Filtering for the case Constraint

Mats Carlsson

Swedish Institute of Computer Science (SICS)

14 juni 2006
The case constraint expresses an $n$-ary relation in a compact form which admits a fast filtering algorithm. The relation is expressed as a DAG.

**Formulation 1**
\[
\text{table}([[X,Y,Z]], \\
[[1,1,10], [2,1,10], [3,1,20], [4,1,20], \\
[5,2,10], [6,2,10], [7,2,30], [8,2,30]]).
\]

**Formulation 2**
\[
\text{element}(X, [1,1,1,2,2,2,2], Y), \\
\text{element}(X, [10,10,20,20,10,10,30,30], Z).
\]
The case constraint
Semantics

Formulation 3

case(\text{f}(A,B,C), [\text{f}(X,Y,Z)],
    [\text{node}(0, A, [(1..2)-1, (3..4)-2, (5..6)-3, (7..8)-4]),
     \text{node}(1, B, [(1..1)-5]),
     \text{node}(2, B, [(1..1)-6]),
     \text{node}(3, B, [(2..2)-5]),
     \text{node}(4, B, [(2..2)-7]),
     \text{node}(5, C, [(10..10)]),
     \text{node}(6, C, [(20..20)]),
     \text{node}(7, C, [(30..30)]))

\[\text{A} \rightarrow \begin{array}{c}3..4 \downarrow 1..2 \downarrow 5..6 \downarrow 7..8 \\
\text{B} \rightarrow \begin{array}{c}1..1 \downarrow 1..1 \downarrow 2..2 \\
\text{C=20} \rightarrow \begin{array}{c}
\text{C=10} \rightarrow \begin{array}{c}
\text{C=30}
\end{array}
\end{array}
\end{array}\]
Data structures

The DAG is represented by *nodes* and *children*. Children are ordered by increasing min:
The main filtering procedure

We have nodes $1, \ldots, m$ and variables $1, \ldots, n$. $P_i$ are
the values for $x_i$ that do not yet have support.

PROCEDURE case()
1:   for $i \leftarrow 1$ to $m$ do
2:       node[$i$].val $\leftarrow$ clear
3:   for $i \leftarrow 1$ to $n$ do
4:       $P_i \leftarrow \text{dom}(x_i)$
5: if $\neg \text{eval}(\text{root})$ then
6:       return fail
7:   for $i \leftarrow 1$ to $n$ do
8:       $\text{dom}(x_i) \leftarrow \text{dom}(x_i) \setminus P_i$
9: if all $x_i$ are ground then
10:      return succeed
11: else
12:      return delay
The recursive procedure

PROCEDURE eval(i)
1: if $i = \text{leaf}$ then
2: return true \{base case\}
3: else if node[i].val $\neq \text{clear}$ then
4: return node[i].val \{node already visited\}
5: $j \leftarrow$ node[i].var
6: $s \leftarrow$ false
7: for $c \leftarrow$ node[i].child to node[i].childend − 1 do
8: $l \leftarrow$ [child[c].min, child[c].max]
9: if dom($x_j$) $\cap$ $l \neq \emptyset$ $\land$ eval(child[c].node) then
10: $P_j \leftarrow P_j \setminus l$ \{found some support\}
11: $s \leftarrow$ true
12: node[i].val $\leftarrow$ $s$
13: return $s$
Some optimizations

- **Structure sharing.** The DAG encodes an $n$-ary relation. Multiple tuples can share one DAG.
- **Incrementality.** Up to backtracking, *false* nodes remain *false*.
- **Using support.** The **for** loop can be exited as soon as all values of the relevant variables have support.
  - The **for** loop can be exited as soon as $\text{child}[c]. \min > \max(x_j)$.
  - The first feasible $c$ in the **for** loop can be found in $O(\log k)$ time, for $k$ children.
- $m$ nodes can be cleared in $O(1)$ time using **timestamps**.
- The SICStus propagator allows to choose between AC and BC, and more.