Uses of Boolean Expressions

Used to drive conditional execution of program sections.

Examples:

IF (a < 17) OR (b = 12) THEN
  ...
else
  ...

WHILE NOT ((x+1) > 39) DO
  ...
end

(In some languages) may be assigned to boolean variables or passed as parameters.

Example:

VAR b : BOOLEAN := (a < 17) OR (b = 12);
...
IF b THEN ...
ELSE ...
...
myproc(b); (* procedure call *)
...

Boolean Expressions

Two representations may be useful:

- **Value Representation.**
  
  Encode `true` and `false` numerically, e.g., as 1 and 0, and treat boolean expressions like arithmetic expressions.
  
  Pro: Language may support boolean values.
  Con: Bad match to hardware.

- **Flow-of-control Representation.**
  
  Position in generated code represents boolean value.
  
  Pro: Good when “short-circuit” evaluation is allowed (or required), e.g., in C expression `e1 | e2`, `e2` should be evaluated only if `e1` is false.
  
  Reminder: Some languages mandate short-circuit evaluation; others prohibit it; still others leave it up to the compiler writer.
  
  Pro: Convenient for control statements.

- For PCAT, we’ll use flow-of-control approach, and convert to values when necessary.

Sample Productions for Value-based Conditional Evaluation

```plaintext
B := E1 '<' E2
B.place = newtemp()
B.code =
  let true = newlabel()
  after = newlabel()
in E1.code @ E2.code @
  [gen(true,if<,E1.place,E2.place),
   gen(B.place,:=,0,_),
   gen(after,goto,_,_),
   gen(true,:,_,_),
   gen(B.place,:=,1,_),
   gen(after,:,_,_)]
```

Generates:

```plaintext
IF E1 < E2 GOTO L1
T := 0
GOTO L2
L1: T := 1
L2: ...
```
More Sample Value-based Productions

\( B := B1 \ OR \ B2 \)

\( B\text{.place} = \text{newtemp()} \)

\( B\text{.code} = B1\text{.code} \ \@ \ B2\text{.code} \ @ \) \[\text{gen}(B\text{.place},|, B1\text{.place},B2\text{.place})]\]

Here \(|\) represents bit-wise OR. Note that this implements NON-short-circuiting form of OR.

\( S := \text{IF } B \text{ THEN } S1 \text{ ELSE } S2 \)

\( S\text{.code} = \text{let } \text{false} = \text{newlabel()} \)

\( \text{after} = \text{newlabel()} \)

\( \text{in } B\text{.code} @ \) \[\text{gen}(\text{false},\text{if=},B\text{.place},0)\] \@ \( S1\text{.code} @ \)

\[\text{gen}(\text{after},\text{goto},_,-,_)\] \@ \[\text{gen}(\text{false},:,-,_)\] \@ \( S2\text{.code} \)

\[\text{gen}(\text{after},:,-,_)\]

Generates:

\[
\begin{align*}
\text{IF } B = 0 & \text{ GOTO L1} \\
S1 & \\
\text{GOTO L2} \\
L1: & S2 \\
L2: & \ldots
\end{align*}
\]

Basic Control-flow Representation

Idea: Code generated for boolean and relational expressions has true and false “exits”, i.e., code evaluates expression and then jumps to one place if true and another place if false.

- Relational expressions perform test and jump to true or false exit accordingly.
- Boolean variables and constants jump directly to appropriate true or false exit.
- Boolean expressions simply adjust/combine true/false exits of their sub-expressions.
- Conditional statements define true and false exits of boolean sub-expression to point to appropriate code blocks, e.g., THEN and ELSE branches.
- If boolean-typed expression must deliver a value, true and false exits are defined to point to code that loads the value.

Example Value-based Code

IF \((a > 7) \ OR \ (b = 5)\) THEN \(x = 7\) ELSE \(y = 2\);

\[
\begin{align*}
t1 := & \text{addr } a \\
t2 := & *t1 \\
t3 := & \text{const } 7 \\
& \text{if } t2 > t3 \text{ goto L1} \\
t4 := & \text{const } 0 \\
& \text{goto L2} \\
L1: & t4 := \text{const } 1 \\
L2: & t5 := \text{addr } b \\
t6 := & *t5 \\
t7 := & \text{const } 5 \\
& \text{if } t6 = t7 \text{ goto L3} \\
t8 := & \text{const } 0 \\
& \text{goto L4} \\
L3: & t8 := \text{const } 1 \\
L4: & t9 := t4 \ OR t8 \\
& \text{if } t9 = 0 \text{ goto L5} \\
t10 := & \text{const } 7 \\
t11 := & \text{addr } x \\
*t11 := & t10 \\
& \text{goto L6} \\
L5: & t12 := \text{const } 2 \\
t13 := & \text{addr } y \\
*t13 := & t12 \\
L6: &
\end{align*}
\]

Example (assuming short-circuiting)

IF \((a > 7) \ OR \ (b = 5)\) THEN \(x = 7\) ELSE \(y = 2\);

\[
\begin{align*}
t1 := & \text{addr } a \\
t2 := & *t1 \\
t3 := & \text{const } 7 \\
& \text{if } t2 > t3 \text{ goto L1} \\
t4 := & \text{const } 0 \\
& \text{goto L2} \\
L1: & t7 := \text{const } 7 \\
t8 := & \text{addr } x \\
*t8 := & t7 \\
& \text{goto L3} \\
L2: & t9 := \text{const } 2 \\
t10 := & \text{addr } y \\
*t10 := & t9 \\
L3: &
\end{align*}
\]
Conditional Statements (Somewhat Naive Approach)

Use control flow representation for boolean-typed expressions; define labels on per-statement basis.

\[
S := \text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \\
B.true = \text{newlabel}(); \\
B.false = \text{newlabel}(); \\
S.code = \\
\text{let after = newlabel() in } \\
B.code @ \text{[gen(B.true,:,\_,\_)]} @ S_1.code @ \text{[gen(after,goto,\_,\_)]} @ \text{[gen(B.false,:,\_,\_)]} @ S_2.code @ \text{[gen(after,:,\_,\_)]} \\
\]

Generates:

\[
\text{IF } B \text{ GOTO L1} \\
\text{GOTO L2} \\
L1: S1 \\
\text{GOTO L3} \\
L2: S2 \\
L3: \\
\]

Relational Expressions

Inherit true and false label attributes. Pass them down to subexpressions, after suitable manipulation; synthesize code attribute.

Again, no place attribute.

\[
B := E1 \text{ OR } E2 \\
B1.true = B.true \\
B1.false = \text{newlabel}() \\
B2.true = B.true \\
B2.false = B.false \\
B.code = B1.code @ \text{[gen(B1.true,:,\_,\_)]} @ \text{[gen(B1.false,:=,1,\_)]} @ \text{[gen(B2.true,:,\_,\_)]} @ \text{[gen(B2.false,:=,0,\_)]} \\
\]

\[
B := E1 \text{ AND } E2 \\
B1.true = \text{newlabel}() \\
B1.false = B.false \\
B2.true = B.true \\
B2.false = B.false \\
B.code = B1.code @ \text{[gen(B1.true,:,\_,\_)]} @ \text{[gen(B1.false,:=,1,\_)]} @ \text{[gen(B2.true,:,\_,\_)]} @ \text{[gen(B2.false,:=,0,\_)]} \\
\]

\[
B := \text{NOT } B \\
B1.true = B.false \\
B1.false = B.true \\
B.code = B1.code \\
\]

Conversions to and from value form

Boolean-typed identifiers (variables, true and false constants) must be “converted” to control-flow form when tested.

\[
B := V \\
B.code = V.code @ \text{[gen(B.false,if=,V.place,0),}} \\
\text{[gen(B.true,goto,\_,\_)]} \\
\]

(Assuming 0 = false, non-0 = true)

Similarly, must convert other way when a value is needed, generating code to build a value into a place.

\[
E := B \\
B.true = \text{newlabel}() \\
B.false = \text{newlabel}() \\
E.place = \text{newtemp}() \\
E.code = \\
\text{let after = newlabel() in } \\
B.code @ \text{[gen(B.true,:,\_,\_),}} \\
\text{[gen(E.place,\_=,1,\_)}, \\
\text{[gen(after,goto,\_,\_),}} \\
\text{[gen(B.false,:,\_,\_),}} \\
\text{[gen(E.place,\_=,0,\_)}, \\
\text{[gen(after,:,\_,\_)}}} \\
\]

\[
\]

\[
\]

\[
\]
Handling Loop Exits

Same label-passing approach can be used to implement break or exit statements that can cause jumps out of loops. We simply add a break inherited attribute to statements!

```
S := BREAK
    S.code = gen(S.break, goto, _, _)
S := LOOP S END
    S.break = newlabel();
    S.code =
        let top = newlabel()
        in [gen(top, :, _, _)] @
            S.code @
            [gen(top, goto, _, _),
             gen(S.break, :, _, _)]
```

Other loop statements (like WHILE) must define and pass a similar appropriate label to their child statement.

All other (non-loop) statement translations must pass the break attribute through (unchanged) to their children!

---

Improving Jump Generation

- Code for each statement always concludes by “falling through” to next statement.
- There is no information flow between code generation for statements.

```
S := S1 ;' ;' S2
    S.code = S1.code @ S2.code
```

This can lead to bad code, e.g.,

```
WHILE B1 DO (WHILE B2 DO S)
```

generates

```
L1: IF B1 GOTO L2
       GOTO L3
L2: IF B2 GOTO L4
       GOTO L5 "jump to jump"
L4: S
       GOTO L2
L5: GOTO L1
L3:
```

We can eliminate problems like this during optimization, but it’s easy to avoid some of them in the first place.

---

Idea: Defer Definition of Target Labels

- Give each statement an inherited attribute .next, which says where to transfer control after statement.
- Code generated for each statement guarantees either to transfer control to .next label or to “fall through.”

```
S := S1 ;' ;' S2
    S1.next = newlabel()
    S2.next = S.next
    S.code = S1.code @
        [gen(S1.next, :, _, _)] @
        S2.code
S := WHILE B DO S1
    B.true = newlabel()
    B.false = S.next
    S1.next = newlabel()
    S.code = [gen(S1.next, :, _, _)] @
        B.code @
        [gen(B.true, :, _, _)] @
        S1.code @
        [gen(S1.next, goto, _, _)]
```

and similarly for other compound statements.
Backpatching

Target label attributes (true, false, break, etc.) are inherited, so won’t work with one-pass bottom-up code generation, e.g. when generating code while doing bottom-up parsing.

Solution: Instead, keep lists of locations of gotos that need to be filled in (“backpatched”) when final target is known. These backpatch lists are synthesized attributes.

Example (to fill in): (a > 7) OR (b = 5)

```
1. t1 := addr a
2. t2 := *t1
3. t3 := const 7
4. if t2 > t3 goto _____
5. goto _____

6. t4 := addr b
7. t5 := *t4
8. t6 := const 5
9. if t5 = t6 goto _____
10. goto _____
```

At reduction for B := B1 OR B2

- Backpatch B1.false list with address of first instruction in B2.
- Merge B1.true and B2.true to form B.true.
- Make B2.false into B.false.

Case Statements

```
case e of
  v1 : s1
  v2 : s2
  ...  
v_n : s_n
else s
end
```

Good code generation for case statement depends on analysis of the values on the case labels v_i.

Options include:

- List of conditional tests and jumps (linear search).
- Binary decision code (binary tree).
- Other search code (e.g., hash table).
- Jump table (constant time).
- Hybrid schemes.

Best option depends on range of values (min and max) and their “density,” i.e., what percentage of the values in the range are used as labels.

Jump tables work well for dense value sets (even if large), but waste lots of space for sparse sets. Linear search works well for small value sets.

Backpatching (Continued)

At reduction for conditional statement, backpatch true and false lists for expression.

E.g.: On reducing if B then S1 else S2, backpatch B.true to location of S1 and B.false to location of S2.

Example (to fill in):

```
IF (a > 7) OR (b = 5) THEN x := 7 ELSE y := 2
```

```
1. t1 := addr a
2. t2 := *t1
3. t3 := const 7
4. if t2 > t3 goto _____
5. goto _____

6. t4 := addr b
7. t5 := *t4
8. t6 := const 5
9. if t5 = t6 goto _____
10. goto _____

11. t7 := const 7
12. t8 := addr x
13. *t8 := t7
14. goto _18_
15. t9 := const 2
16. t10 := addr y
17. *t10 := t9
18. ...
```