15-411: LLVM

Jan Hoffmann

Substantial portions courtesy of Deby Katz

and Gennady Pekhimenko, Olatunji Ruwase,Chris Lattner, Vikram Adve, and David Koes Carnegie

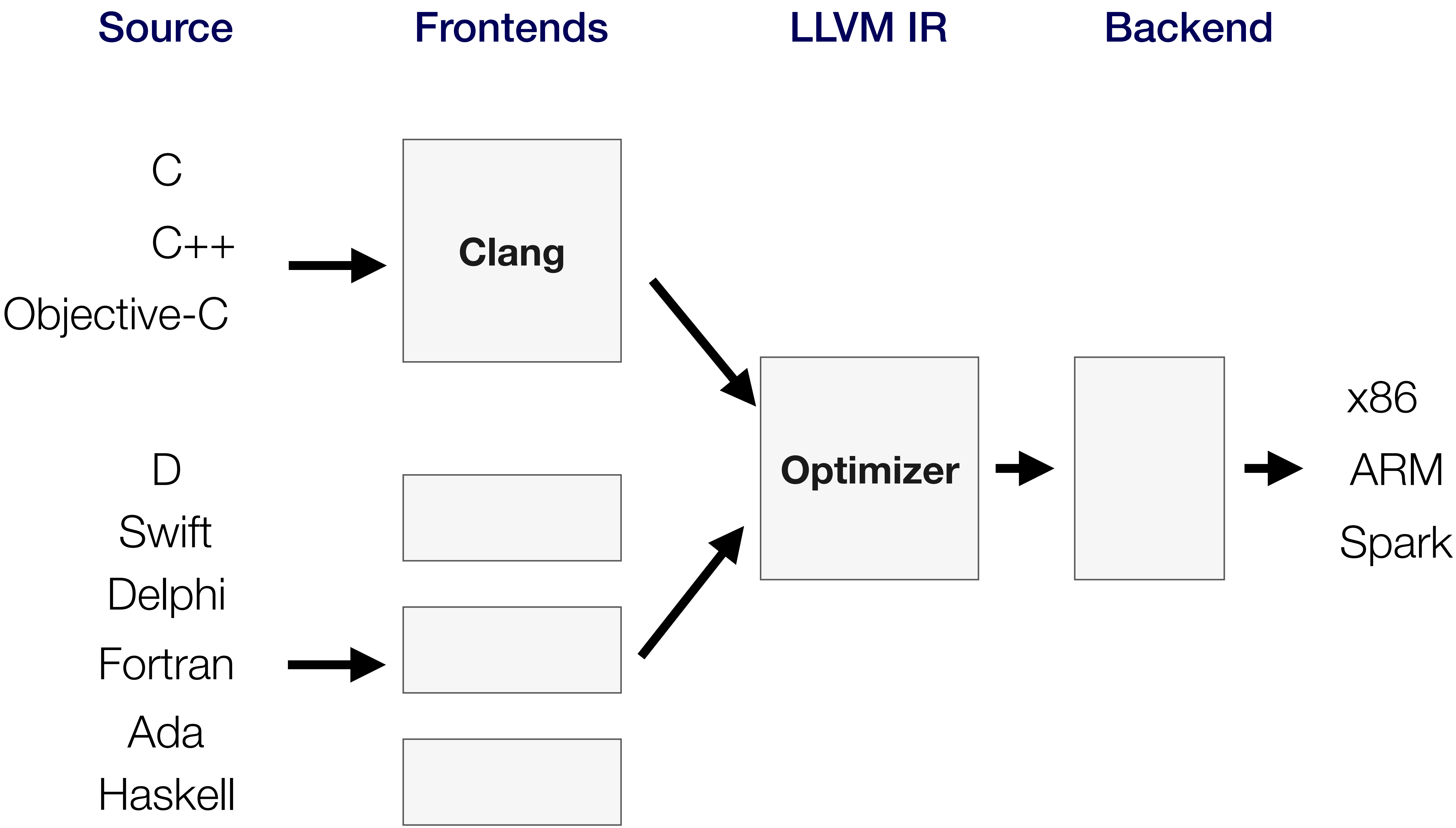
What is LLVM?

A collection of modular and reusable compiler and toolchain technologies

- Implemented in C++
- LLVM has been started by Vikram Adve and Chris Lattner at UIUC in 2000
- Originally ‘Low Level Virtual Machine’ for research on dynamic compilation
- Evolved into an umbrella project for a lot different things

LLVM Components

- **LLVM Core:** optimizer for source- and target independent LLVM IR code generator for many architectures
- **Clang:** C/C++/Objective C compiler that uses LLVM Core Includes the Clang Static Analyzer for bug finding
- **libcc+:** implementation of C++ standard library
- **LLDB:** debugger for C, C++, and Objective C
- **dragonegg:** parser front end for compiling Fortran, Ada, …
- …



LLVM Compiler Framework

LLVM Analysis Passes

- Basic-Block Vectorization
- Profile Guided Block Placement
- Break critical edges in CFG
- Merge Duplicate Global
- Simple constant propagation
- Dead Code Elimination
- Dead Argument Elimination
- Dead Type Elimination
- Dead Instruction Elimination
- Dead Store Elimination
- Deduce function attributes
- Dead Global Elimination
- Global Variable Optimizer
- Global Value Numbering
- Canonicalize Induction Variables
- Function Integration/Inlining
- Combine redundant instructions
- Internalize Global Symbols
- Interprocedural constant propa.
- Jump Threading
- Loop-Closed SSA Form Pass
- Loop Strength Reduction
- Rotate Loops
- Loop Invariant Code Motion

LLVM Analysis Passes

Canonicalize natural loops

Unroll loops

Unswitch loops

-mem2reg:

Promote Memory to Register

MemCpy Optimization

Merge Functions

Unify function exit nodes

Remove unused exception handling

Reassociate expressions

Demote all values to stack slots

Sparse Conditional Cons. Propaga.

Simplify the CFG

Code sinking

Strip all symbols from a module

Strip debug info for unused symbols

Strip Unused Function Prototypes

Strip all llvm.dbg.declare intrinsics

Tail Call Elimination

Delete dead loops

Extract loops into new

LLVM IR

Example 1

```
int add (int x) {
  int y = 8128;
  return x+y;    }
```

Clang

```
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Example 1

Functions are parametrized with arguments and types.

```
int add (int x) {
  int y = 8128;
  return x+y;    }
```

Clang →

```
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Example 1

Functions are parametrized with arguments and types.

```c
int add (int x) {
  int y = 8128;
  return x+y;    }
```

Clang

```llvm
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Local vars are allocated on the stack; not in temps.

Example 1

Functions are parametrized with arguments and types.

```
int add (int x) {
  int y = 8128;
  return x+y;    }
```

Clang →

```
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Local vars are allocated on the stack; not in temps.

Instructions have types: i32 is for 32bit integers.

Example 1

```
int add (int x) {
  int y = 8128;
  return x+y;    }
```

Clang

```
; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}
```

Functions are parametrized with arguments and types.

Local vars are allocated on the stack; not in temps.

Instructions have types: i32 is for 32bit integers.

No signed wrap: result of overflow undefined.

Example2

```
int loop  (int n) {
  int i = n;
  while(i<1000){i++;}
  return i;
}
```

Clang

```
; Function Attrs: nounwind ssp uwtable
define i32 @loop(i32 %n) #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  %2 = load i32* %1, align 4
  store i32 %2, i32* %i, align 4
  br label %3

; <label>:3                                ; preds = %6, %0
  %4 = load i32* %i, align 4
  %5 = icmp slt i32 %4, 1000
  br i1 %5, label %6, label %9

; <label>:6                                ; preds = %3
  %7 = load i32* %i, align 4
  %8 = add nsw i32 %7, 1
  store i32 %8, i32* %i, align 4
  br label %3

; <label>:9                                ; preds = %3
  %10 = load i32* %i, align 4
  ret i32 %10
}
```

Example2

```c
int loop  (int n) {
  int i = n;
  while(i<1000){i++;}
  return i;
}
```

Clang

Basic blocks.

```llvm
; Function Attrs: nounwind ssp uwtable
define i32 @loop(i32 %n) #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  %2 = load i32* %1, align 4
  store i32 %2, i32* %i, align 4
  br label %3

; <label>:3                              ; preds = %6, %0
  %4 = load i32* %i, align 4
  %5 = icmp slt i32 %4, 1000
  br i1 %5, label %6, label %9

; <label>:6                              ; preds = %3
  %7 = load i32* %i, align 4
  %8 = add nsw i32 %7, 1
  store i32 %8, i32* %i, align 4
  br label %3

; <label>:9                              ; preds = %3
  %10 = load i32* %i, align 4
  ret i32 %10
}
```

Example2

```
int loop  (int n) {
  int i = n;
  while(i<1000){i++;}
  return i;
}
```

Clang

```
; Function Attrs: nounwind ssp uwtable
define i32 @loop(i32 %n) #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  %2 = load i32* %1, align 4
  store i32 %2, i32* %i, align 4
  br label %3

; <label>:3                             ; preds = %6, %0
  %4 = load i32* %i, align 4
  %5 = icmp slt i32 %4, 1000
  br i1 %5, label %6, label %9

; <label>:6                             ; preds = %3
  %7 = load i32* %i, align 4
  %8 = add nsw i32 %7, 1
  store i32 %8, i32* %i, align 4
  br label %3

; <label>:9                             ; preds = %3
  %10 = load i32* %i, align 4
  ret i32 %10
}
```

Predecs. in CFG.

Basic blocks.

LLVM IR

- Three address pseudo assembly
- Reduced instruction set computing (RISC)
- Static single assignment (SSA) form
- Infinite register set
- Explicit type info and typed pointer arithmetic
- Basic blocks

LLVM IR

- Three address pseudo assembly
- Reduced instruction set computing (RISC)
- Static single assignment (SSA) form
- Infinite register set
- Explicit type info and typed pointer arithmetic
- Basic blocks

```
for (i = 0; i < N; i++)
   Sum(&A[i], &P);
```

```
loop:                 ; preds = %bb0, %loop
  %i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum(float %AiAddr, %pair* %P)
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.1, %N
  br i1 %exitcond, label %outloop, label %loop
```

LLVM IR

- Three address pseudo assembly
- Reduced instruction set computing (RISC)
- Static single assignment (SSA) form
- Infinite register set
- Explicit type info and typed pointer arithmetic
- Basic blocks

Stack allocated temps eliminated by mem2reg.

```
for (i = 0; i < N; i++)
   Sum(&A[i], &P);
```

```
loop:                    ; preds = %bb0, %loop
  %i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum(float %AiAddr, %pair* %P)
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.1, %N
  br i1 %exitcond, label %outloop, label %loop
```

LLVM IR Structure

- **Module contains Functions and GlobalVariables**
  - Module is unit of compilation, analysis, and optimization
- **Function contains BasicBlocks and Arguments**
  - Functions roughly correspond to functions in C
- **BasicBlock contains list of instructions**
  - Each block ends in a control flow instruction
- **Instruction is opcode + vector of operands**

Type System

- llvm.org:

  “The LLVM type system is one of the most important features of the intermediate representation.

  Being typed enables a number of optimizations to be performed on the intermediate representation directly, without having to do extra analyses on the side before the transformation.

  A strong type system makes it easier to read the generated code and enables novel analyses and transformations that are not feasible to perform on normal three address code representations”

Type System

Greg Morrisett and Karl Crary: Typed Assembly (1998)

- llvm.org:

  "The LLVM type system is one of the most important features of the intermediate representation.

  Being typed enables a number of optimizations to be performed on the intermediate representation directly, without having to do extra analyses on the side before the transformation.

  A strong type system makes it easier to read the generated code and enables novel analyses and transformations that are not feasible to perform on normal three address code representations"

Single Value Types

Integer Types

`iN`

Single Value Types

Integer Types

`iN`

Size.

Single Value Types

Integer Types

Size.

```
iN
```

i1	a single-bit integer.
i32	a 32-bit integer.
i1942652	a really big integer of over 1 million bits.

Single Value Types

Integer Types

`iN`

Size.

i1	a single-bit integer.
i32	a 32-bit integer.
i1942652	a really big integer of over 1 million bits.

Float Types

half	16-bit floating point value
float	32-bit floating point value
double	64-bit floating point value

Functions and Void

Void

```
void
```

Function Types

```
<returntype> (<parameter list>)
```

i32 (i32)	function taking an i32, returning an i32
float (i16, i32 *) *	Pointer to a function that takes an i16 and a pointer to i32, returning float.
i32 (i8*, ...)	A vararg function that takes at least one pointer to i8 (char in C), which returns an integer

Functions and Void

No representation and no size.

Void

```
void
```

Function Types

```
<returntype> (<parameter list>)
```

i32 (i32)	function taking an i32, returning an i32
float (i16, i32 *) *	Pointer to a function that takes an i16 and a pointer to i32, returning float.
i32 (i8*, ...)	A vararg function that takes at least one pointer to i8 (char in C), which returns an integer

Pointers and Vectors

Pointer Types

```
<type>*
```

[4 x i32]*	A pointer to array of four i32 values.
i32 (i32*) *	A pointer to a function that takes an i32*, returning an i32.

Vectors

```
< <# elements> x <elementtype> >
```

<4 x i32>	Vector of 4 32-bit integer values.
<8 x float>	Vector of 8 32-bit floating-point values.
<2 x i64>	Vector of 2 64-bit integer values.

Arrays and Structs

Arrays Types

```
[<# elements> x <elementtype>]
```

[40 x i32]	Array of 40 32-bit integer values.
[12 x [10 x float]]	12x10 array of single precision floating point values.
[2 x [3 x [4 x i16]]]	2x3x4 array of 16-bit integer values.

Struct Types

```
%T1 = type { <type list> }      ; Normal struct type
%T2 = type <{ <type list> }>    ; Packed struct type
```

{ i32, i32, i32 }	A triple of three i32 values
{ float, i32 (i32) *}	A pair, where the first elem. is a float and the second element is a pointer to a function that takes an i32, returning an i32.
<{ i8, i32 }>	A packed struct has no alignment or padding

Arrays and Structs

Unclear what this is for; 0 means unknown?

https://llvm.org/docs/GetElementPtr.html#what-happens-if-an-array-index-is-out-of-bounds

Arrays Types

```
[<# elements> x <elementtype>]
```

[40 x i32]	Array of 40 32-bit integer values.
[12 x [10 x float]]	12x10 array of single precision floating point values.
[2 x [3 x [4 x i16]]]	2x3x4 array of 16-bit integer values.

Struct Types

```
%T1 = type { <type list> }      ; Normal struct type
%T2 = type <{ <type list> }>    ; Packed struct type
```

{ i32, i32, i32 }	A triple of three i32 values
{ float, i32 (i32) *}	A pair, where the first elem. is a float and the second element is a pointer to a function that takes an i32, returning an i32.
<{ i8, i32 }>	A packed struct has no alignment or padding

Generating LLVM Code

High-Level Approach

It is not necessary to directly produce SSA form:

- Allocate all variables on the stack
- Store instructions are not limited by SSA form

```
store i32 %x, i32* %p, align 4
```

- Use LLVM's **mem2reg** optimization to turn stack locations into variables
  - promotes memory references to be register references
  - changes alloca instructions which only have loads and stores as uses
  - introduces phi functions

Options

- Using the LLVM C++ interface & OCaml or Haskell bindings
- Generating an LLVM assembly (.ll file)

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

- Generating LLVM bitcode (.bc file)

```
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
```

Options

- Using the LLVM C++ interface & OCaml or Haskell bindings

- Generating an LLVM assembly (.ll file)

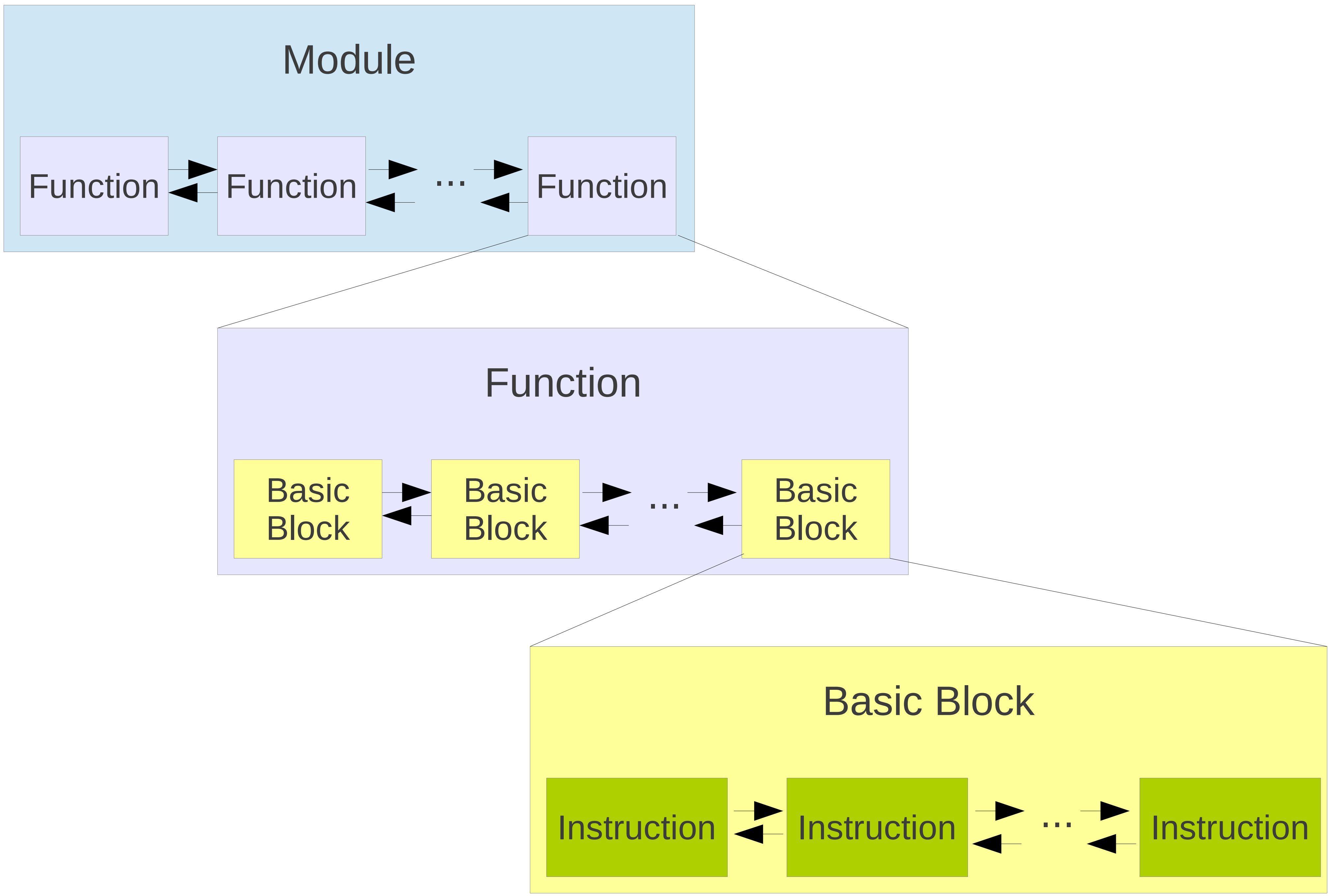
```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
...
```

Recommended.

- Generating LLVM bitcode (.bc file)

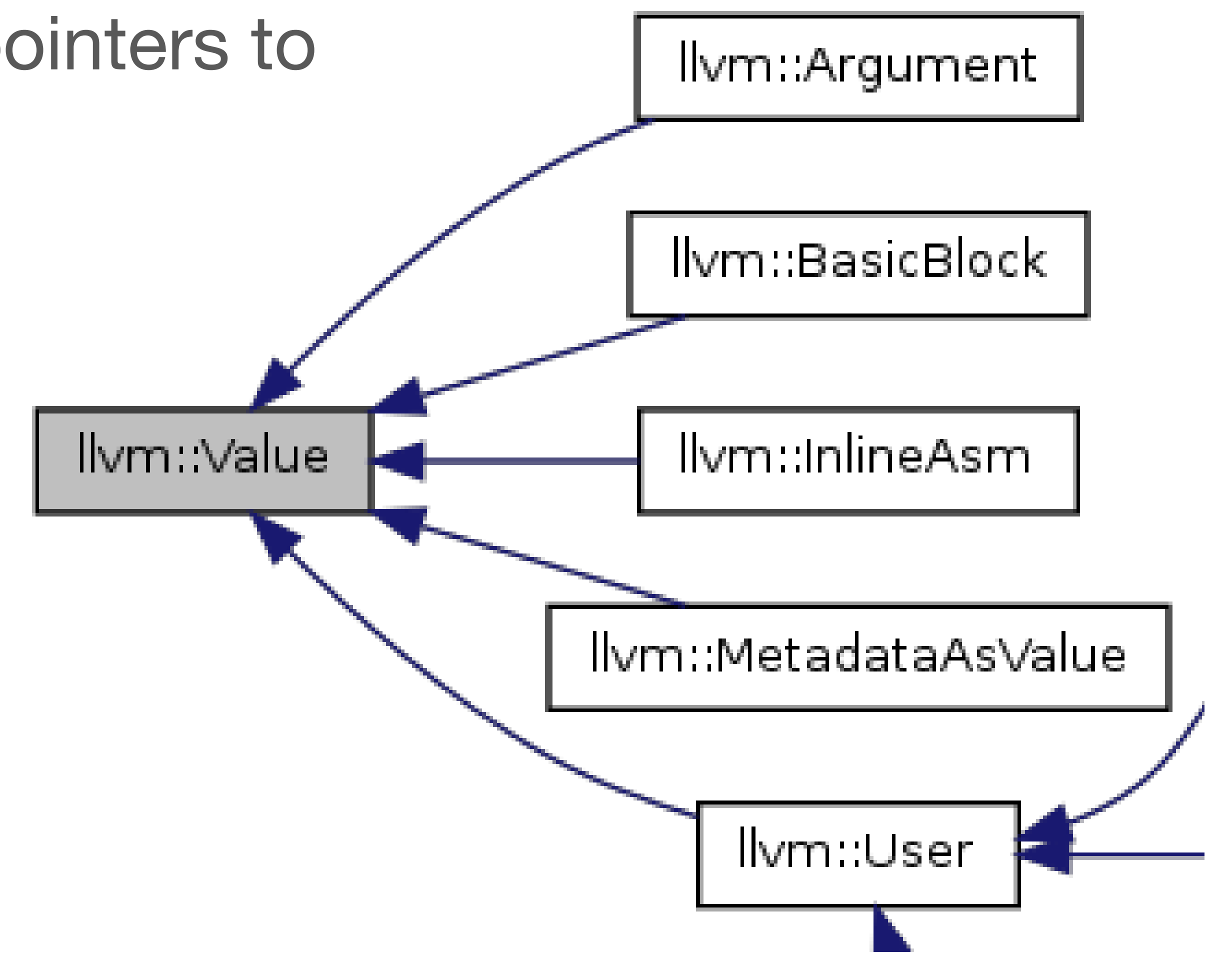
```
42 43 C0 DE  21 0C 00 00
06 10 32 39  92 01 84 0C
0A 32 44 24  48 0A 90 21
18 00 00 00  98 00 00 00
E6 C6 21 1D  E6 A1 1C DA
```

C++ Interface


Module
Function
Function
...
Function
Function
Basic Block
Basic Block
...
Basic Block
Basic Block
Instruction
Instruction
...
Instruction

C++ Interface

- Object oriented
- Instruction doubles as reference for written value
- Every value contains a list of pointers to instructions that use the value



.ll Files

```
; ModuleID = 'llvm.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.10.0"

; Function Attrs: nounwind ssp uwtable
define i32 @add(i32 %x) #0 {
  %1 = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 8128, i32* %y, align 4
  %2 = load i32* %1, align 4
  %3 = load i32* %y, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}

; Function Attrs: nounwind ssp uwtable
define i32 @loop(i32 %n) #0 {
  %1 = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  %2 = load i32* %1, align 4
  store i32 %2, i32* %i, align 4
  br label %3

; <label>:3                                       ; preds = %6, %0
  %4 = load i32* %i, align 4
  %5 = icmp slt i32 %4, 1000
  br i1 %5, label %6, label %9

; <label>:6                                       ; preds = %3
  %7 = load i32* %i, align 4
  %8 = add nsw i32 %7, 1
  store i32 %8, i32* %i, align 4
  br label %3

; <label>:9                                       ; preds = %3
  %10 = load i32* %i, align 4
  ret i32 %10
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false"
"no-frame-pointer-elim"="true" “no-frame-pointer-elim-non-leaf”… }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{!"Apple LLVM version 7.0.0 (clang-700.0.72)"}
```

LLVM Coding Basics

- **Written in modern C++, uses the STL:**
  - Particularly the vector, set, and map classes

- **LLVM IR is almost all doubly-linked lists:**
  - **Module** contains lists of **Functions** & **GlobalVariables**
  - **Function** contains lists of **BasicBlocks** & **Arguments**
  - **BasicBlock** contains list of **Instructions**

- **Linked lists are traversed with iterators:**

```
Function *M = …
for (Function::iterator I = M->begin(); I != M->end(); ++I) {
  BasicBlock &BB = *I;
  ...
```

See also: docs/ProgrammersManual.html

LLVM Pass Manager

- **Compiler is organized as a series of "passes":**
  - Each pass is one analysis or transformation
- **Four types of passes:**
  - 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

See also:
docs/WritingAnLLVMPass.html

LLVM tools

- **Basic LLVM Tools**
  - llvm-dis: Convert from .bc (IR binary) to .ll (human-readable IR text)
  - llvm-as: Convert from .ll (human-readable IR text) to .bc (IR binary)
  - opt: LLVM optimizer
  - llc: LLVM static compiler
  - llvm-link - LLVM bitcode linker
  - llvm-ar - LLVM archiver
- **Some Additional Tools**
  - bugpoint - automatic test case reduction tool
  - llvm-extract - extract a function from an LLVM module
  - llvm-bcanalyzer - LLVM bitcode analyzer
  - FileCheck - Flexible pattern matching file verifier
  - tblgen - Target Description To C++ Code Generator

See also:
http://llvm.org/docs/CommandGuide/

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");
```

Lecture 6
More on the LLVM Compiler

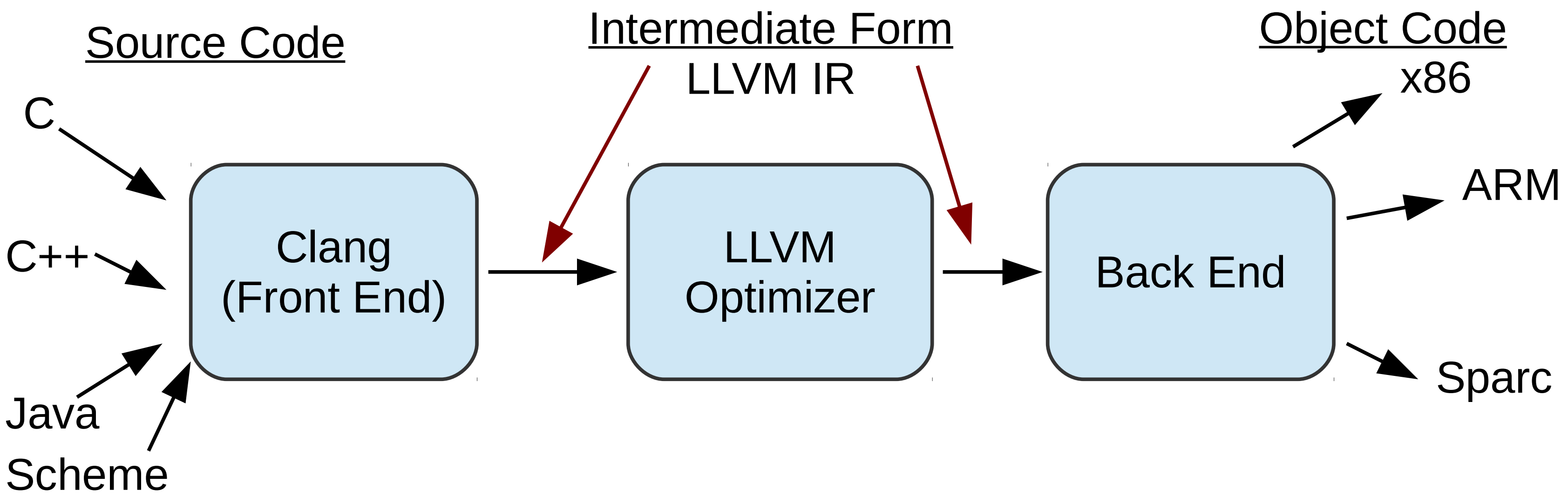
Deby Katz

Thanks to Luke Zarko and Gabe Weisz for some content

Outline

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

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

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

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`
- **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,

```
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

Understanding the LLVM IR - Binary Instructions

- **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

Understanding the LLVM IR - Memory Instructions

- **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:**

```
%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)

Understanding the LLVM IR - SSA

- **LLVM uses Static Single Assignment (SSA) to represent memory**
  - More on SSA later in the class
- **May produce “phi” instructions**
  - First instruction(s) in a BB
  - Give the different potential values for a variable, depending on which block preceded this one
- **Arbitrary/unlimited number of abstract “registers”**
  - Actual register use is determined at a lower level - target dependent
  - Can use as many as you want
  - Really, they are stack locations or SSA values

Understanding the LLVM IR - Instructions as Values

- **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 (iN of N bits, N from 1 to 2^{23}-1), Floating point (half, float, double, etc.)
  - Others (x86mmx, void, etc.)
- **Derived Types**
  - Arrays ([# elements (>= 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**
  - Changing the LLVM IR
- **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,

```
// 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

Writing Passes - Correctness

- **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**

Writing Passes - Module Invariants

- **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!**

Writing Passes - Parameters

- **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

Outline

- **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

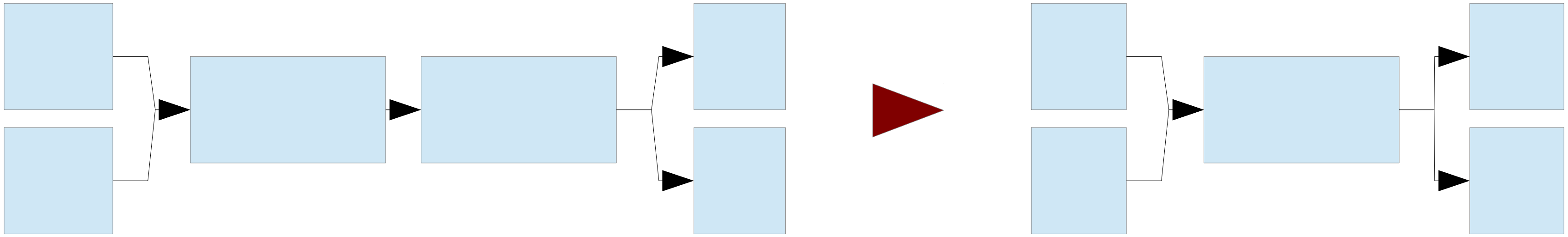
- **If you have alloca instructions that only have loads and stores as uses**
  - Changes them to register references
  - May add SSA features like “phi” instructions
- **Sometimes useful for simplifying IR**
  - Confuses easily

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**
  - 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/
  - May be easier to search the internal documentation from the http://llvm.org front page

Further Reading

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

http://www.cs.cmu.edu/afs/cs/academic/class/15745-s13/public/lectures/L6-LLVM-Detail-1up.pdf