Implementing the

Skolem Abstract Machine

John R. Fisher
jrfisher@cpp.edu

last update = January28, 2020

This report describes an *open* implementation design for a *Skolem Abstract Machine*. This report was started in February 2012. Since then there have been various changes or refinements to the design and corresponding implementations to test the design changes. The last modification/correction date for this report is posted above. The implementation design patterns are as follows:

1 Branch indexing

2 QDF search algorithms

3 QDF Fairness

4 Proof Relevance

5 Distributive rule choices

6 Tabular indexing

7 Consequent forwarding

8 Complexity cut mechanism

9 Projects

The machine code for a Skolem Machine is a *colog theory*. A colog theory is a finite sequence colog rules. See the *colog primer*[1] for various examples of colog theories. Before studying this implementation design report, the reader is advised to read the colog primer[1] and run the examples from the primer using the GUI version of the colog14I prover[2].

1 Branch indexing

A coherent logic theory is a finite ordered sequence of colog rules. This section describes how colog rules are used to expand a search branch for a Skolem Machine implementing the colog theory. We describe structures to hold facts on the branch and a bindings array for each rule.

To review the basic concepts, let us reconsider a simple example from the introduction of [4]. The following *colog* theory has six coherent logic rules. The free variables in the consequent of rule #1 and one of the disjuncts of rule #2 are existential variables. When such a rule is satisfied and fires it introduces a new Skolem constant as value for the existential variable. The | in the consequent of rule #2 is disjunction; conjunctions of atoms are separated with commas.

```
// Example 1.1
true => domain(X), p(X).                      % #1
p(X) => q(X) | r(X) | domain(Y), s(X,Y).  % #2
domain(X) => u(X).                            % #3
u(X), q(X) => false.                          % #4
r(X) => goal.                                 % #5
s(X,Y) => goal.                               % #6
```

Figure 1 shows a QDF proof for the little theory above. The LaTeX source code for search tree pictures were automatically generated by a QDF colog prover.

A colog prover using a QDF search algorithm (Section 2) first saturates the leftmost branch and then backtracks to the branch point (3), saturates the second branch, backtracks to the branch point (3) again, and then saturates the final branch.

Serial implementations use just one active branch. When the machine backtracks, it then reuses the portion of the branch below the branch point and writes over the previous facts left over from the last branch probe. Branches are arrays of facts (*Fact* objects). A depiction of the pictured tree using a single branch might look like this:

```
  0        1          2       3       4             5           6      7
 [true,domian(sk0),p(ski),u(sk0),q(ski),     false                       ...]
 [                                ,r(sk0),      goal                        ...]
 [                                ,domain(sk1),s(sk0,sk1),u(sk1),goal,  ...]
```

The functors of a colog theory (predicates and functions) are themselves indexed by name/arity. For the current theory the predicate indices are

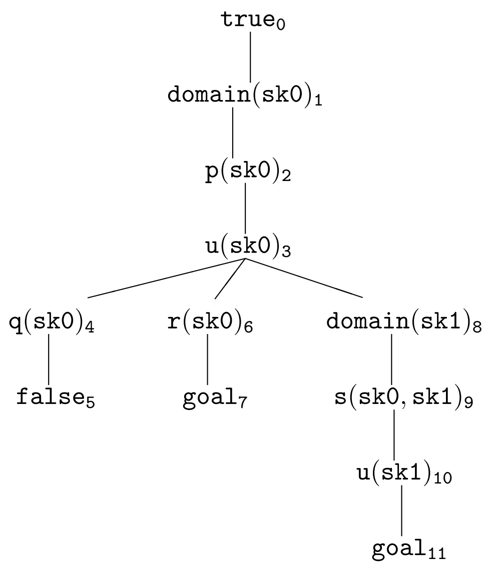


Figure 1: Proof Tree, Example 1.1

```
0 : true
1 : goal
2 : false
3 : domain
4 : p
5 : q
6 : r
7 : s
8 : u
```

The positions of current facts on the branches is managed by three arrays

`first`[−] of size equal to number of predicate names

`last`[−] of size equal to number of predicate names

`jumpto`[−] of size equal to size of branch

These arrays are updated by rule applications as facts are added to the branch, and the arrays are appropriately reset when control returns to a branch point. In particular, reconsider the third branch B of the tree above at the time that it saturates:

```
  0        1          2       3          4           5      6     7
[true,domain(sk0),p(ski),u(sk0),domain(sk1),s(sk0,sk1),u(sk1),goal,...]
```

At this time we have

```
first  = [0,7,*,1,2,*,*,5,3]
last   = [0,7,*,6,2,*,*,5,3]
jumpto[0]=*
jumpto[1]=4
jumpto[2]=*
jumpto[3]=6
jumpto[4]=*
jumpto[5]=*
jumpto[6]=*
jumpto[7]=*  ...
```

The * indicates a sentinal value (say the size of the branch) indicating that there is *no* relevant value. So, for a particular example, the first occurrence of '`domain/1`' (index=3) is located at position first[3]=1 on B, and the next is located at position jumpto[first[3]]=jumpto[1]=4=last[1], and so there are no more to jump to. This illustrates the basic calculation mechanism for predicate position indexing on B.

This basic indexing is enhanced by adding other position information (distributive choices, tabular indexing) to ensure fairness of antecedent matching, avoid resatisfaction of antecedent factors, and speed.

A rule is *applicable* at the focus point on the branch provided that its antecedent conjunction of factors matches facts at or above the focus point and that its consequent (disjunction of conjunctive factors) is not already satisfied using facts at or above the focus point.

Active rules are objects compiled from colog rules. Part of the dynamic data of an active rule is a binding table that is used when attempts are made to match rule factors (antecedent or consequent) against branch facts. The binding table records variable matchings in a factor by remembering which ground term is bound to a variable.

To illustrate how a binding table works consider another small colog theory:

```
// Example 1.2
true => p(a), p(b).
p(X), p(Y) => q(X,Y,Z), p(Z).  // existential rule
q(b,A,B) => goal.
```

Referring to search tree in Figure 2 at branch index 6, the second rule can be satisfied and produce the inference whichs asserts the facts at positions 7 and 8.

A *variable binding table* is a matrix of ground functors indexed on the left by variables (themselves being indexed using nonnegative integers) and indexed across the top by nonnegative integers representing the predicate factors in a rule.

To illustrate, consider the second rule of Example 1.2. This rule has three variables, 2 antecedent factors (index 0 and 1) and two consequent factors (index 2 and 3). The bindings in the table are shown in equation 1.

factor	0	1	2	3
X/0	*b*	-	-	-
Y/1	-	*a*	-	-
Z/2	-	-	sk2	-
choice	2	1		

(1)

The inference in question (and the factors) is

```
 0       1           2            3
p(b), p(a) => q(b,a,sk2), p(sk2).
```

The table shows what the antecedent bindings were matching the antecedent, and the branch position choice that produced that match. The consequent was not already satisfied for those choices, so a binding of `sk2` was created for `Z` in the third factor, and these bindings were used to generate the facts asserted at branch positions 7 and 8.

These details do not explain *why* the particular choices for matching on the branch were made. That involves a distributed choice mechanism explained in Section 5. The next Section 2 explains why a fair choice mechanism is needed, of which the distributive choice mechanism is an excellent specification.

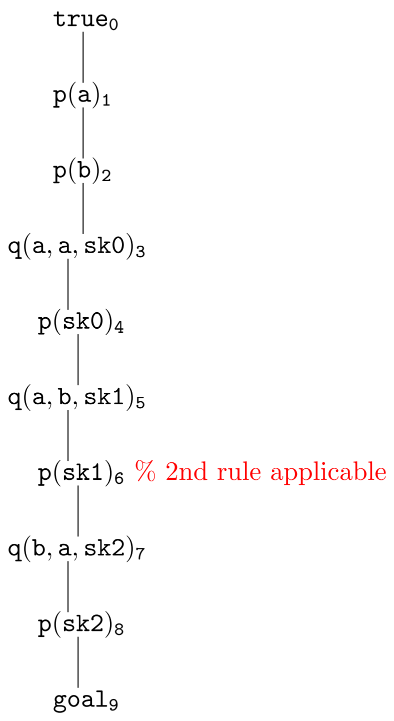


Figure 2: Example 1.2

2 QDF search algorithms

There is an interesting family of depth-first proof search algorithms for a Skolem Machine associated with colog theory. These *QDF* algorithms all use a queueing mechanism to delay the activation of rules that would generate new constants or functions (like rule #1 and #2 in the previous section). Q rules are those rules that the compiler adds to the queue. Other rules are non-Q, or sometimes called *definite* rules. (A smarter compiler can also put complex rules in the queue, such as those that introduce new functions in the antecedent – a detail not discussed here.)

Pseudo-code for QDF search might look something like this . . .

```
start tree with *true* root
until the tree is saturated
    expand the branch {
        D: use all applicable definite rules in order
        until D-saturation and then
        Q: dequeue (and add) until there is an applicable Q rule
        extend branch using the first consequent
        store list of remaining consequents at this branch point
    } until the branch is saturated (*goal* or $false$ leaf or stuck)
    go up to the first relevant branch point (backtrack)
    extend branch using first of consequent list at branch point
```

QDF search algorithm template (2)

At `D`, repeatedly apply definite rules until none apply (saturate branch). At `Q`, dequeue the first Q rule and requeue that rule until a rule applies (or no rule applies). This is a round-robin attempt to find one applicable Q rule. The algorithm generates a counter model when a branch saturates with no goal or false, but of course the algorithm might not terminate if there is an infinite branch.

The role of the rule queue Q is to delay the use of rules that produce new functions (including constants), and allow definite rules to saturate a branch before introducing new functions. The *good effect* of this approach is illustrated in the examples from [1]. (There may be other effective approaches.)

The primary task for ensuring logically complete implementations involves methods that ensure that the application of rules is *fair*, which we illustrate in the next Section 3.

3 QDF Fairness

A QDF search algorithm is *fair* provided that the methods it uses to apply rules does not *hide* any possible antecedent matching choice. A *hidden* matching choice is one which never occurs no matter how long the search algorithm processes rule applications on a branch.

We can illustrate this by reconsidering Example 1.2 from Section 1.

```
// Example 1.2
true => p(a), p(b).
p(X), p(Y) => q(X,Y,Z), p(Z).  // existential rule
q(b,A,B) => goal.
```

The relevant issue (for this particular example) is *how* the choices are determined for matching the antecedent of the second rule.

Let us first investigate an unfair regimen for applying the second rule. Suppose that the branch expands as follows . . .

```
true
p(a)
p(b)
q(a,a,sk0)
p(sk0)
q(a,b,sk1)
p(sk1)
q(a,sk0,sk2)
p(sk2)
q(a,sk1,sk13)
p(sk3)
q(a,sk2,sk4)
p(sk4)
q(a,sk3,sk4)
p(sk4)
... forever
```

The second rule is being applied to each new possible match binding for `Y`, ignoring the fact that `X` might make new matches. This resembles a control order similar to

```
for all X
    for all Y
        ....
```

where the inner loop on `Y` is continues before the outer `X` loop is revisited for new values. The relevant issue is that `X` never *sees* `b` because `Y` continues to have new matches.

This is not the only manner in which unfairness could arise using an antecedent choice mechanism. It is only a simple illustration.

A general solution requires a mechanism for achieving all matches before overall advancement. The advantage to the distributive choices mechanism in Section 5 is that that mechanism also avoids previous choices that would already have been satisfied.

4 Proof Relevance

For a proof branch (`goal` or `false` leaf) a *relevant* branch point is one representing a splitting rule that was actually used to infer the tip of the branch. Here is a simple example having an irrelevant branching point.

```
// Example 4.1
true => p.
p => q | r | s.
p => goal.
```

A proof tree is

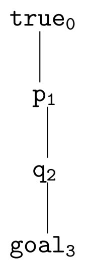


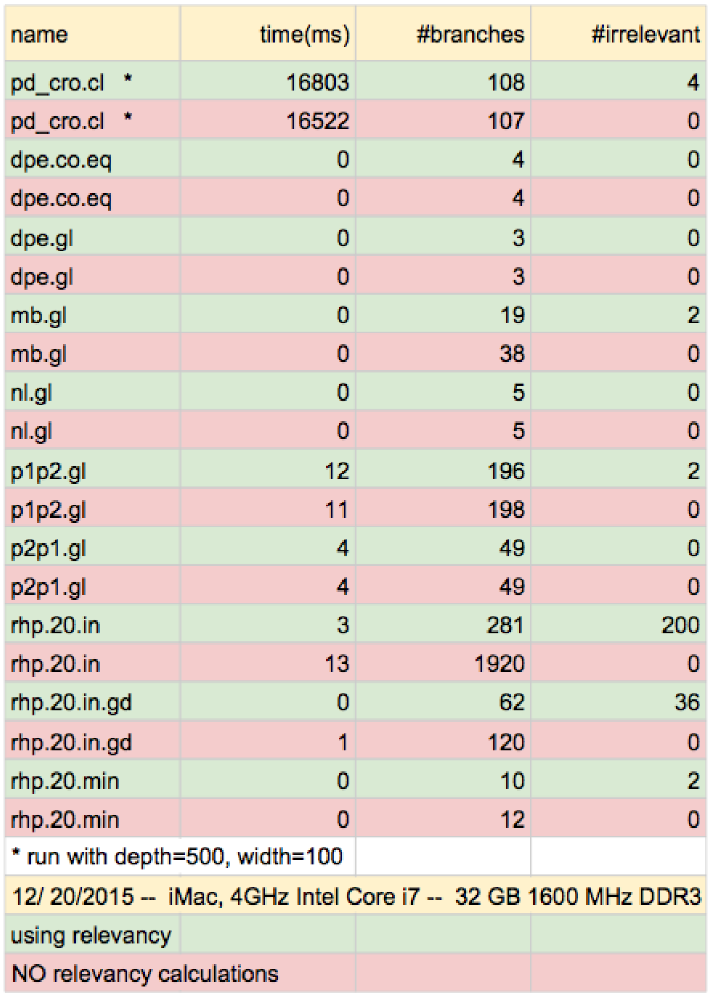
Figure 3: sample tree

and the extracted proof is

```
@0, rule1: true => p
@2, rule3: p => goal
```

The inference at branch point 1, `p` => `q` | `r` | `s` is unneeded for the proof. A simple algorithm for marking used inferences is used to determine relevant branching points and those that are not relevant are ignored. A more precise specificication of an algorithm is left as an exercise for the reader (but see also Example 4.2 and 4.3 below).

Fig. 4 gives some relevancy data for proofs of several theories. In particular, note that the theory `rhp.20.in` (provided by A. Polonsky) has many irrelevant branches, whereas other similar versions – `rhp.20.gd` (type guards automatically added) and `rhp.20.min` (even more type guards, by hand) – encounter many fewer irrelevant branches. Also, the first theory `pd_cro.cl` (M. Bezem and D. Hendriks) has scant irrelevancy, but the run times with and without the relevancy check shows that the checking does not take inordinate time. Conversely, for such a theory, *not* checking does not produce abundant irrelevancy.



name	time(ms)	#branches	#irrelevant
pd_cro.cl *	16803	108	4
pd_cro.cl *	16522	107	0
dpe.co.eq	0	4	0
dpe.co.eq	0	4	0
dpe.gl	0	3	0
dpe.gl	0	3	0
mb.gl	0	19	2
mb.gl	0	38	0
nl.gl	0	5	0
nl.gl	0	5	0
p1p2.gl	12	196	2
p1p2.gl	11	198	0
p2p1.gl	4	49	0
p2p1.gl	4	49	0
rhp.20.in	3	281	200
rhp.20.in	13	1920	0
rhp.20.in.gd	0	62	36
rhp.20.in.gd	1	120	0
rhp.20.min	0	10	2
rhp.20.min	0	12	0
* run with depth=500, width=100			
12/ 20/2015 -- iMac, 4GHz Intel Core i7 -- 32 GB 1600 MHz DDR3			
🟩 using relevancy			
🟥 NO relevancy calculations			

Figure 4: Some Relevancy Data

Computing relevancy proceeds bottom-up from the leaf of a branch, marking the branch positions where inferences were used to produce a fact, and then recursively doing the same for the antecedents of the facts which constitute the antecedent of the rule application, up to root. The choice points on the branch which are marked as used are then the branchings that need to be expanded, and ones not so marked can be ignored. With appropriate data structures for the relevant data, the algorithm is quite fast. But it is important to emphasize that the relevance concept requires that any branch below a choice point might contribute to the choice being relevant (and not just the first completed or leftmost branch). Here is a simple example to illustrate this proof phenomenon.

```
// Example 4.2
true => q | r.
true => a | b.
a => goal.
b, q => goal.   \% delayed relevance for 1st rule
r => goal.
```

Consider the proof tree

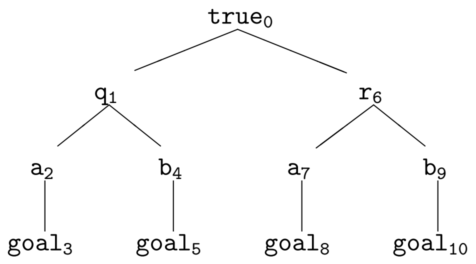


Figure 5: proof tree for Example 4.2

The inference `true=>q|r` at node `0` becomes relevant due to the inference `q,b=>goal` at node `4` because `q` at node `1` is required in the antecedent of the latter rule. Thus it is that the second branch determines the relevancy of the choice at node `0` and not the first branch.

Now, let us modify the example slightly, as follows:

```
// Example 4.3
true => q | r.
true => a | b.
a => goal.
b => goal.   \% q no longer needed, compare Example 4.2
r => goal.
```

The 4th rule now does not need `q` in its antecedent, so the choice at node `0` is no longer relevant, and not expanded in the proof tree.

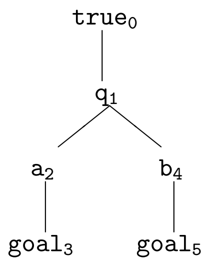


Figure 6: proof tree for Example 4.3

5 Distributive Rule Choices

Distributive rule choices or DC for short is an algorithm for selecting facts on the current branch which satisfy the antecedent of a coherent logic rule. We motivate the algorithm by descriptions first for rules with two antecedent factors in the rule, then three, etc. Each case, dependent on the number of antecedent factors, gives rise to a finite state machine that marshals the matching choices for the rule.

An effective implementation then compiles colog rules into codes that correctly compute the finite-state machines.

Fig. 7 depicts choice ranges for a rule with two antecedents. In the figure the ranges for choices for p are labeled A and C, and the ranges for q are labeled B and D. The C and D ranges depict *extended* ranges. An expression like AB stands for all choice pairs $[i, j]$ where i falls in range A and j falls in range B. The iteration of choice within ranges is enabled using the indexing arrays `first[ ]`, `jumpto[ ]`, and `last[ ]` for the predicates (from Section 1).

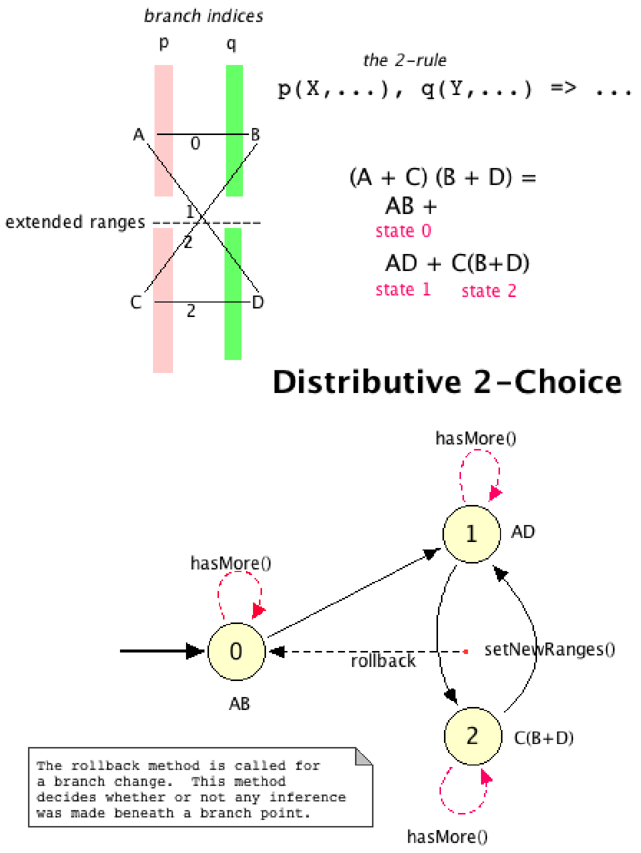
The terminology *distributive ranges* draws on an analogy with the distributive law formula.

$$(A + C)(B + D) = AB + AD + C(B + D) \tag{3}$$

where the range (A+C) signifies A together with C, for example. The formula can be read to mean that, after considering the initial choice range AB, and then extending with C snd D, compute the new choices in a manner indicated by the expressions AD and then $C(B+D)$. Notice that the ranges AB, AD, and $C(B+D)$ are mutually exclusive, and thus the initial or prior range AB is not revisited when considering the extended ranges, and neither does either extended range AD or $C(B+D)$ impinge on the other.

Conventional backtracking algorithms easily apply to give complete iteration of all possible choices within the subranges AB, AD and C(B+D). *If* new range choices *were* dynamically allowed – rather than delayed for later DC state – conventional backtracking algorithms fail to be *fair*: the last item for iteration blocks later items for earlier choices as explained in Section 3. The distributive choices approach, together with saturation of ranges within a DC state, corrects this fairness problem and it also speeds up new inferences because it obviates the need for reconsideration of previous choice ranges. Also, in-state saturation of ranges can easily be designed to not repeat those specific choices.

The 2-distribution formula applies similarly for each subsequent exten-



The rollback method is called for a branch change. This method decides whether or not any inference was made beneath a branch point.

Figure 7: Distributive 2-Choices

sion to the ranges, after the prior range choices are exhausted. The distribution formula is the basis for a finite state machine also depicted in Fig.7. The start state is state 0 (range AB). After depleting choices in the range of state 0, transition to state 1 (range AD). After depleting choices in the range of state 2, transition to state 2 (range $C(B+D)$). The states 1 and 2 then constitute a super-cycle for additional range extensions. One can demonstrate that a subsequent extension E and F distributes in the same 2-pattern as that already considered: $(A+C+E)(B+D+F) = (A+C)(B+D)+\ldots$. Thus, the distributive choice algorithm is dynamically stable as choice ranges extend while the search branch extends.

The 1+2 state machine thus becomes the basis for the compilation of a

2-rule into a process that applies the rule choices on a branch in accordance with the range choices determined by the state machine. The *rollback* transition corresponds to the control adjustments that need to be made when a delayed branch is considered. This depends upon whether the choices are serially considered (rollback must check ranges on the previous branch) or concurrently (rollback does nothing but a new branch is spawned).

For further motivation to establish the general DC pattern, consider the case of a 3-rule, a rule having three antecedent factors. The state-machine picture is depicted in Fig. 8.

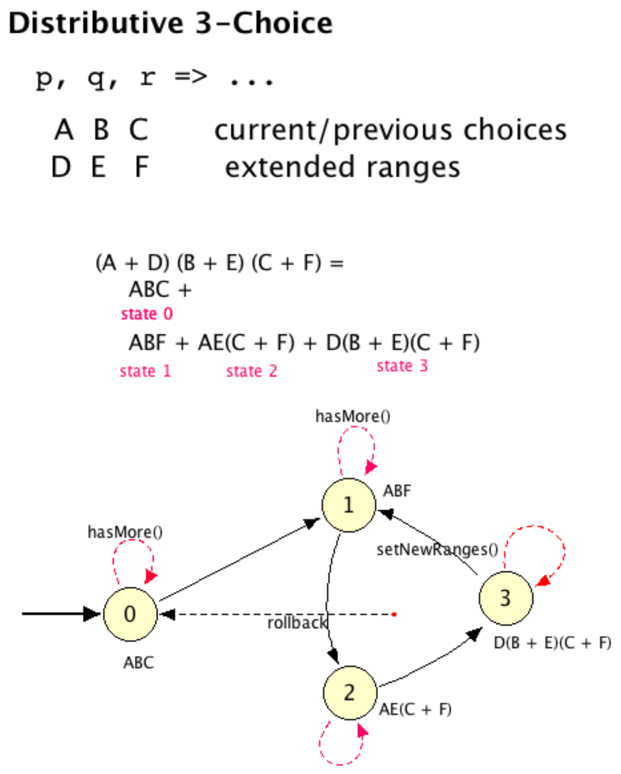


Figure 8: Distributive 3-Choices

In this case, the corresponding state machine has 1+3 states, and a 3-rule is compiled to a process that applies the rule choices on a branch in accordance with the range choices determined by the state machine.

Four a colog rule having 4 antecedent choices, consider the picture in Fig. 9. The red bars indicate active range choices.

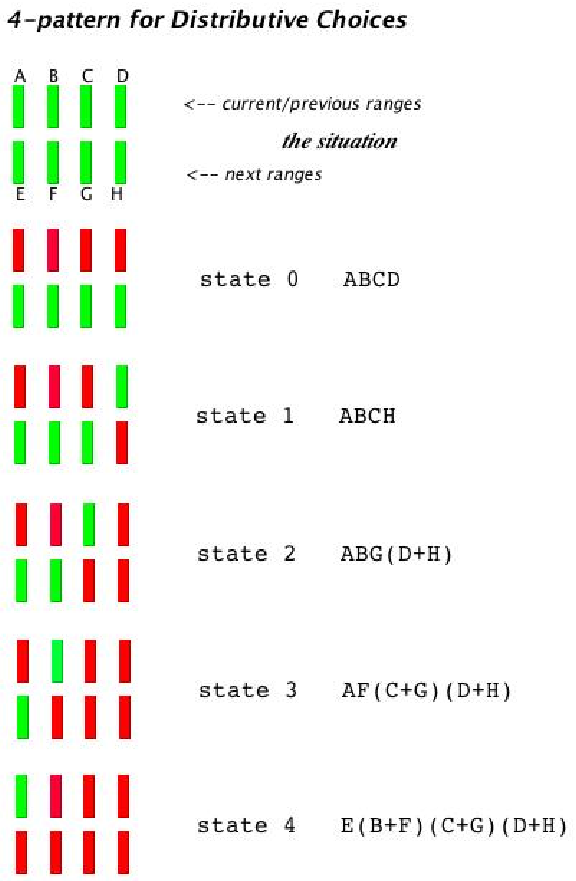


Figure 9: Distributive 4-Choices

The general problem is to consider a colog rule having N factors in its antecedent. The QDF algorithm is modified so that the rule applications are restricted to a current range-choice state until all of those choices have been tried, and then advance to the next state. When all states have ben visited, the choice ranges are enlarged in state 0 to include new branch facts that rules have asserted to the branch since the previous visit to state 0.

A distributed choice rule is inherently *fair*: Given *any* possible antecedent choice on a branch, there is some distributed range that includes the given choice. However, distributed choices do not necessarily produce *earliest-first* choices.

An interesting statistic for measuring the efficiency of a colog implementation is a *ratio of effective inferences*, defined as

$$\rho = \#\texttt{effective inferences} \;/\; \#\texttt{attempted inferences} \tag{4}$$

An *attempted* inference occurs when the rule choices do match the antecedent of a rule. An *effective* inference occurs only when the consequence is new (not subsumed on a branch by previous facts), in which case the inference adds new facts to the branch. If no rule ever revisits previous choices (as with DC) then ρ measures some kind of maximal effectiveness, which is a property of the colog theory itself. Keep in mind that a rule can easily produce repeated consequences for many different choices of antecedent factors, so we would seldom have $\rho = 1$.

5.1 branch switching

Suppose that R is an active rule of the colog theory. A critical computation is one that determines which choice ranges for R are safe to use *after* a branch B is saturated and the appropriate new branch B' is explored at a branch point p. R may have asserted facts to the old branch B below the branch point p, and so those choices are not relevant to B'. R may have used some unsuccessful choices on B which lie below p, and so those choices are again not relevant to B'. In either case, the choice ranges need to be adjusted.

A completely safe strategy is to restart the DC choice ranges after a branch switch, for all active rules. The essential problem with this conservative strategy is that many of the DC choice-range computations (and any successful inferences) may be above the branch point. Deciding how to *rollback* the choice ranges is a challenging problem.

6 Tabular indexing

When the colog reader reads a colog rule it analyses the rule by looking at all of the factors in the antecedent and consequent in left-to-right order. In order to fix ideas let us work with a specific example of a rule.

The idea here is to compute the patterns of bound variables in rule factors as they are encountered left-to-right. These are called *key patterns*. These key patterns can be computed using static analysis of the input theory when it is loaded. Here is a concrete example.

```
   0      1       2       3           0       1       0
p(X), q(Y), r(X,W), r(W,Y)  =>  s(Z), t(X,Z) | w(X,Z).
 [f]    [f]   [t,f]   [t,t]        [f]   [t,t]   [t,t]
```

where t=true and f=false. So for example, the the 2-factor of the antecedent r(W,Y) has X bound and W unbound when an attempt is made by the rule to match this factor. That is, the variable binding table will have a value stored for X but not for W at that time, as outline in the discussion around Equation 1 in Section 1 (using, of course, a table appropriate for the present rule). The key patterns for a factor are stored in the factors as boolean arrays.

Note that the key patterns can also be used to compute a match order for factors. Some efficiency is gained by attempting to match bound variables first, in order to allow fast-fail and also so as to limit the possibilities for branch facts to actually match the factor.

A literal factor is said to be *factual* provided that all of its variables are bound in the rule binding table at match time. A factor is said to be *subfactual* provided that at least one, but not all, of its variables are bound in the rule binding table at match time. If a factor is neither factual nor sub factual, then it is a *free factor*.

For colog14I, if every antecedent literal factor of a rule is a free factor, then an attempt to satisfy the antecedent of the rule uses the basic branch indexing (distributive choices with `jumpto`[ ] as explained in Section 1). Otherwise, the branch fact hash table is employed to satisfy all of the antecendent literal factors of the rule. Checking to see if a consequent conjunction is already satisfied uses either the fact hash table for bound consequent factors or a stepwise approach for existential factors. [1]

[1] A more sophisticated *future* approach would be to use plain DC for a free antecedent factor and employ the fact hash table for a factual or subfactual antecedent factor on a factor-by-factor basis rather than for the entire rule antecedent. Similarly for ”new” consequent checking.

The *branch fact table* is the essential device for tabular indexing. The table is a custom hash table which stores branch positions (indices) of facts on the branch which have hash values associated with key pattern for a rule factor.

Given a rule factor `f` (Functor object),

```
bucket[factor] = tree.table.get(keyHashCode(f))
```

returns a bucket list of positions of facts on the current tree branch table which have the same `keyHashCode` as the factor `f`, and then linear search of this list attempts to find an *exact* computed match of the rule factor to some branch fact. This scenario is typical for table probe mechanisms.

Here is the Java source code for the factor `keyHashCode` calculations:

```
protected int keyHashCode(Functor f) {
   int h = f.index ;
   for (int i = 0 ; i < f.arity ; i++)
      h = (h<<5) + (f.key[i] ? hashCode(f.args[i]) : 0) ;
   return h ;
}

/**
  *  @return factual hash value of terms using the current table bindings.
  *  This call is only made for BOUND Terms.
   */
public int hashCode(Term t) {
   int hc = 0 ;
   if (t.isVariable()) {
      int v = ((Variable)t).index ; // which variable?
      for(int c = 0 ; c < C ; c++)
         if (table[v][c] != null)
            hc= Fact.hashcode(table[v][c]) ;
         // N.B. variable must be bound
   }
   else {  //t is a Functor
      hc = ((Functor)t).index ;
      for (int i = 0 ; i < ((Functor)t).arity ; i++) {
         int h = hashCode(((Functor)t).args[i]) ;
         if (h == 0) return 0 ; // unbound arg   ???
         hc = (hc << 5)  + h ;
      }
   }
   return hc ;
}
```

Although the other details may be sketchy, the essential calculation is the ubiquitous *shift-left 5 dictionary hash code* calculation. Experiments show that this hash function is reasonably diverse, but there are probably sharper functions that could be explored. Table entries are further distinguished as to whether entries in the buckets contain a factual or a subfactual entry. The table is populated when facts are asserted to branch and the table is used to retrieve possible matching branch positions when rule factors are matched against branch facts. Possible matches retrieved by hash code must then actually match, and this may cause new rule bindings for later factors. Of course, one needs to make some allowance for when the shift hash code overflows the return type.

It is important to *emphasize* that tableing described above affords an efficiency improvement for the distributive choice algorithm as formally described in section 5 where branch indexing used only predicate names and arity.

One uses a hash table to store branch positions that hold facts that entangle a more restrictive hash value (e.g., name, arity and combined argument hash values). The more restrictive hash functions eliminate failing choices that would otherwise entangle a factor using a more generous hash value (name, arity). So, more restrictive entanglement profiles eliminate many choices that were bound to fail anyway.

We assume that tabeling uses buckets where branch choices are linearly ordered in a way that if choice `a=branch[i]` is retrieved before choice `b=branch[j]` then `i<j`, and vice versa. In this way, an earliest-first inference regimen would be imposed, which makes run-time inference testing more intuitive.

The GUI version of the colog prover allows inspection of the branch fact table (under ”options” menu).

7 Consequent forwarding

As explained before, an active rule matchs antecedent factors in left-to-right order. One approach to triggering a colog rule is to find a matching for all the the antecedent factors *and then* check to see that none of the consequents are already satisfied the current branch.

As a concrete example consider the following rule (borrowed from Marc Bezem's pd_cro.in theory):

```
i(A,L),i(B,L),i(C,L),i(D,M),i(E,M),i(F,M),
i(B,N),i(F,N),i(G,N),i(C,O),i(E,O),i(G,O),
i(B,P),i(D,P),i(H,P),i(A,Q),i(E,Q),i(H,Q),
i(C,R),i(D,R),i(I,R),i(A,S),i(F,S),i(I,S)
=> l(N,O) | l(P,Q) | l(R,S) |   % degeneracy, false cases
   l(T,T),i(G,T),i(H,T),i(I,T). % goal seeking
```

The antecedent of the rule has 24 factors. Spying on this rule during a proof search with a width limit of 132 branches (an unsuccessful attempt) produced the following statistics for subsumed (repeated) consequents:

```
...
=> l(N,O) |                        25,244,561
   l(P,Q) |                         7,373,678
   l(R,S) |                         2,950,061
   l(T,T),i(G,T),i(H,T),i(I,T).      318,0091
```

If we were to check a factual consequent as soon as its variables are bound in the antecedent, and if the consequent is tabled then we do not have to continue with the matching of antecedent factors. For example, if we had checked `l(N,O)` as soon as `N` and `O` became bound in the antecedent, we would have voided many of the 25,244,561 continued matches for the remaining 14 antecedent factors to the right of `i(C,O)`. So the rule, might be compiled so that, in effect, it would look like this:

```
i(A,L),i(B,L),i(C,L),i(D,M),i(E,M),i(F,M),
i(B,N),i(F,N),i(G,N),i(C,O), \$not(l(N,O)), i(E,O),i(G,O),
i(B,P),i(D,P),i(H,P),i(A,Q), \$not(l(P,Q)), i(E,Q),i(H,Q),
i(C,R),i(D,R),i(I,R),i(A,S), \$not(l(R,S)), i(F,S),i(I,S)
=> l(N,O) | l(P,Q) | l(R,S) |    % degeneracy, false cases
   l(T,T),i(G,T),i(H,T),i(I,T).  % goal seeking
```

where the \$not(C) factors suggest that at their point of occurrence, one should check to see that C is *not* tabled: If C is tabled, then backtrack on the matching process up to this point, or otherwise proceed (a kind of *negation-as-failure*). Experiments show that this can effect considerable speedup in the rule application process. This compilation device is called *consequent forwarding*. Notice that code analysis to determine forwarding data requires only *static* analysis of the relevant input rule, and not run-time checking, so that the forwarding is a compiler optimization.

8 Complexity cut mechanism

The complexity cut mechanism is the only heuristic method currently used to limit inferences for colog provers. [2] It can block logical completeness. It can very effectively shorten proof searches, especially for *algebra* theories, or other colog theories with complex occurrences of functions.

The *complexity* of a colog term is the depth of function nesting. The complexity of a colog predicate (rule factor) is the maximum complexity of its arguments – 0 if there are no arguments. The complexity of a conjunction of factors is the maximum complexity of its factors, and the complexity of a disjunctive rule consequent is the maximum complexity of all of it disjuncts (each of which is a conjunction). Finally, a rule is *complex* if the complexity of it consequent is larger than the complexity of its antecedent.

Complex rules are also added to the rule Queue Q, as described in Section 2. Complex rules might add new logical data for which rules would become newly applicable. Queueing complex rules slows down the proliferation of new data until definite rules have a chance to saturate their consequences.

As an example, see Section 6.3 of the colog primer [1] where an abstract algebra example is given. Algebraic operation can introduce abundant operator terms, and complexity limitation can be very effective for achieving proofs in less time.

[2]except search tree depth and width limits, which seem more like resource limitations

9 Projects

These suggestions are now (January 28, 2020) deprecated. The Autolog project (2018 . . .) addresses some of these issues.

9.1 concurrency/parallelism

One possible approach would be to spawn parallel branch searches for all relevant branches above a saturated leaf. A similar experiment was explored in 2009 [3], before much of the serial indexing was implemented (see appendix to the report).

9.2 answer extraction

Section 2 of [4] defines a query associated with a colog theory. Design a query answer mechanism for a Skolem Machine.

9.3 complexity and other heuristics

Refine complexity calculations or invent other heuristics for guiding or restricting Skolem Machine operations.

9.4 computing conflicted theories

See reference [4], Section 7, concerning *conflicted* colog theories. Design SAM tools to aid in the computation of conflict.

References

[1] Coherent Logic *via* colog (colog language primer)
Linked at the SkolemMachines.org website: `colog.pdf`.

[2] Colog prover, `colog14I.jar`. Available at SkolemMachines.org website. Runs as GUI or from command line. Start GUI by double-click on jar, read info under the `???` tab.

[3] Concurrent coherent logic report (2009).
`http://SkolemMachines.org/reports/cocolog09-19ver/cocolog.pdf`
The runtimes table in the appendix illustrates the large gains in speed due to SAM indexing (rather than parallelism).

[4] John Fisher and Marc Bezem, Skolem Machines, *Fundamenta Informaticae*, 91 (1) 2009, pp.79-103.
Linked at the SkolemMachines.org website: `SkolemMachines.pdf`
(Note that "colog" is called "geolog" in this paper. The link is to a copy with minor corrections.)