Lecture 7

CIS 341: COMPILERS

Announcements

- HW2: X86lite
 - Available on the course web pages.
 - Due: TOMORROW at 11:59:59pm
- HW3: LLVM to X86lite compiler
 - Available by Thursday
 - Due Weds. Feb. 23rd

see: ir-by-hand.ml, ir<X>.ml

INTERMEDIATE REPRESENTATIONS

Eliminating Nested Expressions

- Fundamental problem:
 - Compiling complex & nested expression forms to simple operations.



Idea: *name* intermediate values, make order of evaluation explicit.
– No nested operations.

Translation to SLL

• Given this:

```
Add(Add(Const 1, Var X4),
Add(Const 3, Mul(Var X1,
Const 5)))
```

• Translate to this desired SLL form:

let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
tmp3

- Translation makes the order of evaluation explicit.
- Names intermediate values
- Note: introduced temporaries are never modified

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.

Control-flow Graphs



Intermediate Representations

- IR1: Expressions
 - *immutable* global variables
 - simple arithmetic *expressions*
- IR2: Commands
 - *mutable* global variables
 - commands for update and sequencing
- IR3: Local control flow
 - conditional commands & while loops
 - basic blocks
- IR4: Procedures (top-level functions)
 - local variables
 - call stack
- IR5: "almost" LLVM IR
 - missing *phi-nodes* (explained when we get there)

See Ilvm.org



Zdancewic CIS 341: Compilers

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:
 - Ilvm-gcc (drop-in replacement for gcc)
 - Clang: C, objective C, C++ compiler supported by Apple
 - various languages: Swift, 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.]



IR3/4/5 vs. LLVM

- "let in" and OCaml-style identifiers:
- let tmp1 = add 3L 4L in
- OCaml-style "let-rec" and functions for blocks:

```
let rec entry () =
   let tmp1 = ...
and foo () =
   let tmp2 = ...
```

OCaml-style global variables:
 let varX = ref 0L

• Omits let/in and prefixes local identifiers with %:

%tmp1 = add i64 3, i64 4

• Uses lighter-weight colon notation:

```
entry:
%tmp1 = ...
foo:
%tmp2 = ...
```

 Prefixes globals with @ define @X = i64 0

Example LLVM Code



}

```
Zdancewic CIS 341: Compilers
```

Real LLVM

Decorates values with type information ullet

factorial.

- i64 i64* i1
- Permits numeric ulletidentifiers
- Has alignment • annotations
- Keeps track of ٠ entry edges for each block: preds = %5, %0

```
; 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
                                      ; preds = %5, %0
; <label>:2
  %3 = load i64* %1, align 8
 %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11
; <label>:5
                                      ; preds = \%2
  %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
                                      ; preds = \%2
; <label>:11
 %12 = load i64* %acc, align 8
  ret i64 %12
}
```

Example Control-flow Graph

define @factorial(%n) {



}

LL Basic Blocks and Control-Flow Graphs

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

```
type block = {
    insns : (uid * insn) list;
    term : (uid * 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:

```
type cfg = block * (lbl * block) list
```

LL Storage Model: Locals

- 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 *static single 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

- alloca instruction allocates stack space and returns a reference to it. •
 - The returned reference is stored in local:

%ptr = alloca type

- The amount of space allocated is determined by the type
- The contents of the slot are accessed via the load and store ٠ instructions:

%acc = alloca i64 store i64 341, i64* %acc ; store the integer value 341 x = 10ad i64, i64* acc : load the value 341 into <math>x

- ; allocate a storage slot

- Gives an abstract version of stack slots