CSE 401 – Compilers

Intermediate Representations
Hal Perkins
Winter 2015
Agenda

• Survey of Intermediate Representations
  – Graphical
    • Concrete/Abstract Syntax Trees (ASTs)
    • Control Flow Graph
    • Dependence Graph
  – Linear Representations
    • Stack Based
    • 3-Address

• Several of these will show up as we explore program analysis and optimization
Compiler Structure (review)

Parser

Semantic Analysis

Middle (optimization)

Code Gen

Source

Scanner

characters

tokens

IR

IR (maybe different)

IR (often different)

Assembly or binary code

Target
Intermediate Representations

• In most compilers, the parser builds an intermediate representation of the program
  – Typically an AST, as in the MiniJava project

• Rest of the compiler transforms the IR to improve (“optimize”) it and eventually translate to final code
  – Typically will transform initial IR to one or more different IRs along the way

• Some high-level examples now; more specifics later as needed
IR Design

• Decisions affect speed and efficiency of the rest of the compiler
  – General rule: compile time is important, but performance of generated code often more important
  – Typical case for production code: compile a few times, run many times
    • Although the reverse is true during development
  – So make choices that improve compile time as long as they don’t compromise the result
IR Design

• Desirable properties
  – Easy to generate
  – Easy to manipulate
  – Expressive
  – Appropriate level of abstraction

• Different tradeoffs depending on compiler goals

• Different tradeoffs in different parts of the same compiler
  – So often different IRs in different parts
IR Design Taxonomy

• Structure
  – Graphical (trees, graphs, etc.)
  – Linear (code for some abstract machine)
  – Hybrids are common (e.g., control-flow graphs whose nodes are basic blocks of linear code)

• Abstraction Level
  – High-level, near to source language
  – Low-level, closer to machine (exposes more details to compiler)
Examples: Array Reference

\[ A[i,j] \]

or

\[ t1 \leftarrow A[i,j] \]

\[
\begin{align*}
\text{load} & \quad 1 \quad \Rightarrow \quad r1 \\
\text{sub} & \quad rj, r1 \quad \Rightarrow \quad r2 \\
\text{load} & \quad 10 \quad \Rightarrow \quad r3 \\
\text{mult} & \quad r2, r3 \quad \Rightarrow \quad r4 \\
\text{sub} & \quad ri, r1 \quad \Rightarrow \quad r5 \\
\text{add} & \quad r4, r5 \quad \Rightarrow \quad r6 \\
\text{load} & \quad @A \quad \Rightarrow \quad r7 \\
\text{add} & \quad r7, r6 \quad \Rightarrow \quad r8 \\
\text{load} & \quad r8 \quad \Rightarrow \quad r9
\end{align*}
\]
Levels of Abstraction

• Key design decision: how much detail to expose
  – Affects possibility and profitability of various optimizations
    • Depends on compiler phase: some semantic analysis & optimizations are easier with high-level IRs close to the source code. Low-level usually preferred for other optimizations, register allocation, code generation, etc.
  – Structural (graphical) IRs are typically fairly high-level
    – but are also used for low-level
  – Linear IRs are typically low-level
  – But these generalizations don’t always hold
Graphical IRs

• IRs represented as a graph (or tree)
• Nodes and edges typically reflect some structure of the program
  – E.g., source code, control flow, data dependence
• May be large (especially syntax trees)
• High-level examples: syntax trees, DAGs
  – Generally used in early phases of compilers
• Other examples: control flow graphs and data dependency graphs
  – Often used in optimization and code generation
Concrete Syntax Trees

• The full grammar is needed to guide the parser, but contains many extraneous details
  – Chain productions
  – Rules that control precedence and associativity
• Typically the full syntax tree does not need to be used explicitly
Abstract Syntax Trees

• Want only essential structural information
  – Omit extra junk

• Can be represented explicitly as a tree or in a linear form
  – Example: LISP/Scheme S-expressions are essentially ASTs

• Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimizations
DAGs (Directed Acyclic Graphs)

• Variation on ASTs with shared substructures
• Pro: saves space, exposes redundant sub-expressions
• Con: less flexibility if part needs to be changed
Basic Blocks

• Fundamental concept in analysis/optimization

• A *basic block* is:
  – A sequence of code
  – One entry, one exit
  – Always executes as a single unit (“straightline code”) – so it can be treated as an indivisible block

• Usually represented as some sort of a list although Trees/DAGs are possible
Control Flow Graph (CFG)

• Nodes: *basic blocks*

• Edges: represent possible flow of control from one block to another, i.e., possible execution orderings
  – Edge from A to B if B could execute immediately after A in some possible execution

• Required for much of the analysis done during optimization phases
print("hello");
a=7;
if (x == y) {
    print("same");
    b = 9;
} else {
    b = 10;
}
while (a < b) {
    a++;
    print("bump");
}
print("finis");
Dependency Graphs

• Often used in conjunction with another IR
• Data dependency: edges between nodes that reference common data

Examples
  – Block A defines x then B reads it (RAW – read after write)
  – Block A reads x then B writes it (WAR – “anti-dependence”)
  – Blocks A and B both write x (WAW) – order of blocks must reflect original program semantics

• These restrict reorderings the compiler can do
Linear IRs

• Pseudo-code for some abstract machine
• Level of abstraction varies
• Simple, compact data structures
  – Commonly used: arrays, linked structures
• Examples: 3-address code, stack machine code

```
t1 ← 2
t2 ← b
t3 ← t1 * t2
t4 ← a
t5 ← t4 − t3
```

push 2
push b
multiply
push a
subtract

• Fairly compact
• Compiler can control reuse of names – clever choice can reveal optimizations
• ILOC & similar code

• Each instruction consumes top of stack & pushes result
• Very compact
• Easy to create and interpret
• Java bytecode, MSIL
Abstraction Levels in Linear IR

• Linear IRs can also be close to the source language, very low-level, or somewhere in between.

• Example: Linear IRs for C array reference $a[i][j+2]$

  – High-level: $t1 \leftarrow a[i,j+2]$
IRs for a[i][j+2], cont.

• Medium-level
  
  \[ \begin{align*}
  t1 & \leftarrow j + 2 \\
  t2 & \leftarrow i \times 20 \\
  t3 & \leftarrow t1 + t2 \\
  t4 & \leftarrow 4 \times t3 \\
  t5 & \leftarrow \text{addr a} \\
  t6 & \leftarrow t5 + t4 \\
  t7 & \leftarrow *t6 \\
  \end{align*} \]

• Low-level
  
  \[ \begin{align*}
  r1 & \leftarrow [fp-4] \\
  r2 & \leftarrow r1 + 2 \\
  r3 & \leftarrow [fp-8] \\
  r4 & \leftarrow r3 \times 20 \\
  r5 & \leftarrow r4 + r2 \\
  r6 & \leftarrow 4 \times r5 \\
  r7 & \leftarrow \text{fp} - 216 \\
  f1 & \leftarrow [r7+r6] \\
  \end{align*} \]
Abstraction Level Tradeoffs

• High-level: good for some high-level optimizations, semantic checking, but can’t optimize things that are hidden – like address arithmetic for array subscripting

• Low-level: need for good code generation and resource utilization in back end but loses some semantic knowledge (e.g., variables, data aggregates, source relationships)

• Medium-level: more detail but keeps more higher-level semantic information

• Many compilers use all 3 in different phases
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Control-flow graph is often an example of this
  - Basic IR is a graph
  - Nodes in the graph can be linear lists of IR instructions
What IR to Use?

• Common choice: all(!)
  – AST or other structural representation built by parser and used in early stages of the compiler
    • Closer to source code
    • Good for semantic analysis
    • Facilitates some higher-level optimizations
  – Hybrid IR for optimization phases
  – Transform to low-level IR for later stages of compiler
    • Closer to machine code
    • Exposes machine-related optimizations
    • Use to build control-flow graph
Coming Attractions

• Survey of compiler “optimizations”
  – Analysis and transformations (including SSA)

• Back-end organization in production compilers
  – Instruction selection and scheduling, register allocation

• Other topics depending on time
  – Dynamic languages? JVM? Memory management (garbage collection)? Any preferences?