Uses of Boolean Expressions

Used to drive conditional execution of program sections.

Examples:

\[
\begin{align*}
  &\text{IF } (a < 17) \text{ OR } (b = 12) \text{ THEN} \\
  &\quad \ldots \\
  &\quad \text{else} \\
  &\quad \ldots \\
  &\text{WHILE NOT } ((x+1) > 39) \text{ DO} \\
  &\quad \ldots \\
  &\text{end}
\end{align*}
\]

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

Example:

\[
\begin{align*}
  &\text{VAR } b : \text{BOOLEAN} := (a < 17) \text{ OR } (b = 12); \\
  &\quad \ldots \\
  &\text{IF } b \text{ THEN } \ldots \text{ ELSE } \ldots \\
  &\quad \ldots \\
  &\text{myproc}(b); \quad (* \text{ procedure call } *) \\
  &\quad \ldots 
\end{align*}
\]
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

\[ B := E_1 \ ' '<' E_2 \]
\[ B.\text{place} = \text{newtemp}() \]
\[ B.\text{code} = \]
\[ \text{let } \text{true} = \text{newlabel}() \]
\[ \text{after} = \text{newlabel}() \]
\[ \text{in } E_1.\text{code} @ \]
\[ E_2.\text{code} @ \]
\[ [\text{gen(true,if<,E_1.\text{place},E_2.\text{place})}, \]
\[ \text{gen(B.\text{place},:=,0,_)}, \]
\[ \text{gen(after,goto,_,_}), \]
\[ \text{gen(true,:,_,_)}, \]
\[ \text{gen(B.\text{place},:=,1,_)}, \]
\[ \text{gen(after,:,_,_)}] \]

Generates:

\[
\text{IF } E_1 < E_2 \text{ GOTO L1} \\
T := 0 \\
\text{GOTO L2} \\
L1: \ T := 1 \\
L2: \ ... 
\]
More Sample Value-based Productions

\[ B := B_1 \text{ OR } B_2 \]
\[ B.\text{place} = \text{newtemp()} \]
\[ B.\text{code} = B_1.\text{code} @ B_2.\text{code} @ \]
\[ \text{[gen}(B.\text{place},|, B_1.\text{place},B_2.\text{place})]\]

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

\[ S := \text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \]
\[ S.\text{code} = \text{let } \text{false} = \text{newlabel()} \]
\[ \text{after} = \text{newlabel()} \]
\[ \text{in } B.\text{code} @ \]
\[ \text{[gen}(\text{false},\text{if}=,B.\text{place},0)] @ S_1.\text{code} @ \]
\[ \text{[gen}(\text{after},\text{goto},_,_)\] @ \]
\[ \text{[gen}(\text{false},:,,_)] @ \]
\[ S_2.\text{code} @ \]
\[ \text{[gen}(\text{after},:,,_)]\]

Generates:

\[ \text{IF } B = 0 \text{ GOTO L1} \]
\[ S_1 \]
\[ \text{GOTO L2} \]
\[ \text{L1: S2} \]
\[ \text{L2: ...} \]
Example Value-based Code

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

  t1 := addr a
  t2 := *t1
  t3 := const 7
  if t2 > t3 goto L1
  t4 := const 0
  goto L2
L1:
  t4 := const 1
L2:
  t5 := addr b
  t6 := *t5
  t7 := const 5
  if t6 = t7 goto L3
  t8 := const 0
  goto L4
L3:
  t8 := const 1
L4:
  t9 := t4 | t8
  if t9 = 0 goto L5
  t10 := const 7
  t11 := addr x
  *t11 := t1 0
  goto L6
L5:
  t12 := const 2
  t13 := addr y
  *t13 := t12
L6:
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 (assuming short-circuiting)

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

```
t1 := addr a
t2 := *t1
t3 := const 7
if t2 > t3 goto L1
goto L4
L4:
  t4 := addr b
t5 := *t4
t6 := const 5
if t5 = t6 goto L1
goto L2
L1:
  t7 := const 7
t8 := addr x
  *t8 := t7
goto L3
L2:
  t9 := const 2
t10 := addr y
  *t10 := t9
L3:
```
Conditional Statements (Somewhat Naive Approach)

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

\[
S := \text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2
\]

\[
B.\text{true} = \text{newlabel}();
\]

\[
B.\text{false} = \text{newlabel}();
\]

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

Generates:

\[
\text{IF } B \text{ GOTO L1}
\]

\[
\text{GOTO L2}
\]

\[
\text{L1: S1}
\]

\[
\text{GOTO L3}
\]

\[
\text{L2: S2}
\]

\[
\text{L3:}
\]
Relational Expressions

Inherit true and false label attributes.

Synthesize code to perform appropriate test and jump to appropriate label.

Code doesn’t build a value, so no place attribute.

\[
B := E1 \ '==' \ E2
\]
\[
B.code =
E1.code @
E2.code @
[gen(B.true,if=,E1.place,E2.place),
gen(B.false,goto,_,_)]
\]

\[
B := E1 \ '<' \ E2
\]
\[
B.code =
E1.code @
E2.code @
[gen(B.true,if<,E1.place,E2.place),
gen(B.false,goto,_,_)]
\]

...
Boolean Expressions

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

Again, no `place` attribute.

\[
\begin{align*}
B &:= B_1 \text{ OR } B_2 \\
&\quad \begin{align*}
B_1.true &= B.true \\
B_1.false &= \text{newlabel()} \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \ @ \\
&\quad \begin{align*}
&[\text{gen}(B_1.false,:,_,_) \ ] @ \\
B_2.code
\end{align*}
\end{align*}
\]

\[
\begin{align*}
B &:= B_1 \text{ AND } B_2 \\
&\quad \begin{align*}
B_1.true &= \text{newlabel()} \\
B_1.false &= B.false \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \ @ \\
&\quad \begin{align*}
&[\text{gen}(B_1.true,:,_,_) \ ] @ \\
B_2.code
\end{align*}
\end{align*}
\]

\[
B := \text{NOT } B_1 \\
B_1.true = B.false \\
B_1.false = B.true \\
B.code = B_1.code
\]
Conversions to and from value form

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

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

(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.

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

\[
S := \text{BREAK} \\
\quad S.\text{code} = \text{gen}(S.\text{break, goto, _, _})
\]

\[
S := \text{LOOP } S \text{ END} \\
\quad S.\text{break} = \text{newlabel}(); \\
\quad S.\text{code} = \\
\quad \quad \text{let top} = \text{newlabel}() \\
\quad \quad \text{in } \left[\text{gen}(\text{top, :, _, _})\right] \circ \ \\
\quad \quad \quad S.\text{code} \circ \ \\
\quad \quad \quad \left[\text{gen}(\text{top, goto, _, _}), \right. \ \\
\quad \quad \quad \quad \left. \text{gen}(S.\text{break, :, _, _})\right]
\]

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.

\[ S := S_1 ';' S_2 \]
\[ S.code = S_1.code \oplus S_2.code \]

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

\[
\text{WHILE } B_1 \text{ DO (WHILE } B_2 \text{ DO } S)\]

generates

\[
\begin{align*}
L_1: & \quad \text{IF } B_1 \text{ GOTO } L_2 \\
L_1: & \quad \text{GOTO } L_3 \\
L_2: & \quad \text{IF } B_2 \text{ GOTO } L_4 \\
L_2: & \quad \text{GOTO } L_5 \quad \text{“jump to jump”} \\
L_4: & \quad S \\
L_4: & \quad \text{GOTO } L_2 \\
L_5: & \quad \text{GOTO } L_1 \\
L_3: &
\end{align*}
\]

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.”

$$S := S_1 \ ';$' \ S_2$$

\[
S_1.next = \text{newlabel()}
\]

\[
S_2.next = S.next
\]

\[
S.code = S_1.code @
\]

\[
[\text{gen}(S_1.next,:,:,\_)] @
\]

\[
S_2.code
\]

$$S := \text{WHILE} \ B \ \text{DO} \ S_1$$

\[
B.true = \text{newlabel()}
\]

\[
B.false = S.next
\]

\[
S_1.next = \text{newlabel()}
\]

\[
S.code = [\text{gen}(S_1.next,:,:,\_)] @
\]

\[
B.code @
\]

\[
[\text{gen}(B.true,:,:,\_)] @
\]

\[
S_1.code @
\]

\[
[\text{gen}(S_1.next,\text{goto},\_,\_)]
\]

and similarly for other compound statements.
Deferred Label Definition (continued)

Now get better code, e.g.

\[
\text{WHILE } B1 \text{ DO (WHILE } B2 \text{ DO } S) \]

now generates

\[
\begin{align*}
L1: & \quad \text{IF } B1 \text{ GOTO } L2 \\
& \quad \text{GOTO } L? \\
L2: & \quad \text{IF } B2 \text{ GOTO } L3 \\
& \quad \text{GOTO } L1 \\
L3: & \quad S \\
& \quad \text{GOTO } L2 \\
\ldots \\
L?: &
\end{align*}
\]
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): \((a > 7) \text{ OR } (b = 5)\)

1. \(t_1 := \text{addr } a\)
2. \(t_2 := *t_1\)
3. \(t_3 := \text{const 7}\)
4. if \(t_2 > t_3\) goto _____
5. goto _____

6. \(t_4 := \text{addr } b\)
7. \(t_5 := *t_4\)
8. \(t_6 := \text{const 5}\)
9. if \(t_5 = t_6\) goto _____
10. goto _____

At reduction for \(B := B_1 \text{ OR } B_2\)

- Backpatch \(B_1.\text{false}\) list with address of first instruction in \(B_2\).
- Merge \(B_1.\text{true}\) and \(B_2.\text{true}\) to form \(B.\text{true}\).
- Make \(B_2.\text{false}\) into \(B.\text{false}\).
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__
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. ...
Case Statements

case e of
  v₁ : s₁
  | v₂ : s₂
  | ...  
  | vₙ : sₙ
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.