Lecture 3
More on the LLVM Compiler

Dominic Chen

Thanks to:
Jonathan Burket, Deby Katz, Gabe Weisz,
and Luke Zarko, for their slides

Visualizing the LLVM Compiler System

The LLVM Optimizer is a series of “passes”
- Analysis and optimization passes, run one after another
- Analysis passes do not change code, optimization passes do

LLVM Intermediate Form is a Virtual Instruction Set
- Language- and target-independent form
  Used to perform the same passes for all source and target languages
  - Internal Representation (IR) and external (persistent) representation

Three-Phase Design

LLVM: From Source to Binary

Clang Frontend (clang)
Optimzer (opt)
Static Compiler Backend (llc)

More Language Specific
C Source Code
Clang AST
LLVM IR
SelectionDAG
MachinistDAG
MCInst / Assembly

More Architecture Specific

In-Memory Data Structure

Bitcode (.bc files)

Text Format (.ll files)

define i32 @main() #0 {
  entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  ...
**Doing Things The LLVM Way - Strings**

- LLVM does not use either `char *` or `std::string` to represent strings internally.
  - `char *` uses null terminators to represent strings -> bad for byte strings
  - `std::strings` are ok, but not recommended for performance reasons

- If it takes a `StringRef`, it can take a `std::string` or string literal as well:
  ```cpp
  getFunction("myfunc")  getFunction(std::string("myfunc"))
  getFunction(StringRef("myfunc",6))  All Equivalent!
  ```

- Use `.str()` on a `StringRef` to get a `std::string`
- Should avoid using C-style strings altogether

**Doing Things The LLVM Way - Strings**

- Forget `std::cout` and `std::cerr`! You want `outs()`, `errs()`, and `null()`.
  - Oddly, there's not equivalent of `std::endl` in LLVM

- Printing the Name of a Function:
  ```cpp
  std::cout << F->getName().str() << std::endl;
  outs() << F->getName() << "\n";
  ```

- Printing an Instruction:
  ```cpp
  Instruction *I;
  I->dump();  or  outs() << *I << "\n";
  ```

- Printing an Entire Basic Block:
  ```cpp
  BB->dump();  or  outs() << *BB << "\n";
  ```

**Doing Things the LLVM Way – Data Structures**

- LLVM provides lots of data structures:
  - `BitVector`, `DenseMap`, `DenseSet`, `ImmutableList`, `ImmutableMap`,
  - `ImmutableSet`, `IntervalMap`, `IndexedMap`, `MapVector`, `PriorityQueue`,
  - `SetVector`, `ScopedHashTable`, `SmallBitVector`, `SmallPtrSet`, `SmallSet`,
  - `SmallString`, `SmallVector`, `SparseBitVector`, `SparseSet`, `StringRef`, `StringMap`, `StringRef`,
  - `StringRef`, `StringRef`, `Triple`, `TinyPtrVector`, `PackedVector`, `FoldingSet`, `UniqueVector`,
  - `ValueMap`

- Provide better performance through specialization
- STL data structures work fine as well
- Only use these data structures in HW if you really want to

**Doing Things The LLVM Way – Class Hierarchy**

```
Value
  User
    Constant
      ConstantExpr
      ConstantInt
    Operator
      ConcreteOperator
      BinaryOperator
      UnaryInstruction
    Instruction
      UnderValue
      CallInst
```
### LLVM Instructions <-> Values

```c
int main() {
    int x = 1;
    int y = 2;
    int z = 3;
    x = y+2;
    y = x+z;
    z = x+y;
}
```

Instruction I: “%add1 = add nsw i32 %add, 3”

You can’t “get” %add1 from Instruction I.

Instruction serves as the Value %add1.

```c
clang + mem2reg
```

Instruction I: “%add1 = add nsw i32 %add, 3”

outs() << *(I.getOperand(0));
outs() << *(I.getOperand(0)->getOperand(0));
```

Only makes sense for an SSA Compiler

---

### Doing Things The LLVM Way – Casting and Type Introspection

Given a Value *v*, what kind of Value is it?

```c
isa<Argument>(v)
```

Is *v* an instance of the `Argument` class?

```c
Argument *v = cast<Argument*>(v)
```

I know *v* is an `Argument`, perform the cast.

Causes assertion failure if you are wrong.

```c
Argument *v = dyn_cast<Argument*>(v)
```

Cast *v* to an `Argument` if it is an argument, otherwise return NULL. Combines both `isa` and `cast` in one command.

`dyn_cast` is not to be confused with the C++ `dynamic_cast` operator!
Navigating the LLVM IR: Iterators

- Module::iterator
  - Modules are the large units of analysis
  - Iterates through the functions in the module
- Function::iterator
  - Iterates through a function’s basic blocks
- BasicBlock::iterator
  - Iterates through the instructions in a basic block
- Value::use_iterator
  - Iterates through uses of a value
  - Recall that instructions are treated as values
- User::op_iterator
  - Iterates over the operands of an instruction (the “user” is the instruction)
  - Prefer to use convenient accessors defined by many instruction classes

Navigating the LLVM IR - Hints on Using Iterators

- Be careful when modifying an object while iterating over it (iterator invalidation)
  - Can cause unexpected behavior; use a separate removal list
- Use preincrement (++) rather than postincrement (i++)
  - Avoids problems with iterators doing unexpected things
  - Likely more performant, especially for complex iterators
- Iterators can overlap with each other
  - e.g., InstIterator walks over all instructions in a function, which overlaps with BasicBlock::iterator
Navigating the LLVM IR – More Iterators

- Finding a Basic Block’s predecessors/successors:

```cpp
#include "llvm/Support/CFG.h"
BasicBlock *BB = ...;
for (pred_iterator PI = pred_begin(BB); PI != pred_end(BB); ++PI) {
    BasicBlock *pred = *PI;
    //...
}
```

Many useful iterators are defined outside of Function, BasicBlock, etc.

Navigating the LLVM IR – Casting and Iterators

```cpp
for (Function::iterator FI = func->begin(); FI != func->end(); ++FI) {
    for (BasicBlock::iterator BBI = FI->begin(); BBI != FI->end(); ++BBI) {
        Instruction * I = BBI;
        if (CallInst * CI = dyn_cast<CallInst>(I)) {
            outs() << "I’m a Call Instruction!\n";
        }
        if (UnaryInstruction * UI = dyn_cast<UnaryInstruction>(I)) {
            outs() << "I’m a Unary Instruction!\n";
        }
        if (CastInstruction * CI = dyn_cast<CastInstruction>(I)) {
            outs() << "I’m a Cast Instruction!\n";
        }
        //...
    }
}
```

Very Common Code Pattern!

Navigating the LLVM IR – Visitor Pattern

```cpp
class MyVisitor : public InstVisitor<MyVisitor> {
    void visitCallInst(CallInst &CI) {
        outs() << "I’m a Call Instruction!\n";
    }
    void visitUnaryInstruction(UnaryInstruction &UI) {
        outs() << "I’m a Unary Instruction!\n";
    }
    void visitCastInst(CastInst &CI) {
        outs() << "I’m a Cast Instruction!\n";
    }
    void visitBinaryOperator(BinaryOperator &I) {
        switch (I.getOpcode()) {
            case Instruction::Mul:
                outs() << "I’m a multiplication Instruction!\n";
        }
    }
};
```

No need for iterators or casting!

A given instruction only triggers one method: a CastInst will not call visitUnaryInstruction if visitCastInst is defined.

You can case out on operators too, even if there isn’t a specific class for them!

Writing Passes - Changing the LLVM IR

- Getting rid of Instructions:
  - `eraseFromParent()`
  - `removeFromParent()`
  - `moveBefore/insertBefore/insertAfter` are also available
  - `replaceInstWithValue` and `replaceInstWithInst` are also useful to have
**Writing Passes – Adding New Instructions**

```cpp
define i32 @main() #0 {
    entry:
    %add = add nsw i32 2, 2
    %add1 = add nsw i32 %add, 2
    %mul = mul nsw i32 %add, %add1
    %sub = sub nsw i32 %add1, %mul
    %add2 = add nsw i32 %mul, 5
    ret i32 %sub
}
```

```
Instruction *I = "%mul = mul nsw i32 %add, %add1";
Instruction *newInst = BinaryOperator::Create(Instruction::Add, I.getOperand(0),
ConstantInt::get(I.getOperand(0)->getType(), 0));
I.getParent()->getInstList().insert(&I, newInst)
```

**Writing Passes - Correctness**

- When you modify code, be careful not to change the meaning!
- You can break module invariants while in your pass, but you should repair them before you finish
  - Some module invariant examples:
    - Types of binary operator parameters are the same
    - Terminator instructions only at the end of BasicBlocks
    - Functions are called with correct argument types
    - Instructions belong to Basic blocks
    - Entry node has no predecessor
- `opt` automatically runs a pass (-verify) to check module invariants
  - But it doesn’t check correctness in general!
- Think about multi-threaded correctness

**LLVM Pass Manager**

- Compiler is organized as a series of “passes”:
  - Each pass is an analysis or transformation
  - Each pass can depend on results from previous passes
- Six useful types of passes:
  - `BasicBlockPass`: iterate over basic blocks, in no particular order
  - `CallGraphSCCPass`: iterate over SCC’s, in bottom-up call graph order
  - `FunctionPass`: iterate over functions, in no particular order
  - `LoopPass`: iterate over loops, in reverse nested order
  - `ModulePass`: general interprocedural pass over a program
  - `RegionPass`: iterate over single-entry/exit regions, in reverse nested order
- Passes have different constraints (e.g. `FunctionPass`):
  - `FunctionPass` can only look at the “current function”
  - Cannot maintain state across functions

**LLVM Pass Manager**

- Given a set of passes, the PassManager tries to optimize the execution time of the set of passes
  - Share information between passes
  - Pipeline execution of passes
- PassManager must understand how passes interact
  - Passes may require information from other passes
  - Passes may require transformations applied by other passes
  - Passes may invalidate information or transformations applied by other passes
- `DominatorTreeWrapperPass`, `LoopInfoWrapperPass`, `LoopSimplify`
**LLVM Pass Manager**

- The `getAnalysisUsage()` function defines how a pass interacts with other passes.
- Given `getAnalysisUsage(AnalysisUsage &AU)` for `PassX`:
  - `AU.addRequired<PassY>()` → PassY must be executed first.
  - `AU.addPreserved<PassY>()` → PassY is still preserved by running PassX.
  - `AU.setPreservesAll()` → PassX preserves all previous passes.
  - `AU.setPreservesCFG()` → PassX might make changes, but not to the CFG.
- If nothing is specified, it is assumed that all previous passes are invalidated.

**DeadStoreElimination Pass:**

```cpp
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.setPreservesCFG();
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.addRequired<AliasAnalysis>();
  AU.addRequired<MemoryDependenceAnalysis>();
  AU.addPreserved<AliasAnalysis>();
  AU.addPreserved<DominatorTreeWrapperPass>();
  AU.addPreserved<MemoryDependenceAnalysis>();
}
```

**Using Passes**

- For homework assignments, do not use passes provided by LLVM unless instructed to:
  - We want you to implement the passes yourself!
- For projects, you can use whatever you want:
  - Your own passes or LLVM's passes.

**opt tool: LLVM modular optimizer**

- Invoke arbitrary sequence of passes:
  - Completely control `PassManager` from command line.
  - Supports loading passes as plugins from *.so files.

```
opt -load foo.so -pass1 -pass2 -pass3 x.bc -o y.bc
```

- Passes "register" themselves:
  - When you write a pass, you must write the registration.

```
RegisterPass<FunctionInfo> X("function-info",
  "15745: Function Information");
```

**Useful LLVM Passes: Memory to Register (-mem2reg)**

```cpp
define i32 @main() #0 {
  entry:
    %retval = alloca i32, align 4
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    store i32 0, i32* %retval
    store i32 5, i32* %a, align 4
    store i32 3, i32* %b, align 4
    %0 = load i32* %a, align 4
    %1 = load i32* %b, align 4
    %sub = sub nsw i32 %0, %1
    ret i32 %sub
}
```

```cpp
define i32 @main() #0 {
  entry:
    %sub = sub nsw i32 5, 3
    ret 132 %sub
}
```

**Not always possible:** Sometimes stack operations are too complex.
```c
int main(int argc, char* argv[]) {
    int x;
    if (argc > 5) {
        x = 2;
    } else {
        x = 3;
    }
    return x;
}
```

```c
int main(int argc, char* argv[]) {
    int x = 0;
    for (int i=0; i<argc; i++) {
        x += i;
    }
    return x;
}
```
```c
define i32 @main(i32 %argc, i8** %arv) #0 { 
  entry:
  br label %for.cond
  %for.cond:
  %x.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, %argc
  br i1 %cmp, label %for.body, label %for.end
  %for.body:
  %add = add nsw i32 %x.0, %i.0
  br label %for.inc
  %for.inc:
  %inc = add nsw i32 %i.0, 1
  br label %for.cond
  %for.end:
  ret i32 %x.0
}
```

There are no "Loop" instructions!
Loops are implemented with conditions and jumps

---

**Useful LLVM Passes: Loop Information (-loops)**
- llvm/Analysis/LoopInfo.h
  - Reveals:
    - The basic blocks in a loop
    - Headers and pre-headers
    - Exit and exiting blocks
    - Back edges
    - "Canonical induction variable"
    - Loop Count

---

**Useful LLVM Passes: Simplify CFG (-simplifycfg)**
- Performs basic cleanup
  - Removes unnecessary basic blocks by merging unconditional branches if the second block has only one predecessor
  - Removes basic blocks with no predecessors
  - Eliminates phi nodes for basic blocks with a single predecessor
  - Removes unreachable blocks

---

**Useful LLVM Passes: Others**
- Scalar Evolution (-scalar-evolution)
  - Tracks changes to variables through nested loops
- Target Data (-targetdata)
  - Gives information about data layout on the target machine
  - Useful for generalizing target-specific optimizations
- Alias Analyses (-basicaa, -aa-eval, -scev-aa)
  - Several different passes give information about aliases
  - If you know that different names refer to different locations, you have more freedom to reorder code, etc.
Useful LLVM Passes: Others

- Liveness-based dead code elimination (-dce, adce)
  - Assumes code is dead unless proven otherwise
- Dead global elimination (-globaldce)
  - Deletes all globals that are not live
- Sparse conditional constant propagation (-sccp)
  - Aggressively search for constants
- Loop invariant code motion (-licm)
  - Move code out of loops where possible
- Canonicalize induction variables (-indvars)
  - All loops start from zero and step by one
- Canonicalize loops (-loop-simplify)
  - Put loop structures in standard form

Some Useful LLVM Documentation

- LLVM Coding Standards
  - [http://llvm.org/docs/CodingStandards.html](http://llvm.org/docs/CodingStandards.html)
- LLVM Programmer’s Manual
- LLVM Language Reference Manual
  - [http://llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html)
- Writing an LLVM Pass
  - [http://llvm.org/docs/WritingAnLLVMPass.html](http://llvm.org/docs/WritingAnLLVMPass.html)
- LLVM’s Analysis and Transform Passes
  - [http://llvm.org/docs/Passes.html](http://llvm.org/docs/Passes.html)
- LLVM Internal Documentation

Remember: We’re using LLVM 3.5.0, but the documentation is always for the most up to date code (i.e. for the upcoming LLVM 3.9)