CSE 401/M501 – Compilers

Intermediate Representations

Hal Perkins

Spring 2018
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)

- **Source**: characters → **Scanner** → tokens → **Parser** → IR
- **Semantic Analysis**: IR → IR (maybe different)
- **Middle (optimization)**: IR (often different) → **Code Gen** → Assembly or binary code
- **Target**: Assembly or binary code

UW CSE 401/M501 Spring 2018
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 target code
  – Typically will transform initial IR to one or more different IRs along the way

• Some general 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/quality 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

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

\[
\begin{align*}
\text{loadI 1} & \Rightarrow r1 \\
\text{sub rj,r1} & \Rightarrow r2 \\
\text{loadI 10} & \Rightarrow r3 \\
\text{mult r2,r3} & \Rightarrow r4 \\
\text{sub ri,r1} & \Rightarrow r5 \\
\text{add r4,r5} & \Rightarrow r6 \\
\text{loadI @A} & \Rightarrow r7 \\
\text{add r7,r6} & \Rightarrow r8 \\
\text{load r8} & \Rightarrow r9
\end{align*}
\]
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/Racket 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 subexpressions
- **Con**: less flexibility if part needs to be changed
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
```

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

```
push 2
push b
multiply
push a
subtract
```

- 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 ← a[i,j+2]`
More IRs for a[i][j+2]

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

• Low-level
  \[\begin{align*}
  r1 & \leftarrow [fp-4] \\
  r2 & \leftarrow r1 + 2 \\
  r3 & \leftarrow [fp-8] \\
  r4 & \leftarrow r3 \times 20 \\
  r5 & \leftarrow r4 + r2 \\
  r6 & \leftarrow 4 \times r5 \\
  r7 & \leftarrow fp – 216 \\
  f1 & \leftarrow [r7+r6] \\
  \end{align*}\]
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 – great for machine-independent optimizations. Many (all?) optimizing compilers work at this level

• Many compilers use all 3 in different phases
Three-Address Code (TAC)

• Usual form: $x \leftarrow y \ op \ z$
  – One operator
  – Maximum of 3 names
  – (Copes with: nullary $x \leftarrow y$ and unary $x \leftarrow \ op \ y$)

• Eg: $x = 2 \ * \ (m + n)$ becomes
  
  \[
  t_1 \leftarrow m + n; \quad t_2 \leftarrow 2 \ * \ t_1; \quad x \leftarrow t_2
  \]

  – You may prefer: add $t_1$, $m$, $n$; mul $t_2$, $2$, $t_1$; mov $x$, $t_2$

  – Invent as many new temp names as needed. “expression temps” – don’t correspond to any user variables; de-anonymize expressions

• Store in a quad(ruple)
  – <lhs, rhs1, op, rhs2>
Three Address Code

• Advantages
  – Resembles code for actual machines
  – Explicitly names intermediate results
  – Compact
  – Often easy to rearrange

• Various representations
  – Quadruples, triples, SSA (Static Single Assignment)
  – We will see much more of this...
Stack Machine Code Example

Hypothetical code for \( x = 2 \times (m + n) \)

```
pushaddr x
pushconst 2
pushval n
pushval m
add
mult
store
```

```
m
n
2
@x
?  
m + n
2
@x
?  
2*(m+n)
@x
?  
?  
```
Stack Machine Code

- Originally used for stack-based computers (famous example: B5000, ~1961)
- Often used for virtual machines:
  - Pascal – pcode
  - Forth
  - Java bytecode in a .class files (generated by Java compiler)
  - MSIL in a .dll or .exe assembly (generated by C#/F#/VB compiler)
- Advantages
  - Compact; mostly 0-address opcodes (fast download over network)
  - Easy to generate; easy to write a front-end compiler, leaving the 'heavy lifting' and optimizations to the JIT
  - Simple to interpret or compile to machine code
- Disadvantages
  - Somewhat inconvenient/difficult to optimize directly
  - Does not match up with modern chip architectures
Hybrid IRs

• Combination of structural and linear

• Level of abstraction varies

• Most common example: control-flow graph (CFG)
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
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 unit
    • We’ll ignore exceptions, at least for now
• Usually represented as some sort of a list although Trees/DAGs are possible
print("hello");
a = 7;
if (x == y) {
    print("same");
    b = 9;
} else {
    b = 10;
}
while (a < b) {
    a++;
    print("bump");
}
print("finis");
Basic Blocks: Start with Tuples

1 i = 1
2 j = 1
3 t1 = 10 * i
4 t2 = t1 + j
5 t3 = 8 * t2
6 t4 = t3 - 88
7 a[t4] = 0
8 j = j + 1
9 if j <= 10 goto #3
10 i = i + 1
11 if i <= 10 goto #2
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1
16 i = i + 1
17 if i <= 10 goto #13

Typical "tuple stew" - IR generated by traversing an AST

Partition into Basic Blocks:

• Sequence of consecutive instructions
• No jumps into the middle of a BB
• No jumps out of the middles of a BB
• "I've started, so I'll finish"
• (Ignore exceptions)
Basic Blocks: Leaders

Identify Leaders (first instruction in a basic block):
- First instruction is a leader
- Any target of a branch/jump/goto
- Any instruction immediately after a branch/jump/goto

Leaders in red. Why is each leader a leader?
Basic Blocks: Flowgraph

Control Flow Graph ("CFG", again!)

- 3 loops total
- 2 of the loops are nested

Most of the executions likely spent in loop bodies; that's where to focus efforts at optimization
Identifying Basic Blocks: Recap

• Perform linear scan of instruction stream

• A basic blocks begins at each instruction that is:

  – The beginning of a method
  – The target of a branch
  – Immediately follows a branch or return
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
What IR to Use?

• Common choice: all(!)
  – AST used in early stages of the compiler
    • Closer to source code
    • Good for semantic analysis
    • Facilitates some higher-level optimizations
  – Lower to linear IR for optimization and codegen
    • Closer to machine code
    • Exposes machine-related optimizations
    • Use to build control-flow graph
  – Hybrid (graph + linear IR = CFG) for dataflow & opt
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