COS320: Compiling Techniques

Zak Kincaid

February 14, 2019
• Reminder: HW1 due on Tuesday
• **Office hour change**: Qinshi’s office hours will start at *3pm* on Thursdays
Syntax-directed translation

- Compilation strategy in which *syntax* of the program drives code generation
  - Assembly code generated from AST, or even directly by the parser
  - No substantial code analysis or transformation
- Example: Lecture 2 compiler

```plaintext
let run () =
  let v_X = ref 0 in
  let v_ANS = ref 0 in
  v_X := 6;
  v_ANS := 1;
  while !v_X != 0 do
    v_ANS := (!v_ANS * !v_X);
    v_X := (!v_X + -1)
  done;
  !v_ANS
```
Syntax-directed translation

- Compilation strategy in which *syntax* of the program drives code generation
  - Assembly code generated from AST, or even directly by the parser
  - No substantial code analysis or transformation

- Example: Lecture 2 compiler

- Easy to implement, but:
  - produces inefficient code
  - can be difficult to implement some language features (e.g., first-class functions)
Intermediate Representations
Separation of concerns

- An IR breaks code generation up into two phases. Simpler & easier to implement
- Simplifies optimization
  - E.g., in optimization pass, we don’t have to think about how code motion interacts w/ register use
- Safety: IR can enforce maintenance of invariants (e.g. types)
What makes a good IR?

1. Convenient to translate source language to IR
2. Convenient to generate assembly from IR
3. Convenient to manipulate IR during optimization
Varieties of IR

• In practice, compilers often use several IRs
  • GCC: Source → GENERIC → GIMPLE → RTL → Target

• High-level
  • Preserves high-level structures, but may simplify (e.g., convert for to do/while) or elaborate
  • Some high-level optimizations (e.g., function inlining)

• Mid-level
  • “Abstract assembly language”
    • Still retains some high-level features (e.g., explicit functions, variables, structured data)
  • Machine-independent optimizations

• Low-level
  • Machine-dependent optimizations
A simple let-based IR

1. Makes evaluation order explicit (no nested expressions)
2. Names all intermediate values
3. Distinguish between variables & intermediate values
4. Invariant: there is exactly one assignment to any temporary (warm-up to SSA)