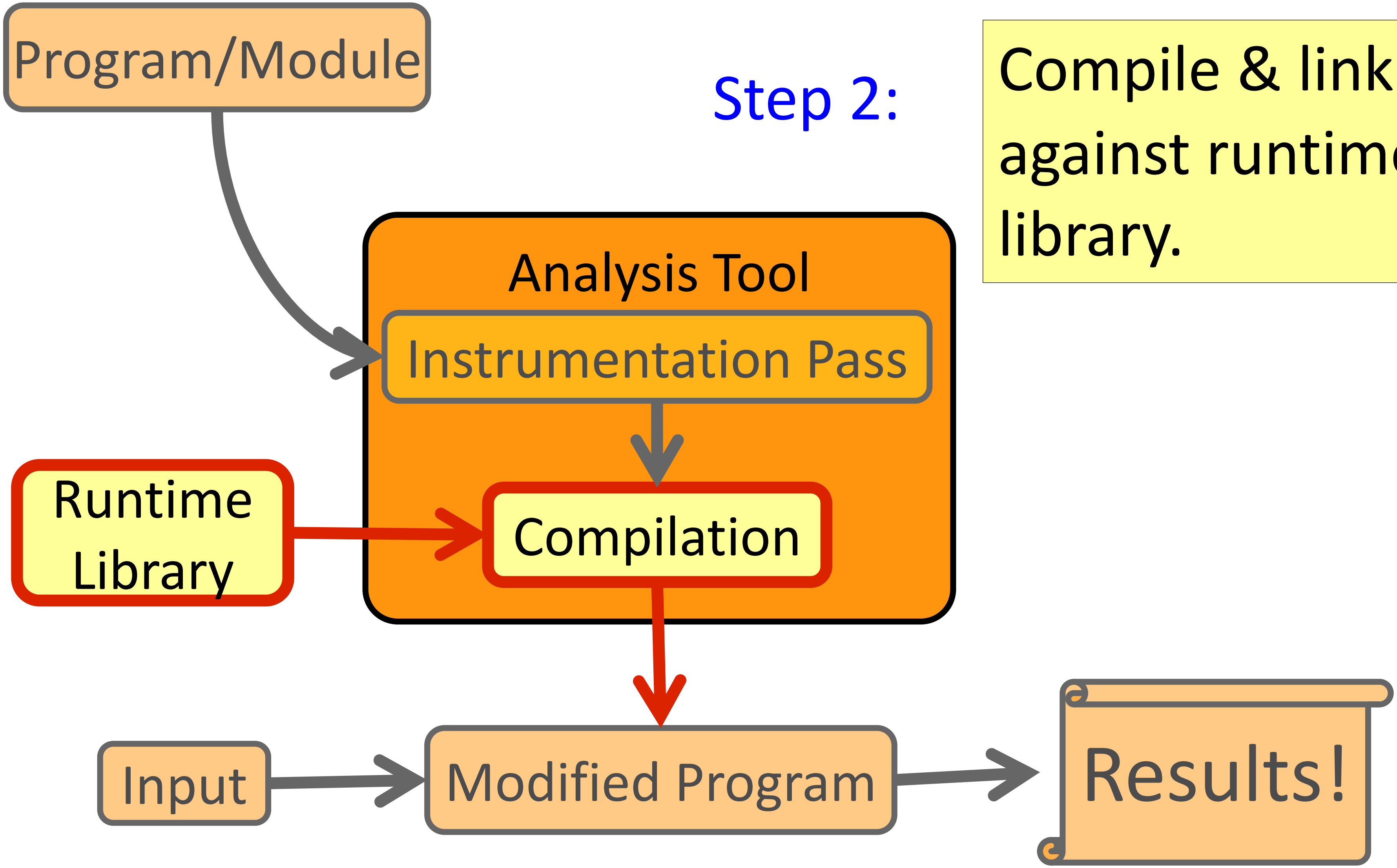


Dynamic Analysis Big Picture



Step 1: Insert useful calls to a runtime library

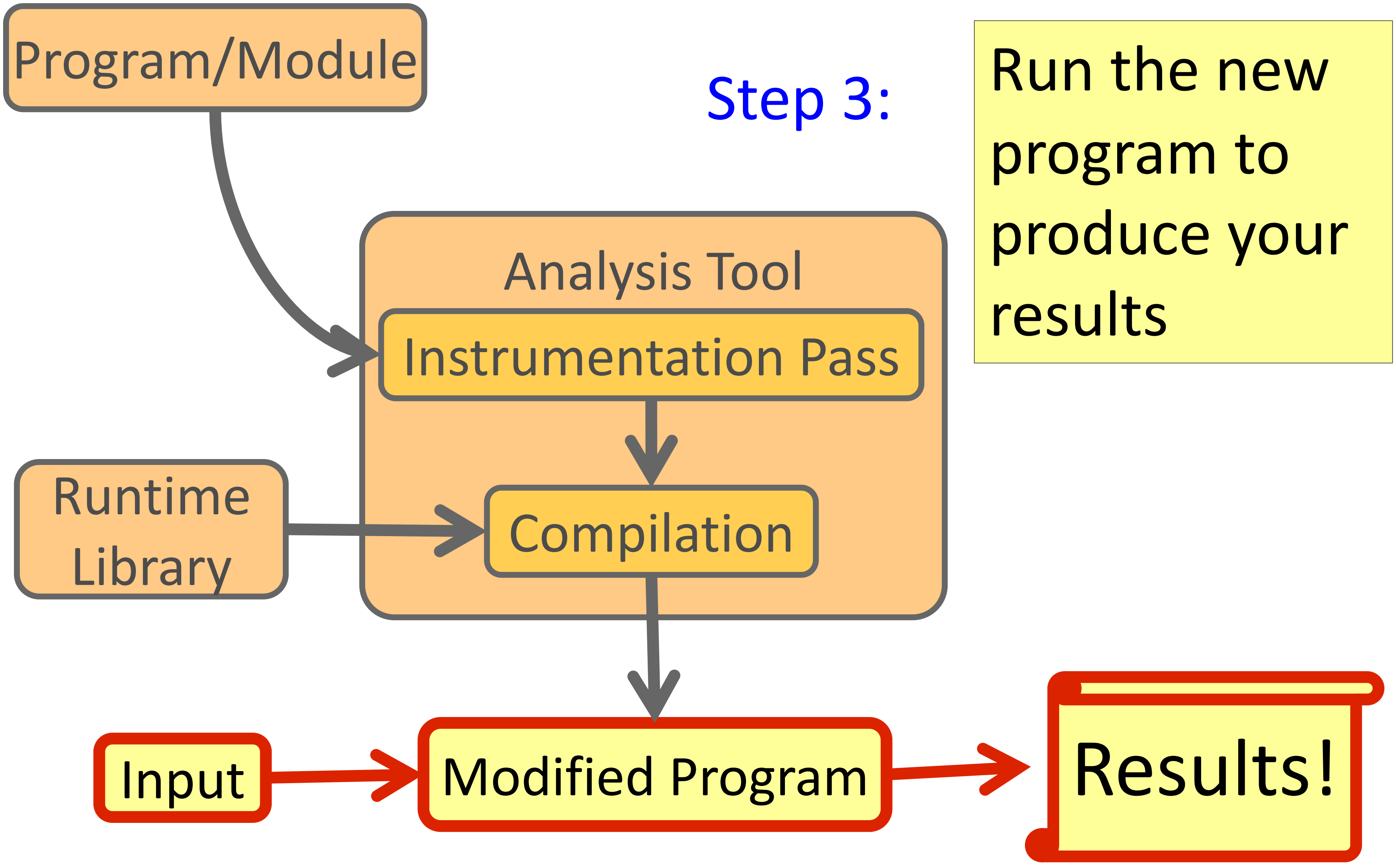
Dynamic Analysis Big Picture



Step 2:

Compile & link against runtime library.

Dynamic Analysis Big Picture



Program/Module

Analysis Tool

Instrumentation Pass

Runtime Library

Compilation

Input

Modified Program

Results!

Step 3:

Run the new program to produce your results

A Roadmap to LLVM Passes

LLVM has a rich set of passes available for you to study

- Clone LLVM source tree

…/llvm/lib/Passes/PassRegistry.def

- A file that registers all core passes in LLVM
- A good reference to see what is available to study

…/llvm/lib/Analysis

- These are the analysis passes

…/llvm/lib/Transforms

- These are the transformation passes
- /Hello gives a “hello world pass”

Explore some LLVM Passes

After developing a basic familiarity with LLVM passes

Explore …/llvm/lib/Transforms/Scalar/

- ConstantProp, DCE

Explore the passes used in the tipc compiler

As you will see the SSA form is key to LLVM

- We will discuss it in some detail in the coming weeks