Languages and Compiler Design Part II

Topics
- Semantics
- Interpreters
- Runtime Organization
- Intermediate Code Generation
- Machine Code Generation
- Optimization

Project
- Build PCAT Interpreter
- Complete PCAT Compiler

Themes
- Mapping from high-level to low-level
- Implementing resource management
- Integration with OS and hardware environment
- Syntax-directed techniques

Translator Structure

Source code
\[ \rightarrow \] Lexical Analysis
\[ \rightarrow \] Tokens
\[ \rightarrow \] Parsing
\[ \rightarrow \] Abstract syntax
\[ \rightarrow \] Type-checking
\[ \rightarrow \] Intermediate Code Generation
\[ \rightarrow \] Optimization
\[ \rightarrow \] Machine Code Generation
\[ \rightarrow \] Machine code

Language Design Issues

Syntax

Data Types

Control Structures

Runtime Models
Translation to Abstract Syntax

\[
\begin{align*}
a &= \text{if } c > 7 \text{ and } b = e \\
\quad &\text{then } 3 \ast (d + e) \\
\quad &\text{else } -c \ast (d + 2)
\end{align*}
\]

Source code

\[
\begin{aligned}
&: = \\
&\quad \text{if} \\
&\quad \quad \text{and} \\
&\quad \quad 3 \ast + - + \\
&\quad \quad c \ast d e c d d 2
\end{aligned}
\]

AST

"Three-address code" - a typical linear I.R.

Generate list of "instructions"

Each has an operator, up to 2 args, and up to 1 result

Instructions can be labeled

Operands are names for locations in some abstract memory (e.g., symbol table entries)

Examples of instructions:

\[
\begin{align*}
A &= B \quad \text{copy} \\
A &= \text{op } B \quad \text{binary ops} \\
A &= \text{op } B \quad \text{unary ops} \\
\text{goto } L \quad \text{jumps} \\
\text{if A relop B goto L} \quad \text{conditional jumps} \\
\text{param A} \quad \text{procedure call setup} \\
\text{call } P, N \quad \text{procedure call} \\
\text{return } N \quad \text{procedure return} \\
A[I] \quad \text{array dereference}
\end{align*}
\]

Intermediate Code Generation

Emit I.C. (or "I.R.") from abstract syntax or directly from parser

Advantages:

- Keeps more of compiler machine-independent
- Facilitates some optimizations

Typical examples:

- Postfix
- Trees or DAGs
- Three-address code (quadruples, triples, etc.)

Abstracts key features of machine architectures

- E.g., sequential execution, explicit jumps

But hides details

- E.g., # of registers, style of conditionals, etc.

Many possible levels

3-Address Code Example

\[
\begin{align*}
a &= \text{if } c > 7 \text{ goto L1} \\
&\quad \text{goto L2} \\
&\quad \text{L1: if } b = e \text{ goto L3} \\
&\quad \text{L2: } t1 := d + 2 \\
&\quad \quad t2 := -c \\
&\quad \quad t3 := t1 \ast t2 \\
&\quad \quad \text{goto L4} \\
&\quad \text{L3: } t1 := d + e \\
&\quad \quad t3 := 3 \ast t1 \\
&\quad \text{L4: } a := t3
\end{align*}
\]

Linearized

Nested conditionals expanded (badly)

Temporaries for all intermediate results
Machine-code Generation

“Read” I.R.; generate assembly language (symbol or binary).
Must cooperate with I.R. to define and “enforce” runtime environment.
Must deal with idiosyncrasies of target machine,
  • e.g., instruction selection
and perform resource management,
  • e.g., register assignment.
Lots of case analysis, especially for complex target architectures.
Can do by hand, but hard.
Tools limited but sometimes useful; mainly based on pattern matching

Optimization

Improve (don't perfect) code by removing inefficiencies:
  • In original program
  • Introduced by compiler itself
Can operate on source, I.R., object code.
Local Improvements
  • Example: changing
    if c > 7 goto L1
    goto L2
    L1: ...
    L2: ...
  • to
    if c <= 7 goto L2
    L1: ...
    L2: ...

Sample machine code

Assumes a global; b,c args; d,e locals.
Illustrates register conventions, delay slots, etc.

```
sethi %hi(_a),%o2
cmp %i1,7
ble L2
or %o2,%lo(_a),%i1
cmp %i0,%i2
bne L4
sub %g0,%i1,%o0
add %i0,%i1,%l1
```

```
sll %o1,1,%o0
add %o0,%i1,%o0
b L3
st %o0, [%o2+%lo(_a)]
L2: sub %g0,%i1,%o0
L4: call .umul,0
add %i0,2,%o1
st %o0, [%l1]
L3:
```

```
```

“Global” Improvements

  • Example: changing
    for (i := 0; i < 1000; i++)
    a[i] := b*c + i;
  • to
    t1 := b * c;
    for (i = 0; i < 1000; i++)
    a[i] = t1 + i;

Interprocedural improvements
  • Example: Inlining a function
Most of a modern compiler is devoted to optimization.
**Interpretation**

Simulate execution of program (source, AST, or other IR) on an abstract machine.

Implement abstract machine on a real machine.

Inputs to interpreter are
- Program to be interpreted
- Input to that program

Simpler than compiling and takes no time up front, but interpreted code runs (~10X) more slowly than compiled code.

Much more portable than real machine code (as for Java).

Helps with semantic definition.