LLVM Tutorial

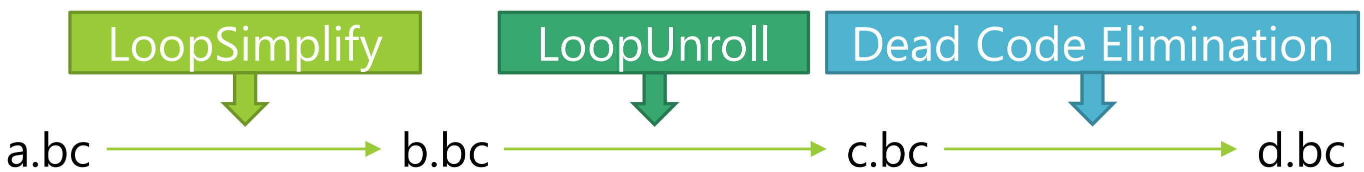
IR Optimization (Part 1)

2019. 04. 10

IR Optimization

- LLVM enables modular optimizations through the LLVM pass framework

- Each **LLVM pass** performs optimizations and transformations on LLVM IR
  - Example



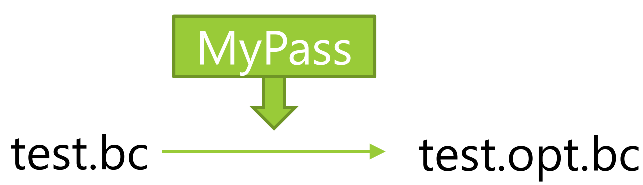
LLVM Pass

- C++ class that inherits the "`Pass`" class in LLVM
  - Implement functionality by overriding virtual methods

    e.g. `runOnModule` or `runOnFunction`

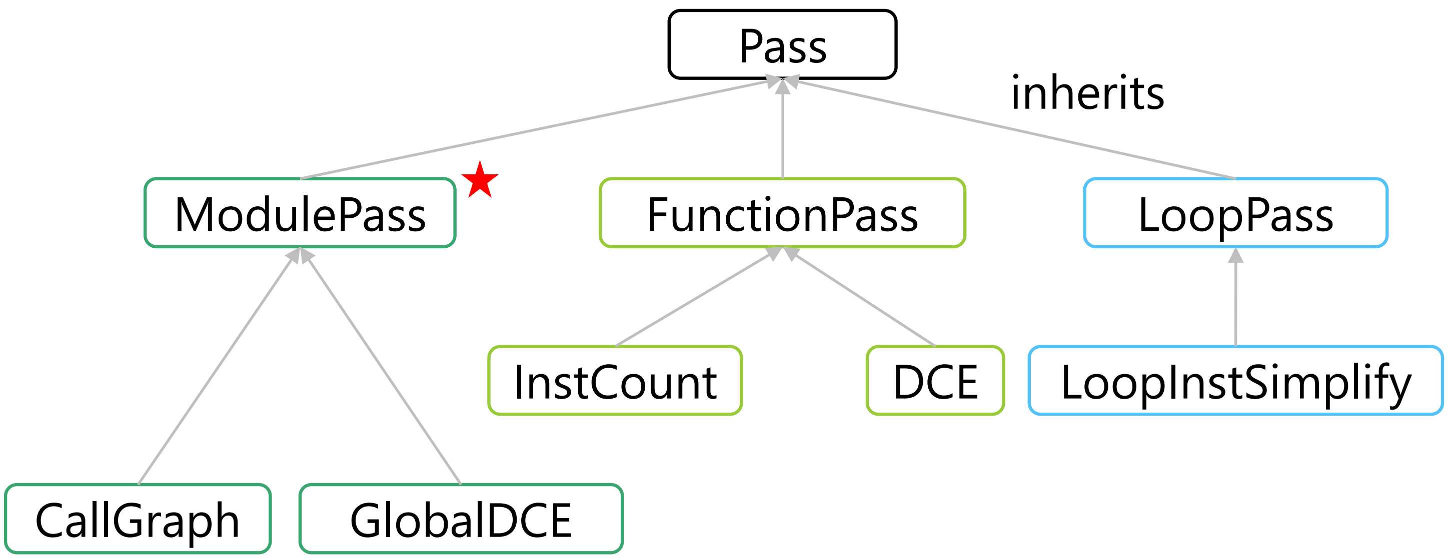
- Dynamically loaded at run-time
  - `opt -load PASS_LIBRARY_PATH -PASS_NAME`
  - `example`

```
$ opt –load ~/lib/MyPass.so –MyPass test.bc –o test.opt.bc
```



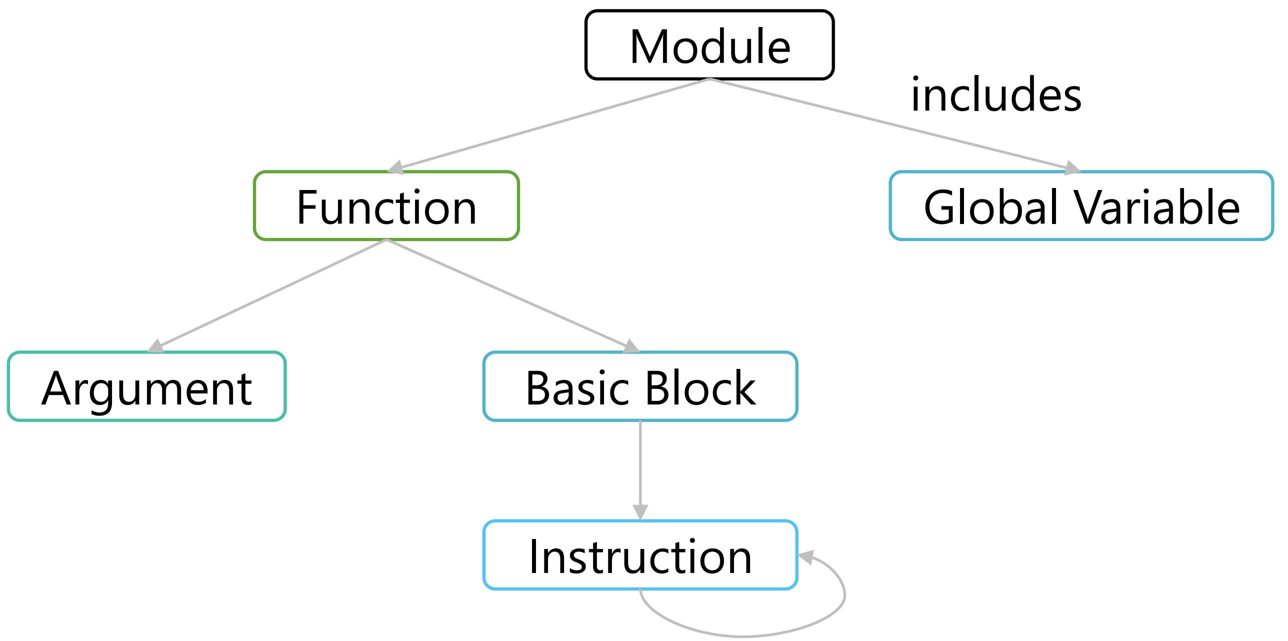
LLVM Pass Classes

- Is-A relationship of LLVM Pass classes
  - `xxxPass`: `xxx` is the unit of optimization



LLVM IR Classes

- Has-a relationship of LLVM IR classes



Skeleton Code: ModulePass

- Header File (`HelloModule.h`)

```
#include "llvm/IR/Module.h"
#include "llvm/Pass.h"

using namespace llvm;

namespace {
  struct HelloModule : public ModulePass {
    static char ID; // Pass identification, replacement for typeid
    HelloModule() : ModulePass(ID) {}

    bool runOnModule(Module &M) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override;
  };
}
```

Skeleton Code: ModulePass

- Source File (`HelloModule.cpp`)

```
#include "HelloModule.h"

#define DEBUG_TYPE "hello"

bool HelloModule::runOnModule(Module &M) {
  return false;
}

void HelloModule::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
}

char HelloModule::ID = 0;
static RegisterPass<HelloModule> X("helloModule", "Hello World Pass ");
```

Do Something to analyze or optimize code

Skeleton Code: FunctionPass

- Header File

```
#include "llvm/IR/Function.h"
#include "llvm/Pass.h"

using namespace llvm;

namespace {
  struct HelloFunction : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    HelloFunction() : FunctionPass(ID) {}

    bool runOnFunction(Function &M) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override;
  };
}
```

How to Run LLVM Pass

Automatically generate compile options

1) Compile LLVM Passes

```
$ clang++ -c -fpic -fno-rtti `llvm-config --cppflags` HelloModule.cpp -o HelloModule.o
```

2) Make a shared library with the LLVM passes

```
$ clang++ -shared -o Hello.so HelloModule.o HelloFunction.o
```

2) Run the LLVM Passes using opt

```
$ opt -load Hello.so -helloModule test.bc -o test.opt.bc
```

Practice 1: First LLVM Pass

- Goal
  - Learn how to write, compile and run passes

- Steps

1) Implement a NamePrinter pass that inherits `FunctionPass`
   - Print "Hello " and the function name
     - Tip 1: To print a debug message, use
       - `dbgs() << "Message"`
     - Tip 2: To get a function name, use
       - F.getName()
2) Compile and test the pass

IR Code Analysis

- Use the member functions of IR Classes!

- References
  - Doxygen
    - http://llvm.org/doxygen/
  - Existing LLVM Passes
    - Want to the usage of a function
    - Find the function call in llvm/lib/Analysis or llvm/lib/Transforms

IR Code Analysis

- class `Module`
  - https://llvm.org/doxygen/classllvm_1_1Module.html

llvm::Module Class Reference

A **Module** instance is used to store all the information related to an LLVM module. More...

```
#include "llvm/IR/Module.h"
```

Function *	**getFunction** (**StringRef Name**) **const** Look up the specified function in the module symbol table. More...
GlobalVariable *	**getGlobalVariable** (**StringRef Name**) **const** Look up the specified global variable in the module symbol table. More...
const DataLayout &	**getDataLayout** () **const** Get the data layout for the module's target platform. More...

IR Code Analysis

- class Function
  - http://llvm.org/doxygen/classllvm_1_1Function.html

llvm::Function Class Reference

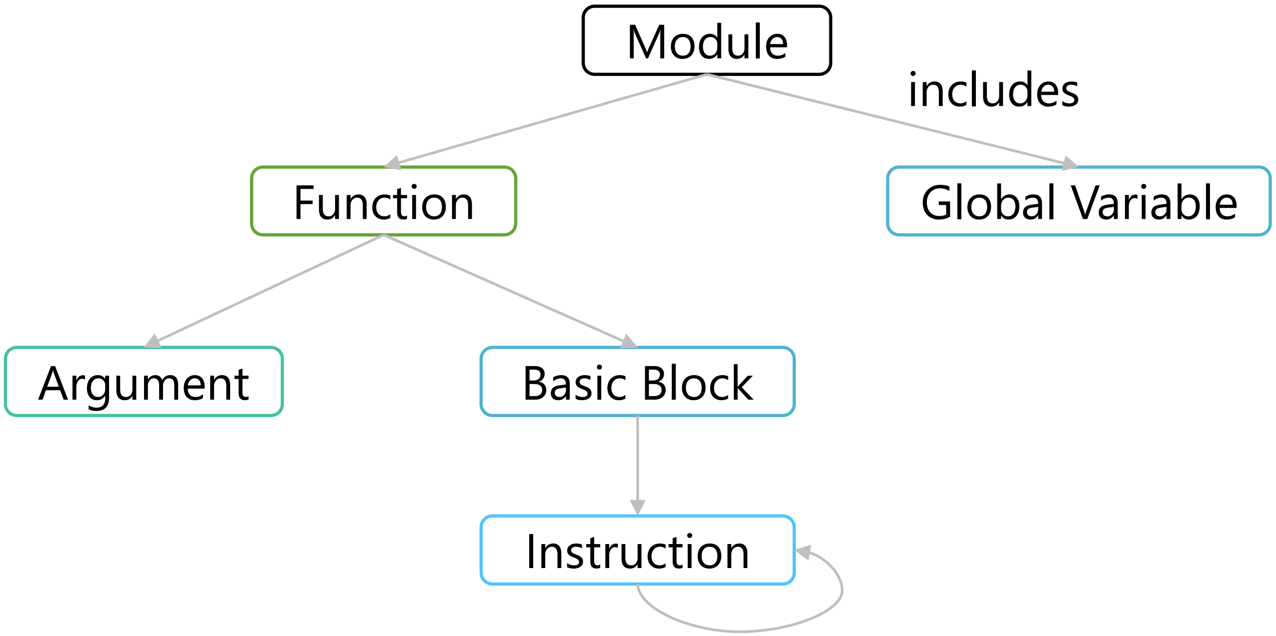
```
#include "llvm/IR/Function.h"
```

FunctionType *	getFunctionType () const Returns the FunctionType for me. More...
Type *	getReturnType () const Returns the type of the ret val. More...

BasicBlock &	getEntryBlock ()

LLVM IR Classes

- Has-a relationship of LLVM IR classes



Useful Basic Iterators

- class `Module`

using	iterator = FunctionListType::iterator The **Function** iterators. More...
using	const_iterator = FunctionListType::const_iterator The **Function** constant iterator. More...

iterator	begin ()
const_iterator	begin () const
iterator	end ()
const_iterator	end () const

- class `Function`

using	iterator = BasicBlockListType::iterator
using	const_iterator = BasicBlockListType::const_iterator

Useful Basic Iterators

- class `BasicBlock`

using	**iterator** = **InstListType::iterator** **Instruction** iterators... More...
using	**const_iterator** = **InstListType::const_iterator**

Useful Basic Iterators

- Example 1
  - Iterate functions with a Module object

```
for(Function &F : M) {
  // Do something with F
}
```

```
Module::iterator Begin = M.begin();
Module::iterator End = M.end();
for (Module::iterator it = Begin; it != End; ++it) {
  Function &F = *it;
  // Do something with F
}
```

Useful Basic Iterators

- Example 2
  - Iterate instructions with a Module object

```
for(Function &F : M) {
  for(BasicBlock &BB : F) {
    for(Instruction &I : BB) {
      // Do something with I
    }
  }
}
```

Other Iterators

- class `Module`

using	global_iterator = GlobalListType::iterator The Global Variable iterator. More...
using	const_global_iterator = GlobalListType::const_iterator The Global Variable constant iterator. More...

Global Variable Iteration

global_iterator	global_begin ()
const_global_iterator	global_begin () const
global_iterator	global_end ()
const_global_iterator	global_end () const
bool	global_empty () const
iterator_range< global_iterator >	globals ()
iterator_range< const_global_iterator >	globals () const

Other Iterators

- class `Function`

using	arg_iterator = Argument *
using	const_arg_iterator = const Argument *

Function Argument Iteration

arg_iterator	arg_begin ()
const_arg_iterator	arg_begin () const
arg_iterator	arg_end ()
const_arg_iterator	arg_end () const
iterator_range< arg_iterator >	args ()
iterator_range< const_arg_iterator >	args () const

Useful Basic Iterators

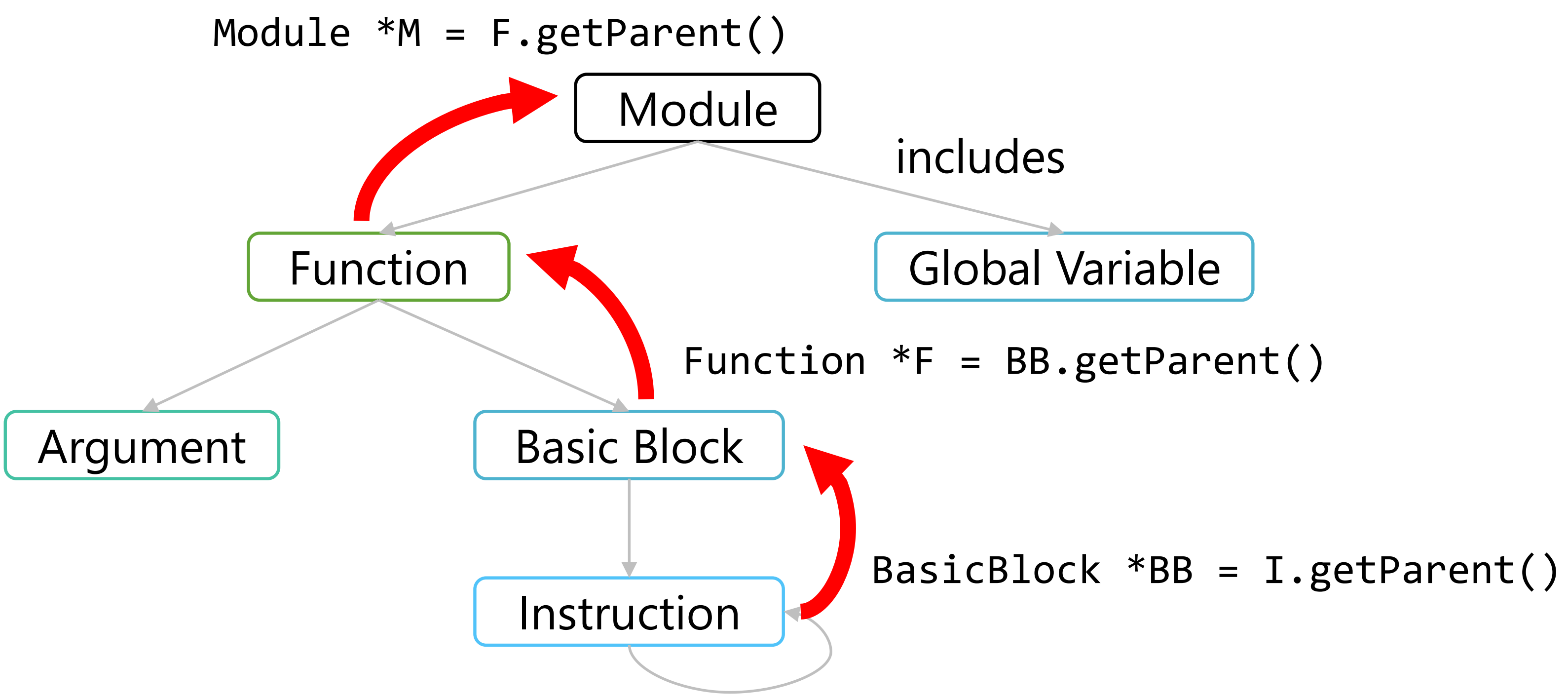
- Example 3
  - Iterate function arguments with a Function object

```
for(Argument *Arg : F.args()) {
  // Do something with Arg
}
```

```
Function::arg_iterator Begin = F.arg_begin();
Function::arg_iterator End = F.arg_end();
for (Function::arg_iterator it = Begin; it != End; ++it) {
  Argument *Arg = *it;
  // Do something with Arg
}
```

Get Parent Instance

- Has-A relationship of LLVM IR classes



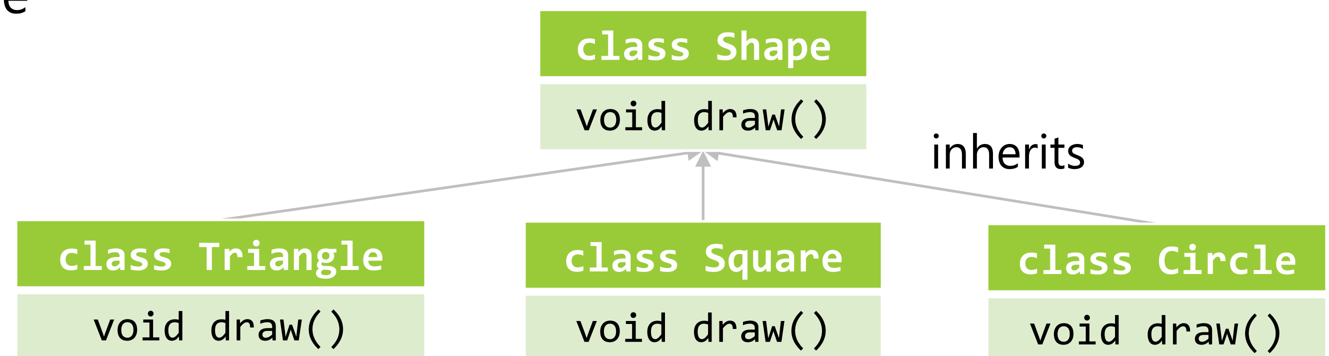
Practice 2: (Static) InstCount

- Goal
  - Learn how to write static analysis pass

- Steps
  1) Impelement a InstCount pass that inherits `FunctionPass`
     - Count the number of instructions in a function
     - Print the function name and the number of instructions
  2) Compile and test the pass

Dynamic Type Casting

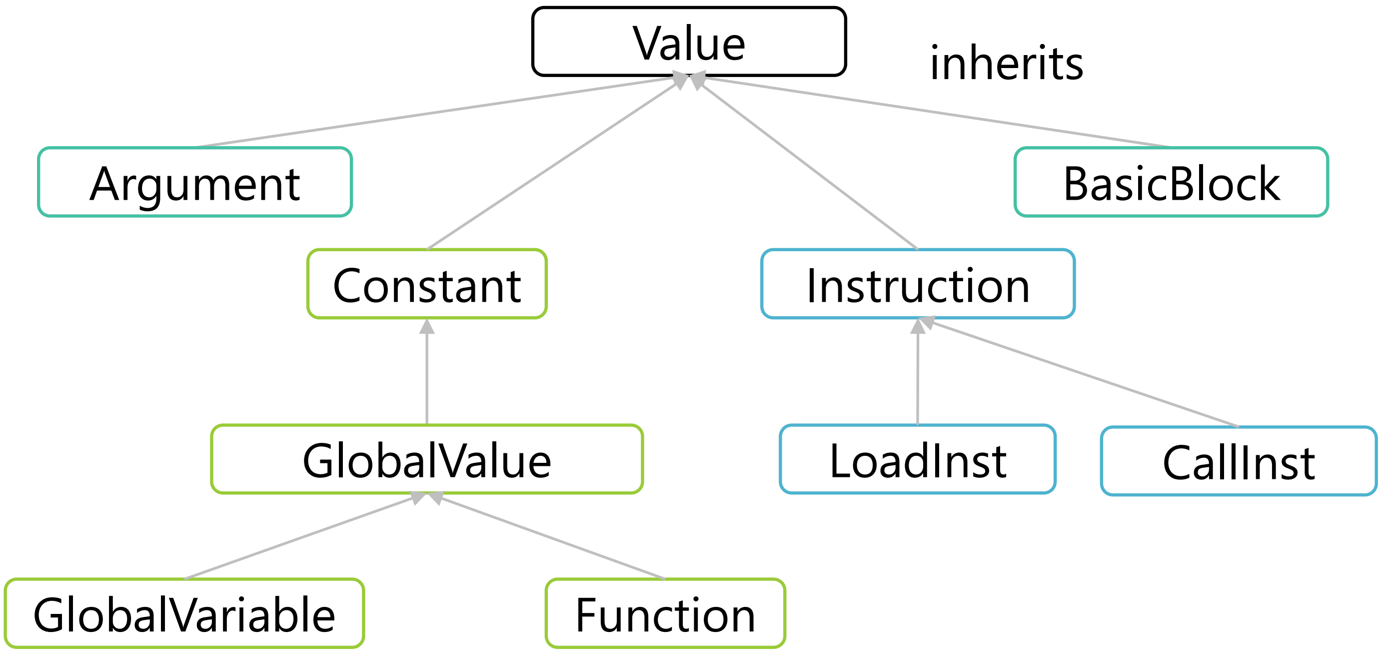
- **Polymorphism** in object-oriented programming
  - A <u>super class type</u> pointer can points a <u>subclass type</u> instanc
e




```
Shape *shape = nullptr;
if (arg == 3) shape = new Triangle();
else if (arg == 4) shape = new Square();
else shape = new Circle();
shape.draw();
```

LLVM IR Classes

- Is-A relationship of IR classes



Dynamic Type Casting

- How can distinguish the type of instructions?

```
for(Function &F : M) {
  for(BasicBlock &BB : F) {
    for(Instruction &I : BB) {
      // Do something with I
    }
  }
}
```

Dynamic Type Casting

- Use C++ operators that support polymorphism
  - Check the type of an instance that a pointer points
    - `isa<Type>`
    - Example

```
if(isa<CallInst>(&I)) {
}
```

  - Cast to the subclass type of a pointer
    - `dyn_cast<Type>`
    - Example

```
CallInst* CI = dyn_cast<CallInst>(&I);
```

```
if(CallInst* CI = dyn_cast<CallInst>(&I)){
}
```

Practice 3: CallInstCount Pass

- Goal
  - Understand runtime types

- Result
  1) Implement a CallInstCount pass that inherits `FunctionPass`
     - Count the number of call instructions in a function
     - Print the function name and the number of call instructions
  2) Compile and test the pass

Interact with Other Passes

- Passes are **dependent** with each other
  - `opt --debug-pass=Structure` shows the dependence relations
  - Example

```
opt --debug-pass=Structure -reg2mem test.bc -o test.opt.bc
```

```
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -break-crit-edges
-reg2mem -verify -write-bitcode
Target Library Information
Target Transform Information
Target Pass Configuration
  ModulePass Manager
    FunctionPass Manager
      Break critical edges in CFG
      Demote all values to stack slots
      Module Verifier
    Bitcode Writer
```

Interact with Other Passes

1) Include the header file of another pass

```
#include "llvm/Analysis/LoopInfo.h"
```

2) Call `addRequired` in `getAnalysisUsage`

```
void Hello::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.addRequired< LoopInfoWrapperPass >();
  AU.setPreservesAll();
}
```

Interact with Other Passes

3) Bring the analysis result of the pass

```
LoopInfo &LI =
getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
```

- Note
  - ModulePass brings FunctionPass

```
getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
```

  - FunctionPass brings FunctionPass

```
getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```

Practice 4: Loop Analysis Pass

- Goal
  - Learn how to get analysis results from other passes

- Steps
  1) Get LoopInfo from LoopInfoWrapperPass
  2) Print the information about loops
     - The number of loops, the depth of a loop, …
     - Refer to http://llvm.org/doxygen/classllvm_1_1Loop.html
  3) Compile and run the pass

Backup Slides