AN APPROACH TO TABLEAU INFERENCING WITH 'HARDWIRED' HEURISTICS

Hal Berghel ‡†
Richard Rankin ‡†
Farukh Burney ‡

‡ University of Arkansas
† National Center for Toxicological Research

ABSTRACT

In this paper we discuss a prototype of the tableau method which derives its rule base from Jeffrey's text [3]. The heuristics are a subset of those outlined for HARP by Oppacher and Suen [4]. The prototype was written in C and intended for use with the IBM family of microcomputers. We shall outline some of our implementation strategies and, insofar as is possible, contrast them with those of the earlier pioneering work of Oppacher and Suen.

INTRODUCTION

One of the major practical breakthroughs of this century in formal logic was the development of the semantic tableau [1][2][7]. This complete and consistent decision procedure has since evolved into the truth tree method [3][8] which preserves the important completeness and consistency properties while making the resulting proof tree more readable. As a result of the convenience of the procedure and the desirable metalogical properties, the truth tree variant of the semantic tableau has become a de facto standard for first order problem solving [3].

From a computational point of view, the method is attractive because it is supportable by fairly clearcut unification and resolution procedures. Since an automated tableau, by definition, supports full first-order logic with identity, there is a great deal of current interest in developing prototypes (see refs. [4][5][6][7] for further details).

THE FORMAL METHOD

As with conventional inference engines (e.g., Prolog), semantic tableaus rely on a reductio ad absurdum or indirect proof strategy. That is, in the case of validity checks, an attempt is made to reconcile the negation of the conclusion with the premises. If the negation of the conclusion is jointly satisfiable with the premises, then the conclusion is not satisfiable, hence the argument is invalid. Unlike conventional inference engines, the tableau method works with full first order expressions rather than restricted subsets (e.g., Horn clauses) and utilizes a set of procedural rules which are complete w.r.t. the Principia Mathematica connectives. This makes the automation of the semantic tableau particularly appealing to the AI community.

The semantic tableau rule base consists of three types of rules: branching rules, non-branching rules and quantifier rules (see Figure 1). We will initially take up the first two categories.

The validity of the parsing rules derives from the truth functional properties of the corresponding formulas. To illustrate, we note that there are two ways in which the formula '$\alpha \vee \beta$' may be true: either 'α' is true or 'β' is true (or both, of course). In the tableau, this fact is captured in a branching rule with '$\alpha \vee \beta$' as the parent and 'α' and 'β' as separate descendants. Similarly, de Morgan's law for conditionals tells us that a negated conditional is true just in case the antecedent is true and the consequent is false. This fact is reflected in a non-branching rule. In general, the parsing rules ensure that every truth functional alternative appears in the proof tree. One may think of each of these parsing rules as a property which holds for any arbitrary interpretation over the variables (e.g., $I(\alpha \rightarrow \beta)=F \leftrightarrow (I(\alpha)=T$ and $I(\beta)=F)$, for any interpretation, I). The parsing rules which pertain to truth functional structure appear in both branching and non-branching versions in Figure 1.

1) Non-Branching Rules

a) $\frac{\sim\sim A}{A}$	b) $\frac{A \wedge B}{\begin{array}{c}A\\B\end{array}}$	c) $\frac{\sim(A \vee B)}{\begin{array}{c}\sim A\\ \sim B\end{array}}$	d) $\frac{\sim(A \rightarrow B)}{\begin{array}{c}A\\ \sim B\end{array}}$

2) Branching Rules

a) $\frac{A \vee B}{A \mid B}$	b) $\frac{A \rightarrow B}{\sim A \mid B}$	c) $\frac{\sim(A \wedge B)}{\sim A \mid \sim B}$	d) $\frac{A \leftrightarrow B}{\begin{array}{c}A \mid \sim A\\ B \mid \sim B\end{array}}$	e) $\frac{\sim(A \leftrightarrow B)}{\begin{array}{c}A \mid \sim A\\ \sim B \mid B\end{array}}$

3) Quantification Rules (restrictions apply - see text)

a) $\frac{\forall x A(x)}{A(t)}$	b) $\frac{\sim\exists x A(x)}{\forall x \sim A(x)}$	c) $\frac{\exists x A(x)}{A(t)}$	d) $\frac{\sim\exists x A(x)}{\forall x \sim A(x)}$

FIGURE 1: TABLEAU PARSING RULES

In addition to the parsing rules, there are two rules which deal with quantification. The rule for existential quantification is depicted as 3.c. in Figure 1. This rule states that for any formula of the form $\exists x\alpha$, if the path containing $\exists x\alpha$ does not contain any formula of the form $\alpha[x/t]$, then we may substitute for $\exists x\alpha$ a new formula $\alpha[x/t]$ for some term, t. In effect, this guarantees that ground instances of existential quantifiers will only result from instantiations of terms new to the path.

Conversely, the rule for universal quantification ensures that grounding will only result from instantiations of terms already on the path. Formally, for each ground term t on the path containing $\forall x\alpha$, we introduce the formula $\alpha[x/t]$. However, unlike all of the other rules, the rule for quantification is never terminal with respect to a formula. In this case, but in no other, our algorithm retains the original formula $\forall x\alpha$ on the path along with each and every ground instance.

The flowchart for the semantic tableau appears as Figure 2 followed by a typical tableau in Figure 3.

HEURISTICS

As we mentioned earlier, one of the advantages of the tableau method is that with the exception of a few anomalies involving quantifier nesting, the rules may be applied in any order without sacrificing the correctness of the procedure. In our system, we apply the rules by means of a subset of the

STAGE 1.List the premises and the negation of the conclusion.

STAGE 2. Apply the rules for the negation of all relevant formula. Close (X) each path that contains both a formula and its negation. Are all paths closed?
NO: GO TO STAGE 3
YES: STOP. THE INFERENCE IS VALID

STAGE 3. Is there a formula (unchecked in an open path) to which one of the parsing rules applies?
NO: GO TO STAGE 4
YES: APPLY IT and GO TO STAGE 2

STAGE 4. Is there a formula (unchecked in an open path) to which the rule for existential quantifiers can be applied.
NO: GO TO STAGE 5
YES: APPLY IT and GO TO STAGE 4

STAGE 5. Is there a formula (in an open path) for which the rule for universal quantifiers can be applied?
NO: GO TO STAGE 6
YES: APPLY IT and GO TO STAGE 5

STAGE 6. Have any changes been made in the tree since last entering stage 2, above?
NO: GOTO STAGE 7
YES: GO TO STAGE 2

STAGE 7. STOP. The inference is invalid.

FIGURE 2: TABLEAU FLOWCHART

$\forall x(Px \wedge Qx) \rightarrow (\forall xPx \wedge \forall xQx)$

1	✔	$\sim(\forall x(Px \wedge Qx) \rightarrow (\forall xPx \wedge \forall xQx))$ [negation of formula]		
2		$\forall x(Px \wedge Qx)$		
3	✔	$\sim(\forall xPx \wedge \forall xQx)$		
4	✔	$\sim\forall xPx$	5 ✔	$\sim\forall xQx$
6	✔	$\exists x\sim Px$	7 ✔	$\exists x\sim Qx$
8		$\sim Pa$	9	$\sim Qb$
10	✔	$Pa \wedge Qa$	11 ✔	$Pb \wedge Qb$
12		Pa	14	Pb
13		Qa	15	Qb
		X (closed)		X (closed)

FIGURE 3: SEMANTIC TABLEAU

heuristics used in HARP. It should be remembered that the objectives for this prototype were portability within the IBM/PC family of microcomputers and efficiency. As a result, heuristics were 'hardwired' into the code. This yields the small code size and increased speed, but comes at the expense of generality and robustness. Further, all of our heuristics are definable as Type A in Oppacher and Suen, e.g. definable as "...heuristics for efficient and human-like proof construction...". As a group, they are 'textbook-level' heuristics which virtually every student of elementary logic applies in tableau-based reasoning.
For completeness, we include in Table 1 the entire set of 14 heuristics used in HARP by Oppacher and Suen. Those which we have implemented are marked with an asterisk.

TABLE 1: Heuristics H0 to H13

H0: Work on a branch until it closes or is known to remain open
H1*: Work on a formula until it is ground
H2*: Avoid unnecessary branching
H3*: Prefer existential instantiation
H4: Favor formulas with nested existential quantifiers
H5: Favor formulas derived from the negation of the conclusion
H6: Avoid clearly useless work
H7: Prioritize branching
H8*: Favor fresh universal quantifiers
H9*: Minimize the proliferation of instantiated terms
H10: Use of theorem Introduction (complementary)
H11: Apply domain specific rules wherever possible
H12*: Identify complete open branches ASAP
H13*: Recognize non-converging $\forall\exists$ patterns

Before we turn to a discussion of the actual implementation, we need to make a fairly substantial caveat. Since this is a prototype, we took considerable liberties in experimentation, even if the experimentation were unjustified from a practical or theoretical point of view. For example, our current search strategy is breadth-first, rather than depth-first (as called for in H0). Since we may determine the invalidity of an argument with only one open path, there is no reason to pursue a second path if we can show that the first one will never close. Hence, H0 is the only viable search strategy. However, a breadth first strategy is far more useful when one is tracing through the program execution because it provides a wider window into the operation of the program. Thus, it was selected. We had similar reasons for omitting other heuristics. In general, if we felt that the time spent was disproportionately large compared with the understanding which the inclusion would have afforded, we left it out. It is important to recognize that our approach is purely experimental within a narrow environment, and not intended as a commercial-grade product.

DEVELOPMENT SYSTEM

Our prototype was totally developed in a microcomputer environment. The development and target machine was an IBM PC class platform with 640k and PC-DOS 3.0 or above. The compiler used was Microsoft Quick C, version 1.0. The resulting program uses 'standard C', and is, therefore, portable, except for the windowing functions used to develop the opening screens. The portability and utility of the system is further enhanced by the fact that the executable module is only 19k in size. Each node in the tableau requires approximately 30 bytes, primarily for pointers to other structures. These additional structures require a substantial portion of the memory usage by the system. We estimate that a tableau with two to three thousand nodes can be easily accomodated in a machine with 640k memory.

LIMITATIONS

Some limitations apply to the implementation of the system, primarily regarding the input schema, and the number of heuristics included.

During input, all connectives must be completely parenthesized. An example of this would be: ((Fa→Ba)→Ca):Da. The input routine checks for the proper matching of parentheses. It appears that HARP [4] adds parentheses as needed, by itself. Also, connectives and quantifier symbols in the prototype have been chosen from the set of printable characters in the lower half of the ASCII set. This increases portability, but decreases the aesthetic appearance of formulas.

HARP maintains nodes awaiting processing in a priority queue. A meta-level supervisor applies the heuristics supported by the system, and selects a new candidate formula for processing based upon the resultant priorities. The prototype applies the heuristics implemented as part of a control loop which uses the sequential ordering of the procedures to apply procedures in the proper order. The ordering of the control structure itself forces heuristics to be applied in the proper order, favoring existential instantiation, for example, over universal instantiation. The protoype system maintains a list of leaf nodes so that full tree traversals are not required to apply splitting rules to leaf nodes or to determine if the entire tree is closed.

HARP has implemented a memory saving schema which allows new nodes to be added without formula duplication. New nodes are represented by pointers to the proper sub-formula of the ancestor formula. This allows for structure sharing among many nodes. This feature was not implemented in the system.

GENERAL CONTROL STRUCTURE

The prototype is implemented by using a control loop which utilizes the ordering of the procedures in the loop to apply the heuristics in proper order. The control loop based upon earlier work of Reeves [5] is as follows:

```
begin
repeat
 changes:=false
 close each path containing a sentence and its negation
 if all paths closed, deliver:entailment is valid
 else
    {1} if a splitting rule can be applied then
        changes:=true
        {2} apply appropriate splitting rule
        mark the sentence as used
    else{3}
        apply the existential (instantiation) rule
        {4} apply the universal (instantiation) rule
until not changes
```

```
deliver: entailment is invalid
end
```

Lines marked above in braces, e.g. {1}, are referenced below.

Our control structure is slightly varied to allow for the inclusion of the heuristics described above. H2, can be incorporated by distinguishing, at section {1} between sentences available for branching rules and non-branching rules. By favoring non-branching rules over branching rules for application at section {2}, one incorporates H2, since non-branching rules *restrict the growth of the seach tree.*

In section {3}, the pseudo-code for our prototype is implemented so that the application of a universal instantiation would occur only if there were no existential instantiation candidate available. This effectively applies H3.

H8 is implemented by maintaining a flag which indicates when a universal quantifier rule is used. Section {4} prefers universal quantifier rules which have not been used. For full implementation of H8, this flag would be changed to an occurrence counter, and the universal rule with the lowest counter value would be selected.

Heuristic H9 is concerned with minimizing the number of new parameters introduced. This heuristic was incorporated through the use of a system incorporating Reeves' dummy variables schema, [5], and is discussed in more detail below.

The resulting control structure for the system, therefore, is as follows:

```
do {
 tree_change := false
 if (tree_closed) then valid := true
 else
    search_for_alpha_rule
    if (no alpha_rule)
       search_for_beta_rule
    if (splitting rule found)
       apply splitting rule found
       tree_change :=true
    else
       search_for_existential_candidate
       if (no existential_candidate_found)
          search_for_universal_rule
       if (existential_candidate_found)
          apply existential_rule
          tree_change := true
       else if (universal_candidate_found)
          apply universal rule
          tree_change :=true
} while (tree_change = true)
if (valid = true) return: entailment valid
else return: entailment invalid
```

H12 is implemented using the dummy variables and scope trees. When a dummy variable is added to a leaf node, and the constraint list is empty, the conditions hold for invoking H12, and the branch may be declared complete.

H13 is invoked when applying instantiations from formulas containing a ∀∃ pattern. This operation involves checking branches for non-converging ∀∃ patterns. A standard form of a non-converging ∀∃ pattern is:

$$\forall x \exists y Fxy, \exists y Fay, Fab, \exists y Fby, Fbc, \exists y Fcy, Fcd$$

The universal and existential quantifiers repeatedly introduce new instantiations which do not converge towards a solution. Implementing H13 involves a search of the current path when instantiations arising from a ∀∃ formula arise. When a non-converging pattern is found, the branch is declared complete.

DUMMY VARIABLES

As mentioned above, the use of dummy variables in universal instantiations can help control the proliferation of unnecessary nodes. Implementation problems with dummy variables generally fall into two categories: quantifier scope and constraint lists.

The first determination to be made when using dummy variables is the scope of the quantifiers involved in the sentence prior to instantiation. For example, consider the sentence:

$$\forall x\,(\forall y \exists u(\ldots) \rightarrow \forall z \exists v(\ldots))$$

From the discussion above, it can be seen that existentially instantiating either u or v should constrain x, since both are within the scope of ∀x. ∃u is within the scope of ∀x and ∀y, but not within the scope of ∀z. ∃v is within the scope of ∀x and ∀z, but not within the scope of ∀y. If an *existential instantiation is made with variable u, the resulting constraint* should only be placed upon the dummy variables for x and y, and not z. To manage this problem, our prototype utilizes scope trees to determine the proper dependencies.

A scope tree is constructed for a sentence with quantifiers. The tree maintains only the quantifier portions of the sentence so that a relation of dependencies can be quickly analyzed. As leaf nodes are added to the tableau, each receives the appropriate copy of a scope tree.

As one instantiates portions of the tableau, it is necessary to determine which values have appeared. Using only scope trees, any instantiation would require a search of the scope tree for every node above the current node. For reasons of efficiency, the prototype incorporates what are called path_frames. A path_frame is a doubly linked list which stores all values which have been inserted in every path of the tableau. Each leaf node maintains a pointer to its path_frame and can check the path_frame for values which have been used along the path to the leaf node. A path_frame is maintained for every leaf node. When a new node is added, the path_frame is moved to the new node. When a splitting rule has been applied, for example, creating two new leaf nodes, each of the new nodes receives a copy of the path_frame of the parent. This allows quick

EXAMPLE 1

Formula: F[1][2], (G[1][3]&R[1][2]),(F[1][2]>(-G[1][3]+-G[1][3]+-R[1][2])
Entailment is valid

NODE	LEFT	RIGHT	CLOSED	CONTENTS
0	1	-	n	F[1][2]
1	2	-	n	(G[1][3]&R[1][2])
2	3	-	n	(F[1][2]>(-G[1][3]+-R[1][2]))
3	4	-	n	G[1][3]
4	5	6	n	R[1][2]
5	-	-	y	-F[1][2]
6	7	8	n	(-G[1][3]+-R[1][2])
7	-	-	y	-G[1][3]
8	-	-	y	-R[1][2]

EXAMPLE 2

Formula: -∃x(Fx+-Fx)
Entailment is valid

NODE	LEFT	RIGHT	CLOSED	CONTENTS
0	1	-	n	∀x-(Fx+-Fx)
1	2	-	n	-(F[#1]+-F[#1])
2	3	-	n	-F[#1]
3	-	-	y	F[#1]

[#1] = 1 Constraint on [#1] : None

EXAMPLE 3

Formula: ∃zBzz, ∀x(Sx>Bxx):-S[7]
Entailment is invalid

NODE	LEFT	RIGHT	CLOSED	CONTENTS
0	1	-	n	∃zBzz
1	2	-	n	∀x(Sx>Bxx)
2	3	-	n	S[7]
3	4	-	n	B[1][1]
4	5	6	n	(S[#1]>B[#1][#1])
5	-	-	y	-S[#1]
6	-	-	n	B[#1][#1]

[#1] = 7 Constraint on [#1] : None

FIGURE 4: SAMPLE OUTPUT

determination of potential instantiations for a given node.

SAMPLE OUTPUT

Output from the prototype is presented as a list of nodes and contents similar to that used in a cursor-based array implementation of a binary tree. This method of presentation was chosen to eliminate the problem of graphically displaying a binary tree of arbitrary depth. As can be seen from the examples in Figure 4, the original formula is presented, a decision as to whether or not the entailment is valid, then a presentation of the tableau. Constraints on dummy variables are also listed, where applicable.

Dummy variables are listed as [#x] where x is an integer value. Constants are represented as positive integers. V is used for universal quantification, E is used for existential quantification, other capital letters are available for predicates. Variables are represented by lower case letters. Connectives are: '&' for AND, '+' for OR, '-' for NEGATION, '>' for IMPLICATION, and '=' for CO-IMPLICATION. The premises are terminated by commas, and the conclusion by a colon. Constraints are listed by dummy variable name. For example, the line:

[#1] = 0 Constraint on [#1] : 1

is equivalent to Reeves' "$X_1 < 1$" where 1 is considered a constant.

CONCLUSION

As we mentioned in the introduction, we approached this prototyping from a special interest point of view. Unlike the broader work of Oppacher and Suen, our work is best described as an investigation into the feasibility of developing a tableau-based inference engine for MS-DOS microcomputers. We found that such a system was both realistic and practicable within this environment.

Subsequent work on memory management and control strategies has already begun. A depth-first search strategy would speed the determination of the validity of the entailment, and will be included in the future. Parallel with this work will be an investigation into the type and variety of extra- and meta-logical mechanisms which may be required of the programming language which will ultimately evolve from this work.

REFERENCES

[1] Beth, E.W., Aspects of Modern Logic, D. Reidel Publishing Co., Dordrecht, 1970.

[2] Beth, E.W., Formal Methods: An Introduction to Symbolic Logic and to the Study ofEffective Operations in Arithmetic and Logic, D. Reidel Publishing Co., Dordrecht,1962.

[3] Jeffrey, R.C., Formal Logic: It's Scope and Limits, McGraw-Hill Book Co., New York, 1967.

[4] Oppacher, F. and E. Suen, "HARP: A Tableau-Based Theorem Prover", Journal of Automated Reasoning, Vol 4, 1988 pp.69-100.

[5] Reeves, S., "Semantic Tableaux as a Framework for Automated Theorem-Proving", manuscript.

[6] Reeves, S., "An Introduction to Semantic Tableaux", CSM-5, Department of Computer Science, University of Essex, March 1983.

[7] Reeves, S., "Adding Equality to Semantic Tableau", Journal of Automated Reasoning, Vol. 3, 1987, pp. 225-246.

[8] Smullyan, R.M., First Order Logic, Springer-Verlag, NY, 1968.