Compiler Development (CMPSC 401)

Intermediate Representations

Janyl Jumadinova

March 28, 2019
Compiler

Diagram:

1. **Scanner**
   - Input: characters
   - Output: tokens

2. **Parser**
   - Input: tokens
   - Output: IR

3. **Semantic Analysis**
   - Input: IR
   - Output: IR (may be different)

4. **Optimizer**
   - Input: IR (often different)
   - Output: IR

5. **Code Gen**
   - Input: IR
   - Output: Assembly or binary code

6. **Target**

Flow:
- Source $\rightarrow$ Scanner $\rightarrow$ Parser $\rightarrow$ Semantic Analysis $\rightarrow$ Optimizer $\rightarrow$ Code Gen $\rightarrow$ Target
Intermediate Representation Generation

- The final phase of the compiler front-end.
- **Goal**: Translate the program into the format expected by the compiler back-end.
Intermediate Representation Generation

- The final phase of the compiler front-end.
- **Goal**: Translate the program into the format expected by the compiler back-end.
- Generated code need not be optimized; that’s handled by later passes.
- Generated code need not be in assembly; that can also be handled by later passes.
Intermediate Representation Generation

Why do IR Generation?

- Machine code has many constraints that inhibit optimization.
- Working with an intermediate language makes optimizations easier and clearer.
- gcc can handle C, C++, Java, Fortran, Ada, and many other languages.
- Each front-end translates source to the GENERIC language.
- Do most optimization on intermediate representation before emitting code targeted at a single machine.
Why do IR Generation?

- **Simplify certain optimizations:**
  - Machine code has many constraints that inhibit optimization.
  - Working with an intermediate language makes optimizations easier and clearer.
Intermediate Representation Generation

Why do IR Generation?

- **Simplify certain optimizations:**
  - Machine code has many constraints that inhibit optimization.
  - Working with an intermediate language makes optimizations easier and clearer.

- **Have many front-ends into a single back-end:**
  - gcc can handle C, C++, Java, Fortran, Ada, and many other languages.
  - Each front-end translates source to the GENERIC language.
Intermediate Representation Generation

Why do IR Generation?

- **Simplify certain optimizations:**
  - Machine code has many constraints that inhibit optimization.
  - Working with an intermediate language makes optimizations easier and clearer.

- **Have many front-ends into a single back-end:**
  - gcc can handle C, C++, Java, Fortran, Ada, and many other languages.
  - Each front-end translates source to the GENERIC language.

- **Have many back-ends from a single front-end:**
  - Do most optimization on intermediate representation before emitting code targeted at a single machine.
IRs are like type systems they are extremely hard to get right.

Need to balance needs of high-level source language and low-level target language.

Too high level: can’t optimize certain implementation details.

Too low level: can’t use high-level knowledge to perform aggressive optimizations.

Often have multiple IRs in a single compiler.
Architecture of gcc

- Source Code
- AST
  - GENERIC
    - High GIMPLE
    - SSA
    - Low GIMPLE
    - RTL
- Machine Code
Survey of Intermediate Representations

- Graphical Representations
  - Control Flow Graph
  - Dependence Graph
  - Concrete/Abstract Syntax Trees (ASTs)
Survey of Intermediate Representations

- Graphical Representations
  - Control Flow Graph
  - Dependence Graph
  - Concrete/Abstract Syntax Trees (ASTs)

- Linear Representations
  - Stack based
  - Three-Address Code
In most compilers, the parser builds an intermediate representation of the program, typically an AST.

Rest of the compiler transforms the IR to improve ("optimize") it and eventually translates it to final code.

Typically will transform initial IR to one or more lower level IRs along the way.
Decisions affect speed and efficiency of the rest of the compiler
General rule: Compile time is important, but performance of the executable is more important.
Decisions affect speed and efficiency of the rest of the compiler.

General rule: Compile time is important, but performance of the executable is more important.

Typical case: compile few times, run many times.

So make choices that improve compile time, as long as they don’t impact performance of generated code.
Desirable properties:
- Easy to generate
- Easy to manipulate
- Expressive
- Appropriate level of abstraction
IR Design Dimensions

- **Structure**:
  - Graphical (trees, graphs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs with linear code in basic blocks)
IR Design Dimensions

- **Structure**:
  - Graphical (trees, graphs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs with linear code in basic blocks)

- **Abstraction Level**:
  - High-level, near to source language
  - Low-level, closer to machine, more exposed to compiler
Survey of Intermediate Representations

- Graphical Representations
  - Control Flow Graph
  - Dependence Graph
  - Concrete/Abstract Syntax Trees (ASTs)
Survey of Intermediate Representations

- **Graphical Representations**
  - Control Flow Graph
  - Dependence Graph
  - Concrete/Abstract Syntax Trees (ASTs)

- **Linear Representations**
  - Stack based
  - Three-Address Code
Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source, control flow, data dependence
Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source, control flow, data dependence
- May be large (especially syntax trees)
Graphical IRs

- IRs represented as a graph (or tree)
- Nodes and edges typically reflect some structure of the program
  - E.g., source, 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 dependence graphs
  - Often used in optimization and code generation
The full grammar is needed to guide the parser, but contains many extraneous details—e.g., syntactic tokens, rules that control precedence. Typically the full syntax tree does not need to be used explicitly.
Graphical IR: Concrete Syntax Trees

- The full grammar is needed to guide the parser, but contains many extraneous details
  - E.g., syntactic tokens, rules that control precedence
- Typically the full syntax tree does not need to be used explicitly
Graphical IR: Abstract Syntax Trees

- Abstract Syntax Trees can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal.

For a \[i+j\], this might be:

- **Subscript**
- **Mult**

  - Id(A)
  - Id(i)
  - Id(j)

Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimization.
Graphical IR: Abstract Syntax Trees

- Want only essential structural information (omit extra junk)
- Can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal.
Graphical IR: Abstract Syntax Trees

- Want only essential structural information (omit extra junk)
- Can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal. For $a[i+j]$, this might be:

  ```
  Subscript
  ID(A)
  Plus
  ID(i)
  ID(j)
  ```
Graphical IR: Abstract Syntax Trees

- Want only essential structural information (omit extra junk)
- Can be represented explicitly as a tree or in a linear form, e.g., in the order of a depth-first traversal. For $a[i+j]$, this might be:

  Subscript
  Id(A)
  Plus
  Id(i)
  Id(j)

- Common output from parser; used for static semantics (type checking, etc.) and sometimes high-level optimization
DAG = Directed Acyclic Graph

In compilers, typically used to refer to an AST like structure, where common components may be reused. E.g, the $2*a$ in $2*a + 2*a*b$ (above).
DAG = Directed Acyclic Graph

- In compilers, typically used to refer to an AST like structure, where common components may be reused. E.g., the $2*a$ in $2*a + 2*a*b$ (above).

- **Pros:** Saves space, makes common subexpressions explicit.

- **Cons:** If want to change just one occurrence, need to split off. If variable value may change between evaluations, may not want to treat as common.
Control Flow Graph (CFG)

- Nodes are **Basic Blocks**
- Code that always executes together (i.e., no branches into or out of the middle of the block).
  - i.e., “straight-line code”
Control Flow Graph (CFG)

- **Nodes are Basic Blocks**
- **Code that always executes together** (i.e., no branches into or out of the middle of the block).
  - i.e., “straight-line code”
- **Edges represent paths** that control flow could take.
  - i.e., possible execution orderings.
  - Edge from Basic Block A to Basic Block B means Block B could execute immediately after Block A completes.
Control Flow Graph (CFG)

- Nodes are **Basic Blocks**
  - Code that always executes together (i.e., no branches into or out of the middle of the block).
    - i.e., “straight-line code”
- Edges represent paths that control flow could take.
  - i.e., possible execution orderings.
  - Edge from Basic Block A to Basic Block B means Block B could execute immediately after Block A completes.
- Required for much of the analysis done in the optimizer.
Control Flow Graph (CFG)

```plaintext
print("hello");
a = 7;
if (x == y) {
    print("equal");
b = 9;
} else {
    b = 10;
}
while (a < b) {
    a++;
    print("increase");
}
print("done");
```

- `print("hello"); a = 7;` is executed.
- The program enters the `if` statement:
  - `x == y` is checked.
  - If `true`, `print("equal"); b = 9;` is executed.
  - If `false`, `b = 10;` is executed.
- The `while` loop runs while `a < b`.
  - `a++` increments `a`.
  - `print("increase");` is executed.
- The program prints “done”.

The control flow graph shows the decision path based on the conditions:
- If `x == y`, then `b = 9`.
- If `false`, then `b = 10`.
- The `while` loop continues with `a++` and `print("increase");` until `a` is no longer less than `b`.
- Finally, “done” is printed.
print(“hello”);
a = 7;
if (a < 10)
{
    print(“Increase”);
    a = a + 1;
}

print(“done”);

Note: There are variations in how function calls in basic blocks are treated. It may depend on the level of abstraction of the IR, as well as the semantics of the language. For example, if a function may throw an exception, the call should terminate the basic block (since there is no guarantee that the call will return to the same point). In low-level IRs, calls may also terminate blocks.
Dependence Graph

- Often used in conjunction with another IR.

In a data dependence graph, edges between nodes represent "dependencies" between the code represented by those nodes.
- If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
- A → B means compiler can't move B before A.
- Granularity of nodes varies. Depends on abstraction level of rest of IR. E.g., nodes could be loads/stores, or whole statements.

- E.g.,
  a = 2; b = 2; c = a + 7;

  Where is the dependence?
Dependence Graph

- Often used in conjunction with another IR.
- In a data dependence graph, edges between nodes represent “dependencies” between the code represented by those nodes.
  - If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
  - A → B means compiler can’t move B before A.
Dependence Graph

- Often used in conjunction with another IR.
- In a data dependence graph, edges between nodes represent “dependencies” between the code represented by those nodes.
  - If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
  - A → B means compiler can’t move B before A.
  - Granularity of nodes varies. Depends on abstraction level of rest of IR. E.g., nodes could be loads/stores, or whole statements.
Dependence Graph

- Often used in conjunction with another IR.
- In a data dependence graph, edges between nodes represent "dependencies" between the code represented by those nodes.
  - If A and B access the same data, and A must occur before B to achieve correct behavior, then there is a dependence edge from A to B.
  - A → B means compiler can’t move B before A.
  - Granularity of nodes varies. Depends on abstraction level of rest of IR. E.g., nodes could be loads/stores, or whole statements.
- E.g., `a = 2; b = 2; c = a + 7;`
- Where is the dependence?
Types of Dependencies

- **Read-after-write (RAW)/ “flow dependence”**
  - E.g., `a = 7; b = a + 1;`
  - The read of ‘a’ must follow the write to ‘a’, otherwise it won’t see the correct value.

- **Write-after-read (WAR)/ “anti dependence”**
  - E.g., `b = a * 2; a = 5;`
  - The write to ‘a’ must follow the read of ‘a’, otherwise the read won’t see the correct value.

- **Write-after-write (WAW)/ “output dependence”**
  - E.g., `a = 1; ... a = 2; ...`
  - The writes to ‘a’ must happen in the correct order, otherwise ‘a’ will have the wrong final value.
Types of Dependencies

- **Read-after-write (RAW)/“flow dependence”**
  
  E.g., \( a = 7; \ b = a + 1; \)

  The read of ‘a’ must follow the write to ‘a’, otherwise it won’t see the correct value.

- **Write-after-read (WAR)/“anti dependence”**
  
  E.g., \( b = a \times 2; \ a = 5; \)

  The write to ‘a’ must follow the read of ‘a’, otherwise the read won’t see the correct value.
Types of Dependencies

- **Read-after-write (RAW)/“flow dependence”**
  - E.g., \( a = 7; \ b = a + 1; \)
  - The read of ‘a’ must follow the write to ‘a’, otherwise it won’t see the correct value.

- **Write-after-read (WAR)/“anti dependence”**
  - E.g., \( b = a \times 2; \ a = 5; \)
  - The write to ‘a’ must follow the read of ‘a’, otherwise the read won’t see the correct value.

- **Write-after-write (WAW)/“output dependence”**
  - E.g., \( a = 1; \ldots a = 2; \ldots \)
  - The writes to ‘a’ must happen in the correct order, otherwise ‘a’ will have the wrong final value.
Loop-Carried Dependence

Loop carried dependence:
A dependence across iterations of a loop

for (i = 0; i < size; i++)
    x = foo(x);
Loop-Carried Dependence

Loop carried dependence:
A dependence across iterations of a loop

```c
for (i = 0; i < size; i++)
  x = foo(x);
```

- **RAW loop carried dependence**: the read of ‘x’ depends on the write of ‘x’ in the previous iteration.
Loop-Carried Dependence

Loop carried dependence:
A dependence across iterations of a loop

```c
for (i = 0; i < size; i++)
    x = foo(x);
```

- **RAW loop carried dependence**: the read of ‘x’ depends on the write of ‘x’ in the previous iteration.
  - If the compiler “understands” the nature of the dependence, it can sometimes be removed or dealt with.
  - Often use sophisticated array subscript analysis for this.
Dependence Graph

```python
a = 7;
print("hello");
while (a < b) {
    print("increase");
a++;
}
print("done");

LCD: Loop-Carried Dependence
```
Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
- *Commonly used:* arrays, linked structures
- *Examples:* Three Address Code, stack machine

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

- Each instruction: pop operands, push result.
- Very compact
- Easy to create interpreter.
- Java bytecode

push 2
push b
multiply
push a
subtract
**Abstraction Level Trade-Offs**

- **High-level**: good for some high-level optimizations, semantic checking, but can’t optimize things that are hidden (e.g., address calculations in subscript operations)

- **Low-level**: Needed for good code generation and resource utilization in back end, but lose some semantic knowledge (e.g., variables)

- **Medium-level**: Exposes more, but still keeps some semantic knowledge.
Abstraction Level Trade-Offs

- **High-level**: good for some high-level optimizations, semantic checking, but can’t optimize things that are hidden (e.g., address calculations in subscript operations)
- **Low-level**: Needed for good code generation and resource utilization in back end, but lose some semantic knowledge (e.g., variables)
Abstraction Level Trade-Offs

- **High-level**: good for some high-level optimizations, semantic checking, but can’t optimize things that are hidden (e.g., address calculations in subscript operations)
- **Low-level**: Needed for good code generation and resource utilization in back end, but lose some semantic knowledge (e.g., variables)
- **Medium-level**: Exposes more, but still keeps some semantic knowledge.
Abstraction Level Trade-Offs

- **High-level**: good for some high-level optimizations, semantic checking, but can’t optimize things that are hidden (e.g., address calculations in subscript operations)
- **Low-level**: Needed for good code generation and resource utilization in back end, but lose some semantic knowledge (e.g., variables)
- **Medium-level**: Exposes more, but still keeps some semantic knowledge.
- Many compilers use all three at different phases
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- *Most common example*: control-flow graph
  - **Nodes**: basic blocks. Within nodes, linear representation of basic block’s code.
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- *Most common example:* control-flow graph
  - **Nodes:** basic blocks. Within nodes, linear representation of basic block’s code.
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies

*Most common example*: control-flow graph
- **Nodes**: basic blocks. Within nodes, linear representation of basic block’s code.

- May also see Dependence Graph implemented as edges between linear instructions.
  - Possibly even inside CFG basic
What IR to use?

Common choice: all(!)
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
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
Lower to low-level linear IR for later stages of compiler
- Closer to machine code
- Exposes machine-related optimizations
- Good for resource allocation and scheduling