Intermediate Code Generation

Createlinear representation of program. Result can be machine code, assembly code, code for an abstract machine (e.g., the JVM), threaded code, or anything in between.

Common representation of intermediate code depend on target machine:
- Anything in between.
- Result can be machine code, assembly code, code for an abstract machine (e.g., the JVM), threaded code, or anything in between.
- Create linear representation of program:

Intermediate Code for Elif parts

Generate (symbolic) labels
Generate quadruples for some descendant node
Place label in code

For each alternative, place in code the current else-label, and generate a new one. All alternatives inherit the end label from parent.

if cond1 then
S1 t1 := cond1
endif
if not t1 goto else
label1
endif

elsif cond2 then
S2
endif
if not t2 goto else
label2
endif

else
S3
endif

In all cases, another (top down) tree traversal may need label for then-statement, for short-circuit conditions:

Intermediate Code for if-statement parts

endif
label1:
else-label:
goto else-label

endif
label2:
else-label:
goto else-label

endif
label:
else-label:
goto else-label

endif
label;
else-label:
goto else-label

endif
label;
else-label:
goto else-label

endif
label;
else-label:
goto else-label

More Details of Code Generation

Outline the part of the code generator that deals with (2-branch) conditional statements:

```plaintext
Function code_gen(node): reg var end_label; integer, end_label := new_label;
    case node.kind of
        if: code_gen(node.child1); issue if(!R t1) then goto L1 (else-label);
            code_gen(node.child2); issue goto L1 (end-label);
            issue goto L1 (end-label);
        end;
```

Example of Code Generation

Consider the statement:

```plaintext
if C1 then if C2 then x := 1 else x := 2
else if C3 then x := 3 else x := 4
```

The generated code is given by:

```plaintext
if !C1 goto L1
if !C2 goto L3
x := 1 goto L2
L3:
x := 2 goto L2
L1:
if !C3 goto L5
x := 3 goto L6
L5:
x := 4 goto L6
L2:
```

Example of an Improved Generation

Reconsider the statement:

```plaintext
if C1 then if C2 then x := 1 else x := 2
else if C3 then x := 3 else x := 4
```

The improved generated code is given by:

```plaintext
if !C1 goto L1
if !C2 goto L3
x := 1 goto L2
L3:
x := 2 goto L2
L1:
if !C3 goto L5
x := 3 goto L6
L5:
x := 4 goto L6
L2:
```
Intermediate Code Generation

A.Pnueli

Code Generation for While Loops

1. Generate additional labels for composite short-circuit
2. Inherit target labels from enclosing control structure
3. Else-label:
4. Quadruples for S1
5. Then-label:
6. If not B2 goto else-label
7. If B1 goto then-label
8. { \( \text{if } B1 \text{ then } S1 \ldots \text{ if } (B1 \lor B2) S1 \ldots \text{ if } B2 \text{ else } S1 \ldots \) }
9. Short-circuit expressions are treated as control

Code for Short-Circuit Expressions

1. Place Test at End to Utilize Loop Instruction

Semantics: Loop not executed if range is null, so must test before first body execution.

Intermediate Code for Numeric Loops

1. Code for Short-Circuit Expressions

2. Place Test at End to Utilize Loop Instruction

3. Generate additional labels for composite short-circuit

4. Inherit target labels from enclosing control structure

5. Else-label:

6. Quadruples for S1

7. Then-label:

8. If not B2 goto else-label

9. If B1 goto then-label

10. \{ \( \text{if } K \leq 1 \text{ goto start-label} \) \}

11. \( K := K + 1 \)

12. { \( \text{test-label: quadruples for S1} \) }

13. \( K := 11 - 1 \)

14. If \( \text{cond} \) break; \( \text{S1} \)

15. \( \text{end-loop: goto start-loop} \)

16. \( \text{end-loop: goto end-loop} \)

17. \( \text{end-loop for S1} \)

18. \( \text{end-loop for S2} \)

19. \( \text{end-loop for S3} \)

20. \( \text{S3:} \)

21. \( \text{S2:} \)

22. \( \text{S1:} \)

23. \( \text{while (\text{cond})} \)

24. \( \text{emit loop instruction} \)

25. \( \text{for } J \text{ in expr1..expr2} \)

26. \( \text{goto start-loop} \)

27. \( \text{goto end-loop} \)

28. \( \text{S1:} \)

29. \( \text{S2:} \)

30. \( \text{S3:} \)

31. \( \text{S1:} \)

32. \( \text{S2:} \)

33. \( \text{S3:} \)
Intermediate Code for Case Statements

- If range is small and most cases are defined, create a jump table as an array of code addresses, and generate indirect jumps.
- $d = b - 4 * c$
- $d = b - 4 * c$
- Load from memory
- Duplicate value on top of stack
- Push constant
- Multiply
- Push
- Multiply
- Push
- Multiply
- Push
- Duplicate
- Load

Evaluating Expressions: Stack Machines

- Zero-address instructions: push, pop, arithmetic
- Binary operations: remove two operands, push result
- Jump table: array of code addresses, and generate indirect jumps.
Aho-Sethi Algorithm for Minimal Registers Use

For a constant:
return 1

For a variable:
return 1

For an expression $arg1 op arg2$:
Let $min1 = \text{minimum for } arg1$
Let $min2 = \text{minimum for } arg2$
If $min1 \neq min2$
then return $\max(min1, min2)$
elsereturn $min1 + 1$

Optimal register use:

- Else return $\min1 + 1$
- If $\min1 \neq \min2$ then return $\max(\min1, \min2)$

For an expression $arg1 op arg2$:
- Let $\min1 = \text{minimum for } arg1$
- Let $\min2 = \text{minimum for } arg2$

For a variable $arg2$:
- For a variable: Return 1
- For a constant: Return 1

Example

Program:
\[ \text{exp-float is part of runtime library linked with user} \]
\[ Y := \text{exp-float}(x, n) \]
\[ Y := x^{**}n ; \]

General case requires run-time support:
\[ Y = T1 * T1 \]
\[ T1 = x^{**}4 ; \]
\[ T1 = x^{**}2 ; \]
\[ Y = x ; \]

Simple cases are computed efficiently:

Exponentiation

Optimal register use:

At each step, compute subtree with largest weight

Complete weight of each node

Code Generation for More Complex Constructs

- Treetransformations
- exponentiation
- Inline expansion
- Exponentiation
- CodeGenerationforMoreComplex

Calls to run-time routines
- Dispatching calls

Storage management
- 64-bit arithmetic
- 64-bit arithmetic