INTRODUCTION TO LLVM

Bo Wang
SA 2016 Fall
OUTLINE

- LLVM Basic
- LLVM IR
- LLVM Pass
What is LLVM?

- LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces.
  - Implemented in C++
  - Several front-ends
  - Several back-ends
  - First release: 2003
  - The original author: Chris Lattner (PhD of UIUC)
  - Open source http://llvm.org/
LLVM is a Compilation Infrastructure

It is a framework that comes with a lots of tools to compile and optimize code.
A First Look

1. PATH/clang –emit-llvm –c hello.c –o hello.bc
2. PATH/lli hello.bc
3. PATH/llvm-dis < hello.bc | less
   or
   PATH/llvm-dis hello.bc
   or
   PATH/clang –emit-llvm –S hello.c
Why to learn LLVM?

- Intensively used in the academia:
  - LLVM: A compilation framework for lifelong program analysis & transformation

- Widely used in the industry
  - LLVM is supported by Apple
  - ARM, NVIDIA, Mozilla, etc.

- Clean and modular interfaces

- Awards: ACM Software System Award 2012
  - UNIX, TCP/IP, WWW, Java, Apache, Eclipse, gcc, make, VMware, LLVM...
Big Picture of LLVM

- LLVM implements the entire compilation flow.
  - Front-end, e.g., clang (C), clang++ (C++)
  - Middle-end, e.g., analyses and optimizations
  - Back-end, for different computer architectures, e.g., MIPS, x86, ARM
Middle-end: LLVM IR

- **IR**: Intermediate Representation
  - RISC like instruction set
  - Well typed representation
  - SSA format: Each variable noun has only one definition
  - Three types of format
    - in memory (JIT)
    - byte code (.bc)
    - human readable (.ll)
A First Look at IR

CMD : YOUR_BUILD_PATH/bin/clang -emit-llvm -S 1st.c

1st.c

```c
int foo(int a){
    int res;
    if(a > 0){
        res = 1;
    }else{
        res = 0;
    }
    return res;
}
```

1st.ll

```llvm
; Function Attrs: nounwind uwtable
define i32 @foo(i32 %a) #0 {
  entry:
  %a.addr = alloca i32, align 4
  %res = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 1, i32* %res, align 4
  br label %if.end

if.else:
  store i32 0, i32* %res, align 4
  br label %if.end

if.end:
  %1 = load i32, i32* %res, align 4
  ret i32 %1
}
```

- All the types of IR:
  - llvm/include/llvm/IR/Instruction.def

- Document:
  - http://llvm.org/docs/LangRef.html
LLVM Core Hierarchy

- Module contains Functions/GlobalVariables
  - Module is unit of compilation/analysis/optimization
- Function contains BasicBlocks/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
  - All operands have types
  - Instruction result is typed
The Module

- What is the modules?
  - Modules represent the top-level structure in an LLVM program.
  - An LLVM module is effectively a translation unit or a collection of translation units merged together.

- Why C need modules?
  - Python: interpreter-based
  - Java: All members of a class within a java src
  - C/C++: linkage, the scope of identifiers
The Function

- Name
- Argument list
- Return type
- Extends from `GlobalValue`, has properties of linkage visibility.
The Value

- Value: can be treated as arbitrary num of registers.
- Locals start with %, globals with @
- All instructions that produce values can have a name (Not assignments: store, br)
Type

- Not exactly what PL people think of as types
- All values have a static type
- Integer: \( \text{iN} \); for C --- \( \text{i1, i8, i16, i32, i64} \)…
- Float: float, double, half
- Arrays: can get num of elements
- Structures: can get members, like \( \{\text{i32, i32, i8}\} \)
- Pointers: can get the pointed value
- Void
Note on Integer Types

- There are no signed or unsigned integers
- LLVM views integers as bit vectors
- Frontends destroyed signed/unsigned information
- Operations are interpreted as signed or unsigned based on instructions they are used in
  - icmp sgt v.s. icmp ugt
  - sdiv v.s. udiv
BasicBlock & Instruction

- Classify Instructions
  - Terminator Instructions: ret, switch, br (cond & uncond)...
  - Binary operators: add, sub...
  - Logical operators: and, or, shl...
  - Memory operators: alloca, load, store...
  - Cast operators ...
  - Others: icmp, phi, call...

- Contains a list of Instructions

- In general, every basic block must end with a Terminator Instruction
More Detail of Phi nodes

- Phi nodes – construct to handle cases where a variable may have more than one value
  - May be self referential (in loops)
  - Inside a block – select statement sometimes used

- In LLVM:
  - Must be at the beginning of the block
  - Must have exactly 1 entry for every predecessor
  - Must have at least one entry
  - May include undef values
LLVM Pass

- Normal Compiler Organization

```
Program
  Front end
  Abstract Syntax Tree
    High-level IR
      High-level optimization
      Parallelization
      Loop transformations
    Machine-Independent Intermediate Representations
      Low-level IR
        Low-level optimization
        Redundancy elimination
      Code generation
        Machine code
        Register allocation
        Instruction scheduling
```

Passes shall stay here!
LLVM Pass

- LLVM applies a chain of analyses and transformations on the target program.
- Each of these analyses or transformations is called a pass.
- Some passes, which are machine independent, are invoked by opt.
- A pass may require information provided by other passes. Such dependencies must be explicitly stated.
LLVM Pass

- A pass is an instance of the LLVM class *Pass*
- There are many kinds of passes

In this lesson we will focus on Function Passes, which analyze whole functions.
A First Look at LLVM Passes

- Memory To Register (-mem2reg)

```
int foo(int a){
    int res;
    if(a > 0){
        res = 1;
    }else{
        res = 0;
    }
    return res;
}
```

YOURPATH/opt -mem2reg 1st.bc -S -o 1stm2r.ll

```
; Function Attrs: nounwind uwtable
define i32 @foo(i32 %a) #0 {
  entry:
  %a.addr = alloca i32, align 4
  %res = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else
  if.then:
    br label %if.end
  if.else:
    br label %if.end
  if.end:
    %res.0 = phi i32 [ %0, %if.else ], [ %1, %if.then ]
  ret i32 %res.0
}
```

YOURPATH/clang -emit-llvm -S 1st.c -o 1st.ll
A First Look at LLVM Passes

- **Draw a CGF (-mem2reg)**
  1. `sudo apt-get install graphviz`
  2. `opt --dot-cfg hello.bc`
  3. `dot -Tpng -o cfg.png cfg.foo.dot`
Review: Liveness Analysis

- If we assume an infinite supply of registers, then a variable \( v \) should be in a register at a program point \( p \), whenever:
  1. There is a path \( P \) from \( p \) to another program point \( p_w \), where \( v \) is used.
  2. The path \( P \) does not go across any definition of \( v \).

Why is the second condition really necessary?

Can you define the liveness problem more formally?
Review: Textbook Liveness Analysis

- Liveness analysis: **Backwards, may, union.**

**Algorithm**

```plaintext
for each node n in CFG
    in[n] = ∅; out[n] = ∅  \{ Initialize solutions \}
repeat
    for each node n in CFG in reverse topsort order
        in'[n] = in[n]
        out'[n] = out[n]
        out[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]
        in[n] = \text{use}[n] \cup (out[n] - \text{def}[n])  \{ Save current results \}
    until in'[n]=in[n] and out'[n]=out[n] for all n  \{ Test for convergence \}
```
Review: Textbook Liveness Analysis

- **Complexity**
- **Time**
  - Worst case: $O(n^4)$
  - Typical case: $O(N)$ to $O(N^2)$
- **Space**
  - $O(N^2)$
Can you point where i2 is alive in this program?
SSA Form Liveness Analysis

Can you point where \( i2 \) is alive in this program?

Why the phi-node \( i4 \) is excluded?
SSA Form Liveness Analysis

Without traversing the CFG to reach a fixed point.

Space: $O(N)$
Time: $O(N)$ to $O(N^2)$

For each statement $S$ in the program:
$$IN[S] = OUT[S] = \{\}$$

For each variable $v$ in the program:
For each statement $S$ that uses $v$:
$$\text{live}(S, v)$$

$$\text{live}(S, v):$$
$$IN[S] = IN[S] \cup \{v\}$$
For each $P$ in $\text{pred}(S)$:
$$\text{OUT}[P] = \text{OUT}[P] \cup \{v\}$$
if $P$ does not define $v$
$$\text{live}(P, v)$$
Is Traditional DA Useless?

- Where should we add a phi-function for the definition of $i$ at $L2$. 

```
L0: j = 1

L1: if j < 100 goto L2

L2: i = j + 1
L3: if j < 20 goto L5

L4: j = i + 1
L5: j = i + 2

L6: goto L3

L7: ret j
```
Is Traditional DA Useful?

- The phi-function at $L1$ exists even though it is not useful at all.
- We can add a liveness check to the algorithm that inserts phi-functions.
The LLVM Pass in Action

- Naive Liveness Analysis for LLVM IR
- Function Pass
- LLVM API
  - Iterating basic blocks, instructions and operands.
  - Instruction casting
  - ...
- The code
  - http://pan.baidu.com/s/1pLRfCEn