An experiment in reducing run time costs for a model constructor in Partial Information Logic

COSC 4235

April 4, 1997

Prepared for Professor J. J. Rajnovich
by Michelle Kraushaar
900806320

Filename: C:\MICHELLE\THESIS\PAPER.WPD

Abstract

This paper discusses an experiment in implementing a Modus Ponens and Contextual Modus Ponens reducer function for Partial Information Logic model generation. The reducer is a preprocessor that is to be used with the PIL implementation of a beth tableau generator as developed by Rajnovich and Nait Abdallah.

Table of Contents

List of Tables

Introduction

Partial information logic, as presented by Nait Abdallah in *The Logic of Partial Information*, is an extension of classical logic intended to provide a computable formalism for dealing with knowledge which is tentative. It makes minor additions to the syntax of both propositional logic and first-order logic to allow for tentative reasoning, which in most real world problems is the rule, not the exception. It makes, however, a major shift in the semantics of the logic. The result is a logic for tentative reasoning which has the interesting property that it is computable (a rareity in current research in AI). A proof method is provided based on an extension of the classical tableau method. This method is, however, intractable.

My research has focused on the issue of tractability when dealing with real world problems. How efficiently can we generate models for practical problems? Can we find special cases which are easier to deal with than others? I have started with an implementation of a theorem prover for propositional partial information logic as developed by J. J. Rajnovich. The task is to improve the performance of this software in terms of time and memory use.

Background

Partial information logic uses classical logic as its basis.[1] From here, it adds some syntax to allow for tentative reasoning.

Classical logicis a total logic in which every atomic formula is either True (T) or False (F). In partial information logic, we allow for gaps in aour knowledge. Formulae can be unknown. The symbol $\perp$ is used to represent this. This does not, however, constitute a three-value logic. The $\perp$ is simply a placeholder to indicate an unknown value.

Because the logic is partial, we have four truth turnstiles to indicate, instead of the classical two (see Table 1). In classical (or total) logic, we have the two turnstiles: $\models$ for true and $\not\models$ for false. In partial information logic, however, we have four. The first one is the same in both logics. The difference begins at the second turnstile. $\not\models$ now means 'not true'. This would seem to be the same as false, but the difference is subtle. Recall that in partial logic, there are times when we do not know the truth values of a formula. If we have an unknown value in a formula, we cannot say with absolute certainty that it is false, so, instead, we say that it is simply not true.

---

[1]This discussion assumes the reader has an understanding of classical logic (both propositional and first-order).

The third and fourth turnstiles are unique to partial logic. ||= means 'potentially true'. This is used to indicate that it is possible for the formula to be true (ie T or ⊥). The final turnstile, ||≠, means 'not potentially true'. This is equivalent to the classical false.

Turnstile	*Total (Classical) Logic*	*Partial Logic*
|=	True	True
|≠	False	Not True
||=	--	Potentially True
||≠	--	Not Potentially True (F)

Table 1. Logical Turnstiles

The next syntactic addition is an increase in the number of negation operators from one to three (see Table 2). In classic logic, there is only one negation operator (¬). In partial logic, there are three; ¬ is classical negation, ~ is called pessimistic negation, and ~′ is optimistic negation. Under all three negation operators, if a is true, then $\neg a$, $\sim a$, and $\sim' a$ evaluate to false and vice versa if a is false. The difference arises when you consider the unknown value (when a is ⊥). Under classical negation, $\neg a$ when a is unknown is still unknown. This makes sense. Under pessimistic negation, $\sim a$, we assume that a was false to start with, and therefore, $\sim a$ is true. Under optimistic negation ($\sim' a$), we make the opposite assumption (that a was true to start with), so $\sim' a$ is false.

a	¬a	~a	~'a
T	F	F	F
F	T	T	T
⊥	⊥	T	F

Table 2. Negation operators

There are other syntactic changes, but they are not necessary for this discussion.

Ions

This covers most of the additional syntax, but how is tentativeness shown? This is done with a binary operator *(. . ., . . .) called an ion.

The ion is a binary operator of the form *(*Justification, SoftConclusion*). The ion is presented in the *Logic of Partial Information* as a generalization of a classical atom whose meaning is based on a generalization of classical implication. For example, the implication $a \rightarrow b$ is generalized to the ionic formula *ion(a, b)*[2].This means that if the formula *a* is an acceptable justification, then *b* is true in a soft sense. If *a* is acceptable as a justification, but *b* is false in a soft sense, then the ion is false.[3]

---

[2] or **(a, b)*

[3]It should be noted that there is more than one variety of ion, but for the purposed of this discussion, we will deal only with "free" ions.

Being a formula, an ion can appear anywhere a classical formula can appear. This is what differentiates ions from other logics currently being used.[4] It is also important to note that the justification and soft conclusion parts of an ion can also be formulae (so nesting of ions is possible). For the purposes of this experiment, it was decided to restrict the area of study to non-nested ions.

Let's look at a simple example.

A group is going to go hiking. If the forecast is for rain, bring a cost. If the forecast is for sun, bring sunglasses. What should you bring with you?

1. **(fr, bc)* {If forecast rain, bring coat}

2. **(¬fr, bg)* {If not forecast rain, bring glasses}

One final element of importance to the discussion is the acceptance or rejection of justifications. In order to signify the acceptance of a justification the +* turnstile is used. Conversely, when a justification is rejected, the -* turnstile is used. When a justification is accepted, we have to represent the "soft conclusion". To do this, we use $|=_s$ to represent true in a soft sense.[5]

---

[4] Ions can be indefinitely nested. In this work, we restrict attention to non-nested ions. Such ions can easily be used to capture the intent of Reiter defaults. Defaults, however, are inference rules, not formulae.

[5] There are four more turnstiles in all.

Models, Model Schemes and Model Scheme Patterns

To show that a set of formulae is valid, we need to generate a model. A model is an interpretation of the formulae which makes them true. Interpretation is the assignment of truth values to each atomic formula (atom). Then, we can answer the question "Under what conditions is this formula true?".

The implementation that has been developed by Rajnovich uses an extension of the Beth tableau method for classical logic to generate the models.

Where partial information is concerned, what are generally referred to as models, are actually model schemes. Each scheme denotes a family (potentially infinite) of partial models.

The tableau construction procedure creates branches, syntactic objects which each represent a model scheme pattern. A model scheme pattern has undecomposed justification expressions in it. Inconsistent branches are closed or pruned. The remaining branches give the model scheme patterns. Decomposing the justification expression will give a family of models for each model scheme pattern.

A model scheme is a model scheme pattern in which each justification expression is replaced by its individual mini-tableau. A model is an instance of a model scheme in which specific truth valuations are assigned to its variables.[6]

As a notational convention, a model scheme pattern will be represented as a 4-tuple of the form $<\{H\}, \{R\}, \{A\}, \{S\}>$, where:

H is the set of hard knowledge,

R is the set of rejected justifications,

A is the set of accepted justifications, and

S is the set of soft knowledge.

The logic Tableau

As discussed, a tableau is used to decompose the formula(e). Of special note is that in the worst case, a tableau will have $(2^i)(3^d)$ open branches, where i is the number of ionic operators and d is the number of beta operators[6].

A single implication $(a \rightarrow b)$ is decomposed by first converting it to the equivalent disjunction $(\neg a \vee b)$. Then, the tableau would look like below:

---

[6]A beta operator is an operator that acts like a disjunction. Examples of this would be implications and negated conjunctions.

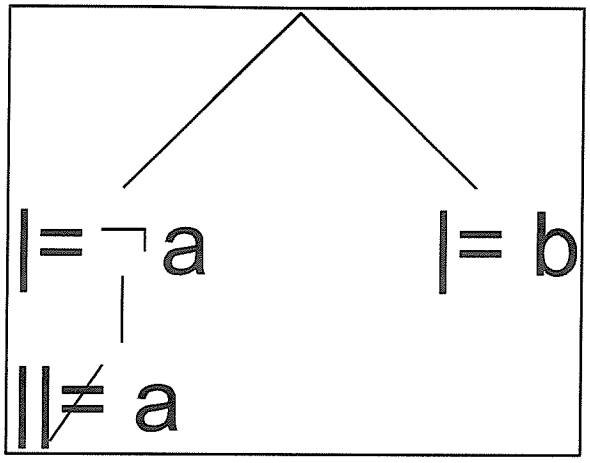


Figure 1 Tableau decomposition for $a \rightarrow b$.

A single ion is decomposed as below:

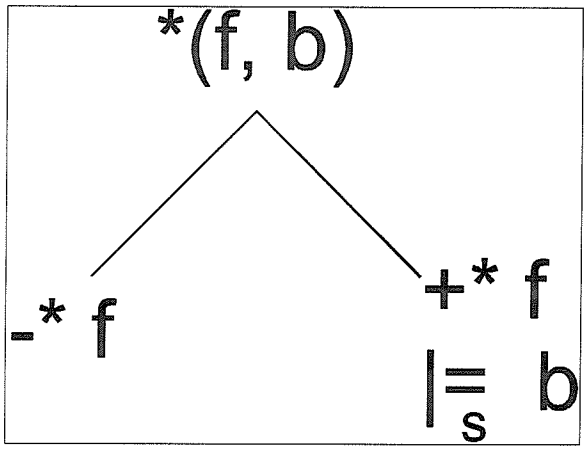


Figure 2 Simple ion decomposition.

The left branch is the rejection of the justification, so no new knowledge is obtained. The second branch represents the acceptance of the justification *f* and in accepting the justification we gain *b* as "soft" knowledge. This means that provided the justification *f* holds, we will believe that *b* is true.

Let's look back at the 'hiking' problem. It's tableau looks like this:

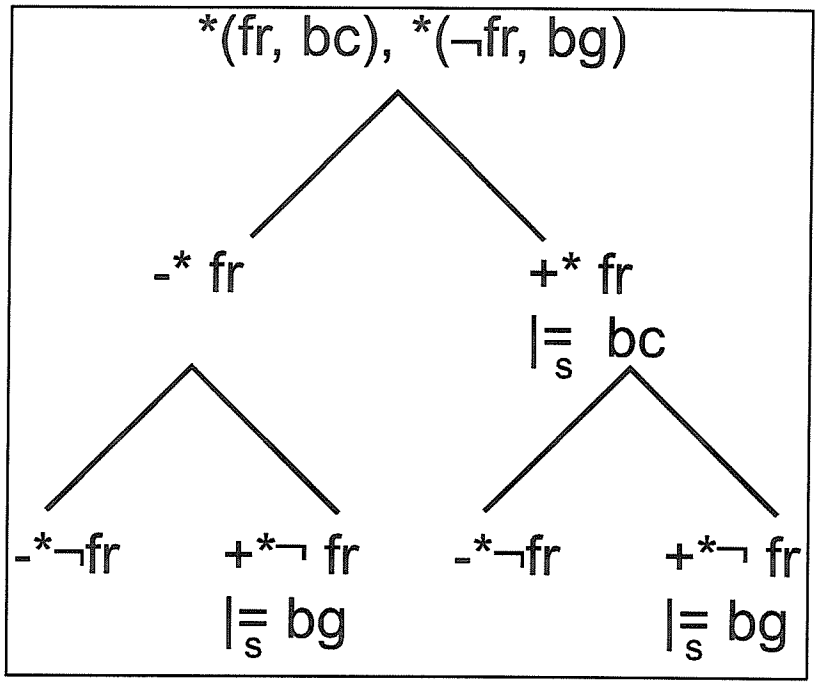


Figure 3 Logic tableau for the hiking problem.

The associated model scheme patterns look like this:

1. <{}, {-*fr, -*¬fr}, {}, {}>

2. <{}, {-*fr}, {+*¬fr}, {|=bg}>

3. <{}, {-*¬fr}, {+*fr}, {|=bc}>

4. <{}, {}, {+*fr, +*¬fr}, {|=bc, |=bg}>

Pattern 1 refers to the instance where it is rejected that is going to rain and rejected

that it is sunny, so, as a hiker, do not bring anything. Pattern 2 is the case where we accept that it is not going to rain, so bring sunglasses. The third refers to the belief that it will rain, so bring a raincoat, and the fourth is where it is decided that it could be rainy or sunny, so bring both.

Horn Clause Logic Problems and Modus Ponens Reduction

Since the development of efficient Prolog compilers, it has been recognized that a large number of problems can be represented in a subset of classical logic called Horn clause logic. Many of the current problems in the literature are in (or are easily converted into) Horn clause form.

It is from these problems that the idea of a preprocessor that checks for Horn clause form and performs Modus Ponens (MP) reduction was born. For the classical total logic problems, Modus Ponens reduction will allow the reduction of the size of the problem (in terms of the number of implications), and in doing so, reduces the size of the tableau needed to generate the models.

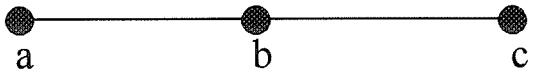
In partial information logic, there is a similar method of reduction called

Contextual-Modus Ponens (called C-MP from now on). The specification of MP and C-MP are listed in Table 3.

Modus Ponens	C-Modus Ponens
from an implication and the premise of the implication, you can infer the conclusion. $\frac{\alpha \rightarrow \beta \ , \ \alpha}{\beta}$	$\frac{*(\Gamma, \alpha \rightarrow \beta), *(\Delta, \alpha)}{*(\Gamma \cup \Delta, \beta)}$ *This can also be representedas:* $\frac{\alpha \rightarrow \beta, *(\Delta, \alpha)}{*(\Delta, \beta)}$

Table 3. Modus Ponens and C-Modus Ponens Specification

To illustrate the savings, let us consider a simple network fault location problem.



Network administrators are trying to figure out if a link is available from host ***a*** *to host* ***c****. There is a link from* ***a*** *to* ***b****, and a link from* ***b*** *to* ***c****. You can send from* ***a*** *to* ***b*** *if there is a link from* ***a*** *to* ***b*** *and that link is up. Similarly, you can send from* ***b*** *to* ***c*** *if there is a link from* ***b*** *to* ***c*** *and the link is up. You can therefore send from* ***a*** *to* ***c*** *if there is a link from* ***a*** *to* ***b*** *and you can send from* ***b*** *to* ***c****. The link is up*

*from **a** to **b**, and from **b** to **c**.*

This can be represented in the following manner:

1. link (a, b)
2. link (b, c)
3. send (x, y) ← link (x, y), uplink (x, y)
4. send (x, z) ← link (x, y), uplink (x, y), send (y, z)
5. uplink (a, b)
6. uplink (b, c)

We can then project this example to propositional form.

1. lab
2. lbc
3. sab :- lab, ulab
4. sbc :- lbc, ulbc
5. sac :- lab, ulab, sbc
6. ulab
7. ulbc

It is now possible to perform MP on this set of formulae since this is now presented in Horn clause form.

(1 + 3 + 6)

8. sab

(2 + 4 + 7)

9. sbc

(1 + 4 + 5)

10. sac

By performing Modus Ponens on this set of formulae, we are able to reduce the number of implications (actually, to eliminate implications altogether). This gives as final results, the following formulae which would go to the tableau generator:

1. lab

2. lbc

6. uab

7. ubc

8. sab

9. sbc

10. sac

Clearly, this is a final solution and there is no need to generate a tableau. If we had generated the tableau for this problem, it would look like this:

***** Tableau here

So, where do ions come into play? Ions permit us to assert tentative knowledge about the network. In this case, links are typically up. This is not hard knowledge.

The case of a link being up is a tentative thing. There is a chance, though, that the link is down between two network nodes. The problem, as now modified, is represented as below:

1. link (a, b)
2. link(b, c)
3. send (x, y) ← link (x, y), uplink (x, y)
4. send (x, z) ← link (x, y), uplink(x, y), send (y, z)
5. *(uplink(x, y), uplink(x, y)) ← link (x, y) [the link is typically up, provided it exists]

Observe that ions are required only in clause five.

This can now be projected into propositional (and Horn clause) form:[7]

1. lab
2. lbc
3. sab :- lab, ulab
4. sbc :- lbc, ulbc
5. sac :- lab, ulab, sbc
6. *(ulab, ulab) :- lab
7. *(ulbc, ulbc) :- lbc

If a tableau were to be generated for the above set of formulae, there would be

---

[7] ":-" as it appears in these clauses in place of the ← symbol. This is a Prolog convention. Since the program (PIL) uses these prolog conventions for its input files,they are used in the examples in this text.

(potentially) 6565 branches (before pruning of contradictory branches). The question is, what is the worst case once C-Modus Ponens is performed? Let's look at this example closely.

8. sab ← uab (1, 3)

9. sbc ← ubc (2, 4)

10. sac ← sbc (1, 5)

11. *(uab, uab) (1, 6)

12. *(uab, sab) (11, 8)

13. *(ubc, ubc) (2, 7)

14. *(ubc, sbc) (13, 9)

15. *(ubc, sac) (14, 10)

This gives us the following formulae to send to the tableau generator:

1. lab

2. lbc

11. *(uab, uab)

12. *(uab, sab)

13. *(ubc, ubc)

14. *(ubc, sbc)

15. *(ubc, sac)

In this case, we would not eliminate the tableau generation altogether, but we would greatly reduce the size of the tableau required. In this reduced form, there

would only a maximum of $2^5*3^0 = 32$ [8]. This is clearly a significant difference and would result in significant time and space savings at runtime.

Clearly, by identifying Horn clause form and performing MP or C-MP before generating the tableau can save time and heap space.

[8]Note that it is actually fewer than this. There are several branches that are terminated due to contradiction.

Experiment Design

Since tableau based theorem provers are NP-Complete (solvable in exponential time), the issue is how to increase the size of the problem solvable on a given computer.

One of the first things investigated in reducing the runtime use of memory resources, was to change the size of some of the variables used. There was one type of variable that was reduced with promising results (see results section). One of the main issues was to consider if reducing the size of the variable (32-bit integer to 16 or even 8 bits) would limit the program in terms of size of problem solvable. Upon further study, it was noted that there were no further significant savings to be found.

By focusing on a specific subset of problems (knowledge bases in Horn clause form), it was decided that a preprocessor that implemented Modus Ponens and C-Modus Ponens would assist in reducing the size of the generated tableau. We will restrict attention to quite simple logical formulae which are still of great computational interest. We will also restrict attention to non-nested ions in the head of the clauses.

The first step was to develop an executable prototype in Prolog. This maintains consistency and permits integration with the initial tableau generator created by

Rajnovich and Nait Abdallah, PIL, which was written in Prolog. One of the benefits of prototyping in Prolog is to have an executable specification which can be debugged before beginning the final coding. This prototype can then be easily converted to the language of choice.

The next step is to take the specification and write the code in C++. This permits integration with the existing PIL++, the C++ version of the partial information logic tableau generator.

When the two implementations are ready (with the preprocessor and without) a number of problems will be run through both with time and heap space requirement recorded and compared. It is expected that if the problem is in Horn clause form, there will be a significant savings. The problems used are in Appendix D.

The program will be run on several machines to test for variations related to hardware.

The Algorithm

The first step is the process of reduction is to determine if a set of formulae is in Horn Claus form. This is done as follows:

```
Function Horn
{
     horn = true;
     while (not (end of list) or (horn = false)) do
```

```
            if (not ((implication and (conjunction
            (antecedent) or atomic(antecedent)) and
            atomic (consequent)) or (atomic (Formula)))
            then
                 horn = false;
            next formula
}
```

This formula will return true if all of the formulae in the input list are in Horn Claus form, else it will return false and quit.

The algorithm for Modus Ponens is as follows:

```
Procedure ModusPonensReducer
{
     if (Horn(FormulaList) then
     {
          while (not (end of formula list)) do
          {
               if Atomic(Formula) then
                    add to atomic list
               else
                    add to non-atomic list;
               next formula
          }
          change = true
          if (Atomic list is non-empty) then
          {
               while ((notEmpty(Non-Atomic)) and
               (change = true)) do
               {
                    Break the Antecedent into
                    components
                    Search the atomic list for the
                    components
                    if FoundAllComponents then
                    {
                         add consequent to Atomic list
                         remove formula from non-Atomic
                         list
                         change = true
                    }
                    Next non-Atomic Formula
                    if end(Non-Atomic list) then
                    {
                         change = false
```

```
                                return to head of non-Atomic
                                list
                            }
                        }
                    }
                }
                FormulaList = append(atomic, non-atomic)
                return FormulaList
}
```

This algorithm calls the Horn function. If the set of formulae is in Horn Claus form, then it breaks the formulae into two lists: Atomic and Non-Atomic formulae. Then, provided the atomic list is not empty, it steps through the Non-Atomic list, breaks each formula into antecedents and consequents and checks the list of Atomic formulae for the atoms in the antecedent. If all elements of the antecedent are in the list of atomic formulae, then the consequent is added to the list of atomics and the formula is removed from the non-atomic list.[9] This will repeat until either the non-atomic list is emptied, or a fix-point is found (ie there is no further reduction possible).

For C-Modus Ponens, there would only be a few changes. First, the test for atomic elements would have to recognize ions as atomic. Second, the procedure for searching for components must check the conclusions of the ion for the match. And finally, if the match was with an ion, the add consequent procedure must add the soft consequent to the atomic list. With these simple changes, one procedure

---

[9]It should be noted here that partial modus ponens could be performed, but it was decided for these purposes that we would require all of the elements of the antecedent to be present in the atomic list.

will be able to perform both classical and contextual Modus Ponens!

One question remains: is this solution actually going to improve the results? (ie, does this algorithm actually take less time and use less space than the savings from the reduction produces?) A simple analysis of the algorithm answers this.

Consider the case where there are *i* formulae. Then, the following possibilities exist:

non-atomic list	$k = i$	$k = i\text{-}1$	$k = i\text{-}2$	. . .	$k = 0$
atomic list	$j=0$	$j=1$	$j=2$	. . .	$j = k$

This will give (in the worst case) $\sum_{x=1}^{i-1} x = \frac{i(i-1)}{2}$.

This algorithm is quadratic in terms of the number of formulae. The tableau generator is exponential in terms of the beta operators (all operators that behave like disjunction). There is a clear savings in terms of time and space between exponential and quadratic.

Results & Discussion

Modifying Variable Size

The differences were significant. It was discovered that a savings of 20% in heap space usage was achieved. See Table 6 for details on the results.

Example Problem	Number of Statements	Heap Usage (in bytes) Before Change	Heap Usage (in bytes) After Change	Savings in Heap Usage (in bytes)	% Memory Savings
broken.frm	3	1656	1350	306	18%
clev.frm	2	224	185	39	17%
diamond.frm	5	1672	1356	316	19%
dracula1.frm	12	26632	21541	5091	19%
hiking.frm	2	672	551	121	18%
minker.frm	6	23504	19085	4419	19%
netabc.frm	11	38284	30946	7338	19%
tweety.frm	2	408	334	74	18%

Table 6. Results from the change of one type from 32-bit to 8-bit integer.

Preprocessing Results

The results from the preprocessor are equally impressive. For the network problem previously discussed, the unreduced set of formulae required 12,985 bytes of heap space. With the preprocessed set, this was reduced to 1,537 bytes. Clearly, this is

an order of magnitude difference. Appendix C contains the results from other problems run.

References

[1] Chitta Baral, Sarit Kraus, Jack Minker and V. S. Subrahmanian. Combining Multiple knowledge Bases Consisting of First-Order Theories. *Computational Intelligence*, 8(1):45-71, 1992

[2] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. *Proceedings of 3rd Annual ACM Symposium on Theory of Computing*, pages 151-157, May 1971.

[3] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer Verlag, 1990.

[4] M. A. Nait Abdallah. *The Logic of Partial Information*. Springer Verlag, 1995.

[5] M. A. Nait Abdallah and J. J. Rajnovich. *Implementing Partial Information Ionic Logic: A Prolog Experiment*. Department of Computer Science, University of Western Ontario, May 1996.

[6] J. J. Rajnovich. *Tentative Reasoning as Respectful Dialogue Tractability Issues in Applied Ionic Logic.* Department of Computer Science, University of Western Ontario, June 1996.

[7] Reiter, R. *A Logic For Default Reasoning*. Artificial Intelligence 12 (1,2), pp. 81-132, 1987.

Appendix A

The Prolog prototype goes here

Appendix B

The C++ Preprocessor goes here

Appendix C

Results from Preprocessor goes here

Appendix D

Sample problems here.