

October 1970

DESIGN IMPLICATIONS OF THEOREM-PROVING STRATEGIES

by

Robert E. Kling

Artificial Intelligence Group

Technical Note 44

SRI Project 8776

The research reported herein was sponsored by National
Science Foundation Grant GJ-1060.

QA3.6 is a new resolution theorem prover that allows a user to flexibly select subsets of pre-set strategies and to easily state strategies of his own. Presumably, some strategies will in fact be expressible within hours after conception, while others may require days or weeks and require modifying the system. Currently, when a user wishes to add a new strategy to the old QA3.5 he must modify certain system functions. Frequently several functions must be modified and the flow of information rerouted nontrivially. Furthermore, a QA3.5 user must be intimately familiar with the various internal representations and system structure in order to write a successful strategy.

We hope to design QA3.6 so that a strategy writer need not know much, if anything about the internal representations and information flows. He will be able to work in a language which is closer to the vernacular of resolution logic, e.g., clauses, literals, resolvents, etc. than to the language of implementation, e.g., caar, cadar, etc. In order to facilitate this goal, it is likely that the best design of QA3.6 would create a series of check past which all information of certain kinds would flow. Thus, to add a particular (selection or deletion) strategy would require modifying only one well understood section of QA3.6 rather than several clever, but idiosyncratically-chosen spots. Presumably, such well-defined changes could be automated for most users.

Unfortunately, we barely understand the implications of these criteria. The class of acceptable strategies is presently ill-defined, and the design requirements to implement this set are still unknown. In order to define the set of acceptable strategies and pursue their implications, I am describing a set of strategies that might be acceptable

to QA3.6. Some of these strategies may in fact be rather good, while others may cause the search-efficiency to deteriorate. All of these strategies are "reasonable." A user, knowing the primitive abilities available within the system would want to pose some strategies like these. Each of these strategies poses somewhat different demands upon the system design.

This memo present a set of plausible strategies that a user may want to test on QA3.6. Some of these may in fact be rather good while others may be quite poor; the quality of these strategies are irrelevant. Rather, these strategies are provided in order to define the set of acceptable strategies by inclusion and by exclusion. Each of the strategies also poses somewhat different design criteria which we do not yet understand and want to illuminate by studying system structures they imply.

Attention (Ordering) Strategies

(Which clause pair shall we consider next?)

1. In QA3.6 we do not want to force ourselves to be bound to unit preference strategy. However, there are many varieties of unit preference, and a user might want to devise his own.

Consider the following version:

Perform in sequence all unit resolutions (e.g. $l \times n$). When no additional unit resolutions are possible, enter the non-unitsection

e.g., $k \times n$, $k_1 n > 1$ and resolve clauses C_1 and C_2 on literals L_1 and L_2 ($L_1 \in C_1$, $L_2 \in C_2$) $\Leftrightarrow L_1$ or L_2 is a ground literal. If a pass through the nonunit section does not yield any resolvents, remove the ground-literal restriction and recycle through the non-unitsection without it.

If the nonunit section has been entered 3(or k) times and generated some resolvents with the ground literal restriction, remove that restriction for the next pass through it.

2. Order the search as in the preceding strategy but change the acceptance criteria to be that the unification of L_1 and L_2 must be ground. Thus, $p[a;y]$ and $\neg p[x;b]$ will be acceptable literals now, although neither are ground literals and would fail the preceding test.

3. If a long clause is entered into CLAUSEARRAY, and will be used in a proof, then we want to chip it down as soon as possible into a "workable short clause." Two ways to do this are to emphasize unit (ground) resolutions with long clauses ($k > 3$),

and to allow $2 \times n$ resolutions with "long clauses." At worst, a $2 \times n$ resolutions resolution results in a clause of length n . But, $2 \times n$ resolutions are likely to reduce the length of a resolvent $< n$ if there are repeated literals induced by the unification. These observations can result in a variety of strategies, but consider just the following:

- (a) Allow the unit section to resolve $1 \times n$ clauses for $k = 2$, to ... n . Where n - maximum length clause.
- (b) Define a 2-section that will resolve $2 \times n$ clauses for $k = n$, ..., down to 3.
- (c) Define a 2×2 section that resolves two clauses with two clauses
- (d) Define a multisection that resolves $k \times \ell$ clauses for $k > 2$, $\ell > 2$, $k + \ell = m_1$ incremented on n . $m = 6 \dots 2_n$.

If the unit section fails, operate the 2-section, 2×2 section, and multisection in that order until some resolvents are found. After completing a run through a successful section, branch back to the unitsection.

4. In QA3.5, when a resolvent is generated, all possible unit sections are performed with it and its unit-resolvent descendents. Whenever a unit resolvent is obtained, a check for its resolving with another unit to produce \square is automatically performed. How can we formally add these features to the expression of the preceding strategies?

5. Suppose that whenever a unit resolvent is added, we will resolve it against all 2-clauses in addition to the unit-check. How do we express this? If we want to limit these 2-clause resolutions to those which result in ground units--how could we express that too?

6. Search strategies that involve graded or partitioned memories will be described in a later section.

Selection Strategies

(Given a set of clauses, shall they be resolved?)

1. In the classical T-support strategy, two clauses are not resolved unless either has T-support. Consider an expanded notion of T-support where 2 axioms will resolve if some (all) their predicates belong to the set of predicates in all the clauses so far resolved. This criterion is closely related to purity. To be more formal;

let $\text{pred}[C]$ = list of all predicates in a clause C.

$C - L$ = set of literals in a clause C minus literal L

$\text{tsupp}[C]$ = T iff c has T-support, else Nil.

and $\underline{\text{preds}} = \{p \mid \exists C p \in \text{preds}[C] \wedge \text{tsupp}[C]\}$.

Then, C_1 is resolvable with C_2 on L_1 and L_2 respectively iff.

$$\begin{aligned} & \text{tsupp}[C_1] \vee \text{tsupp}[C_2] \vee \text{not}[\text{null}[\text{intersection}[\text{pred}[C_1 - L_1]; \underline{\text{preds}}]]] \\ & \vee \text{not}[\text{null}[\text{intersection}[\text{pred}[C_2 - L_2]; \underline{\text{preds}}]]] \quad . \end{aligned}$$

Note, that after each new resolvent is entered, the system must update preds. For a new resolvent R $\underline{\text{preds}} := \text{Union}[\underline{\text{preds}}; \text{pred}[R]]$.

2. For a more refined version of the preceding strategy, note that a clause $\neg p(a) \vee C_0$ is a useful resolvent iff (at sometime) a clause of the form $p(x) \vee C_1$ enters clause array. Thus, to narrow the expanded T-support condition, we can use information regarding the predicates expressed in the positive and negative literals of clauses on CLAUSEARRAY.

Formally, let:

$+ \text{preds}[C]: =$ positive predicates in C

$- \text{preds}[C]: =$ negated predicates in C

$$\underline{\text{nspreds}} = \{p \mid \exists C p \in - \text{preds}[C] \wedge \text{tsupp}[C]\}$$

$$\underline{\text{pspreds}} = \{p \mid \exists C p \in + \text{preds}[C] \wedge \text{tsupp}[C]\} \quad .$$

Accept C_1 and C_2 to resolve on L_1 and L_2 iff:

$$\begin{aligned} & \text{tsupp}[C_1] \vee \text{tsupp}[C_2] \vee \\ & \text{not}[\text{null}[\text{intersection}[+ \text{preds}[C_1 \cup C_2 - L_1 - L_2]; \underline{\text{nspreds}}]]] \\ & \vee \text{not}[\text{null}[\text{intersection}[- \text{preds}[C_1 \cup C_2 - L_1 - L_2]; \underline{\text{pspreds}}]]] \quad . \end{aligned}$$

Again, pspreds and nspreds must be updated after each successful resolution.

3. Let $\varphi(p)$ be a partial ordering of predicates. Let $\varphi(p)$ be represented by a computable $\text{predval}(p): p \Rightarrow [0, 1]$. I will resolve C_1 on L_1 with C_2 on L_2

iff $\forall p \in C_1 - L_1 \text{ predval}(p_1) \geq \text{predval}(p)$
and $\forall p \in C_2 - L_2 \text{ predval}(p_2) \geq \text{predval}(p)$
where $p_j =$ predicate in literal L_j .

Deletion (Editing) Strategies

(Given a resolvent, should I retain it?).

1. Often a user will want to specify in advance function orderings that are either acceptable or rejectable when they appear in some term in a resolvent. Let fns be a list of acceptable (rejectable) function nestings. Then accept (reject) a clause C iff one of these patterns matches the nesting in a term of one of its literals.

2. Consider a resolvent R between an axiom A and some clause C . R will be accepted iff there exists some clause R_2 on CLAUSEARRAY such that R and R_2 resolve. (This may be a particularly good strategy for editing out irrelevant axioms after the search is fairly well developed.)

Graded Memory

In QA3.5 the axiomatic data base is kept separate from an active set of clauses which includes $\neg T$, and resolved axioms, and all the resolvents. All axioms in MEMARRAY are "equal" (except for a unit-preference ordering). Suppose a user wants to place a metric φ in the axioms so that each axiom A is now assigned to some "grade" $\varphi(A) \in [0, 1]$.

If $\varphi(A) = 1$, A should receive frequent attention, while as $\varphi(A) \rightarrow 0$, A should receive correspondingly less attention. In addition, when the theorem proving search is going well, then the low grade axiom should be ignored, while at times when the search is poor a few lower grade axioms may be considered for inclusion. Thus, we would like a way to compute a dynamic grade $\varphi_d(A)$ which changes with the state of the search. For example, if a proof attempt is well developed, then a new candidate axiom A_1 should resolve with many of the clauses developed in the search, rather than just one which would bring it into CLAUSEARRAY. A stronger criterion would be that A_1 should resolve with several units. Suppose that the resolvent between A_1 and some unit in CLAUSEARRAY is R_1 , and the resolvent between R_1 and another active unit is R_2 until we reach some R_n . We can approximate:

$$\varphi_d(A) = \varphi(A) + \frac{\text{length}[A] - \text{length}[R_n]}{\text{length}[A]}$$

$$2 \geq \varphi(A) + 1 \geq \varphi_d(A) \geq \varphi(A) \quad .$$

Now, let the user set a threshold T_d (or let the program increment or decrement T_d based on its own successes). Then, let A_1 resolve with a unit C_1 iff

$$\varphi_d(A_1) \geq T_d \quad .$$

(Note that if A_1 is acceptable we would like to keep R_n and its parents through A_1 , and otherwise throw it away.)

Formally, we want to have a recursive definition of resolving units with resolvents of A_1 , naming the maximal (final) resolvents, computing its length, etc.

In addition we need ways to describe the progress of the theorem prover and changes in T_d . In the beginning of a search we have little need to go through such a long computation, e.g. $\varphi_d(A)$ to decide upon the acceptability of A . $\varphi(A)$ alone should suffice. However, when all latent resolutions have been performed without reaching \square or the theorem prover is spending exorbitant amounts of time in a non-unit section, then either T_d should be decremented (for comparison with $\varphi(A)$), or $\varphi_d(A)$ should be computed in a new search through memory. Thus, in a graded memory, the "breadth" of our acceptance strategies may vary with time and we need ways to express that too.