Lecture 6
More on the LLVM Compiler

Deby Katz
Thanks to Luke Zarko and Gabe Weisz for some content

Understanding the LLVM IR

- Recall that LLVM uses an intermediate representation for intermediate steps
  - Used for all steps between the front end (translating from source code) and the back end (translating to machine code)
  - Language- and mostly target-independent form
    - Target dictates alignment and pointer sizes in the IR, little else

Outline

- Understanding and Navigating the LLVM IR
- Writing Passes
  - Changing the LLVM IR
- Using Passes
- Useful Documentation

Understanding the LLVM IR - Processing Programs

- Iterators for modules, functions, blocks, and uses
  - Use these to access nearly every part of the IR data structure

- There are functions to inspect data types and constants

- Many classes have dump() member functions that print information to standard error
  - In GDB, use `p obj->dump()` to print the contents of that object
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

Hints on Using Iterators

- Be very careful about modifying any part of the object iterated over during iteration
  - Can cause unexpected behavior

- Use ++i rather than i++ and pre-compute the end
  - Avoid problems with iterators doing unexpected things while you are iterating
  - Especially for fancier iterators

- There is overlap among iterators
  - E.g., InstIterator walks over all instructions in a function, overlapping range of Function::iterator and BasicBlock::iterator

- Most iterators automatically convert a pointer to the appropriate object type
  - Not all: InstIterator does not

Understanding the LLVM IR -
Interpreting An Instruction

- The Instruction class has several subclasses, for various types of operations
  - E.g., LoadInst, StoreInst, AllocaInst, CallInst, BranchInst
  - Use the dyn_cast<> operator to check to see if the instruction is of the specified type
    - If so, returns a pointer to it
    - If not, returns a null pointer
    - For example,
      ```c++
      if (AllocationInst *AI = dyn_cast<AllocationInst>(Val)) {
          // ... If we get here, *AI is an alloca instruction
      }
      ```

- Several classifications of instructions:
  - Terminator instructions, binary instructions, bitwise binary instructions, memory instructions, and other instructions

Understanding the LLVM IR -
Terminator Instructions

- Every BasicBlock must end with a terminator instruction
  - Terminator instructions can only go at the end of a BB

- **ret**, **br**, **switch**, **indirectbr**, **invoke**, **resume**, and **unreachable**
  - **ret** - return control flow to calling function
  - **br**, **switch**, **indirectbr** - transfer control flow to another BB in the same function
  - **invoke** - transfers control flow to another function
Binary operations do most of the computation in a program

- Handle nearly all of the arithmetic

Two operands of the same type; result value has same type

- E.g., `add`, `fadd`, `sub`, `fsub`, `mul`, `fmul`, `udiv`, `sdiv`, `fdiv`
  - 'f' indicates floating point, 's' indicates signed, 'u' indicates unsigned

Bitwise binary operations

- Frequently used for optimizations
- Two operands of the same type; one result value of the same type

LLVM IR does not represent memory locations (SSA)

- Instead, uses named locations

alloca

- Allocates memory on the stack frame of the current function, reclaimed at return

load

- Reads from memory, often in a location named by a previous alloca

store

- Writes to memory, often in a location named by a previous alloca

For example:

```cpp
%ptr = alloca i32 ; yields {i32*}:ptr
store i32 3, i32* %ptr ; stores 3 in the location named by %ptr
%val = load i32* %ptr ; yields {i32}:val = i32 3
```

getelementptr

- Gets the address of a sub-element of an aggregate data structure (derived type)

SSA representation means that an Instruction is treated as the same as the Value it produces

- Values start with % or @
  - % indicates a local variable
  - @ indicates a global variable
  - Instructions that produce values can be named

%foo = inst in the LLVM IR just gives a name to the instruction in the syntax
Understanding the LLVM IR - Types in the LLVM IR

- Strong type system enables some optimizations without additional analysis

- Primitive Types
  - Integers (\(\text{N} \) bits, \(\text{N} \) from 1 to \(2^{23}-1\)), Floating point (half, float, double, etc.)
  - Others (x86mmx, void, etc.)

- Derived Types
  - Arrays ([# elements ( \(\geq 0\) ) x element type])
  - Functions (returntype (paramlist))
  - Pointers (type*, type addrspace(N)*)
  - Vectors (\(<# elements ( > 0) x element type\>)
  - Structures({ typelist })

- All derived types of a particular “shape” are considered the same
  - Does not matter if same-shaped types have different names
  - LLVM may rename them

Outline

- Understanding and Navigating the LLVM IR
- Writing Passes
- Using Passes
- Useful Documentation

Writing Passes - Changing the LLVM IR

- eraseFromParent()
  - Remove the instruction from basic block, drop all references, delete

- removeFromParent()
  - Remove remove the instruction from basic block
  - Use if you will re-attach the instruction
  - Does not drop references (or clear the use list), so if you don't re-attach it Bad Things will happen

- moveBefore/insertBefore/insertAfter are also available

- ReplaceInstWithValue and ReplaceInstWithInst are also useful to have

Writing Passes - Analysis Passes vs. Optimization Passes

- Two Major kinds of passes:
  - Analysis: provide information (Like FunctionInfo)
  - Transform: modify the program (Like LocalOpts)

- getAnalysisUsage method
  - Defines how this pass interacts with other passes
  - For example,

```c++
// A pass that modifies the program, but does not modify the CFG
// The pass requires the LoopInfo pass
void LICM::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesCFG();
  AU.addRequired<LoopInfo>();
}
```

- setPreservesAll - used in analysis pass that does not modify the program
When you modify code, be careful not to change the meaning!
- For our assignments, and in most situations, you want the effect of the code to be the same as before you altered it

Think about multi-threaded correctness

You can change the meaning of code while you are modifying the code within your pass, but you should restore the meaning before the pass finishes

You need to check for correctness on your own, because LLVM has very limited built-in correctness checking

LLVM has module invariants that should stay the same before and after your pass
- 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

You can break module invariants while in your pass, but you should repair them before you finish

opt automatically runs a pass (-verify) to check module invariants
- But it doesn't check correctness in general!

The CommandLine library allows you to add command line parameters very quickly
- Conflicts in parameter names won't show up until runtime, since passes are loaded dynamically

Understanding and Navigating the LLVM IR
- Writing Passes
  - Changing the LLVM IR
- Using Passes
- Useful Documentation
Using Passes

- For homework assignments, do not use passes provided by LLVM unless instructed to
  - We want you to implement the passes yourself to understand how they really work
- For projects, you can use whatever you want
  - Your own passes or LLVM’s passes
- Some useful LLVM passes follow

Some Useful Passes - mem2reg transform pass

- If you have alloca instructions that only have loads and stores as uses
  - Changes them to register references
  - May add SSA features like “phi” instructions
- Sometimes useful for simplifying IR
  - Confuses easily

Some Useful 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

Some Useful 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
Some Useful Passes

- **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**
  - Several different passes give information about aliases
  - E.g., does *A point to the same location as *B?
  - If you know that different names refer to different locations, you have more freedom to reorder code, etc.

Other Useful Passes

- **Liveness-based dead code elimination**
  - Assumes code is dead unless proven otherwise
- **Sparse conditional constant propagation**
  - Aggressively search for constants
- **Correlated propagation**
  - Replace select instructions that select on a constant
- **Loop invariant code motion**
  - Move code out of loops where possible
- **Dead global elimination**
- **Canonicalize induction variables**
  - All loops start from 0
- **Canonicalize loops**
  - Put loop structures in standard form

Outline

- **Understanding and Navigating the LLVM IR**
- **Writing Passes**
  - Changing the LLVM IR
- **Using Passes**
- **Useful Documentation**

Some Useful LLVM Documentation

- **LLVM Programmer’s Manual**
- **LLVM Language Reference Manual**
  - http://llvm.org/docs/LangRef.html
- **Writing an LLVM Pass**
  - http://llvm.org/docs/WritingAnLLVMPass.html
- **LLVM’s Analysis and Transform Passes**
  - http://llvm.org/docs/Passes.html
- **LLVM Internal Documentation**
  - May be easier to search the internal documentation from the http://llvm.org front page