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 documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Multistage 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

Front Ends
- CodeGen/
Optimizations/
Transformations

Optimized IR

Typed SSA

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
             | 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: \( \texttt{llvm} \)
  - Abstract locations: references to storage created by the \texttt{alloca} instruction

• Local variables:
  - Defined by the instructions of the form \( \texttt{llvm} = ... \)
  - Must satisfy the single static assignment invariant
    • Each \( \texttt{llvm} \) appears on the left-hand side of an assignment only once in the entire control flow graph.
    • The value of a \( \texttt{llvm} \) remains unchanged throughout its lifetime
  - Analogous to "let \( \texttt{llvm} = \ldots \)" 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.

Example LLVM Code

• LLVM offers a textual representation of its IR
  – files ending in .ll

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

Real LLVM

• Decorates values with type information
  \texttt{i32}
  \texttt{i32*}

```
define i32 @factorial(i32) {
  ret i32...
  i32...
  i32...
  i32...
  i32...
  i32...
}
```

Structured Data in LLVM

• LLVM IR uses types to describe the structure of data.

```
struct { i32, i32 }
struct { i32, i32, i32, i32, i32, i32 }
```

Example LL Types

• An array of 132 integers:

```
[ 341 x 132 ]
```

• 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):

```
typedef = type { i32, uNode* }
```

• Structures from a C program:

```
struct = { uPoint, uPoint, uPoint, uPoint, uPoint }
```

Example LL Type

```
functor = { ... }
```

4/1/15
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.

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

```c
insn ::= ...

|     %uid = getelementptr %*, %val, t1 idx1, t2 idx2 ...

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

Example:

```c
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];
}
```

```c
%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
}
```

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-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.

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

Dominator Dataflow Analysis

- We can define $\text{Dom}(n)$ as a forward dataflow analysis:
  - Using the framework we saw earlier: $\text{Dom}(n) = \text{out}(n)$ where:
    - $\text{in}(n) := \bigcap_{n' \in \text{pred}(n)} \text{out}(n')$
    - $\text{out}(n) := \text{in}(n) \cup \{n\}$
- Formally: $L = \text{set of nodes ordered by } \subseteq$
  - $T = (\text{all nodes})$
  - $F(n) = \text{in}(n) \cup \{n\}$
  - $n$ is $\in$
- Easy to show monotonicity and that $F_n$ distributes over meet.
  - So algorithm terminates

Improving the Algorithm

- $\text{Dom}(b)$ contains just those nodes along the path in the dominator tree from the root to $b$:
  - e.g. $\text{Dom}(8) = \{1, 2, 4, 8\}$, $\text{Dom}(7) = \{1, 2, 4, 3, 7\}$
  - There is a lot of sharing among the nodes
- More efficient way to represent $\text{Dom}$ sets is to store the dominator tree:
  - $\text{doms}(b) = \text{immediate dominator of } b$
  - $\text{doms}(8) = 4$, $\text{doms}(7) = 5$
- To compute $\text{Dom}(b)$ walk through $\text{doms}(b)$
- Need to efficiently compute intersections of $\text{Dom}(a)$ and $\text{Dom}(b)$
  - Traverse up tree, looking for least common ancestor:
    - $\text{Dom}(8) \cap \text{Dom}(7) = \text{Dom}(4)$
- See: “A Simple, Fast Dominance Algorithm” Cooper, Harvey, and Kennedy

Completing Control-flow Analysis

- Dominator analysis identifies back edges:
  - Edge $n \rightarrow h$ 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 $n \rightarrow h$, find the natural loop:
  - $\{n' \in G - \{h\} \cup \{h\}$
- 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

Example Natural Loops

- Control Tree:
  - The control tree depicts the nesting structure of the program.
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.

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
  - Note that dataflow analyses had these kill[n] sets because of updates to variables...
- Naive 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:
  - Directly map source variables into %uids?
  - Does this always work?

What about If-then-else?

- How do we translate this into SSA?

entry:
  "y1" = ...
  "x1" = ...
  "z1" = ...
  "p" = icmp ...
  br i1 "p", label "then", label "else"

then:
  "x2" = add i32 "y1", 1
  br label "merge"

else:
  "x3" = mul i32 "y1", 2
  merge:
  "x4" = phi i32 "x2", "then", "x3", "else"
  "z2" = add i32 "x4", 3

Phi Functions

- Solution: φ functions
  - Fictitious operator, used only for analysis
  - Implemented by inters at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.

entry:
  "y1" = ...
  "x1" = ...
  "z1" = ...
  "p" = icmp ...
  "x2" = phi i32 "y1", "then", "x1", "else"
  "x3" = add i32 "x2", 1
  "z2" = add i32 "x3", 2

What about If-then-else?

- How do we translate this into SSA?

entry:
  "y1" = ...
  "x1" = ...
  "z1" = ...
  "p" = icmp ...
  br i1 "p", label "then", label "else"

then:
  "x2" = add i32 "y1", 1
  br label "merge"

else:
  "x3" = mul 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)

```ml
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

```ml
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 malloc(i32 4)
  store i32** %y, %x // store the pointer into the heap
```

**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

**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] U {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 $\mathcal{J}[N]$ be the least fixed point of the sequence:
  $$DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq \ldots$$
  That is, $\mathcal{J}[N] = DF[N]$ for some $k$ such that $DF_k[N] = DF_{k+1}[N]$.
- $\mathcal{J}[N]$ is called the "join points" for the set $N$.
- We insert $\phi$ functions for the variable $x$ at each such join point.
  - In practice, $\mathcal{J}[N]$ is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF[N]$ 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...

**Example Join-point Calculation**

- Suppose the variable $x$ is modified at nodes 2 and 6
  - Where would we need to add phi nodes?

<table>
<thead>
<tr>
<th>Step</th>
<th>$DF_0[{2,6}]$</th>
<th>$DF_1[{2,6}]$</th>
<th>$DF_2[{2,6}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${1,2,8}$</td>
<td>${1,2,8,6}$</td>
<td>${1,2,8,0}$</td>
</tr>
<tr>
<td>1</td>
<td>${1,2,8,6}$</td>
<td>${1,2,8,0}$</td>
<td>${1,2,8,0}$</td>
</tr>
</tbody>
</table>

- So $\mathcal{J}[\{2,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.