15-411: LLVM

Jan Hoffmann

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 the C++ standard library

- **LLDB:** debugger for C, C++, and Objective C

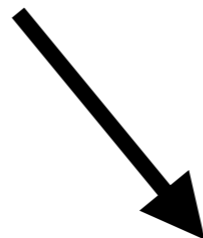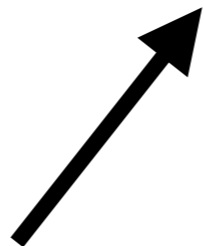- **dragonegg:** parser front end for compiling Fortran, Ada, …

- …

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

S

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

…

LLVM IR

Example 1

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

Clang →

Functions are parametrized with arguments and types.

```
; 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.

No signed wrap: result of overflow undefined.

Example2

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

Clang →

Basic blocks.

Predecs. in CFG.

```
; 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
}
```

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"

Single Value Types

Integer Types

Size.

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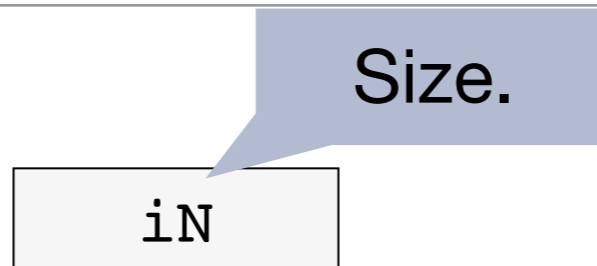
i1	a single-bit integer.
i32	a 32-bit integer.
i1942652	a really big integer of over 1 million bits.

Float Types

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

Functions and Void

Void

No representation and no size.

```
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

Unclear what this is for; 0 means unknown?

```
[<# 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

  ...
```
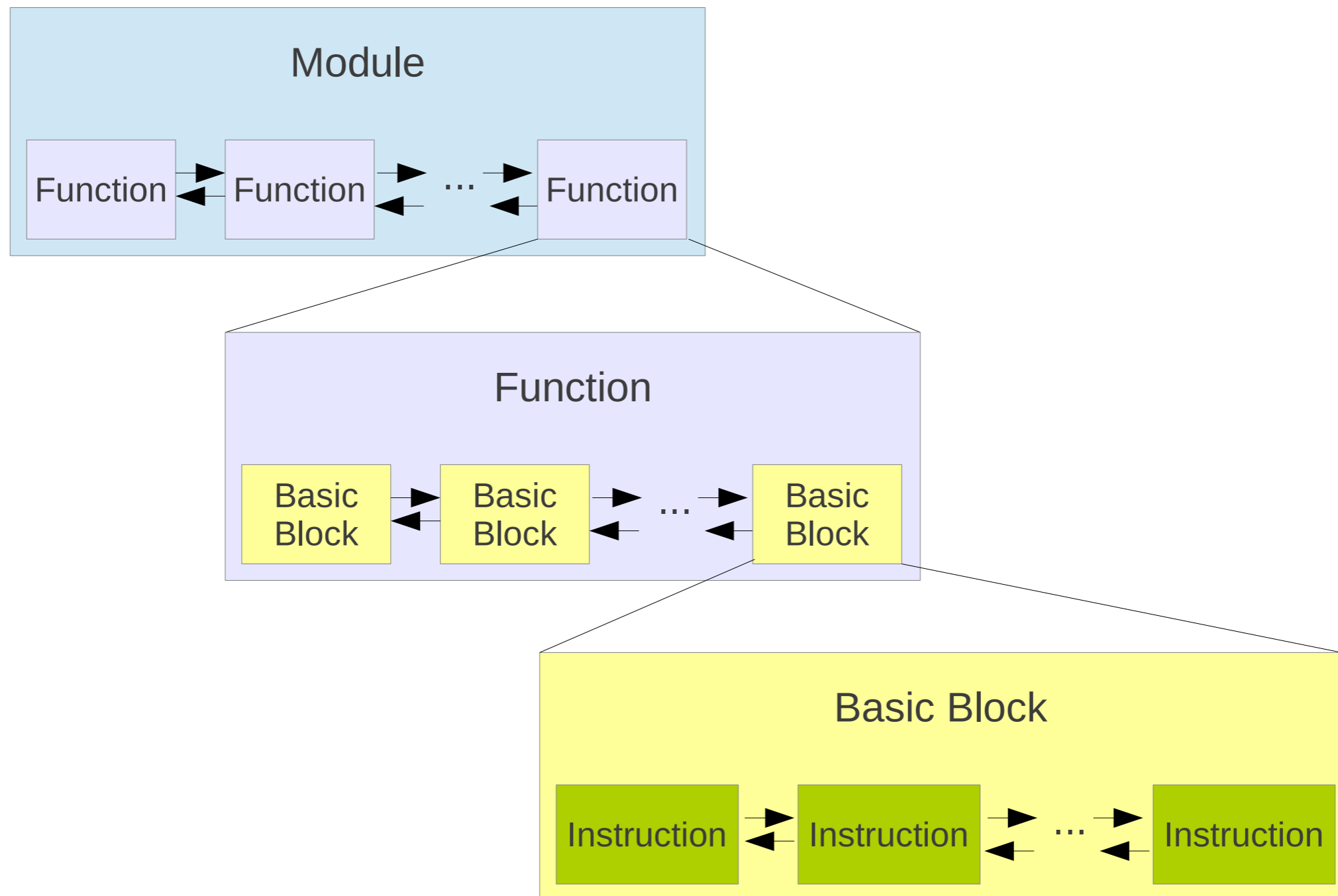
Recommended.
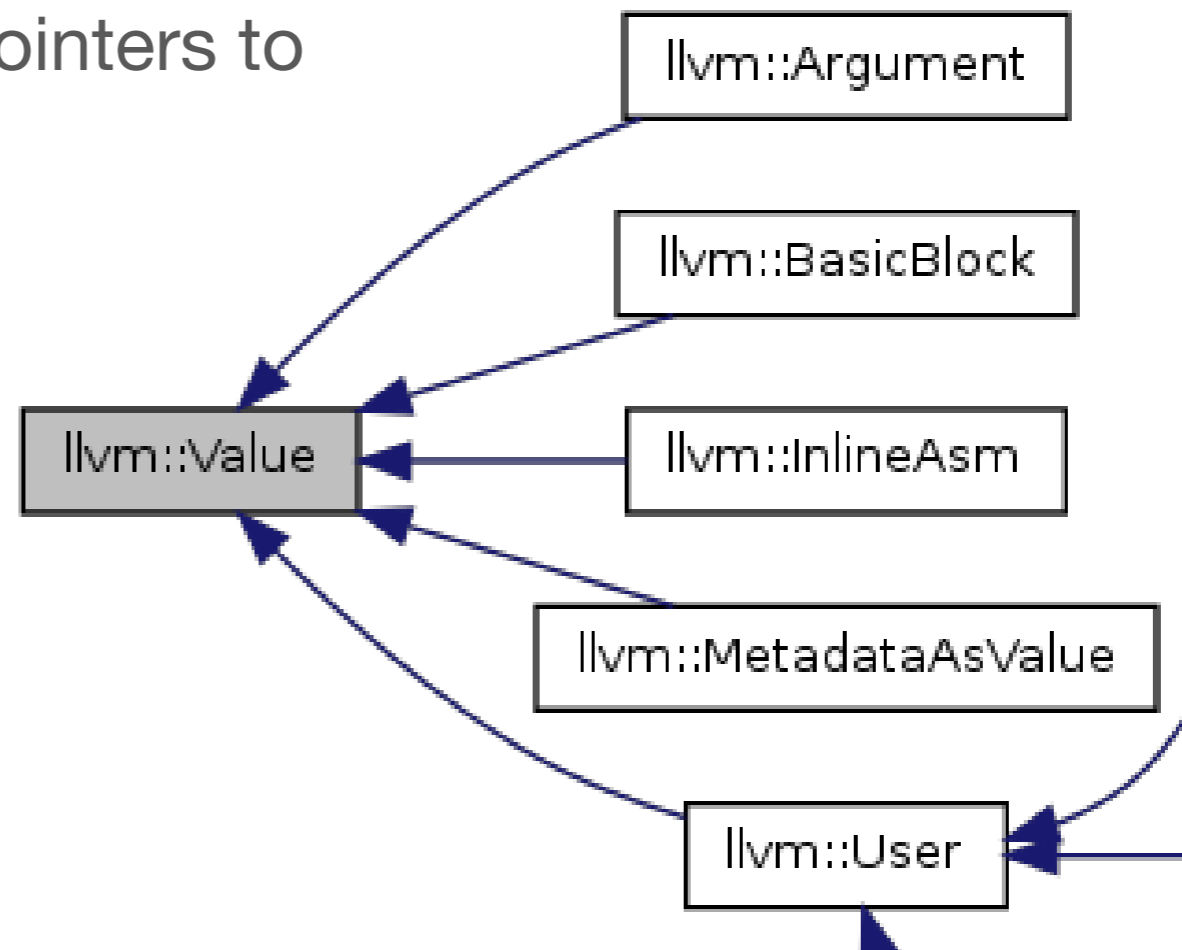
- Generating LLVM bitcode (.bc file)

```
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

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)"}
```

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