# LLVM AND SSA

Slides adapted from those prepared by Steve Zdancewic at Penn



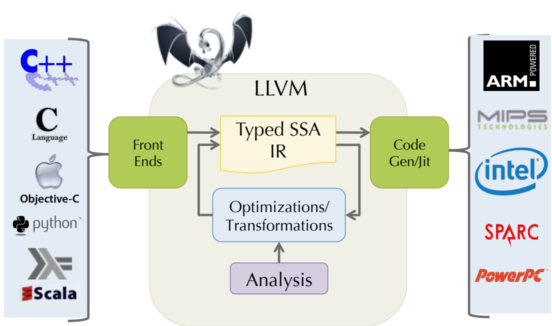
## Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documntation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Mult-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
  - llvm-gcc (drop-in replacement for gcc)
  - Clang: C, objective C, C++ compiler supported by Apple
  - various languages: ADA, Scala, Haskell, …
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects



## LLVM Compiler Infrastructure

[Lattner et al.]




## LL: A Subset of LLVM

```
op     ::= %uid | constant

bop    ::= add | sub | mul | shl | …

cmpop  ::= eq | ne | slt | sle | …

insn ::=
  | %uid = bop op1, op2
  | %uid = alloca
  | %uid = load op1
  | store op1, op2
  | %uid = icmp cmpop op1, op2

terminator ::=
  | ret op
  | br op label %lbl1, label %lbl2
  | br label %lbl
```



## Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the "link"
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target
- Basic blocks can be arranged into a *control-flow graph*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.



## LL Basic Blocks and Control-Flow Graphs

- LLVM enforces (some of) the basic block invariants syntactically.
- Representation in OCaml:

```
type bblock = {
    label : lbl;
    insns : insn list;
    terminator : terminator
}
```

- A *control flow graph* is represented as a list of basic blocks with these invariants:
  - No two blocks have the same label
  - All terminators mention only labels that are defined among the set of basic blocks
  - There is a distinguished entry point label (which labels a block)

```
type prog = {ll_cfg : bblock list; ll_entry : lbl}
```

# LL Storage Model: Locals

- Two kinds of storage:
  - Local variables: `%uid`
  - Abstract locations: references to storage created by the `alloca` instruction
- Local variables:
  - Defined by the instructions of the form `%uid = …`
  - Must satisfy the *single static assignment* invariant
    - Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph.
  - The value of a `%uid` remains unchanged throughout its lifetime
  - Analogous to "`let %uid = e in …`" in OCaml
- Intended to be an abstract version of machine registers.
- We'll see later how to extend SSA to allow richer use of local variables.



# LL Storage Model: `alloca`

- The alloca instruction allocates a fresh (32-bit) slot and returns a reference to it.
  - The returned reference is stored in local:
    `%ptr = alloca`
- The contents of the slot are accessed via the `load` and `store` instructions:

```
%acc = alloca        ; allocate a storage slot
store 341, %acc      ; store the integer value 341
%x = load %acc       ; load the value 341 into %x
```

- Gives an abstract version of stack slots



# Example LLVM Code

- LLVM offers a textual representation of its IR
  - files ending in `.ll`

example.c

```
unsigned factorial(unsigned n) {
  unsigned acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n -1;
  }
  return acc;
}
```

example.ll

```
define @factorial(%n) {
entry:
  %1 = alloca
  %acc = alloca
  store %n,  %1
  store 1,  %acc
  br label %start

start:
  %3 = load %1
  %4 = icmp ugt %3, 0
  br %4, label %then, label %else

then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

else:
  %12 = load %acc, align 4
  ret %12
}
```



# Real LLVM

- Decorates values with type information
  `i32`
  `i32*`
  `i1`
- Has alignment annotations
- Keeps track of entry edges for each block:
  `preds = %start`

```
define i32 @factorial(i32 %n) nounwind uwtable ssp {
entry:
  %1 = alloca i32, align 4
  %acc = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %acc, align 4
  br label %start

start:                                 ; preds = %entry, %else
  %3 = load i32* %1, align 4
  %4 = icmp ugt i32 %3, 0
  br i1 %4, label %then, label %else

then:                                  ; preds = %start
  %6 = load i32* %acc, align 4
  %7 = load i32* %1, align 4
  %8 = mul i32 %6, %7
  store i32 %8, i32* %acc, align 4
  %9 = load i32* %1, align 4
  %10 = sub i32 %9, 1
  store i32 %10, i32* %1, align 4
  br label %start

else:                                  ; preds = %start
  %12 = load i32* %acc, align 4
  ret i32 %12
}
```



# Structured Data in LLVM

- LLVM's IR is uses types to describe the structure of data.

| | |
|---|---|
| `t ::=` | |
| `i32` | *32-bit integers* |
| `[<#elts> x t]` | *arrays* |
| $r\ (t_1, t_2, \ldots, t_n)$ | *function types* |
| $\{t_1, t_2, \ldots, t_n\}$ | *structures* |
| `t*` | *pointers* |
| `%Tident` | *named (identified) type* |
| `r ::=` | *Return Types* |
| `t` | *first-class type* |
| `void` | *no return value* |

- <#elts> is an integer constant >= 0
- Structure types can be named at the top level:

  $\%T1 = \texttt{type}\ \{t_1, t_2, \ldots, t_n\}$

  - Such structure types can be recursive



# Example LL Types

- An array of 341 integers: `[ 341 x i32 ]`
- A two-dimensional array of integers: `[ 3 x [ 4 x i32 ] ]`
- Structure for representing arrays with their length:
  `{ i32 , [0 x i32] }`
  - There is no array-bounds check; the static type information is only used for calculating pointer offsets.
- C-style linked lists (declared at the top level):
  `%Node = type { i32, %Node*}`
- Structs from a C program:

```
%Rect = { %Point, %Point, %Point, %Point }
%Point = { i32, i32 }
```

## GetElementPtr

- LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a "path" through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
  - It is a "type indexed" operation, since the sizes computations involved depend on the type

```
insn ::= …
      |  %uid = getelementptr t*, %val, t1 idx1, t2 idx2 ,…
```

- Example: access the x component of the first point of a rectangle:

```
%tmp1 = getelementptr %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 0
```



## Example*

```
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `sizeof(struct ST)`.
3. Compute the index of the `Z` field by adding `sizeof(struct RT) + sizeof(int)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `sizeof(int)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

Final answer: ADDR + `sizeof(struct ST) + sizeof(struct RT) + sizeof(int)`
`+ sizeof(int) + 5*20*sizeof(int) + 13*sizeof(int)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

## LOOPS AND DOMINATORS



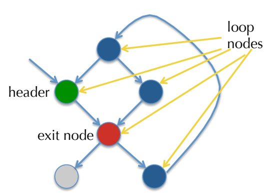
## Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important
- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.
- Loops may be hard to recognize at the quadruple IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…
- Problem: *How do we identify loops in the control-flow graph?*



## Definition of a Loop

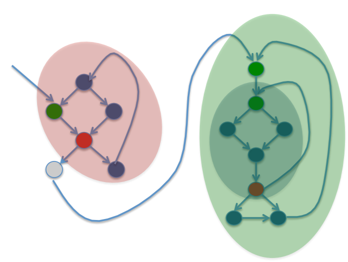
- A *loop* is a set of nodes in the control flow graph.
  - One distinguished entry point called the *header*
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*
- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes



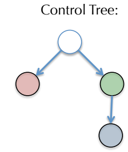



## Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:



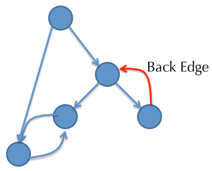
Control Tree:



The control tree depicts the nesting structure of the program.

## Control-flow Analysis

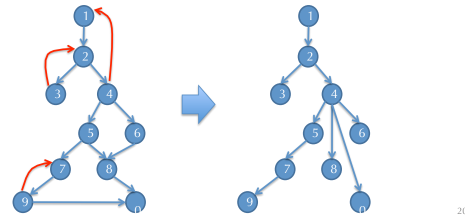
- Goal: Identify the loops and nesting structure of the CFG.
- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.
- An edge in the graph is a *back edge* if the target node dominates the source node.
- A loop contains at least one back edge.




19

## Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation



20

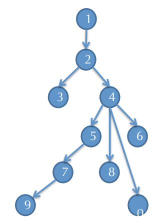
## Dominator Dataflow Analysis

- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- "Every node dominates itself."
  - $out[n] := in[n] \cup \{n\}$
- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $T = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ is $\cap$
- Easy to show monotonicity and that $F_n$ distributes over meet.
  - So algorithm terminates

21

## Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:
  - Dom[8] ∩ Dom[7] = Dom[4]

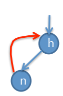


- See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

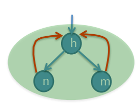
22

## Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
  - Edge n → h where h dominates n
- Each back edge has a *natural loop*:
  - h is the header
  - All nodes reachable from h that also reach n without going through h



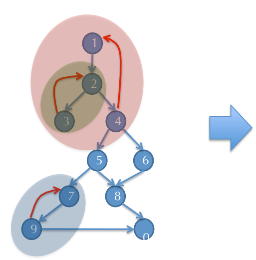
- For each back edge n → h, find the natural loop:
  - {n' | n is reachable from n' in G – {h}} ∪ {h}
- Two loops may share the same header: merge them



- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

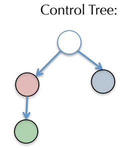
23

## Example Natural Loops



Natural Loops

Control Tree:



The control tree depicts the nesting structure of the program.

24

## Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
  - Deeply nested loops pay off the most for optimization.
- Need to know loop headers / back edges for doing
  - loop invariant code motion
  - loop unrolling
- Dominance information also plays a role in converting to SSA form
  - Used internally by LLVM to do register allocation.



## STATIC SINGLE ASSIGNMENT (SSA)



## Single Static Assignment (SSA)

- LLVM IR names (via %uids) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each %uid is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation: map %uids to stack slots
- Better implementation: map %uids to registers (as much as possible)
- Question: How do we convert a source program to make maximal use of %uids, rather than alloca-created storage?
  - two problems: control flow & location in memory



## Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i32
%y = alloca i32
store i32* %x, 3
store i32* %y, 0
%x1 = load %i32* %x
%tmp1 = add i32 %x1, 1
store i32* %x, %tmp1
%x2 = load %i32* %x
%tmp2 = add i32 %x2, 2
store i32* %y, %tmp2
```

- Directly map source variables into %uids?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i32 3, 0
%y1 = add i32 0, 0
%x2 = add i32 %x1, 1
%y2 = add i32 %x2, 2
```

- Does this always work?



## What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %z2 = %add i32 ???, 3
```

- What do we put for ???



## Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.
    %uid = phi <ty> $v_1$, <$label_1$>, … , $v_n$, <$label_n$>

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %x4 = phi i32 %x2, %then, %x3, %else
  %z2 = %add i32 %x4, 3
```

## Phi Nodes and Loops

- Importantly, the `%uids` on the right-hand side of a phi node can be defined "later" in the control-flow graph.
  - Means that `%uids` can hold values "around a loop"
  - Scope of `%uids` is defined by dominance (discussed soon)

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i32 %x1, %entry, %x3, %body
  %x3 = add i32 %x2, %y1
  %p = icmp slt i32, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```



## Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i32        // %x cannot be promoted
  %y = call malloc(i32 4)
  store i32** %y, %x     // store the pointer into the heap
```

```
entry:
  %x = alloca i32        // %x cannot be promoted
  %y = call foo(i32* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated



## Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"
- Replace loads/stores to alloc'ed variables with freshly-generated %uids
- Eliminate the now unneeded load/store/alloca instructions.



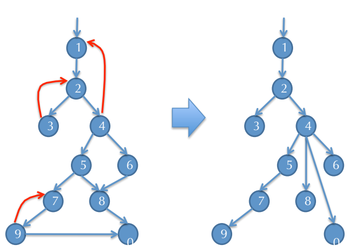
## Where to Place ϕ functions?

- Need to calculate the "Dominance Frontier"
- Node A *strictly dominates* node B if A dominates B and A ≠ B.
- The *dominance frontier* of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
- Write DF[n] for the dominance frontier of node n.



## Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {}, DF[2] = {1,2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}




Control-flow Graph    Dominator Tree



## Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)

```
for all nodes b
   if #(pred[b]) ≥ 2
    for each p ∈pred[b]
        runner := p
        while (runner ≠ doms[b])
            DF[runner] := DF[runner] ∪ {b}
            runner := doms[runner]
```

## Insert $\phi$ at Join Points

- Lift the DF[n] to a set of nodes N in the obvious way:
  $DF[N] = \bigcup_{n \in N} DF[n]$
- Suppose that at variable x is defined at a set of nodes N.
- $DF_0[N] = DF[N]$
  $DF_{i+1}[N] = DF[DF_i[N] \cup N]$
- Let J[N] be the *least fixed point* of the sequence:
  $DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$
  - That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$
- J[N] is called the "join points" for the set N
- We insert $\phi$ functions for the variable x at each such join point.
  - $x = \phi(x, x, \ldots, x)$; (one "x" argument for each predecessor of the node)
  - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes.
- Intuition:
  - If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too…

37

## Example Join-point Calculation

- Suppose the variable x is modified at nodes 2 and 6
  - Where would we need to add phi nodes?
- $DF_0[\{2,6\}] = DF[\{2,6\}] = DF[2] \cup DF[6] = \{1,2,8\}$
- $DF_1[\{2,6\}]$
  $= DF[DF_0\{2,6\} \cup \{2,6\}]$
  $= DF[\{1,2,8,6\}]$
  $= DF[1] \cup DF[2] \cup DF[8] \cup DF[6]$
  $= \{\} \cup \{1,2\} \cup \{0\} \cup \{8\} = \{1,2,8,0\}$
- $DF_2[\{2,6\}]$
  $= \ldots$
  $= \{1,2,8,0\}$
- So $J[\{2,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.

38