Lecture 5

CIS 341: COMPILERS
Announcements

• HW2: X86lite
  – Available on the course web pages.
  – Due: Monday, February 2\textsuperscript{nd} at 11:59:59pm
  – Pair-programming:
    • There’s a pair-search survey on Piazza
    • Register the group on the submission page
    • Submission by any group member counts for the group
INTERMEDIATE REPRESENTATIONS
Directly Translating AST to Assembly

• For simple languages, no need for intermediate representation.
  – e.g. the arithmetic expression language from

• Main Idea: Maintain invariants
  – e.g. Code emitted for a given expression computes the answer into rax

• Key Challenges:
  – storing intermediate values needed to compute complex expressions
  – some instructions use specific registers (e.g. shift)
One Simple Strategy

• Compilation is the process of “emitting” instructions into an instruction stream.
• To compile an expression, we recursively compile sub expressions and then process the results.
• Invariants:
  – Compilation of an expression yields its result in rax
  – Argument (Xi) is stored in a dedicated operand
  – Intermediate values are pushed onto the stack
  – Stack slot is popped after use (so the space is reclaimed)
• Resulting code is wrapped to comply with cdecl calling conventions:
  • See the compile.ml from last lecture for example code.
Why do something else?

• This is a simple *syntax-directed* translation
  – Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  – It works fine for simple languages.

But…

• The resulting code quality is poor.
• Richer source language features are hard to encode
  – Structured data types, objects, first-class functions, etc.
• It’s hard to optimize the resulting assembly code.
  – The representation is too concrete – e.g. it has committed to using certain registers and the stack
  – Only a fixed number of registers
  – Some instructions have restrictions on where the operands are located
• Control-flow is not structured:
  – Arbitrary jumps from one code block to another
  – Implicit fall-through makes sequences of code non-modular (i.e. you can’t rearrange sequences of code easily)
• Retargeting the compiler to a new architecture is hard.
  – Target assembly code is hard-wired into the translation
Intermediate Representations (IR’s)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization.

Diagram:
- AST → IR
- IR → x86
- IR → Java Bytecode
- IR → Arm
- Optimization

CIS 341: Compilers 7
Multiple IR’s

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)
What makes a good IR?

• Easy translation target (from the level above)
• Easy to translate (to the level below)
• Narrow interface
  – Fewer constructs means simpler phases/optimizations

• Example: Source language might have “while”, “for”, and “foreach” loops (and maybe more variants)
  – IR might have only “while” loops and sequencing
  – Translation eliminates “for” and “foreach”

\[
\text{⟦for(pre; cond; post) \{body\}⟧} = \\
\text{⟦pre; while(cond) \{body;post\}⟧}
\]

  – Here the notation \text{⟦cmd⟧} denotes the “translation” or “compilation” of the command cmd.
IR’s at the extreme

• High-level IR’s
  – Abstract syntax + new node types not generated by the parser
    • e.g. Type checking information or disambiguated syntax nodes
  – Typically preserves the high-level language constructs
    • Structured control flow, variable names, methods, functions, etc.
    • May do some simplification (e.g. convert for to while)
  – Allows high-level optimizations based on program structure
    • e.g. inlining “small” functions, reuse of constants, etc.
  – Useful for semantic analyses like type checking

• Low-level IR’s
  – Machine dependent assembly code + extra pseudo-instructions
    • e.g. a pseudo instruction for interfacing with garbage collector or memory allocator
      (parts of the language runtime system)
    • e.g. (on x86) a imulq instruction that doesn’t restrict register usage
  – Source structure of the program is lost:
    • Translation to assembly code is straightforward
  – Allows low-level optimizations based on target architecture
    • e.g. register allocation, instruction selection, memory layout, etc.

• What’s in between?
Mid-level IR’s: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values might be named to facilitate optimizations that attempt to minimize stack/register usage

- Many examples:
  - Triples: \( \text{OP} \ a \ b \)
    - Useful for instruction selection on X86 via “tiling”
  - Quadruples: \( a = b \ \text{OP} \ c \) (“three address form”)
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM: industrial-strength IR, based on SSA
  - Stack-based:
    - Easy to generate
    - e.g. Java Bytecode, UCODE
Growing an IR

- Develop an IR in detail… starting from the very basic.

- Start: a (very) simple intermediate representation for the arithmetic language
  - Very high level
  - No control flow

- Goal: A simple subset of the LLVM IR
  - LLVM = “Low-level Virtual Machine”
  - Used in HW3+

- Add features needed to compile rich source languages
SIMPLE LET-BASED IR
Eliminating Nested Expressions

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

Source

```
((1 + X4) + (3 + (X1 * 5)))
```

AST

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

IR

```
?
```

- Idea: *name* intermediate values, make order of evaluation explicit.
  - No nested operations.
Translation to SLL

• Given this:
  \[
  \text{Add(Add(Const 1, Var X4),} \\
  \text{Add(Const 3, Mul(Var X1,} \\
  \text{Const 5))))}
  \]

• 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