Compiler Design

Intermediate Representation

Hwansoo Han
Intermediate Representations

- Front end - produces an intermediate representation (IR)
- Middle end - transforms the IR into an equivalent IR that runs more efficiently (Potentially multiple IR's can be used)
- Back end - transforms the IR into native code

**IR encodes the compiler’s knowledge of the program**

- AST, DAG
- Stack-machine code (one address code) : Java bytecode
- Three address code : Low-level, RISC-like IR
Intermediate Representations

- Decisions in \textit{IR} design affect the speed and efficiency of the compiler

- Some important \textit{IR} properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction

- The importance of different properties varies between compilers
  - Selecting an appropriate \textit{IR} for a compiler is critical
Types of Intermediate Representations

- **Structural IR**
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large
  - Examples: Trees, DAGs

- **Linear IR**
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange
  - Examples: 3 address code, Stack machine code

- **Hybrid IR**
  - Combination of graphs and linear code
  - Example: control-flow graph
  - Example: Control-flow graph
Level of Abstraction (1)

- The level of detail exposed in an IR
  - Influences the profitability and feasibility of different optimizations.
  - Two different representations of an array reference:

```plaintext
char A[10][10];
A[j][i] = A+(j-1)*10+(i-1)
```

High level AST:
- Good for memory disambiguation

Low level linear code:
- Good for address calculation
Level of Abstraction (2)

- Structural IRs are usually considered high-level
- Linear IRs are usually considered low-level
- Not necessarily true:

Low level AST

load

High level linear code

loadArray A,i,j

\[
\begin{align*}
  & - j \\
  & 10 \quad + \quad i \quad 1 \\
  & \quad + \quad @A \\
  & \quad + \quad \text{low level AST} \\
  & \quad \text{loadArray A,i,j} \\
  & \quad \text{High level linear code}
\end{align*}
\]
Abstract Syntax Tree

- An abstract syntax tree is the procedure’s parse tree with the non-terminal nodes removed.

Parse Tree

- Can use linearized form of the tree.
  - Easier to manipulate than pointers to nodes.

  \[
  x - 2 * y
  \]

  \[
  x - 2 * y
  \]

  AST

  \[
  - \frac{2}{y}
  \]

  \[
  \frac{x}{y}
  \]
A directed acyclic graph (DAG) is an AST with a unique node for each value.

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

- Originally used for stack-based computers, now Java bytecode
  - Example: \(x - 2 * y\) becomes
  - Push x
  - Push 2
  - Push y
  - Multiply
  - Subtract

- Advantages
  - Compact form
  - Introduced names are *implicit*
  - Simple to generate and execute code

- Useful where code is *transmitted over* slow communication links

Implicit names take up no space, where explicit ones do!
Several different representations of three address code

- In general, three address code has statements of the form:
  \[ x \leftarrow y \ op z \]
- With 1 operator (\( op \)) and, at most, 3 names (\( x, y, z \))

Example:
\[ z \leftarrow x - 2 \times y \]
becomes
\[ t \leftarrow 2 \times y \]
\[ z \leftarrow x - t \]

Advantages:
- Resembles many machines
- Introduces a new set of names
- Compact form
Three Address Code: Quadruples

- Naïve representation of three address code
  - Table of k * 4 small integers
  - Simple record structure
  - Easy to reorder
  - Explicit names

```
load  r1, y
loadl r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

| load    | 1 | y |
| loadi   | 2 | 2 |
| mult    | 3 | 2 | 1 |
| load    | 4 | x |
| sub     | 5 | 4 | 3 |

RISC assembly code

Quadruples
Two Address Code

- Allows statements of the form 
  \[ x \leftarrow x \ op \ y \]
  
  Has 1 operator \( \textit{op} \) and, at most, 2 names (\( x \) and \( y \))

- Example:
  \[ z \leftarrow x - 2 \times y \]
  becomes
  
  \[
  \begin{align*}
  t_1 & \leftarrow 2 \\
  t_2 & \leftarrow \text{load } y \\
  t_2 & \leftarrow t_2 \times t_1 \\
  z & \leftarrow \text{load } x \\
  z & \leftarrow z - t_2
  \end{align*}
  \]

- Can be \textit{very} compact

- Problems
  - Machines no longer rely on destructive operations
  - Difficult name space
    - Destructive operations make reuse hard
    - Good model for machines with destructive ops (PDP-11, x86)
Static Single Assignment Form

- **Single assignments generated by compiler**
  - A variable represents a value
  - Only a single assignment to a variable within the code

```plaintext
if (A < B)  then
    A := 7;
else
    A := 0;
fi
B := A;
```

```plaintext
if (A < B)  then
    A1 := 7;
else
    A2 := 0;
fi
A3 := φ(A1,A2)
B := A3;
```
Control-flow Graph

- Models the transfer of control in the procedure
  - Nodes in the graph are basic blocks
    - Can be represented with quads or any other linear representation
  - Edges in the graph represent control flow

- Example

Diagram:

```
if (x = y)

a ← 2
b ← 5

a ← 3
b ← 4

c ← a * b
```

Basic blocks — Maximal length sequences of straight-line code
Using Multiple Representations

Repeatedly lower the level of the intermediate representation
- Each intermediate representation is suited towards certain optimizations

Example: the Open64 compiler
- WHIRL intermediate format
  - Consists of 5 different IRs that are progressively more detailed
Memory Models

Two major models

- Register-to-register model
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores

- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers just before they are used
  - Compiler back-end can remove redundant loads and stores
Summary

- **Intermediate representation**
  - AST, DAG
  - Stack machine code, 3-address code
  - Control flow

- **Consider different level of abstraction**

- **Expose as much detail as possible**