

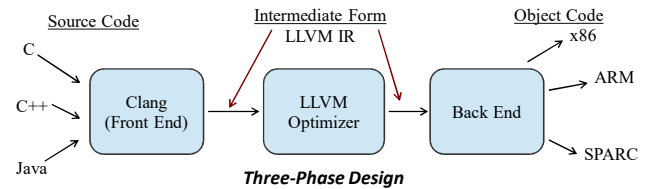
## Lecture 3 More on the LLVM Compiler

Dominic Chen

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

Carnegie Mellon

### Visualizing the LLVM Compiler System



The LLVM Optimizer is a series of “passes”

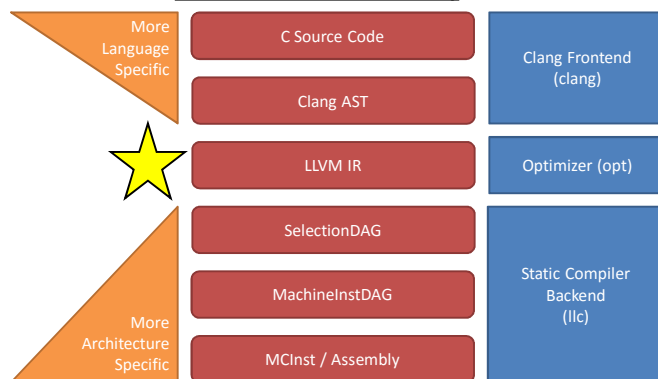
- Analysis and optimization passes, run one after another
- Analysis passes do not change code, optimization passes do

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

Carnegie Mellon

### LLVM: From Source to Binary



Carnegie Mellon

### 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
...
    
```

Text Format (.ll files)

```

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

llvm-dis

llvm-asm

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

Carnegie Mellon

### 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

Carnegie Mellon

### 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";
```

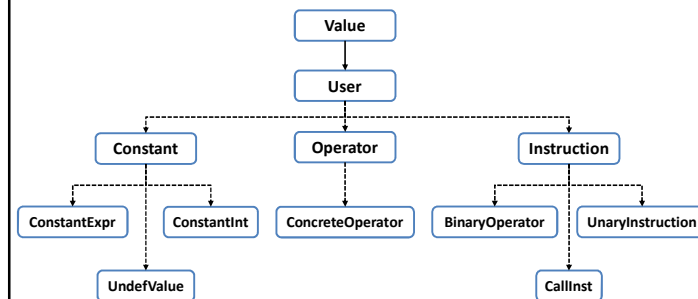
Carnegie Mellon

### 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

Carnegie Mellon

### Doing Things The LLVM Way – Class Hierarchy



Carnegie Mellon

### LLVM Instructions <--> Values

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

; 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
}
```

clang + mem2reg

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

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

Operand 1      Operand 2

Carnegie Mellon

### LLVM Instructions <--> Values

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

; 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
}
```

clang + mem2reg

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

Carnegie Mellon

### 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?

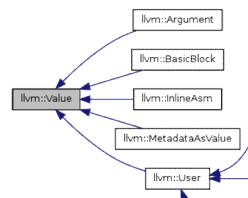
Argument \*v = cast<Argument>(v)

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!

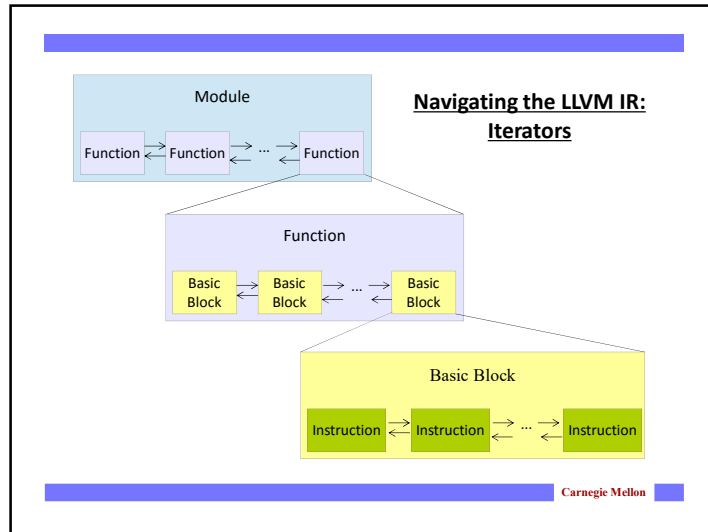


Carnegie Mellon

### 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";
    }
    ...
}
```

Carnegie Mellon



**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

Carnegie Mellon

**Navigating the LLVM IR - Iterators**

- Iterate through every instruction in a function:
 

```

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

Carnegie Mellon

**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 (++i) 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**

Carnegie Mellon

### 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); PI != pred_end(BB); ++PI) {
    BasicBlock *pred = *PI;
    // ...
}
```

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

Carnegie Mellon

### Navigating the LLVM IR – Casting and Iterators

```
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!

Carnegie Mellon

### Navigating the LLVM IR – Visitor Pattern

```
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!

Carnegie Mellon

### 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

Carnegie Mellon

## 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
}
```

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)

Carnegie Mellon

## 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

Carnegie Mellon

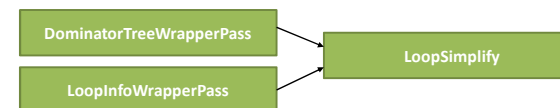
## LLVM Pass Manager

- 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

Carnegie Mellon

## 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



Carnegie Mellon

## 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:

```
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>();
}
```

Carnegie Mellon

## 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");
```

Carnegie Mellon

## 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

Carnegie Mellon

## Useful LLVM Passes: Memory to Register (-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
}
```



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

Not always possible:  
Sometimes stack operations  
are too complex

Carnegie Mellon

```

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

define i32 @main(i32 %argc,
                i8** %argv) #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
}

```

clang + mem2reg

%x.0 = 2 from %if.then  
%x.0 = 3 from %if.else

Carnegie Mellon

```

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;
}

```

Carnegie Mellon

```

@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 ☹

Carnegie Mellon

```

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

```

Carnegie Mellon

```

define i32 @main(i32 %argc, i8** %argv) #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

Carnegie Mellon

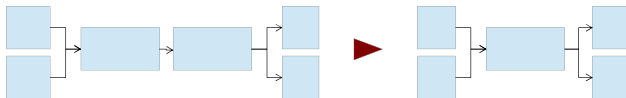
### Useful LLVM Passes: Loop Information (-loops)

- 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

Carnegie Mellon

### 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



Carnegie Mellon

### 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.

Carnegie Mellon

### 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

Carnegie Mellon

### Some Useful LLVM Documentation

- LLVM Coding Standards
  - <http://llvm.org/docs/CodingStandards.html>
- 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 3.5.0, but the documentation is always for the most up to date code (i.e. for the upcoming LLVM 3.9)

Carnegie Mellon