An implementation of propositional plausible logic

Author

Rock, Andrew, Billington, David

Published

2000

Conference Title

Proceedings of 23rd Australasian Computer Science Conference ACSC 2000

DOI

10.1109/ACSC.2000.824404

An Implementation of Propositional Plausible Logic

Andrew Rock and David Billington
School of Computing and Information Technology,
Nathan Campus, Griffith University,
Brisbane, Queensland 4111,Australia
{A.Rock, D.Billington}@cit.gu.edu.au

Abstract

We report the first complete implementation of propositional Plausible Logic. Plausible Logic is an extension of Defeasible Logic that overcomes the latter's inability to represent or prove disjunctions. This advantage is significant in dealing with practical applications such as the modeling of regulations. The system has a web interface, which makes it available to researchers and students everywhere. The implementation language chosen was Haskell and some advantages and consequences of this choice are discussed.

1. Introduction

Non-monotonic reasoning [1] is a branch of artificial intelligence which is concerned with the problem of deducing conclusions from incomplete or uncertain information. Worst-case complexity results [2] about the well-known non-monotonic formalisms or logics are discouraging. However, despite these poor complexity results, non-monotonic logics have been applied to, for example, software engineering [3, 4, 5]. As noted by Cholewinski et al [6], experimenting with implementations of non-monotonic logics is the most direct, and perhaps the only, way to see if non-monotonic logics are computationally useful. It is therefore very important to have implementations of non-monotonic logics, and even more important for such implementations to be widely accessible.

One of the problems with the well-known non-monotonic logics is that they were not designed to be implemented. However in the late 1980s Nute [7] introduced a non-monotonic logic, called Defeasible Logic, which was from the very beginning designed to be implemented. Substantial theoretical results for Defeasible Logic have been obtained by Billington et al [8], Billington [9], Nute [10], and Antoniou et al [11]. An implementation of Defeasible Logic can be found in chapter 11 of Covington et al [12], and some applications of Defeasible Logic are given by Covington [13, 14].

The expressivity of Defeasible Logic is limited by its inability to represent or prove disjunctions. Broadly, Plausible Logic is an extension of Defeasible Logic which overcomes these limitations. Both logics are sceptical, rather than credulous, and hence have no multiple "extensions". Both have a constructively defined proof theory which is very easy to use compared to other non-monotonic logics. Neither logic uses negation as failure, nor does any consistency checking.

Recent work on the modeling of regulations [15] has shown that the ability to accommodate disjunctions is important. Of course the increased expressivity also increases the complexity. However initial indications are that in modeling actual regulations the use of disjunctions does not destroy the polynomial nature of the model.

Plausible Logic not only accommodates disjunction, and hence arbitrary propositions, but also generalizes the idea of competing rules. This allows a more natural representation of some problems, which is important for building confidence that the representation is faithful.

After a brief overview of Plausible Logic in section 2, this paper describes, in section 3, an implementation of Plausible Logic, which is accessible from the web. The results of some performance experiments, presented in section 4, indicate that thousands of default rules are within the capabilities of this implementation.

2. An Overview of Plausible Logic

A reasoning situation is defined by a *plausible description*, which consists of three sets. The set of facts describes the known indisputable truths of the situation. Definitions and taxonomic hierarchies form typical facts. Each fact is represented as a propositional formula, or formula for short. The next two sets are sets of rules. A rule r consists of a finite set $A(r)$ of formulas on the left, called its *antecedent*, followed by an arrow in the middle, followed by a formula $c(r)$ on the right, called its *consequent*. For example, {large, common, furry, pet} $\Rightarrow$ dog, is a rule indicating that a large common furry pet is likely to be a dog. The set of plausible rules, which are indicated by the plausible arrow $\Rightarrow$, describe reasonable assumptions, typical truths, rules of thumb, and default knowledge, which may have a few exceptions.

The set of defeater rules, which are indicated by the defeater arrow $\rightsquigarrow$, disallow conclusions which are too risky, without supporting the negation of the conclusion.

These three sets enable very natural descriptions of reasoning situations. However they are difficult to compute with, so they are transformed into a single set R of rules, for which computation is easy. Later we shall denote this transformation by Θ. A *deductive rule* is a rule whose antecedent is a set of clauses, and whose consequent is a literal. A *simple rule* is a deductive rule whose antecedent is a set of literals, rather than clauses. Although all the rules in R are deductive, most are simple. Each fact is transformed into several strict simple rules, which are indicated by the strict arrow $\rightarrow$. Each plausible [resp. defeater] rule is transformed into several simple plausible [defeater] rules. To these simple rules some non-simple deductive rules are added to form R.

To this set, R, of deductive rules we add a *priority relation*, $>$, from R to R_{pd}, the set of non-strict rules. This relation must be acyclic. The pair $(R,>)$ is called a *plausible theory*. A *plausible logic* consists of a plausible theory together with several inference conditions, which we shall outline below.

We shall only be concerned with deducing or proving formulas in conjunctive normal form, which we abbreviate to cnf-formulas. Formulas not in conjunctive normal form can be transformed into equivalent cnf-formulas prior to their proofs. Cnf-formulas can be proved at three different levels of certainty or confidence. In decreasing certainty they are: the definite level, the defeasible level, and the supported level. The definite level is like classical monotonic proof in that modus ponens is used and so more information cannot defeat a previous proof. Definite proof is indicated by the tag $+\Delta$. If a formula is proved definitely then one should behave as if it is true.

Proof at the defeasible level is non-monotonic, that is more information may defeat a previous proof. However one should still behave as if a defeasibly proved formula is true, even though this may be wrong. Within the defeasible level of proof there are many different sublevels. We shall concentrate on two, indicated by the tags $+\delta$ and $+\partial$. $+\delta$-defeasibility is more cautious or sceptical than $+\partial$-defeasibility, and propagates ambiguity whereas $+\partial$-defeasibility does not.

At the supported level of proof more information may defeat a previous proof. However one should not behave as if a supported formula is true. Indeed a formula and its negation are often both supported. Within the supported level of proof there are many sublevels. We shall concentrate on one, indicated by the tag $+\int$, which is thought of as an elongated S for supported. The main, although not only, purpose of the supported level is to enable different defeasible sublevels to be defined. Another purpose, not considered in this paper, concerns postulates for governing rational change of defeasible theories.

To prove a cnf-formula f defeasibly we need to prove that all the evidence against f is defeated. Let g be a cnf-formula which is evidence against f, for instance g may be $\sim f$ (not f); and let d indicate a level of certainty, that is $d \in \{\Delta, \delta, \partial, \int\}$. Then we need to prove that $+dg$ is not provable. In a proof $-dg$ indicates that the non-provability of $+dg$ is proved. But what level of certainty is suitable? Different choices give different defeasible sublevels.

A tag is any element of $\{+\Delta, -\Delta, +\delta, -\delta, +\partial, -\partial, +\int, -\int\}$; and a tagged cnf-formula is a tag followed by a cnf-formula.

A *formal proof* or *derivation*, P, is a finite sequence, $P = (P(1), ..., P(n))$, of ordered pairs $(\mathcal{T},\pm df)$ such that $\mathcal{T}$ is a plausible theory, $\pm \in \{+,-\}$, $d \in \{\Delta,\delta,\partial,\int\}$, f is a cnf-formula and for all i in $[0..n-1]$ the following inference conditions $+\wedge$, $-\wedge$, $+\vee$, $-\vee$, $+\Delta$, $-\Delta$, $+\delta$, $-\delta$, $+\partial$, $-\partial$, $+\int$, and $-\int$ all hold. The elements of a derivation are called *lines* of the derivation. We define the *deducibility* or *provability* relation $\vdash$ between plausible theories and tagged cnf-formulas by $\mathcal{T} \vdash \pm df$ iff $(\mathcal{T}, \pm df)$ is a line in some derivation. We read $\mathcal{T} \vdash \pm df$ variously as $\pm df$ is *deducible* or *provable* from $\mathcal{T}$, or there is a derivation of $\pm df$ from $\mathcal{T}$, or f is d-provable from $\mathcal{T}$. In stating these inference conditions $\mathcal{T}=(R,>)$ will be a plausible theory, F will be a finite set of two or more clauses, q will be a literal, and d will be in $\{\Delta,\delta,\partial,\int\}$. Also it is sometimes convenient to abbreviate "there exists" by $\exists$, and "for all" by $\forall$. $P[1..i]$ denotes the first i lines of the proof P.

Consider what would justify $(\mathcal{T},+d/\wedge F)$ being placed in a derivation. A conjunction is proved only if all of its conjuncts are already proved. Hence the positive conjunctive inference condition $+\wedge$.

$+\wedge$) If $P(i+1) = (\mathcal{T},+d/\wedge F)$ then $\forall f \in F$, $(\mathcal{T},+df) \in P[1..i]$.

The inference conditions come in pairs, one positive and one negative. We shall not state the negative one as it is just the "strong" or "constructive" negative of the positive condition. That is, if $+I$ is a positive inference condition then $-I$ is the negation of $+I$ but with $(\_,+d\_) \notin P[1..i]$ replaced by $(\_,-d\_) \in P[1..i]$, and $(\_,-d\_) \notin P[1..i]$ replaced by $(\_,+d\_) \in P[1..i]$.

Before we consider what would justify $(\mathcal{T}, +d\vee F)$ being placed in a proof, we shall need the following definitions. Define $\mathcal{P}^-(F) = \{G : G \subseteq F$ and $\{\} \neq G$ and $G \neq F\}$ to be the set of non-empty proper subsets of F, and $\mathcal{P}_{\geq 1}(F) = \{G : G \subseteq F$ and $|G| \geq 1\}$ to be the set of non-empty subsets of F. Because we are considering $\vee$F we can regard F as a set of literals rather than clauses. A disjunction is proved if a non-empty proper subset of its disjuncts is already proved. Hence $+\vee.1$ below. Also if each literal in F could be proved in the theory formed by adding the complement of all the other literals in F to $\mathcal{T}$, then we could add $(\mathcal{T}, +d\vee F)$ to any proof in $\mathcal{T}$. Hence $+\vee.2$ below.

+$\bigvee$) If $P(i+1) = (\mathcal{T}, +d\bigvee F)$ then either
.1) $\exists G \in \mathcal{P}^{\cdot}(F)$, $(\mathcal{T}, +d\bigvee G) \in P[1..i]$; or
.2) $\forall f \in F$, $((\Theta(R \cup \sim(F-\{f\})), >), +df) \in P[1..i]$.

Since F is a set of literals we have $\Theta(R \cup \sim(F-\{f\})) = R \cup \{\{\} \rightarrow \sim q : q \in (F-\{f\})\}$. Furthermore +$\bigvee$.1 and +$\bigvee$.2 can be combined to give the following positive disjunctive inference condition +$\bigvee$.

+$\bigvee$) If $P(i+1) = (\mathcal{T}, +d\bigvee F)$ and F is a set of literals then
$\exists F' \in \mathcal{P}_{\geq 1}(F)\ \forall f \in F'\ ((R \cup \{\{\} \rightarrow \sim q :$
$q \in (F'-\{f\})\}, >), +df) \in P[1..i]$.

Having dealt with conjunctions and disjunctions we now consider single literals. Consider what would justify $(\mathcal{T},+\Delta q)$ being placed in a proof. Now $(\mathcal{T},+\Delta q)$ would only be justified if q was the consequent of a strict rule, r, and every member of the antecedent of r, tagged with $+\Delta$, was already in the proof. This is just modus ponens for strict rules and definite proof. Hence $+\Delta$ below. If S is any set of rules then we define $S[q] = \{r \in S : c(r) = q\}$ to be the set of all rules in S which end in q. Also S_s, S_p, and S_d denote respectively the strict, plausible, and defeater rules in S.

+Δ) If $P(i+1) = (\mathcal{T},+\Delta q)$ then
$\exists r \in R_s[q]\ \forall a \in A(r)$, $(\mathcal{T},+\Delta a) \in P[1..i]$.

Before we consider the defeasible level of proof we need to define what we mean by evidence against a literal. Rules which end with $\sim q$ are direct evidence against q, and compete with rules that end with q. However there can be indirect evidence against q. Consider the simple strict rule $\{a,q\} \rightarrow c$ and its set $\{a,q,\sim c\}$ of inconsistent literals. A pair of rules, one ending with a and the other ending with $\sim c$, are indirect evidence against q, and compete with rules ending with q. We now formally define these ideas and some other convenient terms.

A rule, r, is *supportive* iff r is strict or plausible. The consequent, $c(r)$, of a supportive rule, r, is said to be *supported by* r. Let d be in $\{\Delta, \delta, \partial, \int\}$. A rule is *d-applicable* in $\mathcal{T}$ iff each member of its antecedent is *d*-provable from $\mathcal{T}$. A rule is *d-useless* in $\mathcal{T}$ iff there is a member of its antecedent which is $(-d)$-provable from $\mathcal{T}$. A set of literals is *inconsistent with* R iff it is a member of $\mathcal{I}(R)$, where $\mathcal{I}(R) = \{A(r) \cup \{\sim c(r)\} : r$ is a simple strict rule in $R\} \cup \{\{q,\sim q\} : q$ is a literal$\}$. If C is a set of rules then we denote by $c(C) = \{c(r) : r \in C\}$ the set of consequents of the rules in C. Define $\mathcal{C}(R) = \{C \subseteq R : c(C) \in \mathcal{I}(R)$ and $|c(C)| = |C|\}$. $|B|$ is the cardinality of B. Each member of $\mathcal{C}(R)$ is a *set of competing rules*. That is, a set of competing rules is a minimal set of rules whose consequents form an inconsistent set of literals. Define $\mathcal{C}(R,q) = \{S-\{r\} : S \in \mathcal{C}(R)$ and $r \in S[q]\}$. Each member of $\mathcal{C}(R,q)$ is a set of rules which together is evidence against or contrary to q.

Consider what would justify $(\mathcal{T},+\partial q)$ being placed in a proof. Suppose $\{a,b\} \rightarrow c$ is a strict rule in a Plausible theory. This rule would have been produced by the transformation of the fact $\sim a \vee \sim b \vee c$ in a Plausible description. So if $(\mathcal{T},+\partial a)$ and $(\mathcal{T},+\partial b)$ are already in the proof then it is reasonable to add $(\mathcal{T},+\partial c)$ to the proof. Generalizing this example leads to $+\partial$.1 below, which is modus ponens for strict rules and defeasible proof.

Part $+\partial$.1 below deals with the case where q is supported by an applicable strict rule. But it is also reasonable to add $(\mathcal{T},+\partial q)$ to a proof if q is supported by a plausible rule which is ∂-applicable in $\mathcal{T}$, and every set of evidence against q is "nullified". Part $+\partial$.2 requires q to be supported by a plausible rule, r, and $+\partial$.2.1 requires r to be ∂-applicable in $\mathcal{T}$. Part $+\partial$.2.2 requires every set C of evidence against q to be "nullified". It is reasonable to regard C as nullified if a member, s, of C is either ∂-useless in $\mathcal{T}$, or is beaten (using a priority $t > s$) by a rule t which supports q and is ∂-applicable in $\mathcal{T}$. In view of $+\partial$.1, we can require t to be plausible. Part $+\partial$.2.2.1 requires s to be ∂-useless in $\mathcal{T}$. Part $+\partial$.2.2.2 requires t to support q and be plausible. Part $+\partial$.2.2.2.1 requires t to be ∂-applicable in $\mathcal{T}$, and $+\partial$.2.2.2.2 requires t to beat s.

+∂) If $P(i+1) = (\mathcal{T},+\partial q)$ then either
.1) $\exists r \in R_s[q]\ \forall a \in A(r)$, $(\mathcal{T},+\partial a) \in P[1..i]$; or
.2) $\exists r \in R_p[q]$ such that
.1) $\forall a \in A(r)$, $(\mathcal{T},+\partial a) \in P[1..i]$, and
.2) $\forall C \in \mathcal{C}(R,q)\ \exists s \in C$ such that either
.1) $\exists a \in A(s)$, $(\mathcal{T},-\partial a) \in P[1..i]$; or
.2) $\exists t \in R_p[q]$ such that
.1) $\forall a \in A(t)$, $(\mathcal{T},+\partial a) \in P[1..i]$, and
.2) $t > s$.

One way to make ∂-provability more cautious is to change $+\partial$.2.2.1 so that a member of the antecedent of s is not just provably not ∂-provable, but provably not even supported. That is instead of s being ∂-useless in $\mathcal{T}$ we now require it to be $\int$-useless in $\mathcal{T}$. Therefore the δ-defeasible sublevel is obtained from the ∂-defeasible sublevel by changing $-\partial$ in $+\partial$.2.2.1 to $-\int$, and all other occurrences of ∂ to δ.

The $+\int$ inference condition results from weakening the $+\partial$ inference condition.

+$\int$) If $P(i+1) = (\mathcal{T},+\int q)$ then either
.1) $\exists r \in R_s[q]\ \forall a \in A(r)$, $(\mathcal{T},+\int a) \in P[1..i]$; or
.2) $\exists r \in R_p[q]$ such that
.1) $\forall a \in A(r)$, $(\mathcal{T},+\int a) \in P[1..i]$, and
.2) $\forall C \in \mathcal{C}(R,q)\ \exists s \in C$ such that either
.1) $\exists a \in A(s)$, $(\mathcal{T},-\delta a) \in P[1..i]$; or
.2) not$(s > r)$.

All the details of Plausible Logic, as well as results showing that Plausible Logic is well behaved, and many examples may be obtain from the second author. The examples and all the inference conditions may be viewed at `http://www.cit.gu.edu.au/~arock/plausible/Plausible.cgi`.

3. Implementation

The first complete implementation of Plausible Logic is a system for proving formulas at any of the proof levels described above.

Since this is a tool that supports the ongoing research, the requirements of the system are that the system should be, in decreasing order of significance: (1) correct; (2) traceable; (3) easy to modify as theoretical options are explored; and (4) efficient. With these goals in mind, Haskell was chosen as the implementation language. The system has been implemented and tested using the Hugs interactive Haskell interpreter for Macintosh, Windows-95 and Solaris and using the Glasgow Haskell Compiler in Windows-95 and Solaris. The language version targeted is Haskell 1.3, but the program is compatible with the current Haskell-98.

The core of the system is definition of the data types for the representation of the literals, formulas, descriptions and theories of Plausible Logic. The system is factored into abstract data type modules that define these data types and the functions that manipulate these types.

Among these are functions for parsing and generating textual representations of these data. Aside from mapping the special symbols used in the theory to those available on a keyboard, there are no implementation-imposed restrictions placed upon the descriptions or formulas entered by users. Textual inputs are parsed using the combinator parsing strategy. See for example [16]. This entails building (in Haskell) a parser function type and combinators for building more complex parsers out of simpler ones, for example to accept items in sequence or as alternates. This coding style was used as a model for the prover components of the system.

The transformation of a Plausible Description to a Plausible Theory is implemented as a pure function exported by the module which defines a Plausible Theory. A program, Description2Theory, provides a command line tool for reading a description and writing the corresponding theory. For example, this description:

```
qu.       % Nixon is a quaker.
r.        % Nixon is a republican.
qu => d.  % Quakers are usually doves.
r => h.   % Republicans are usually hawks.
d => ~h.  % Doves are usually not hawks.
h => ~d.  % Hawks are usually not doves.
d => pa.  % Doves and hawks are ...
h => pa.  %    ... usually politically active
```

can be transformed into this theory:

```
R1: {} -> qu.
R2: {} -> r.
R3: {qu} => d.
R4: {r} => h.
R5: {d} => ~h.
R6: {h} => ~d.
R7: {d} => pa.
R8: {h} => pa.
R9: {d | h} => pa.
```

where the symbols `R1`, `R2`, ... `R9` label each rule so that priorities, such as `R5 > R4`, can be added. (Priorities need not be added to this particular example.) In the transformation, facts have become strict rules and the non-simple rule `R9` has been introduced. There will be three possible paths to a defeasible proof of `pa`: (1) prove (definitely or defeasibly) `d`; (2) prove `h`; or (3) prove `d` *or* `h`.

The most important part of the system is the prover, which attempts to prove a formula at a selected level of proof. This must be traceable, to permit the verification of the inference conditions. Pure functional programming languages are traditionally well suited to the expression of algorithms for symbolic computations, but their purity with regard to side effects has also made it difficult to do pragmatic things like input or output in the middle of a computation. Haskell's monadic I/O system made it possible to do these things at the cost of having to complicate the prover functions by making them I/O functions. I/O functions may call pure functions, but pure functions may not call I/O functions. It was found useful to employ the following abstractions as it led to a very clear expression of the inference conditions that has proved easy to verify and modify.

All of the tests required by the inference conditions can be implemented with the types:

```
data ProofStatus
       =  Yes | No | Bottom | Pending
type ThreadedTest state
       = state -> IO (ProofStatus, state)
```

The `ProofStatus` values indicate success, definite failure, loop detected and proof still in progress respectively. The type synonym `ThreadedTest` defines the template for functions to be used as test that may perform I/O and may alter some `state`. The function that attempts to prove a cnf-formula to a specified level of proof has the type:

```
prove :: PlusMinus -> ProofSymbol -> Formula
         -> ThreadedTest State
```

where

```
data PlusMinus = Plus | Minus
data ProofSymbol = PS_D | PS_da | PS_d | PS_S
```

and `State` is the actual data type threaded through the proof. The `ProofSymbol` values represent Δ, δ, ∂, and ∫.

The inference conditions combine sub-goals and priority tests with conjunctions ("and" and ∀) and disjunctions ("or" and ∃) that can be implemented with appropriate combinators: andC, fA, orC, and tE. `andC` applies two tests in sequence returning the net `ProofStatus` and the final `state`. If the first test returns a negative result or a loop is detected, the second test is not applied. A `Pending` result from either test should never occur if correctly implemented.

```
andC :: ThreadedTest s -> ThreadedTest s
        -> ThreadedTest s
andC t1 t2 s
   = do (r1,s1) <- t1 s
        case r1 of
           Yes     -> t2 s1
           No      -> return (r1, s1)
           Bottom  -> return (r1, s1)
           Pending -> error "Pending in andC"
```

`fA` applies a list of tests returning a positive result iff all the tests succeed.

```
fA :: [ThreadedTest s] -> ThreadedTest s
fA ts s
   = case ts of
        [] ->
           return (Yes, s)
        [t] ->
           t s
        (t1:t2:ts) -> do
           (r1,s1) <- t1 s
           case r1 of
              Yes     -> fA (t2:ts) s1
              No      -> return (r1,s1)
              Bottom  -> return (r1,s1)
              Pending -> error "Pending in fA"
```

The disjunctive cases are implemented similarly.

Using these combinators the $+\partial$ inference condition above is expressible as:

```
tE [fA [prove Plus PS_d a
       | a <- ant r]
   | r <- rsq q] `orC`
tE [
    fA [prove Plus PS_d a | a <- ant r] `andC`
    fA [tE [
            tE [prove Minus PS_d a
               | a <- ant s] `orC`
            tE [
                fA [ prove Plus PS_d a
                   | a <- ant t] `andC`
                t `beats` s
               | t <- rpq q]
           | s <- c]
        | c <- competers_q q]
   | r <- rpq q]
```

This expression is excerpted from the `prove` function, the type signature for which was given above, and is slightly simplified compared to the code in the actual system, where there are some strategically placed calls to a function that prints extra details to the trace. The one-to-one correspondence between the inference condition and its representation as a Haskell expression ensures that the implementation is easy to verify and easy to modify as new inference conditions are developed. As the type of the expression is `ThreadedTest`, evaluation of it may also perform I/O and a `state` value can be updated. The `prove` function prints a trace of all sub-goals. The state threaded through the evaluation of the inference conditions contains a name supply for new theories (generated by the $+\bigvee$ and $-\bigvee$ inference conditions), a counter that counts the number of times the proof function recurs, and a history of all the sub-goals so far encountered and their corresponding `ProofStatus`. The history enables some loop detection and saves re-establishing previously encountered goals.

The prover is available as a command line tool, `Prover`, which parses a Plausible Theory and attempts to prove tagged formulas, printing a trace. For example, for the theory above, the proof of $+\delta\ pa$ is shown below. (Δ, δ, ∂, and ∫ are represented as D, da, d, and S respectively.)

```
To prove: (T, +da pa)
.  To prove: (T, +da d)
      (11 lines deleted)
.  Not proved: (T, +da d)
.  To prove: (T, +da h)
      (11 lines deleted)
.  Not proved: (T, +da h)
.  To prove: (T, +da \/{d, h})
.  .  Not proved previously: (T, +da d)
.  .  Not proved previously: (T, +da h)
.  T.1 = T U [{} -> ~h]
.  .  To prove: (T.1, +da d)
.  .  .  To prove: (T.1, +da qu)
.  .  .  Proved: (T.1, +da qu)
.  .  c = [R6: {h} => ~d]
.  .  .  To prove: (T.1, -S h)
.  .  .  .  To prove: (T.1, -S r)
.  .  .  .  Not proved: (T.1, -S r)
.  .  .  c = [R5: {d} => ~h]
.  .  .  .  Loop detected: (T.1, +da d)
.  .  .  c = [{} -> ~h]
.  .  .  {} -> ~h is not > R4: {r} => h
.  .  .  c = []
.  .  .  Proved: (T.1, -S h)
.  .  Proved: (T.1, +da d)
.  T.2 = T U [{} -> ~d]
.  .  To prove: (T.2, +da h)
.  .  .  To prove: (T.2, +da r)
```

```
.  .  .  Proved: (T.2, +da r)
.  .  c = [R5: {d} => ~h]
.  .  .  To prove: (T.2, -S d)
.  .  .  .  To prove: (T.2, -S qu)
.  .  .  .  Not proved: (T.2, -S qu)
.  .  .  c = [R6: {h} => ~d]
.  .  .  .  Loop detected: (T.2, +da h)
.  .  .  c = [{} -> ~d]
.  .  .  {} -> ~d is not > R3: {qu} => d
.  .  .  c = []
.  .  .  Proved: (T.2, -S d)
.  .  Proved: (T.2, +da h)
.  Proved: (T, +da \/{d, h})
Proved: (T, +da pa)
Proved!
Number of goals: 26
CPU time for proof (s): 2.0e-2
```

This trace shows that the proof of $+\delta\ pa$ was not achievable by proving either $+\delta\ d$ or $+\delta\ h$ independently, but only by proving their disjunction, $+\delta\ d \vee h$. It also demonstrates the necessity of the history that is maintained of all sub-goals. Without it, many futile steps would be repeated, and none of the loops would have been detected. This proof only terminates because of the loop detection. The number of goals reported is the total number of times the `prove` function is called.

An alternate user interface is provided by a CGI version, `http://www.cit.gu.edu.au/~arock/plausible/Plausible.cgi`, which is also written in Haskell. The CGI version allows the user to use any of a set of pre-prepared plausible descriptions and theories or enter and work with new ones. This web-accessible version is the preferred interface for most users of the system and has links to explanatory information such as the inference conditions, the detailed syntax for all inputs and help. The present system now consists of about 4000 lines of Haskell code.

An implementation of Defeasible Logic has also been developed by copying the implementation of Plausible Logic, simplifying the data structures by basing rules on literals instead of formulas and changing the inference conditions appropriately. That this (excluding the web interface) was completed within hours is a testament to the design of this system and of Haskell. The web-accessible defeasible implementation is at `http://www.cit.gu.edu.au/~arock/defeasible/Defeasible.cgi`.

4. Results

The implementation is a success as it is currently significantly aiding the theoretical development of Plausible Logic, relieving tedium and improving accuracy. New inference conditions can be added or removed within minutes. The present definitions of the $+\delta$, $-\delta$, $+\int$, and $-\int$ inference conditions were developed with the assistance of this system.

The system's success is due to the attention paid to the requirements: (1) correctness; (2) traceability; (3) ease of modification. The fourth and least important requirement of the present system is efficiency. This is the only requirement put at risk by using a lazy functional programming language. However, performance measurements with a variety of scalable, automatically generated plausible theories has shown that: (a) The prover can operate with theories consisting of thousands of rules and priorities. This is certainly adequate for the regulations problem domain. (b) Once $\mathcal{C}(R)$ is calculated, the prover has a time complexity per sub-goal that is close to linear with respect to the number of rules in the theory. This is a consequence of storing rules and priorities in lists, and may be improved upon in the future by presorting rules and priorities into a data structure that enables more direct access. (c) The complexity of the calculation of $\mathcal{C}(R)$ is non-trivially dependent on the details of the plausible theory and is in some cases the most significant contributor to the overall time taken to perform proofs. No obvious optimizations have been found yet, but they a worth seeking. In the absence of disjunctions, $\mathcal{C}(R)$ need only be computed once per theory, and the cost can be amortized over many proofs. (d) The number of sub-goals is dependent on the theory and the formula to be proved. The worst cases involve the $+\vee$ and $-\vee$ inference conditions which may require 2^n sub-goals, where n is the number of disjuncts, with the added penalty that for the majority of cases $\mathcal{C}(R)$ must be recomputed for the augmented theory required. Fortunately large numbers of disjuncts are rare in practice.

Assertion (b) that the prover has a time complexity per sub-goal that is close to linear with respect to the number of rules in the theory means that there is no unexpected performance penalty in using Haskell to implement this system. As evidence for this assertion, we report the results of experiments with a scalable test theory that has a cascade of rules for $i \in [0..n]$, $\{\} \Rightarrow a_i$ and $\{a_{i+1}\} \Rightarrow \sim a_i$ with a priority for odd i that the latter rule is superior. A proof of $+\partial\ a_0$ exercises every rule and priority in the theory. In figure 1 we plot the time taken in seconds for a proof of $+\partial\ a_0$ divided by the number of goals (the intrinsic size of the proof) versus the number of rules in the theory. It shows a close to linear growth in the time taken per goal as the size of the theory increases. As stated above, this is a consequence of storing rules and priorities in lists which must be sequentially searched many times. This cost is our first and easiest target for optimization. The lists can easily be replaced with arrays.

5. Conclusion

The present system is the first complete implementation of propositional Plausible Logic. It is and will be a valuable tool for the theoretical development of the logic. The web accessibility makes it available as a resource for students and researchers everywhere. The choice of Haskell as the implementation language has

enabled the implementation to be swift, clear, concise and easy to maintain. Planned further work on this implementation includes refining and optimizing the time and space complexity of the propositional implementation and then to extend it by adding variables.

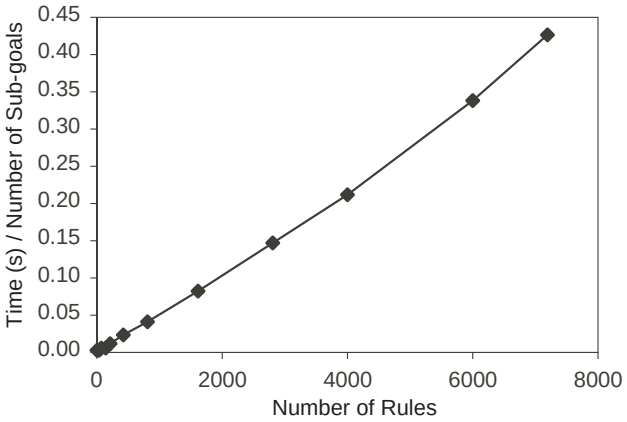

Figure 1: CPU time (s) per sub-goal as a function of the number of rules in a plausible theory. The proof in each case used all of the rules and one priority for every four rules.

References

[1] Antoniou, G., *Nonmonotonic Reasoning*, MIT Press, Cambridge, Mass., (1997).

[2] Gotlob, G., "Complexity Results for Nonmonotonic Logics", *Journal of Logic and Computation*, vol. 2, pp. 397-425, (1992).

[3] Antoniou, G., "On the Role of Nonmonotonic Representations in Requirements Engineering", *International Journal of Software Engineering and Knowledge Engineering*, vol. 8, pp. 385-399, (1998).

[4] Luqi, and D.E. Cooke, "How to Combine Nonmonotonic Logic and Rapid Prototyping to Help Maintain Software", *International Journal on Software Engineering and Knowledge Engineering*, vol. 5, pp. 89-118, (1995).

[5] Zowghi, D., A.K. Ghose, and P. Peppas, "A Framework for Reasoning about Requirements Evolution", *Proceedings of the Fourth Pacific Rim International Conference on Artificial Intelligence*, Springer, LNAI 1114, pp. 157-168, (1996).

[6] Cholewinski, P, V.W. Marek, A. Mikitiuk, and M. Truszczynski, "Experimenting with Nonmonotonic Reasoning", *Proceedings of the 12th International Conference on Logic Programming*, MIT Press, (1995).

[7] Nute, D., "Defeasible Reasoning: A Philosophical Analysis in Prolog", *Aspects of Artificial Intelligence*, J.H. Fetzer (ed.), Kluwer Academic, pp. 251-288, (1988).

[8] Billington, D., K. De Coster, and D. Nute, "A Modular Translation from Defeasible Nets to Defeasible Logics", *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 2, pp. 151-177, (1990).

[9] Billington, D., "Defeasible Logic is Stable", *Journal of Logic and Computation*, vol. 3, pp. 379-400, (1993).

[10] Nute, D., "A Decidable Quantified Defeasible Logic", *Logic, Methodology and Philosophy of Science IX*, D. Prawitz, B. Skyrms, and D. Weasterstahl (Eds.), Elsevier Science B.V., pp. 263-284, (1994).

[11] Antoniou, G., D. Billington, and M. J. Maher, "Normal Forms for Defeasible Logic", *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pp. 160-174, (1998).

[12] Covington, M.A., D. Nute, and A. Vellino, *Prolog Programming in Depth*, Prentice Hall, (1997).

[13] Covington, M.A., "Defeasible Logic on an Embedded Microcontroller", *Proceedings of the Tenth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems,* (1997). Also to appear in *Applied Intelligence*.

[14] Covington, M.A., "Logical Control of an Elevator with Defeasible Logic", *IEEE Transactions on Automatic Control*, to appear, (2000).

[15] Antoniou, G., D. Billington, and M. J. Maher, "On the Analysis of Regulations using Defeasible Rules", *Proceedings of the 32nd Hawaii International Conference on Systems Science*, IEEE Press, (1999).

[16] Hutton, G., "Higher-Order Functions for Parsing", *Journal of Functional Programming*, vol. 2, pp. 323-343, (1992).