Lecture 7

CIS 341: COMPILERS
Announcements

• HW2: X86lite
  – Available on the course web pages.
  – Due: Thursday, February 2\textsuperscript{nd} at 11:59:59pm
  – Pair-programming:
    • Register the group on the submission page
    • Submission by any group member counts for the group
Intermediate Representations

- **IR1: Expressions**
  - simple arithmetic expressions, immutable global variables

- **IR2: Commands**
  - global *mutable* variables
  - commands for update and sequencing

- **IR3: Local control flow**
  - conditional commands & while loops
  - basic blocks

- **IR4: Procedures (top-level functions)**
  - local state
  - call stack
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.
See llvm.org
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 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
  - Here at Penn: SoftBound, Vellvm
LLVM Compiler Infrastructure

[Lattner et al.]
Example LLVM Code

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

factorial64.c

```c
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
    int64_t acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}
```

factorial-pretty.ll

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

start:
  %3 = load %1
  %4 = icmp sgt %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
  ret %12
}```
Real LLVM

- Decorates values with type information
  - \texttt{i64}
  - \texttt{i64*}
  - \texttt{i1}
- Permits numeric identifiers
- Has alignment annotations
- Keeps track of entry edges for each block:
  - \texttt{preds = \%5, \%0}

```llvm
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2

; <label>:2
  %3 = load i64* %1, align 8
  %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11

; <label>:5
  %6 = load i64* %acc, align 8
  %7 = load i64* %1, align 8
  %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc, align 8
  %9 = load i64* %1, align 8
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1, align 8
  br label %2

; <label>:11
  %12 = load i64* %acc, align 8
  ret i64 %12
} factorial.ll
```
define @factorial(%n) {
    entry:
    %1 = alloca
    %acc = alloca
    store %n, %1
    store 1, %acc
    br label %start

    start:
    %3 = load %1
    %4 = icmp sgt %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
    ret %12
}

}
LL Basic Blocks and Control-Flow Graphs

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

```camel
type block = {
  insns : (uid * insn) list;
  terminator : terminator
}
```

- A control flow graph is represented as a list of labeled 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, unlabeled, entry block:

```camel
type cfg = block * (lbl * block) list
```
• Several kinds of storage:
  – Local variables (or temporaries): \%uid
  – Global declarations (e.g. for string constants): @gid
  – Abstract locations: references to (stack-allocated) storage created by the 'alloca' instruction
  – Heap-allocated structures created by external calls (e.g. to malloc)

• 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
  – *phi nodes*
LL Storage Model: `alloca`

- The `alloca` instruction allocates stack space and returns a reference to it.
  - The returned reference is stored in local:
    \[
    \%ptr = \text{alloca typ}
    \]
  - The amount of space allocated is determined by the type

- The contents of the slot are accessed via the `load` and `store` instructions:

  \[
  \%acc = \text{alloca i64} \quad ; \text{allocate a storage slot}
  
  \text{store 341, } \%acc \quad ; \text{store the integer value 341}
  
  \%x = \text{load } \%acc \quad ; \text{load the value 341 into } \%x
  \]

- Gives an abstract version of stack slots