Intermediate Code

CS 471
October 29, 2007

Motivation

What we have so far...
• An abstract syntax tree
  – With all the program information
  – Known to be correct
• Well-typed
• Nothing missing
• No ambiguities

What we need...
• Something "Executable"
• Closer to actual machine level of abstraction

Intermediate Code

• Abstract machine code – (Intermediate Representation)
• Allows machine-independent code generation, optimization

Why an Intermediate Representation?

• What is the IR used for?
  – Portability
  – Optimization
  – Component Interface
  – Program understanding

• Compiler
  – Front end does lexical analysis, parsing, semantic analysis, translation to IR
  – Back end does optimization of IR, translation to machine instructions

• Try to keep machine dependences out of IR for as long as possible

What Makes a Good IR?

• Easy to translate from AST
• Easy to translate to assembly
• Narrow interface: small number of node types (instructions)
  – Easy to optimize
  – Easy to retarget

AST (>40 node types) ⇒ IR (13 node types) ⇒ x86 (>200 opcodes)
Intermediate Representations

• Any representation between the AST and ASM
  – 3 address code: triples, quads (low-level)
  – Expression trees (high-level)
• Tiger intermediate code:

Intermediate representation

Components
• code representation
• symbol table
• analysis information
• string table

Issues
• Use an existing IR or design a new one?
  – Many available: RTLs, SSA, LVM, etc
• How close should it be to the source/target?

IR selection

Using an existing IR
• cost savings due to reuse
• it must be expressive and appropriate for the compiler operations

Designing an IR
• decide how close to machine code it should be
• decide how expressive it should be
• decide its structure
• consider combining different kinds of IRs

The IR Machine

A machine with
• Infinite number of temporaries (think registers)
• Simple instructions
  – 3-operands
  – Branching
  – Calls with simple calling convention
• Simple code structure
  – Array of instructions
• Labels to define targets of branches

Temporaries

The machine has an infinite number of temporaries
• Call them t0, t1, t2, ....
• Temporaries can hold values of any type
• The type of the temporary is derived from the generation
• Temporaries go out of scope with each function

Optimizing Compilers

• Goal: get program closer to machine code without losing information needed to do useful optimizations
• Need multiple IR stages
High-Level IR (HIR)
- used early in the process
- usually converted to lower form later on
- Preserves high-level language constructs
  - structured flow, variables, methods
- Allows high-level optimizations based on properties of source language (e.g. inlining, reuse of constant variables)
- Example: AST

Medium-Level IR (MIR)
- Try to reflect the range of features in the source language in a language-independent way
- Intermediate between AST and assembly
- Unstructured jumps, registers, memory locations
- Convenient for translation to high-quality machine code
- Other MIRs:
  - quadruples: `a = b OP c` (“a” is explicit, not arc)
  - UCODE: stack machine based (like Java bytecode)
  - advantage of tree IR: easy to generate, easier to do reasonable instruction selection
  - advantage of quadruples: easier optimization

Low-Level IR (LIR)
- Assembly code + extra pseudo instructions
- Machine dependent
- Translation to assembly code is trivial
- Allows optimization of code for low-level considerations: scheduling, memory layout

IR classification: Level
```
for i := op1 to op2 step op3
  instructions
endfor
```

High-level

```
if step < 0 goto L2
L1: if i > op2 goto L3
  instructions
  i := i + step
goto L1
L2: if i < op2 goto L3
  instructions
  i := i + step
goto L2
L3:
```

Medium-level

IR classification: Structure

Graphical
- Trees, graphs
- Not easy to rearrange
- Large structures

Linear
- Looks like pseudocode
- Easy to rearrange

Hybrid
- Combine graphical and linear IRs
- Example:
  - low-level linear IR for basic blocks, and
  - graph to represent flow of control

Graphical IRs

Parse tree
Abstract syntax tree
- High-level
- Useful for source-level information
- Retains syntactic structure
- Common uses
  - source-to-source translation
  - semantic analysis
  - syntax-directed editors
Graphical IRs: Often Use Basic Blocks

**Basic block** = a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

**Partitioning a sequence of statements into BBs**
1. Determine leaders (first statements of BBs)
   - The first statement is a leader
   - The target of a conditional is a leader
   - A statement following a branch is a leader
2. For each leader, its basic block consists of the leader and all the statements up to but not including the next leader.

```c
unsigned int fibonacci(unsigned int n) {
    unsigned int f0, f1, f2;
    f0 = 0;
    f1 = 1;
    if (n <= 1) return n;
    for (int i=2; i<=n; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f2;
}
```

Leaders:
- `read(n)`
- `f0 := 0`
- `f1 := 1`
- `n <= 1` goto `L0`
- `i := 2`
- `L2: if i<=n goto L1`
- `return f2`
- `L1: f2 := f0+f1`
- `f0 := f1`
- `f1 := f2`
- `i := i+1`
- `go to L2`
- `L0: return n`

Graphical IRs

**Directed acyclic graphs (DAGs)**
- Like compressed trees
  - leaves: variables, constants available on entry
  - internal nodes: operators
  - annotated with variable names?
  - distinct left/right children
- Used for basic blocks (DAGs don’t show control flow)
- Can generate efficient code.
  - Note: DAGs encode common expressions
  - But difficult to transform
  - Good for analysis

**Generating DAGs**
- Check whether an operand is already present
  - if not, create a leaf for it
- Check whether there is a parent of the operand that represents the same operation
  - if not create one, then label the node representing the result with the name of the destination variable, and remove that label from all other nodes in the DAG.
Graphical IRs
Directed acyclic graphs (DAGs)

- Example
  \[ m := 2 \times y \times z \]
  \[ n := 3 \times y \times z \]
  \[ p := 2 \times y - z \]

Graphical IRs
Control flow graphs (CFGs)

- Each node corresponds to a
  - basic block, or
  - part of a basic block, or
    - may need to determine facts at specific points within BB
  - a single statement
    - more space and time
  - Each edge represents flow of control

Graphical IRs
Dependence graphs: they represent constraints on the sequencing of operations

- Dependence = a relation between two statements that puts a constraint on their execution order.
  - Control dependence
    - Based on the program's control flow.
  - Data dependence
    - Based on the flow of data between statements.
  - Nodes represent statements
  - Edges represent dependences
    - Labels on the edges represent types of dependences
  - Built for specific optimizations, then discarded

Graphical IRs
Dependence graphs

Example:

\[
\begin{align*}
  s_1 & : a := b + c \\
  s_2 & : \text{if } a > 10 \text{ goto } L_1 \\
  s_3 & : d := b \times e \\
  s_4 & : e := d + 1 \\
  s_5 & : L_1: d := e / 2
\end{align*}
\]

- control dependence: \( s_3 \) and \( s_4 \) are executed only when \( a \leq 10 \)
- true or flow dependence: \( s_2 \) uses a value defined in \( s_1 \)
  This is read-after-write dependence
- antidependence: \( s_4 \) defines a value used in \( s_3 \)
  This is write-after-read dependence
- output dependence: \( s_5 \) defines a value also defined in \( s_3 \)
  This is write-after-write dependence

Input dependence:
\( s_5 \) uses a value also used in \( s_3 \)
This is read-after-read situation. It places no constraints in the execution order, but is used in some optimizations.

IR classification: Structure

Graphical
- Trees, graphs
- Not easy to rearrange
- Large structures

Linear
- Looks like pseudocode
- Easy to rearrange

Hybrid
- Combine graphical and linear IRs
  - low-level linear IR for basic blocks, and
  - graph to represent flow of control

Linear IRs
Sequence of instructions that execute in order of appearance

Control flow is represented by conditional branches and jumps

Common representations
- stack machine code
- three-address code
Linear IRs

Stack machine code
- Assumes presence of operand stack
- Useful for stack architectures, JVM
- Operations typically pop operands and push results.
- Advantages
  - Easy code generation
  - Compact form
- Disadvantages
  - Difficult to rearrange
  - Difficult to reuse expressions

Example: Three-Address Code

Each instruction is of the form
\[ x := y \text{ op } z \]
- \( y \) and \( z \) can be only registers or constants
- Just like assembly

Common form of intermediate code
The AST expression \( x + y * z \) is translated as
\[ t_1 := y * z \]
\[ t_2 := x + t_1 \]
- Each subexpression has a “home”

Three Address Code

- Result, operand, operand, operator
  - \( x := y \text{ op } z \), where op is a binary operator and \( x, y, z \) can be variables, constants or compiler generated temporaries (intermediate results)
- Can write this as a shorthand
  - \( <\text{op}, \text{arg1}, \text{arg2}, \text{result}> \) -- quadruples
- Statements
  - Assignment \( x := y \text{ op } z \)
  - Copy stmts \( x := y \)
  - Goto L

Bigger Example

Consider the AST
\[ s = b * (d + b * t_1) \]

\[
\begin{align*}
\ t_1 &:= -c \\
\ t_2 &:= b * t_1 \\
\ t_3 &:= -c \\
\ t_4 &:= b * t_3 \\
\ t_5 &:= t_2 + t_4 \\
\ a &:= t_5
\end{align*}
\]

Summary
- IRs provide the interface between the front and back ends of the compiler
- Should be machine and language independent
- Should be amenable to optimization

Next Time: Tiger IR