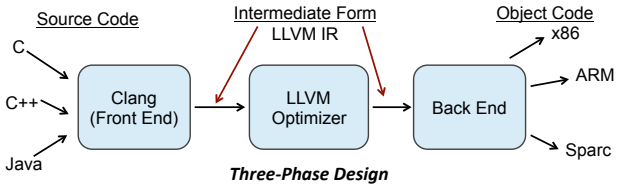
Lecture 6
More on the LLVM Compiler

Jonathan Burket

Special thanks to Deby Katz,
Luke Zarko, and Gabe Weisz for their slides

Visualizing the LLVM Compiler System

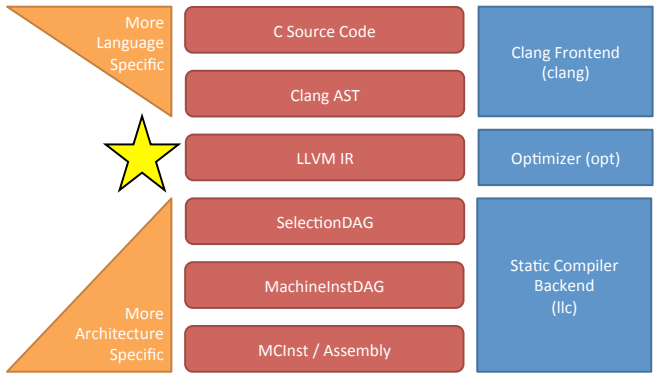


Three-Phase Design

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

LLVM: From Source to Binary



LLVM IR

In-Memory Data Structure

Bitcode (.bc files)

```
42 43 C0 DE  21 0C 00 00
06 10 32 39  92 01 84 0C
0A 32 44 24  48 0A 90 21
18 00 00 00  98 00 00 00
E6 C6 21 1D  E6 A1 1C DA
...
```

llvm-dis
llvm-asm

Text Format (.ll files)

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
...
```

Bitcode files and LLVM IR text files are **lossless serialization formats**!
We can pause optimization and come back later.

Doing Things The LLVM Way - Strings

- LLVM does not use either *char ** or *std::string* to represent strings internally.
  - *char **s use 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:

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:

~~std::cout << F->getName().str() << std::endl;~~

outs() << F->getName() << "\n";

Printing an Instruction:

Instruction *I;

I->dump(); or outs() << *I << "\n";

Printing an Entire Basic Block:

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, StringMap, StringRef, StringSet, 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

LLVM Instructions <--> Values

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

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

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

Operand 1 Operand 2

You can't "get" %add1 from Instruction I.
Instruction serves as the Value %add1.

LLVM Instructions <--> Values

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

clang + mem2reg

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %add = add nsw i32 2, 3
  %add1 = add nsw i32 %add, 3
  %add2 = add nsw i32 %add, %add1
  ret i32 0
}
```

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

outs() << *(I.getOperand(0)); ⟶ "%add = add nsw i32 2, 3"

outs() << *(I.getOperand(0)->getOperand(0)); ⟶ "2"

Only makes sense for an SSA Compiler [More on this Later]

Doing Things The LLVM Way – Casting and Type Introspection

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

isa<Argument>(v)

Is v an instance of the Argument class?

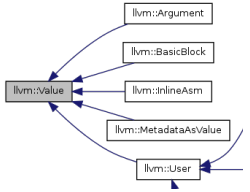
Argument * v = **cast**<Argument>(v)

I *know* v is an Argument, perform the cast. Causes assertion failure if you are wrong.

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.

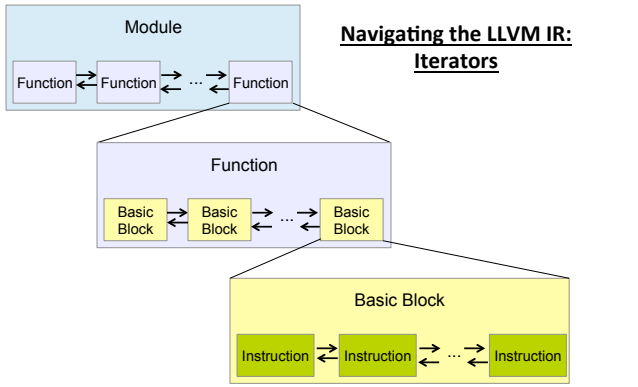
dyn_cast is not to be confused with the C++ *dynamic_cast* operator!



Doing Things The LLVM Way – Casting and Type Introspection

```
void analyzeInstruction(Instruction * I) {
        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";
        }
        …
}
```


Navigating the LLVM IR: Iterators



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 - Iterators

Iterate through every instruction in a function:

```
for (Function::iterator FI = func->begin(), FE = func->end(); FI != FE; ++FI) {
        for (BasicBlock::iterator BBI = FI->begin(), BBE = FI->end(); BBI != BBE; ++BBI) {
            outs() << "Instruction: " << *BBI << "\n";
        }
}
```

Using *InstIterator* (Provided by "llvm/IR/InstIterator.h"):

```
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
        outs() << *I << "\n";
```


Navigating the LLVM IR - 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

Navigating the LLVM IR – More Iterators

Finding a Basic Block's predecessors/successors:

```
#include "llvm/Support/CFG.h"
BasicBlock *BB = ...;

for (pred_iterator PI = pred_begin(BB), E = pred_end(BB); PI != E; ++PI) {
  BasicBlock *Pred = *PI;
  // ...
}
```

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

Navigating the LLVM IR – Casting and Iterators

```
for (Function::iterator FI = func->begin(), FE = func->end(); FI != FE; ++FI) {
        for (BasicBlock::iterator BBI = FI->begin(), BBE = FI->end(); BBI != BBE; ++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

```
struct 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 visitMul(BinaryOperator &I) {
        outs() << "I'm a multiplication Instruction!\n";
    }
}
MyVisitor MV;
MV.visit(F);
```

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()**
    - Remove from basic block, drop all references, delete
  - **removeFromParent()**
    - Remove 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 – Adding New Instructions

```
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
}
```

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

Let I = The "%mul = mul nsw i32 %add, %add1" instruction

```
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

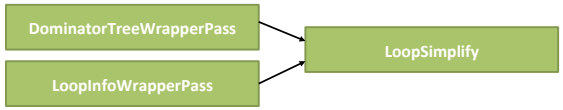
- When you modify code, be careful not to change the meaning!
- You can break module invariants while in your pass, but you should repair them before you finish
  - Some module invariant examples:
    - Types of binary operator parameters are the same
    - Terminator instructions only at the end of 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 one analysis or transformation
- ~~**Four**~~ **Seven types of passes:**
  - ImmutablePass: doesn't do much
  - LoopPass: process loops
  - RegionPass: process single-entry, single-exit portions of code
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time
- **Constraints imposed (e.g. FunctionPass):**
  - FunctionPass can only look at "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



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 some changes, but not to the CFG
  - If nothing is specified, it is assumed all previous passes are invalidated

DeadStoreElimination Pass:

```
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>();
}
```

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");
```


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

Useful LLVM Passes: mem2reg

```
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
}
```

mem2reg →

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

Not always possible:
Sometimes stack operations are too complex


```
int main(int argc,
         char * arv[]) {
  int x;
  if (argc > 5) {
    x = 2;
  }
  else {
    x = 3;
  }
  return x;
}
```

clang + mem2reg →

```
define i32 @main(i32 %argc,
                 i8** %arv) #0 {
entry:
  %cmp = icmp sgt i32 %argc, 5
  br i1 %cmp, label %if.then,
              label %if.else
if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %x.0 = phi i32
     [ 2, %if.then ],
     [ 3, %if.else ]
  ret i32 %x.0
}
```

%x.0 = 2 if we go through %if.then
%x.0 = 3 if we go through %if.else

```
int main(int argc, char * argv[]) {
  int vals[4] = {2,4,8,16};
  int x = 0;
  vals[1] = 3;
  x += vals[0];
  x += vals[1];
  x += vals[2];
  return x;
}
```



```
@main.vals = private unnamed_addr constant [4 x i32]
                        [i32 2, i32 4, i32 8, i32 16], align 4
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %vals = alloca [4 x i32], align 4
  %0 = bitcast [4 x i32]* %vals to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0, i8* bitcast ([4 x i32]*
@main.vals to i8*), i32 16, i32 4, i1 false)
  %arrayidx = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 1
  store i32 3, i32* %arrayidx, align 4
  %arrayidx1 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 0
  %1 = load i32* %arrayidx1, align 4
  %add = add nsw i32 0, %1
  %arrayidx2 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 1
  %2 = load i32* %arrayidx2, align 4
  %add3 = add nsw i32 %add, %2
  %arrayidx4 = getelementptr inbounds [4 x i32]* %vals, i32 0, i32 2
  %3 = load i32* %arrayidx4, align 4
  %add5 = add nsw i32 %add3, %3
  ret i32 %add5
}
```

mem2reg does not solve all of our pointer problems ⊗


```
int main(int argc, char * arv[]) {
  int x = 0;
  for (int i=0; i<argc; i++) {
    x += i;
  }
  return x;
}
```



```
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
}
```

Names are just strings, they should not be used to identify loops

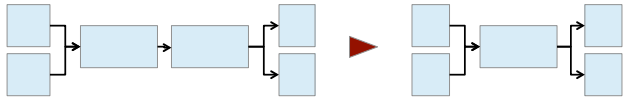
There are no "Loop" Instructions: Loops are implemented with conditions and jumps

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

- 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
  - 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

Some Useful LLVM Documentation

LLVM Programmer's Manual

http://llvm.org/docs/ProgrammersManual.html

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

http://llvm.org/docs/doxygen/html/

Remember: We're using LLVM 3.5, but the documentation is always for the most up to date code (i.e. for the upcoming LLVM 3.6)