INTRODUCTION TO LLVM

Bo Wang
wangbo_pku_15[AT]163.com
SA Class, 2017 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/

Fortran
C/C++
Java
Your cool language

X86
ARM
PowerPC
SPARC
MIPS

LLVM is a Compilation Infra-Structure

It is a framework that comes with a lots of tools to compile and optimize code. **clang, clang++, llc, lli, llvm-dis, opt...**

```
nightwish@nightwish-TP [02:30:42 PM] [~/code/git/newest_llvm/llvm/build/bin] [master *]
-> % ls
arcmt-test              llvm-config       llvm-PerfectShuffle
bugpoint                llvm-cov          llvm-profdata
c-arcmt-test            llvm-c-test       llvm-ranlib
c-index-test            llvm-cxxdump      llvm-readobj
clang                   llvm-cxxfilt      llvm-rtdyld
clang++                 llvm-diff         llvm-size
clang-4.0               llvm-dis          llvm-split
clang-check             llvm-dsymutil     llvm-stress
clang-cl                llvm-dwarfdump    llvm-symbolizer
clang-cpp               llvm-dwp          llvm-tblgen
clang-format            llvm-extract      not
clang-offload-bundler   llvm-lib          obj2yaml
clang-tblgen            llvm-link         opt
count                   llvm-lit          sancov
diagtool                llvm-lto          sanstats
FileCheck               llvm-lto2         scan-build
llc                     llvm-mc           scan-view
lli                     llvm-mcmarkup     verify-uselistorder
lli-child-target        llvm-nm           yaml2obj
llvm-ar                 llvm-objdump      yaml-bench
llvm-as                 llvm-opt-report
```

LLVM is a Compilation Infra-Structure

- Compile a C program:

```
$> echo "int main(){return 26;}" > test.c
$> ~/llvm/build/bin/clang test.c
$> ./a.out
$> echo $?
 26
```

Usually, clang/clang++ have faster compilation times than gcc, and the compilation error message is much more readable.

Why to learn LLVM?

- Intensively used in the academia:

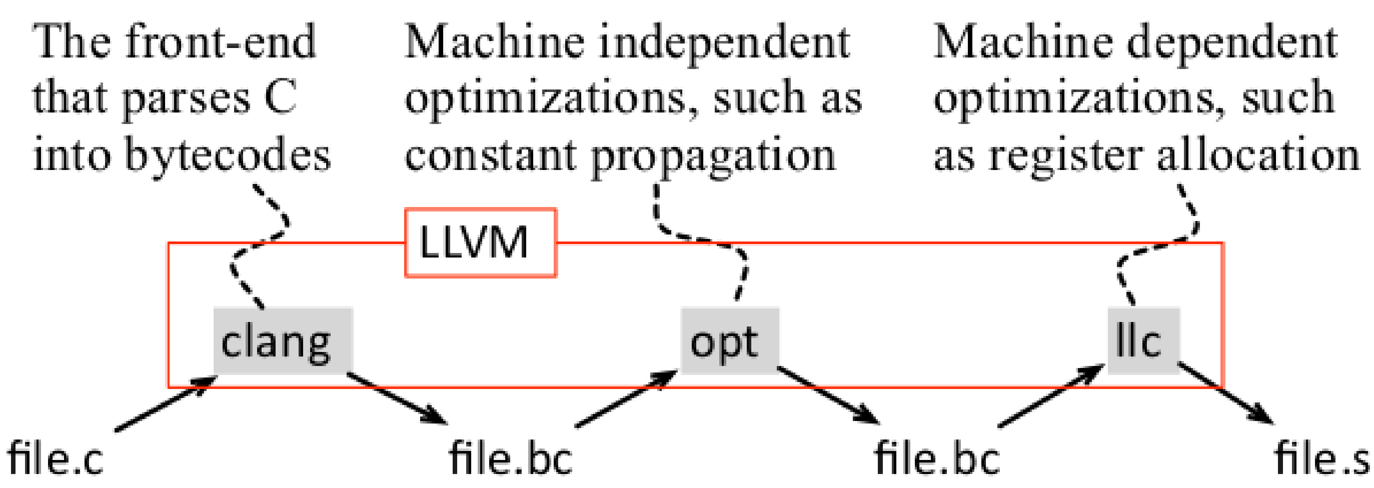
LLVM: A compilation framework for lifelong program analysis & transformation
C Lattner, V Adve - Proceedings of the international symposium on Code …, 2004 - dl.acm.org
Abstract This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support transparent, lifelongprogram analysis and transformation for arbitrary programs, by providing high-level information to compilertransformations at compile-time, link-time, run-time, and inidle time between runs. LLVM defines a common, low-levelcode representation in Static Single Assignment (SSA) form, with several novel features: a …
☆ 99 被引用次数：3261 相关文章 所有 58 个版本

In Prof. Xiong's Group:
- ICSE'15 (MemLeak)
- ICSE'16 (Compiler Testing)
- ICSE'17 (Compiler Testing)
- ISSTA'17 (Testing)

- 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, Apahe, 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



Off-the-shell Optimizations

```
$> opt –help
General options:
  -O0                                    - Optimization level 0. Similar to clang -O0
  -O1                                    - Optimization level 1. Similar to clang -O1
  -O2                                    - Optimization level 2. Similar to clang -O2
  -O3                                    - Optimization level 3. Similar to clang -O3
  -Os                                    - Like -O2 with extra optimizations for size. Similar to clang -Os
  -Oz                                    - Like -Os but reduces code size further. Similar to clang -Oz

Optimizations available:
……..
-globaldce                               - Dead Global Elimination
-dot-cfg                                 - Print CFG of function to 'dot' file
-dot-callgraph                           - Print call graph to 'dot' file
-dot-dom                                 - Print dominance tree of function to 'dot' file
-dce                                     - Dead Code Elimination
-adce                                    - Aggressive Dead Code Elimination
-always-inline                           - Inliner for always_inline functions
……..
```

Levels of Optimizations

llvm-as: assembler of LLVM. It reads human-readable LLVM-IR, translates it to LLVM bytecode, and writes the result in to a file.

```
$> llvm-as < /dev/null | opt -O1 -disable-output -debug-pass=Arguments
Pass Arguments:  -tti -tbaa -scoped-noalias -assumption-cache-tracker...
…
...
```

You can get your passes used by -O1 level.
In my system, -O1 gives me:

Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -instcombine -simplifycfg -pgo-icall-prom -basiccg -globals-aa -prune-eh -always-inline -functionattrs -domtree -sroa -early-cse -speculative-execution -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -basicaa -aa -instcombine -tailcallelim…

Virtual Register Allocation

- One of the most basic optimizations that opt maps memory slots into variables.
- This optimization is very useful, because clang maps every variable to memory

```
#include<stdio.h>
int main(){
    int c1 = 11;
    int c2 = 15;
    int c3 = c1 + c2;
    printf("%d\n", c3);
}
```

```
%0:
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 store i32 11, i32* %1, align 4
 store i32 15, i32* %2, align 4
 %4 = load i32, i32* %1, align 4
 %5 = load i32, i32* %2, align 4
 %6 = add nsw i32 %4, %5
 store i32 %6, i32* %3, align 4
 %7 = load i32, i32* %3, align 4
 %8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
... i8]* @.str, i32 0, i32 0), i32 %7)
 ret i32 0
```

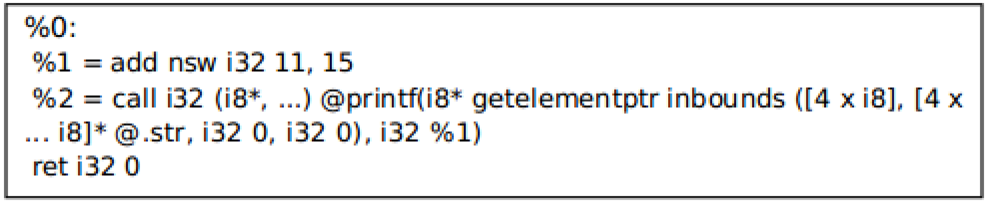
CFG for 'main' function

```
$>clang -c -emit-llvm test.c -o test.bc
$>opt --view-cfg test.bc    #maybe you need sudo apt-get install xdot
```

Virtual Register Allocation

- One of the most basic optimizations that opt maps memory slops into variables.
- We can map memory slots into registers with the **mem2reg** pass.

```
#include<stdio.h>
int main(){
     int c1 = 11;
     int c2 = 15;
     int c3 = c1 + c2;
     printf("%d\n", c3);
}
```



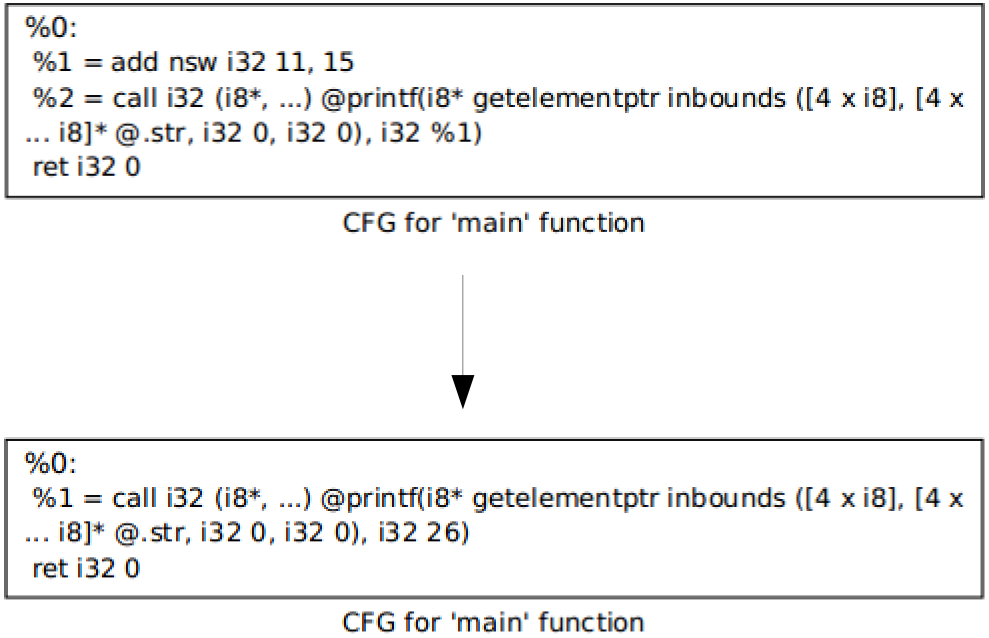
CFG for 'main' function

```
$>opt -mem2reg test.bc > test.reg.bc
$>opt --view-cfg test.reg.bc    #maybe you need sudo apt-get install xdot
```

Constant Propagation

- Constant folding by **constprop** pass

```
#include<stdio.h>
int main(){
     int c1 = 11;
     int c2 = 15;
     int c3 = c1 + c2;
     printf("%d\n", c3);
}
```




```
$>opt -constprop test.reg.bc > test.cp.bc
$>opt --view-cfg test.cp.bc     #maybe you need sudo apt-get install xdot
```

OUTLINE

- LLVM Basic
- **LLVM IR**
- LLVM Pass

A First Look at IR

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

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

1st.c

```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
}
```

1st.ll

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

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

Middle-end: LLVM IR

- IR: Intermediate Representation
  - RISC like instruction set: *add, mul, or, branch, load, store...*
  - Well typed representation: *%0 = load i32* %addr*
  - SSA format: *Each variable noun has only one definition*
  - The LLVM optimizations manipulate these bytecodes
  - We can program directly on them.
  - We can also interpret them

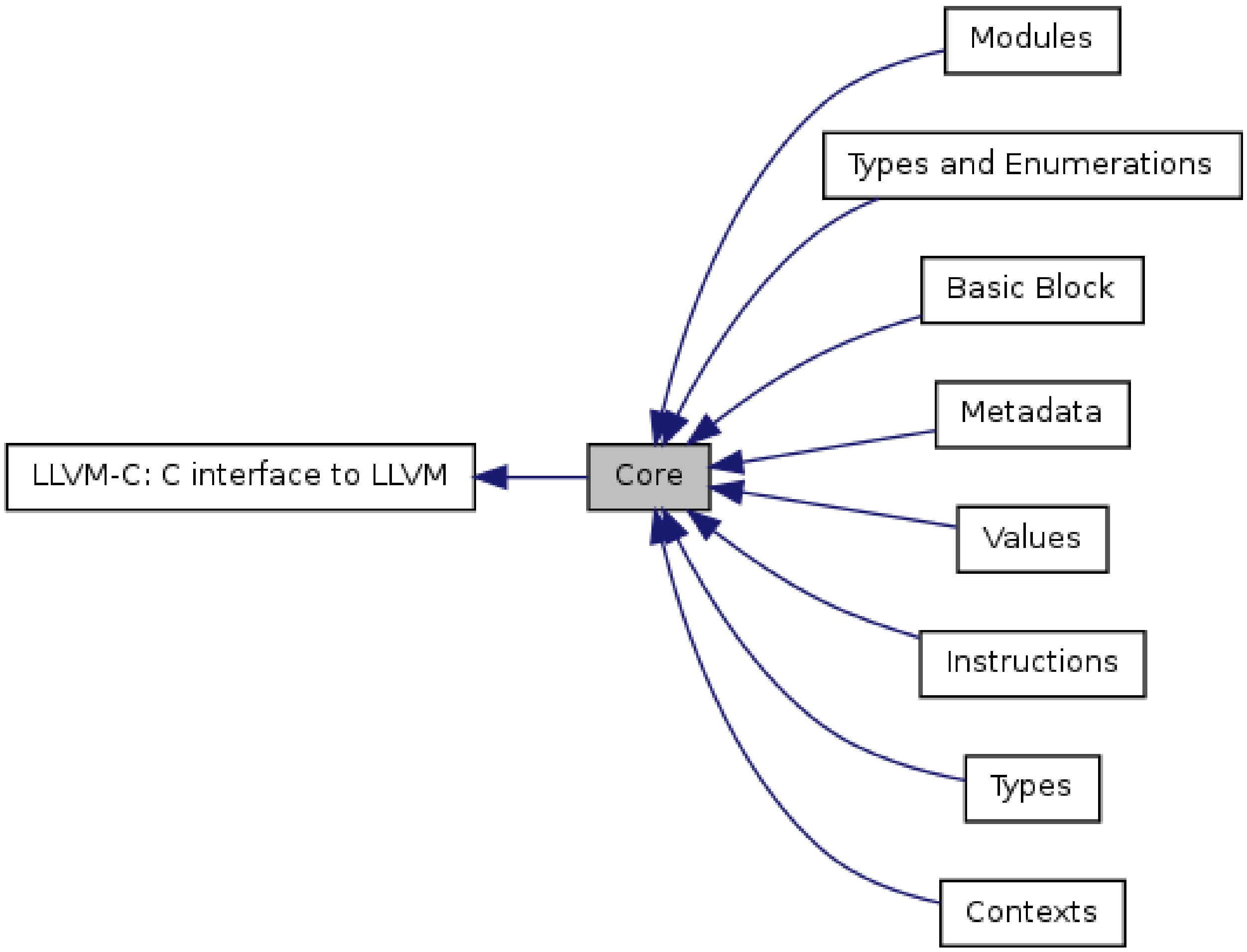
```
$> lli test.bc
```

Back-end: From IR to Machine Code

- llc: the tool to perform translation from IR to architecture specified machine code.

```
$> llc –version
……
$> llc -march=x86 test.cp.bc -o test.x86.S
$> cat test.x86.S
…...
```

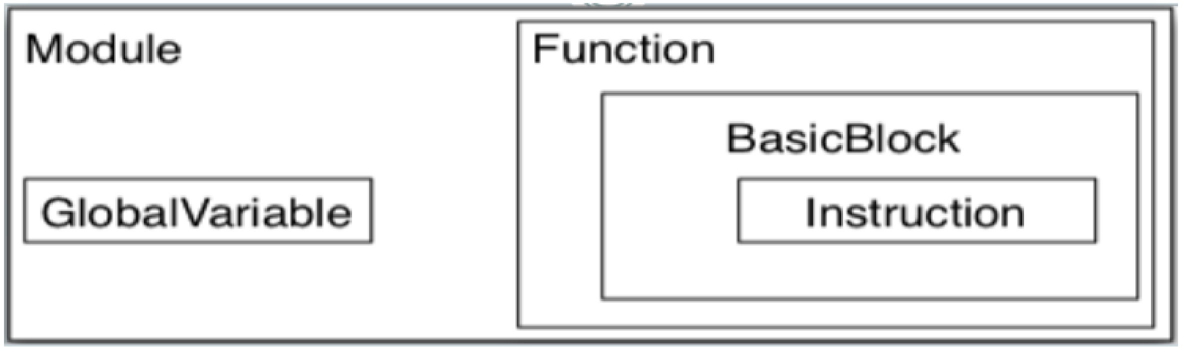
LLVM-IR Core



LLVM-C: C interface to LLVM
Core
Modules
Types and Enumerations
Basic Block
Metadata
Values
Instructions
Types
Contexts

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: iN; for C --- i1, i8, i32, i64…
- Float: float, double, half
- Arrays: can get num of elements
- Structures: can get members, like {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

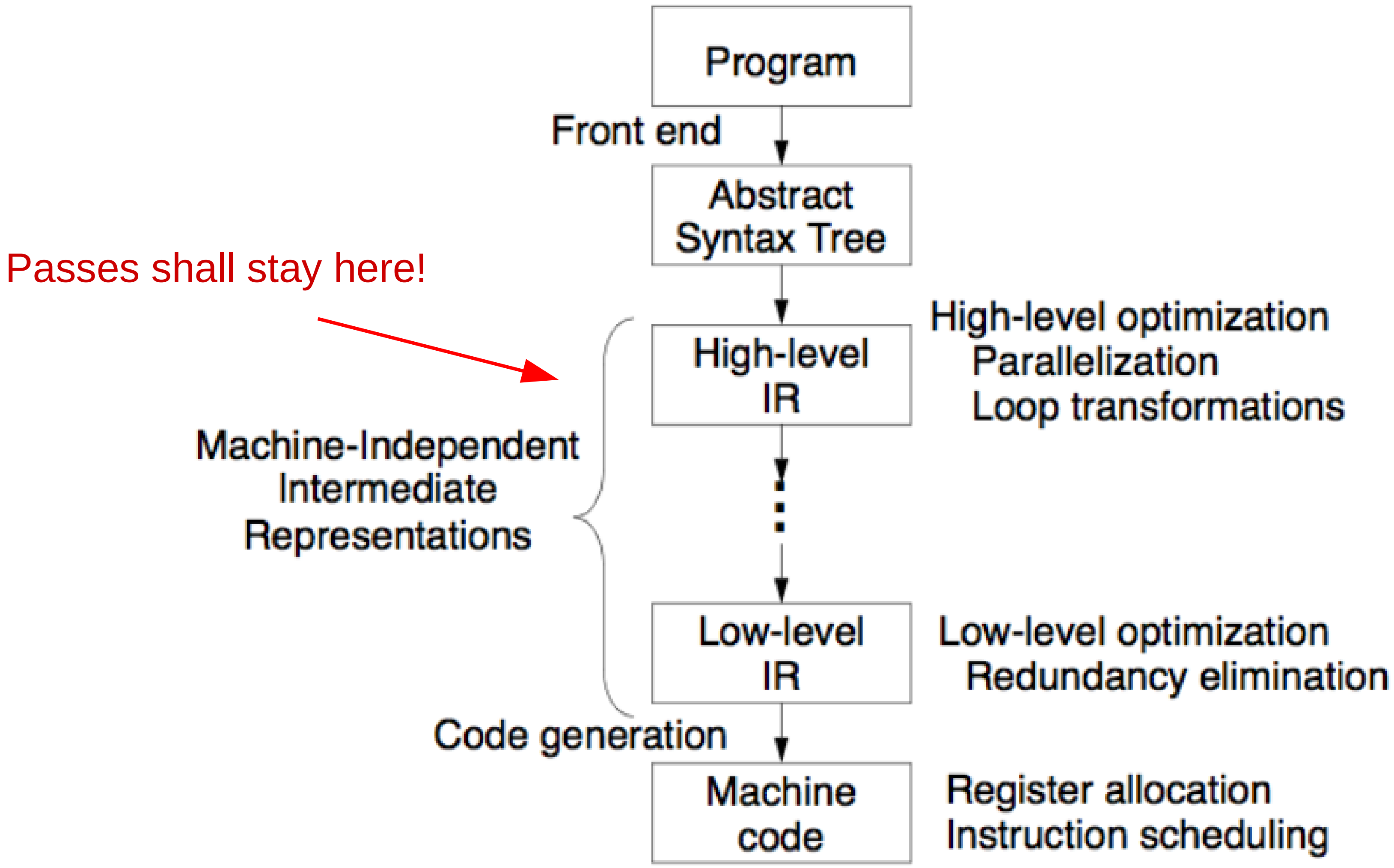
- Phi nodes – construct to handle cases where a variable may have more than one value
  - May be self referential (in loops)
  - Inside a block – select statement sometimes used
- In LLVM:
  - Must be at the beginning of the block
  - Must have exactly 1 entry for every predecessor
  - Must have at least one entry
  - May include undef values

OUTLINE

- LLVM Basic
- LLVM IR
- **LLVM Pass**

LLVM Pass

- Normal Compiler Organization

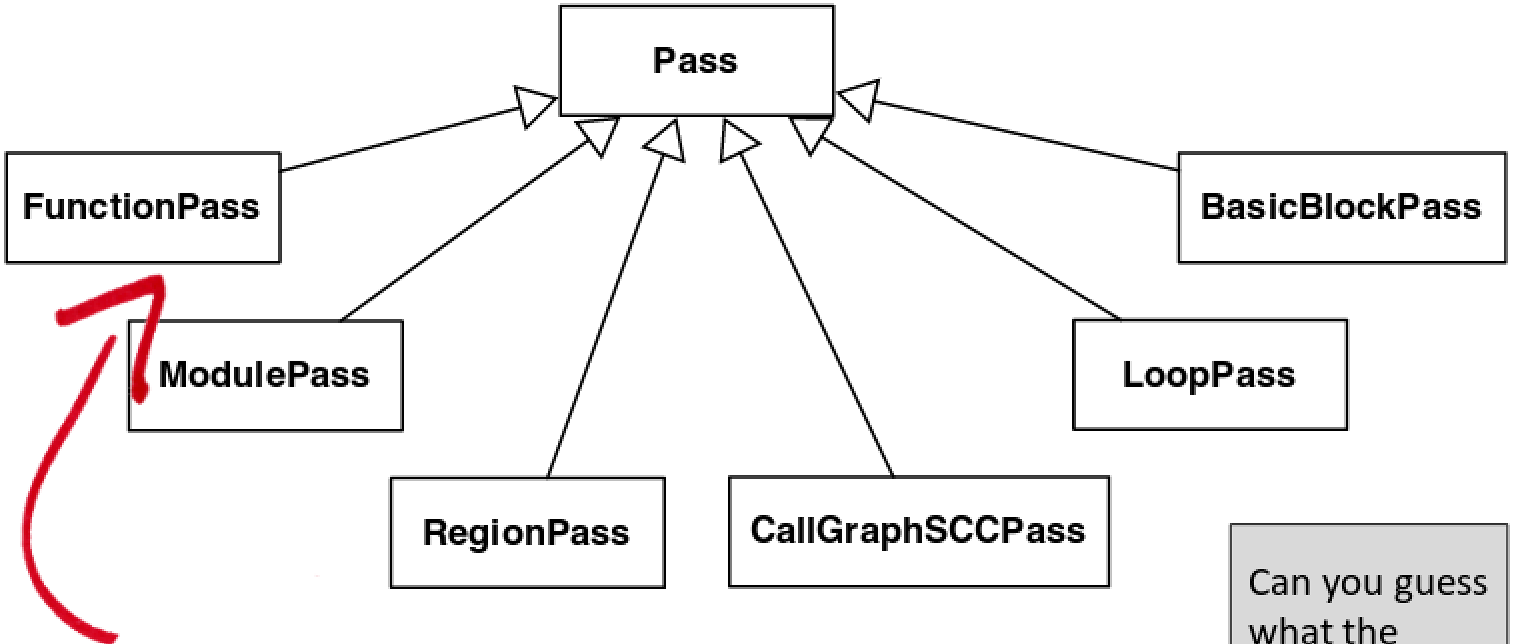


LLVM Pass

- 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.

Can you guess what the other passes are good for?

A First Look at LLVM Passes

- Memory To Register (-mem2reg)

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

1

YOURPATH/clang -emit-llvm -S 1st.c -o 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
}
```

2

```llvm
; Function Attrs: nounwind uwtable
define i32 @foo(i32 %a) #0 {
entry:
  %cmp = icmp sgt i32 %a, 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 [ 1, %if.then ], [ 0, %if.else ]
  ret i32 %res.0
}
```

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

Writing Hello World Pass

- The hello world pass is in the path *llvm/lib/Transforms/Hello/*
- Don’t forget the CMake files in the path and its parent path.
- Don’t forget pass ID and pass registration
- Run the pass with **opt**
- Learn **errs()**

```
$> clang -c -emit-llvm hello.c -o hello.bc
$> opt -load ~/llvm/build/lib/LLVMHello.so -hello < hello.bc > /dev/null
```

http://llvm.org/docs/WritingAnLLVMPass.htm

Counting Opcode Pass

- Let’s write a pass that counts the number of times that each opcode appears in a given function.
- Learn how iterate the data structures.

Counting Opcode Pass

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;
namespace {
  struct CountOp : public FunctionPass {
    std::map<std::string, int> opCounter;
    static char ID;
    CountOp() : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
      errs() << "Function " << F.getName() << '\n';
      for (Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
        for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
          if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
            opCounter[i->getOpcodeName()] = 1;
          } else {
            opCounter[i->getOpcodeName()] += 1;
          }
        }
      }
      std::map <std::string, int>::iterator i = opCounter.begin();
      std::map <std::string, int>::iterator e = opCounter.end();
      while (i != e) {
        errs() << i->first << ": " << i->second << "\n";
        i++;
      }
      errs() << "\n";
      opCounter.clear();
      return false;
    }
  };
}
char CountOp::ID = 0;
static RegisterPass<CountOp> X("opCounter", "Counts opcodes per functions", false, false);
```

1) Make dir
2) Add *CmakeList.txt* (follow the form of *Hello* pass)
3) Modify *CMakeList.txt* in the parent folder
4) Add cpp file with the right-hand code
5) Make and run

Counting Opcode Pass

- Let's write a pass that counts the number of times that each opcode appears in a given function.
- Learn how iterate the data structures.

```c
int foo(int n, int m){
    int sum = 0;
    int c0;
    for(c0 = n; c0 > 0; c0--){
        int c1 = m;
        for(; c1 > 0; c1--){
            sum += c0 > c1 ? 1 : 0;
        }
    }
    return sum;
}
```

```
Function foo
add: 3
alloca: 5
br: 8
icmp: 3
load: 10
ret: 1
select: 1
store: 8
```

```
$> sudo make
$> clang -c -emit-llvm hello.c -o hello.bc
$> opt -load ~/llvm/build/lib/CountOp.so -opCounter < hello.bc > /dev/null
```

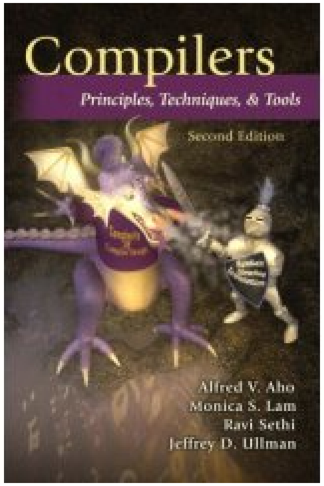
Reading DCE of LLVM

- Dead instruction elimination
  - A single basicblock pass
- Dead code elimination
  - A function pass with fixed point algorithm
  - Call dead instruction elimination pass until fixed.
- Learn how to remove an instruction, discern the type of an instruction and find the usage of a value
- What is ADCE?
  - Starts from the exit points of a function
  - Exit points: ret, memory options...
  - Only preserve instructions related to the exit points

- llvm/lib/Transforms/Scalar/DCE.cpp
- llvm/lib/Transforms/Utils/Local.cpp

Review: Textbook Liveness Analysis

- Liveness analysis: Backwards, may, union.
- Important in register allocation



Algorithm

for each node n in CFG
  $in[n] = \varnothing$; $out[n] = \varnothing$ } Initialize solutions

repeat
  for each node n in CFG **in reverse topsort order**
    $in'[n] = in[n]$
    $out'[n] = out[n]$ } Save current results
    $\mathbf{out[n]} = \bigcup_{s \in succ[n]} \mathbf{in[s]}$
    $\mathbf{in[n] = use[n] \cup (out[n] - def[n])}$ } Solve data-flow equations

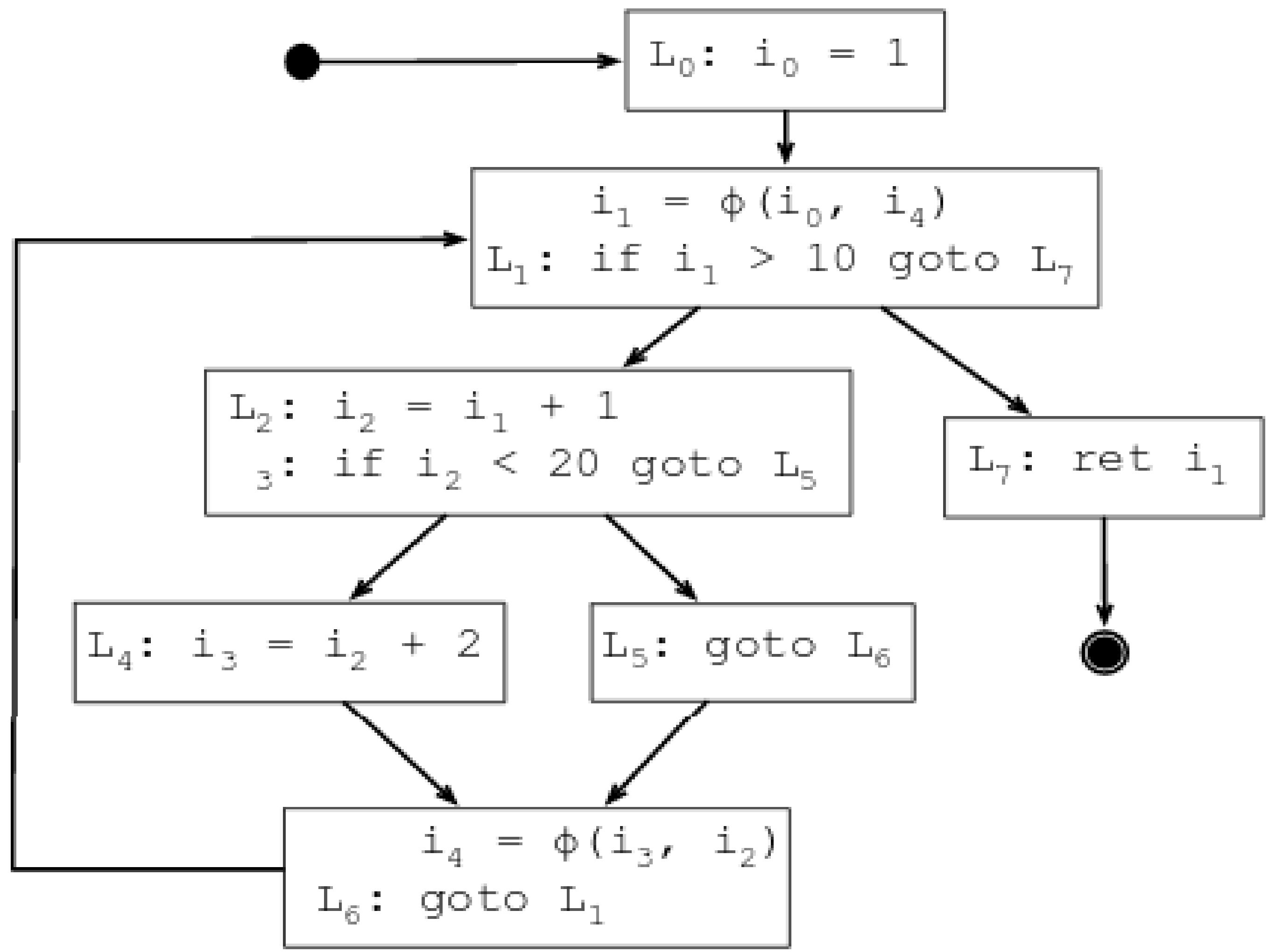
until $in'[n]=in[n]$ and $out'[n]=out[n]$ for all n } Test for convergence

Review: Textbook Liveness Analysis

- Complexity
- Time
  - Worst case: $O(n^4)$
  - Typical case: $O(N)$ to $O(N^2)$
- Space
  - $O(N^2)$

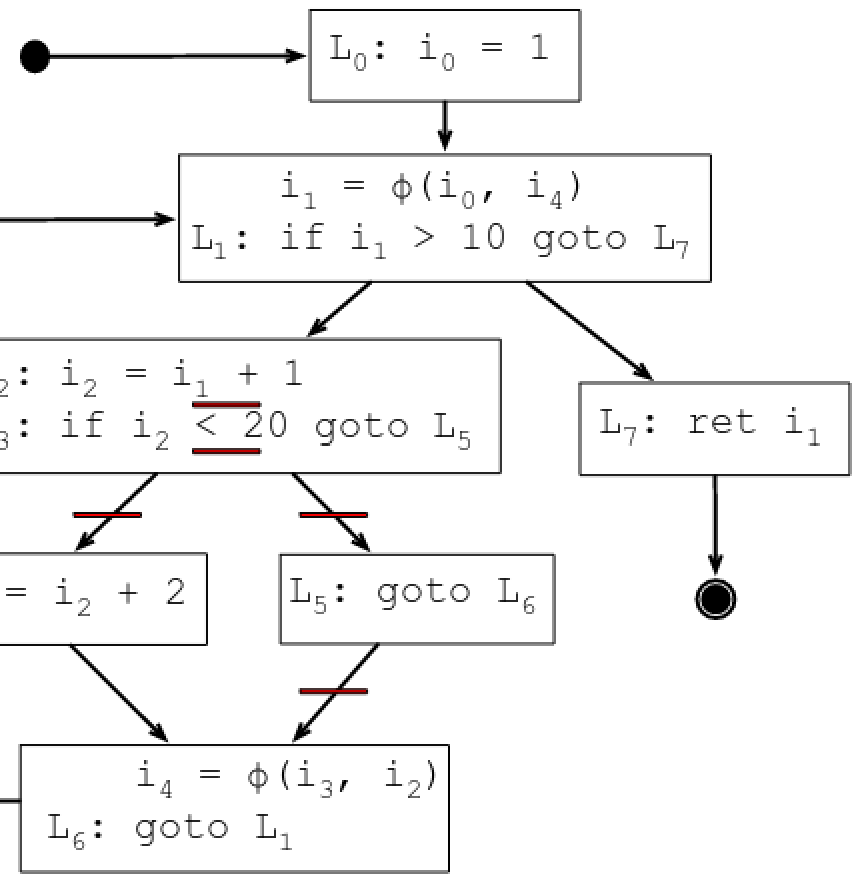
SSA Form Liveness Analysis

Can you point where i2 is alive in this program?



SSA Form Liveness Analysis

Can you point where
i2 is alive in this program?



SSA Form Liveness Analysis

- Without traversing the CFG to reach a fixed point.

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

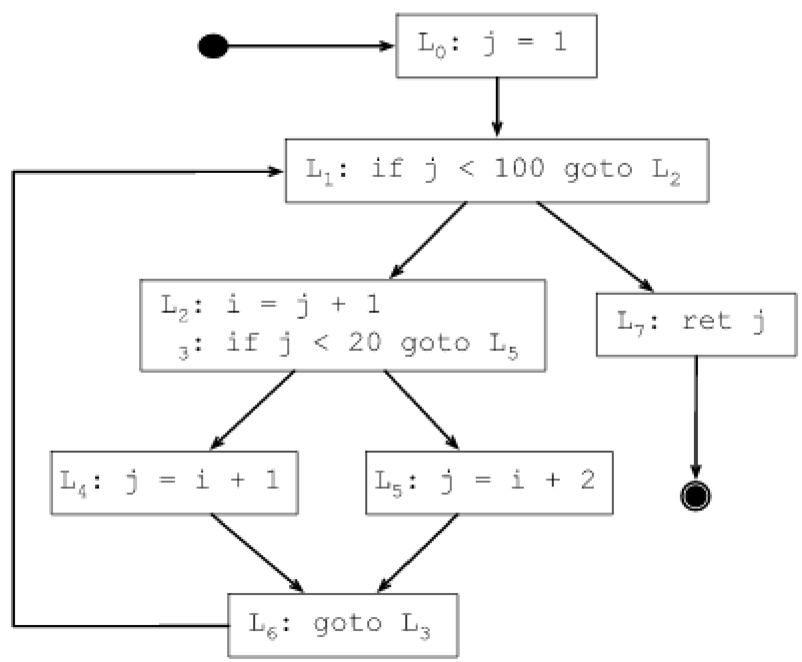
```
For each statement S in the program:
    IN[S] = OUT[S] = {}

For each variable v in the program:
    For each statement S that uses v:
        live(S, v)

live(S, v):
    IN[S] = IN[S] ∪ {v}
    For each P in pred(S):
        OUT[P] = OUT[P] ∪ {v}
        if P does not define v
            live(P, v)
```

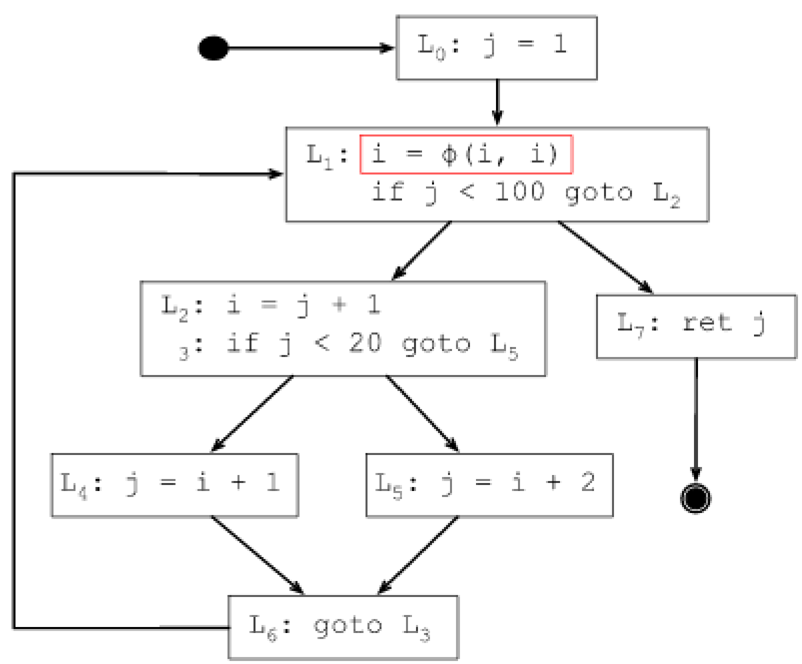
Is Traditional DA Useless?

- Where should we add a phi-function for the defination of *i* at *L2*.



Is Traditional DA Useless?

- 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.



LLVM Pass in Action – A Challenge Job

- Naive Liveness Analysis for LLVM IR
- Function Pass
- LLVM API
  - Iterating basic blocks, instructions and operands.
  - Instruction casting
  - Fix-point algorithm
  - ...

Thank you