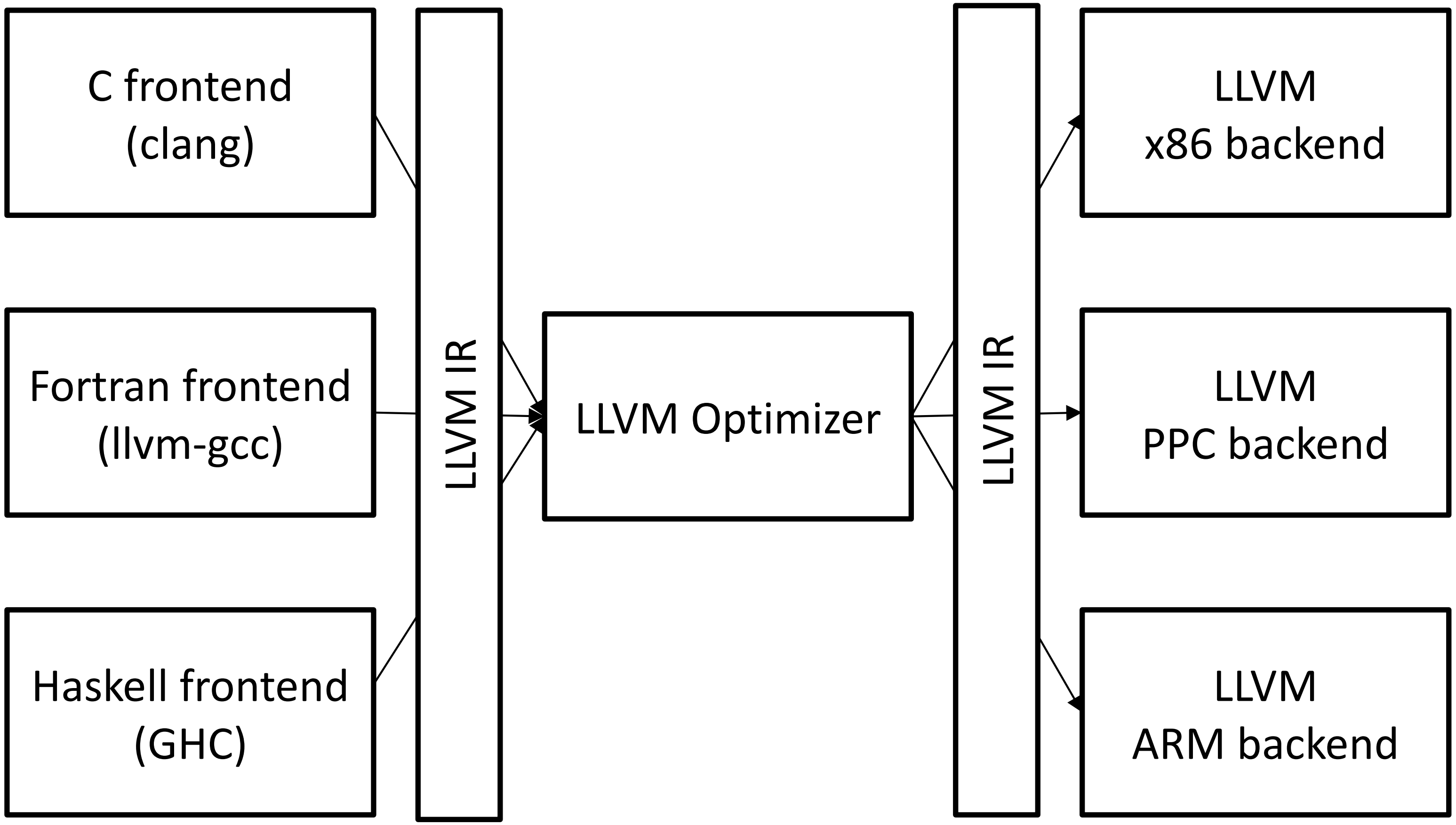
CS451 – Software Analysis

Lecture 21

Developing an LLVM Pass

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

Compiler design with LLVM



LLVM pipeline

- The frontend (e.g., clang) transforms high-level code (e.g., C) to LLVM IR
- The optimizer applies several passes that transform an existing IR to a new, optimized, IR
  - Several analysis tasks can be carried out, in this phase

```
$ bin/opt -print-passes |wc -l
328
```

- The backend transforms the optimized IR to native code

Our first pass

- We will develop a first pass that is very simple
  - The pass should print all the function names that are compiled
- The code of the pass will be written in C++
- Once the pass is compiled, we will be able to use the pass when we compile new source using clang

Pass files

- The pass is in `llvm/lib/Transformations` and in the folder `PrintFunctions`

```
(llvm/lib/Transforms/PrintFunctions)$ ls
CMakeLists.txt  PrintFunctions.cpp
```

CMakeLists.txt

- This file is used by the build system
  - We define the name of the shared library that implements the pass
  - It may include dependencies for more complicated projects

```
add_llvm_library(libLLVMPrintFunctions MODULE
    PrintFunctions.cpp
)
```

- You need to modify `CMakeLists.txt` of `Transforms/` and add

```
add_subdirectory(PrintFunctions)
```

Pass source - Initialization

```
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;
```

Print all function names

```
namespace {

   struct PrintFunctionsPass : public FunctionPass {
      static char ID;
      PrintFunctionsPass (): FunctionPass(ID) {}

      virtual bool runOnFunction(Function &F) {
            // Prints the name of each function.
            errs() << F.getName() << "\n";
            return false;
      }
   };

}

char PrintFunctionsPass::ID = 0;
```

Registering the pass

```
static
RegisterPass<PrintFunctionsPass> X("PrintFunctions",
    "Print Functions Pass",
    false /* Only looks at CFG */,
    false /* Analysis Pass */);

static llvm::RegisterStandardPasses Y(
    llvm::PassManagerBuilder::EP_EarlyAsPossible,
    [](const llvm::PassManagerBuilder &Builder,
    llvm::legacy::PassManagerBase &PM) {
        PM.add(new PrintFunctionsPass());
    }
);
```

Test the pass

- After building LLVM again, the new pass is a shared library
  - The name is `libLLVMPrintFunctions.so` and is in `build/lib`
- LLVM has an old and a new system to enabled LLVM passes
  - We use the old system

```
(build/work)$ ../bin/clang -flegacy-pass-manager
   -Xclang -load -Xclang
   ../lib/libLLVMPrintFunctions.so toy.c -o toy
foo
main
```

Run the pass with opt

```
(build/work)$ ../bin/opt
     -enable-new-pm=0
     -load ../lib/libLLVMPrintFunctions.so
     -PrintFunctions < toy.bc > /dev/null
```

How to print all functions and their total number

- The pass we developed processes each function individually
  - For printing their name
  - We have developed a `FunctionPass`
- What if we wanted to analyze some information related to many function
  - For instance, count all the functions

Count all functions

```
namespace {
   struct CountFunctions : public ModulePass  {
        static char ID;
        CountFunctions (): ModulePass(ID) {}
        virtual bool runOnModule(Module &M) {
            int counter = 0;

            for (Function& func: M) {
                counter++;
                errs() << func.getName() << "\n";
            }
            errs()<< "Total number of functions: "
                  << counter <<"\n";
            return false;
      }
   };
}
```

Active passes

- We have developed, so far, passive analysis passes
  - Print the function names
  - Count all functions
- Can we make a more active pass?
  - Create some additional code

Clone a function

```
namespace {
    struct Cloner: public ModulePass {
        static char ID;
        Cloner() : ModulePass(ID) {}

        bool runOnModule(Module &M) override {
            for (auto &F : M) {
                if (!F.getName().compare("foo")) {
                    Cloner::cloneFunction(F);
                    break;
                }
            }
            return true;
        }
        bool doFinalization(Module &M) override {
        ...
        }
        bool static cloneFunction(Function &F) {
        ...
        }
    };
}
```

doFinalization()

```
        bool doFinalization(Module &M) override {
#ifdef _DEBUG
            errs().write_escaped("Module is done.");
#endif
            return true;
        }
```

cloneFunction()

```
         bool static cloneFunction(Function &F) {
#ifdef _DEBUG
          errs() << "Cloner: ";
          errs().write_escaped(F.getName()) << '\n';
#endif
            ValueToValueMapTy vmap;
            ClonedCodeInfo cc;

#ifdef _DEBUF
            errs().write_escaped("Has the following arguments: ");
            for (Function::const_arg_iterator argI = F.arg_begin();
                 argI != F.arg_end();
                 ++argI) {
                errs().write_escaped(argI->getName()) << ' ';
            }
            errs() << '\n';
#endif
            llvm::CloneFunction(&F, vmap, &cc);

            return false;
        }
```