CS42/413

Introduction to Compilers
Radu Rugina

Lecture 16: Intermediate Representation
01 Mar 04

Semantic Analysis

• Type checking
  – Use type checking rules
  – Static semantics = formal framework to specify type-
    checking rules

• There are also control flow errors:
  – Must verify that a break or continue statement is
    always enclosed by a while (or for) statement
  – Java: must verify that a break X statement is
    enclosed by a for loop with label X
  – Can easily check control-flow errors by recursively
    traversing the AST

Where We Are

Source code (character stream) → Token stream

Lexical Analysis

Syntactic Analysis

Abstract syntax tree + symbol tables, types

Semantic Analysis

Intermediate Code Generation

Intermediate Code

• IR = Intermediate Representation
• Allows language-independent, machine-
  independent optimizations and transformations

Easy to translate from AST

Easy to translate to assembly

Narrow interface: small number of node types
  (instructions)
  – Easy to optimize
  – Easy to retarget

What Makes a Good IR?

AST (>40 node types)

IR (13 node types)

Pentium (>200 opcodes)
Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code

Machine Optimizations

- ... some other optimizations take advantage of the features of the target machine
- Machine-specific optimizations

Next Lectures

- Next few lectures: intermediate representation
- Optimizations covered later

Multiple IRs

- Usually two IRs:
  - **High-level IR**
    - Language-independent (but closer to language)
  - **Low-level IR**
    - Machine independent (but closer to machine)

- Another benefit: a significant part of the translation from high-level to low-level is
  - Language-independent
  - Machine-independent
High-Level IR
- High-level intermediate representation is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch, etc.
  - variables, methods
- Allows high-level optimizations based on properties of source language (e.g. inlining)

Low-Level IR
- Low-level representation is essentially an abstract machine
- Has low-level constructs
  - Unstructured jumps, instructions
- Allows optimizations specific to these constructs (e.g. register allocation, branch prediction)

Low-Level IR
- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    \[ a = b \text{ OP } c \]
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)
- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: easier to generate

Three-Address Code
- In this class: three-address code
  \[ a = b \text{ OP } c \]
- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: \( (a, b, c, \text{ OP}) \)
- Example:
  \[ a = (b+c)*(-e); \quad t1 = b + c \]
  \[ t2 = - e \]
  \[ a = t1 * t2 \]

Low IR Instructions
- Assignment instructions:
  - Binary operations: \( a = b \text{ OP } c \)
    - Arithmetic, logic, comparisons
  - Unary operation \( a = \text{ OP } b \)
  - Arithmetic, logic
  - Copy instruction: \( a = b \)
  - Load/store: \( a = \#b, \#a = b \)
  - Other data movement instructions

Low IR Instructions (Ctd)
- Flow of control instructions:
  - label \( L \): label instruction
  - jump \( L \): Unconditional jump
  - cjump \( a L \): conditional jump
- Function call
  - call \( f(a_1, ..., a_n) \)
  - \( a = \text{ call } f(a_1, ..., a_n) \)
    - Is an extension to quads
- ... IR describes the Instruction Set of an abstract machine
**Temporary Variables**

- The operands in the quadruples can be:
  - Program variables
  - Integer constants
  - Temporary variables

- **Temporary variables** = new locations
  - Use temporary variables to store intermediate values

**Arithmetic / Logic Instructions**

- Abstract machine supports a variety of different operations
  - \( a = b \text{ OP } c \)
  - \( a = \text{ OP } b \)

- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LEQ, GE, GEQ
- Unary operations: MINUS, NEG

---

**Data Movement**

- Copy instruction: \( a = b \)
- Load/store instructions:
  - \( a = *b \)
  - \( *a = b \)
  - Models a load/store machine
- Address-of instruction: \( a = &b \)
- Array accesses:
  - \( a = b[i] \)
  - \( a[i] = b \)
- Field accesses:
  - \( a = b.f \)
  - \( a.f = b \)

---

**Branch Instructions**

- Label instruction:
  - \( \text{label L} \)
- Unconditional jump: go to statement after label L
  - \( \text{jump L} \)
- Conditional jump: test condition variable \( a \); if true, jump to label L
  - \( \text{jump a L} \)
- Alternative: two conditional jumps:
  - \( \text{tjump a L} \)
  - \( \text{fjump a L} \)

---

**Call Instruction**

- Supports function call statements
  - \( \text{call } f(a_1, \ldots, a_n) \)
- ... and function call assignments:
  - \( a = \text{call } f(a_1, \ldots, a_n) \)
- No explicit representation of argument passing, stack frame setup, etc.

---

**Example**

\[
\begin{align*}
n &= 0; \\
\text{while } (n < 10) \{ \\
n &= n + 1 \\
\}\end{align*}
\]

\[
\begin{align*}
n &= 0; \\
\text{label test} \\
t2 &= n < 10 \\
t3 &= \text{not } t2 \\
\text{jump } t3 \text{ end label body} \\
n &= n + 1 \\
\text{jump test label end}
\end{align*}
\]
Another Example

```c
m = 0;
if (c == 0) {
    m = m + n * n;
} else {
    m = m + n;
}
```

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST