

# Efficient Defeasible Reasoning Systems

M.J. Maher, A. Rock, G. Antoniou, D. Billington, T. Miller

School of Computing & Information Technology

Griffith University

QLD 4111, Australia

{mjm,arock,ga,db,xerxes}@cit.gu.edu.au

## Abstract

*For many years, the non-monotonic reasoning community has focussed on highly expressive logics. Such logics have turned out to be computationally expensive, and have given little support to the practical use of non-monotonic reasoning. In this work we discuss defeasible logic, a less-expressive but more efficient non-monotonic logic. We report on two new implemented systems for defeasible logic: a query answering system employing a backward-chaining approach, and a forward-chaining implementation that computes all conclusions. Our experimental evaluation demonstrates that the systems can deal with large theories (up to hundreds of thousands of rules). We show that defeasible logic has linear complexity, which contrasts markedly with most other non-monotonic logics and helps to explain the impressive experimental results. We believe that defeasible logic, with its efficiency and simplicity, is a good candidate to be used as a modelling language for practical applications, including modelling of regulations and business rules.*

## 1 Introduction

Nonmonotonic reasoning was originally introduced to address certain aspects of commonsense reasoning, mainly reasoning with incomplete information. The motivation was to be able to “jump to conclusions” in cases where not all necessary information is available, yet certain plausible assumptions can be made.

A great amount of research has been conducted in non-monotonic reasoning [25, 1]. Despite many conceptual advances some negative aspects have become apparent. The first one comes from the computational complexity analysis: it turns out that most nonmonotonic reasoning systems have high computational complexity [21, 15] which seems to be contrary to the original motivation of “jumping to conclusions”. The second negative observation is the failure of

mainstream nonmonotonic systems to find their way into applications. Only quite recently did applications in reasoning about action [14, 22, 31] and the solution of NP-hard problems [27] appear.

Our paper is not concerned with the classes of nonmonotonic reasoning approaches mentioned above. Rather, it focuses on another research stream within nonmonotonic reasoning – an often neglected one – which is prepared to sacrifice expressive power in favour of simplicity, efficiency and easy implementability. Defeasible logic [28, 29] is an early such logic, and the one we will be dealing with. It is closely related [7] to inheritance networks [19], another formalism with an efficient implementation [32]. Recently several other systems in this class were proposed, for example Courteous Logic Programs [17] and sceptical Logic Programming without Negation as Failure [11]. There has been recent evidence that this is a practicable approach [26].

Defeasible logic is a sceptical nonmonotonic reasoning system based on rules and a priority relation between rules that is used to resolve conflicts among rules, where possible. The logic has been recently subjected to a thorough theoretical analysis by our research group. Results include representational properties and properties of the proof theory [3, 23], and establishing its relationship with negation-as-failure [24], argumentation [16] and other logics [6].

Also, we have embarked on investigating its applicability to the modelling and analysis of regulations and business rules [4]. We believe that defeasible logic is suitable for such practical applications because (i) its basic concepts (simple rules and priorities) can be easily understood by non-experts, and (ii) because the logic is sufficiently efficient. More generally, we believe that these kinds of non-monotonic approaches can be used as simple and efficient modelling languages for situations where one needs to deal quickly and flexibly with incomplete and conflicting information (a point that is, independently, propagated by Grosz [18]). Electronic commerce, where decisions (e.g. on pricing or the granting of credit) need to be made in real time 24 hours a day, is a particularly promising domain [5].

The contribution of this paper is to study and demonstrate the efficiency of defeasible logic. In particular we describe two implemented systems: one for query evaluation, and one that computes all conclusions of a given theory. For each of the systems we describe their design, and provide a summary of their experimental evaluation. We also show that defeasible logic has linear complexity (in the number of symbols in a defeasible theory).

## 2 Defeasible Logic

We begin by presenting the basic ingredients of defeasible logic<sup>1</sup>. A defeasible theory contains five different kinds of knowledge: facts, strict rules, defeasible rules, defeaters, and a superiority relation.

*Facts* are indisputable statements, for example, “Tweety is an emu”. In the propositional logic, this might be expressed as *emu*.

*Strict rules* are rules in the classical sense: whenever the premises are indisputable (e.g. facts) then so is the conclusion. An example of a strict rule is “Emus are birds”. Written formally:

$$emu \rightarrow bird$$

*Defeasible rules* are rules that can be defeated by contrary evidence. An example of such a rule is “Birds typically fly”; written formally:

$$bird \Rightarrow flies.$$

The idea is that if we know that something is a bird, then we may conclude that it flies, *unless there is other evidence suggesting that it may not fly*.

*Defeaters* are rules that cannot be used to draw any conclusions. Their only use is to prevent some conclusions. In other words, they are used to defeat some defeasible rules by producing evidence to the contrary. An example is “If an animal is heavy then it might not be able to fly”. Formally:

$$heavy \rightsquigarrow \neg flies$$

The main point is that the information that an animal is heavy is not sufficient evidence to conclude that it doesn’t fly. It is only evidence that the animal *may* not be able to fly. In other words, we don’t wish to conclude  $\neg flies$  if *heavy*, we simply want to prevent a conclusion *flies*.

The *superiority relation* among rules is used to define priorities among rules, that is, where one rule may override the conclusion of another rule. For example, given the defeasible rules

$$\begin{aligned} r : \quad & bird \Rightarrow flies \\ r' : \quad & brokenWing \Rightarrow \neg flies \end{aligned}$$

which contradict one another, no conclusive decision can be made about whether a bird with a broken wing can fly. But if we introduce a superiority relation  $>$  with  $r' > r$ , then we can indeed conclude that the bird cannot fly. It turns out that we only need to define the superiority relation over rules with contradictory conclusions.

It is not possible, in an extended abstract, to give a complete formal description of the logic. However, we hope to give enough information about the logic to make the discussion of the implementations intelligible. The full version of the paper will contain more details. For more thorough treatments.

A *rule*  $r$  consists of its *antecedent* (or *body*)  $A(r)$  which is a finite set of literals, an arrow, and its *consequent* (or *head*)  $C(r)$  which is a literal. Given a set  $R$  of rules, we denote the set of all strict rules in  $R$  by  $R_s$ , the set of strict and defeasible rules in  $R$  by  $R_{s,d}$ , the set of defeasible rules in  $R$  by  $R_d$ , and the set of defeaters in  $R$  by  $R_{dft}$ .  $R[q]$  denotes the set of rules in  $R$  with consequent  $q$ . If  $q$  is a literal,  $\sim q$  denotes the complementary literal (if  $q$  is a positive literal  $p$  then  $\sim q$  is  $\neg p$ ; and if  $q$  is  $\neg p$ , then  $\sim q$  is  $p$ ).

A *defeasible theory*  $D$  is a triple  $(F, R, >)$  where  $F$  is a finite set of literals (called *facts*),  $R$  a finite set of rules, and  $>$  a superiority relation on  $R$ .

A *conclusion* of  $D$  is a tagged literal and can have one of the following four forms:

$+\Delta q$ , which is intended to mean that  $q$  is definitely provable in  $D$  (i.e., using only facts and strict rules).

$-\Delta q$ , which is intended to mean that we have proved that  $q$  is not definitely provable in  $D$ .

$+\partial q$ , which is intended to mean that  $q$  is defeasibly provable in  $D$ .

$-\partial q$  which is intended to mean that we have proved that  $q$  is not defeasibly provable in  $D$ .

Provability is based on the concept of a *derivation* (or proof) in  $D = (F, R, >)$ . A derivation is a finite sequence  $P = (P(1), \dots, P(n))$  of tagged literals constructed by inference rules. There are four inference rules (corresponding to the four kinds of conclusion) that specify how a derivation can be extended. Here we briefly state the inference rules for the two positive conclusions.  $(P(1..i))$  denotes the initial part of the sequence  $P$  of length  $i$ :

$$\begin{aligned} +\Delta: \text{ We may append } P(i+1) = +\Delta q \text{ if either} \\ q \in F \text{ or} \\ \exists r \in R_s[q] \forall a \in A(r) : +\Delta a \in P(1..i) \end{aligned}$$

This means, to prove  $+\Delta q$  we need to establish a proof for  $q$  using facts and strict rules only. This is a deduction in the classical sense – no proofs for the negation of  $q$  need

<sup>1</sup>In this paper we restrict attention to propositional defeasible logic.

to be considered (in contrast to defeasible provability below, where opposing chains of reasoning must be taken into account, too).

- $+\partial$ : We may append  $P(i + 1) = +\partial q$  if either
- (1)  $+\Delta q \in P(1..i)$  or
  - (2) (2.1)  $\exists r \in R_{sd}[q] \forall a \in A(r) : +\partial a \in P(1..i)$  and  
 (2.2)  $-\Delta \sim q \in P(1..i)$  and  
 (2.3)  $\forall s \in R[\sim q]$  either
    - (2.3.1)  $\exists a \in A(s) : -\partial a \in P(1..i)$  or
    - (2.3.2)  $\exists t \in R_{sd}[q]$  such that  
 $\forall a \in A(t) : +\partial a \in P(1..i)$  and  $t > s$

Let us work through this inference rule. To show that  $q$  is provable defeasibly we have two choices: (1) We show that  $q$  is already definitely provable; or (2) we need to argue using the defeasible part of  $D$  as well. In particular, we require that there must be a strict or defeasible rule with head  $q$  which can be applied (2.1). But now we need to consider possible ‘‘attacks’’, that is, reasoning chains in support of  $\sim q$ . To be more specific: to prove  $q$  defeasibly we must show that  $\sim q$  is not definitely provable (2.2). Also (2.3) we must consider the set of all rules which are not known to be inapplicable and which have head  $\sim q$  (note that here we consider defeaters, too, whereas they could not be used to support the conclusion  $q$ ; this is in line with the motivation of defeaters given earlier). Essentially each such rule  $s$  attacks the conclusion  $q$ . For  $q$  to be provable, each such rule  $s$  must be counterattacked by a rule  $t$  with head  $q$  with the following properties: (i)  $t$  must be applicable at this point, and (ii)  $t$  must be stronger than  $s$ . Thus each attack on the conclusion  $q$  must be counterattacked by a stronger rule.

### 3 A System for Query Evaluation

The query answering system, *Deimos*, is a suite of tools that supports our ongoing research in defeasible logic. The centre of the system is the prover. It implements a backward-chaining theorem prover for defeasible logic based almost directly on the inference rules, such as that in Section 2. The system also includes a program that generates the scalable theories used as test cases in this paper. It is accessible through a command line interface and a CGI interface at <http://www.cit.gu.edu.au/~arock/defeasible/Defeasible.cgi>. The system is implemented in about 4000 lines of Haskell<sup>2</sup>.

*Deimos* has been designed primarily for flexibility (so that we can explore variants of defeasible logic) and traceability (so that we can understand the computational be-

<sup>2</sup>Much of this code, along with the design strategy, is common to the *Phobos* query answering system for Plausible logic[30] which has been developed in parallel with *Deimos*.

haviour of the logics and their implementations). Nevertheless, significant effort has been expended to make the system reasonably efficient.

The present implementation performs a depth-first search, with memoization and loop-checking, for a proof in defeasible logic. Memoization allows the system to recognise that a conclusion has already been proved (or disproved), while loop-checking also detects when a conclusion occurs twice in a branch of the search tree. Loop-checking is necessary for the depth-first search to be complete, whereas memoization is purely a matter of efficiency. Loop-checking and memoization are implemented using a balanced binary tree of data.

A proof is performed by a pair of mutually recursive functions  $|--$  and  $|-$ . The former defines the inference rules, and the latter performs any state modifications (for example, updating the record of conclusions proved and I/O).

The function  $|--$  is defined by an equation for each inference rule in defeasible logic. Each equation is defined in terms of logic combinators ( $\&\&\&$ ,  $|||$ ,  $fA$  and  $tE$ ) and functions such as  $rsdq$  ( $rsdq\ t\ q$  returns  $R_{sd}[q]$ ), and  $beats$  ( $beats\ t\ u\ s$  returns  $u > s$ ). The  $+\partial$  inference rule above is expressed as:

```
(|--) t (Plus PS_d q) (|-) =
  t |- Plus PS_D q |||
  tE (rsdq t q) (\r ->
    fA (ants t r)
      (\a -> t |- Plus PS_d a)) &&&
  t |- Minus PS_D (neg q) &&&
  fA (rq t (neg q)) (\s ->
    tE (ants t s)
      (\a -> t |- Minus PS_d a) |||
    tE (rsdq t q) (\u ->
      fA (ants t u)
        (\a -> t |- Plus PS_d a) &&&
        beats t u s))
```

The one-to-one correspondence between the inference rule and its representation as a Haskell expression ensures that the implementation is easy to verify and easy to modify as new inference rules are developed for variants of defeasible logic. The system provides different definitions of  $|-$  so that memoization and/or loop-checking can be turned off. Similarly, only the logic combinators, which specify depth-first search, need to be redefined to specify other search strategies.

In fact, there are several searches required to prove  $+\partial p$ . First there is the search for a (strict or defeasible) rule for  $p$  whose body is proved defeasibly. Then there is the search for a proof of  $-\Delta \sim p$ . Then, a search for a rule for  $\sim p$  whose body is proved defeasibly, and, finally, a search for a rule for  $p$  that will overrule the rule for  $\sim p$ . The order

```

initialize  $S$ 
 $K = \emptyset$ 

while (  $S \neq \emptyset$  )
  choose  $s \in S$ 
  add  $s$  to  $K$ 
  case  $s$  of
     $+\partial p$ :
      delete all occurrences of  $p$  in rule bodies
      whenever a body with head  $h$  becomes empty
        record  $+\sigma h$ 
        CheckInference(  $+\sigma h, S$  )
     $-\partial p$ :
      delete all rules where  $p$  occurs in the body
      whenever there are no more rules for a literal  $h$ 
        record  $-\sigma h$ 
        CheckInference(  $-\sigma h, S$  )
  end case
end while

```

**Figure 1. All conclusions algorithm**

of these searches follows the order in the presentation of the  $+\partial$  inference rule. While this ordering is not always the best – it is not possible to find a good ordering a priori – the use of memoization and loop-checking minimize bad effects of the search order.

A defeasible logic theory is stored in a data structure containing: a balanced tree and array for mapping from textual literal names to integral representations and back; an array of booleans indexed by the literals to represent the facts; parallel arrays to represent the consequent of, body of, and set of indices of rules beaten by, each rule; and arrays, indexed by head, of the indices of the rules  $R_s[q]$ ,  $R_{sd}[q]$  and  $R[q]$ . Access to the lists of rule indices required by any of the inference rules can be gained in constant time; facts can be tested in constant time and priorities can be tested in  $O(\log n)$  time where  $n$  is the number of rules that a rule beats ( $n$  will usually be small).

## 4 A System for Computing All Conclusions

The system that computes all conclusions, Delores, is based on forward chaining, but this is only for the positive conclusions. The negative conclusions are derived by a dual process. The system is implemented in about 4,000 lines of C. We begin by presenting the algorithm for defeasible theories containing only defeasible rules (i.e. without strict rules, defeaters or superiority relation).

In the algorithm presented in Figure 1,  $p$  ranges over literals and  $s$  ranges over conclusions.  $K$  and  $S$  are sets

of conclusions.  $K$  accumulates the set of conclusions that have been proved, while  $S$  holds those proven conclusions that have not yet been used to establish more conclusions.

To begin the algorithm we initialize the set  $S$  with those conclusions that can immediately be established: all facts are provable, while those literals with no rules for them are unprovable. Thus  $S$  contains  $+\partial f$  for each fact  $f$  and  $-\partial p$  for each proposition  $p$  not appearing in the head of a rule.

The algorithm proceeds by modifying the rules in the theory. When inferring positive consequences, the algorithm is somewhat similar to unit resolution for definite clauses in classical logic: when an atom is proved, it can be eliminated from the bodies of all other definite clauses. In this case, when a literal is established defeasibly it can be deleted from the body of all rules. Similarly, when it is established that a literal  $p$  cannot be proved then those rules which have  $p$  as a pre-condition cannot be used to prove the head, and so they can be deleted.

However, in inferring a positive conclusion  $+\partial p$ , defeasible provability is complicated, in comparison to definite clauses, by the need to consider rules for  $\sim p$ . We first define notation for the “uncomplicated” inference and then relate it to defeasible provability. Let  $+\sigma q$  denote that  $\exists r \in R_{sd}[q] \forall a \in A(r) : +\partial a$  and  $-\sigma q$  denote that  $\forall r \in R_{sd}[q] \exists a \in A(r) : -\partial a$ . Thus we can conclude  $+\sigma q$  precisely when the body of a rule for  $q$  becomes empty, and  $-\sigma q$  precisely when there are no more rules for  $q$ .

If we examine the inference rule for  $+\partial$ , in the absence of defeaters and superiority relation it can be simplified to

$$+\partial p \text{ iff } +\Delta p \text{ or } (+\sigma p \text{ and } -\Delta \sim p \text{ and } -\sigma \sim p)$$

Similarly, we can simplify the inference rule for  $-\partial$  to

$$-\partial p \text{ iff } -\Delta p \text{ and } (-\sigma p \text{ or } +\Delta \sim p \text{ or } +\sigma \sim p)$$

Each time a statement such as  $+\sigma p$  is inferred by the system the statement is *recorded* and we check to see whether either of the above simplified inference rules can be applied, using all recorded information. This task is performed by *CheckInference*, which will add either  $+\partial p$  or  $-\partial p$ , if justified, to the set  $S$ <sup>3</sup>.

The key to an efficient implementation of this algorithm is the data structure used to represent the rules. It is exemplified (albeit incompletely) in Figure 2 for the theory

$$\begin{array}{lcl}
r_1 : & b, c, d \Rightarrow & a \\
r_2 : & \neg b, d, \neg e \Rightarrow & a \\
r_3 : & d, \neg e \Rightarrow & a
\end{array}$$

Each rule body is represented as a doubly-linked list (horizontal arrows in Figure 2). Furthermore, for each literal  $p$  there are doubly-linked lists of the occurrences of  $p$

<sup>3</sup>Note that defeasible logic will never infer both  $+\partial p$  and  $-\partial p$  [8].

in the bodies of rules (diagonal arrows). For each literal  $p$ , there is a doubly-linked list of rules with head  $p$  (dashed arrows). Each literal occurrence has a link to the record for the rule it occurs in (not shown in Figure 2).

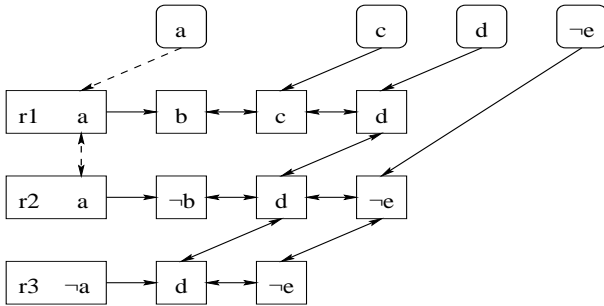


Figure 2. Data Structure for Rules

This data structure allows the deletion of literals and rules in time proportional to the number of literals deleted. Furthermore, we can detect in constant time whether a literal deleted was the only literal in that body, and whether a rule deleted with head  $h$  was the only rule for  $h$ . Each literal occurrence is deleted at most once, and the test for empty body is made at most once per deletion. Similarly, each rule is deleted at most once, and the test for no more rules is made once per deletion. Thus the cost of the algorithm is  $O(N)$ , where  $N$  is the number of literal occurrences in  $D$ .

This algorithm, for positive conclusions, is similar to the bottom-up linear algorithm for determining satisfiability of Horn clauses of Dowling and Gallier [12, 13]. One difference is in the data structures: the Dowling-Gallier algorithm keeps a count of the number of atoms in the body of a rule, rather than keep track of the body. The latter results in greater memory usage, but allows us to reconstruct the residue of the computation: the simplified rules that remain. This residue is useful in understanding the behaviour of a theory.

When we admit strict rules, the algorithm is complicated by

- the need to consider four kinds of conclusions, instead of two;
- the relationship between  $+\Delta$  and  $+\partial$ , and  $-\Delta$  and  $-\partial$ ; and
- the fact that strict rules can be used for both definite and defeasible reasoning.

The resulting algorithm has the same structure as Figure 1 but more details. The data structure also retains the same structure, but there are more lists and strict rules are represented twice.

The algorithm extends to general defeasible theories through the use of a pre-processing transformation that

eliminates all uses of defeaters and superiority relation. The transformation was designed to provide incremental transformation of defeasible theories, and systematically uses new atoms and new defeasible rules to simulate the eliminated features. Presentation of the transformation occupies too much space to give it here. Parts of the transformation were presented in [3]. A full treatment of the transformation, including proofs of correctness and other properties, is presented in [2].

The transformation can increase the size of the theory by at most a factor of 12. Furthermore, the time taken to produce the transformed theory is linear in the size of the input theory. Consequently, the implementation of full defeasible logic by first transforming the input theory to a theory without defeaters and superiority statements, and then applying an algorithm like Figure 1 to the transformed theory provides a linear implementation of defeasible logic.

**Theorem 4.1** *The consequences of a defeasible theory  $D$  can be computed in  $O(N)$  time, where  $N$  is the number of symbols in  $D$ .*

A more complete argument of correctness and complexity analysis for the full algorithm will be presented elsewhere.

## 5 d-Prolog

In addition to the two implementations described above, there is another implementation of defeasible logic. d-Prolog [10] is a query-answering interpreter for defeasible logic implemented in about 300 lines of Prolog. Its intended input is mostly small, non-recursive inheritance problems. The strict rules are implemented directly as Prolog rules. Thus when we time the execution of a theory with only strict rules, we are measuring the speed of the underlying Prolog system. The search for a defeasible proof follows the same pattern as used in *Deimos*, but with no loop-checking or memoization.

The interpreter is designed to allow experimentation, and includes an implicit definition of the superiority relation in terms of specificity; that capability was disabled for our experiments. d-Prolog also treats strict rules slightly differently from the formulation of defeasible logic that we use, and it has been modified so that it implements the same semantics as *Deimos* and *Delores*.

Unfortunately, the d-Prolog implementation of defeasible logic is flawed. The interpreter follows the Prolog computation rule and consequently has the same incompleteness that Prolog has.

This behaviour will be visible in some of the experiments. However, most of the experiments do not contain cyclic dependencies among literals so that for these experiments the flaw has no effect.

## 6 Experimental Evaluation

In the experiments, we ran d-Prolog compiled to Sicstus Prolog 3.7 fastcode, using the default memory allocation. The times presented in the experiments are those measured by the Sicstus Prolog *statistics* built-in. When timing several experiments in the same Prolog session the first experiment consistently took significantly longer than later identical experiments. In our data we have omitted the first timing in a session.

*Deimos* is compiled using the Glasgow Haskell Compiler 4.04, with optimization flags, and times are measured using the `CPUTime` library. The system begins with a stack space of 40M and a heap of 100M. The lazy execution strategy of Haskell can make timing of just part of an execution difficult. Care has been taken to force the complete evaluation of the theory data structure before starting timing of a proof. This avoids mis-allocation of work that could be deferred to run-time by the laziness of the language.

Delores is written in C and compiled using `gcc` without optimization flags. In the experiments, the atom and rule symbol tables have size 1,000,000. Memory is allocated in chunks of 65536 bytes. This system is still under development: the implementation of the basic algorithm for strict and defeasible rules is complete, but the implementation of the pre-processing transformation still requires tuning. For this reason we chose to measure both the full system and the partial system that omits the transformation. Furthermore, the generation of artificial names by the transformation interacts badly with the current hash function, causing congestion in the hash tables. This does not affect the time required to perform inferences, but it substantially affects the time to load a theory. This limited some of the experiments we ran.

All experiments were performed on the same lightly loaded Sun Ultra 2. Each timing datum is the mean of several executions. There was no substantial variation among the executions, except as noted.

### 6.1 Design of Experiments

Our initial experiments are on parameterized problems designed to test different aspects of the implementations. We have not yet been able to create realistic random problems. Since defeasible logic has linear complexity, the approach of [9], which maps NP-complete graph problems to default rules, is not applicable. In the experiments we focus on defeasible inference.

The first group of problems test only undisputed inferences. In `empty()` there are no rules. In `chain(n)`,  $a_0$  is at the end of a chain of  $n$  rules  $a_{i+1} \Rightarrow a_i$ . In `circle(n)`,  $a_0$  is part of a circle of  $n$  rules  $a_{i+1 \bmod n} \Rightarrow a_i$ . `chains(n)` and `circles(n)` are versions of the above using strict rules.

In `tree(n,k)`,  $a_0$  is the root of a  $k$ -branching tree of depth  $n$  in which every literal occurs once. In `dag(n,k)`,  $a_0$  is the root of a  $k$ -branching tree of depth  $nk$  in which every literal occurs  $k$  times.

In `levels-(n)`, there is a cascade of  $n$  disputed conclusions: there are rules  $true \Rightarrow a_i$  and  $a_{i+1} \Rightarrow \neg a_i$ , for  $0 \leq i < n$ . In `levels(n)`, there are, in addition, superiority statements stating that, for odd  $i$ , the latter rule is superior. In `teams(n)`, every literal is disputed, with two rules for  $a_i$  and two rules for  $\neg a_i$ , and the rules for  $a_i$  are superior to the rules for  $\neg a_i$ . This situation is repeated recursively to a depth  $n$ . All the above problems involve only defeasible rules. In `mix(m,n,k)`, there are  $m$  defeasible rules for  $a_0$  and  $m$  defeaters against  $a_0$ , where each rule has  $n$  atoms in its body. Each atom can be established by a chain of strict rules of length  $k$ .

### 6.2 Experimental Results

We use as a measure of problem size the total number of non-label, non-arrow symbols in the theory, that is, the sum of the number of literal occurrences and superiority statements. The tables describe the time (in cpu seconds) required to find the appropriate conclusion for  $a_0$ . Note that Delores finds conclusions for all literals, not simply  $a_0$ , whereas *Deimos* and d-Prolog terminate when  $a_0$  is proved. However, our experiments are designed to exercise all rules and literals, so that, for these experiments, *Deimos* will have conclusions memoized for all atoms.

The times for *Deimos* include time spent garbage collecting, whereas the times for d-Prolog do not. This adds significantly to the time in problems where the space usage approaches the heap space allocated to the Haskell run-time environment.

In the tables,  $\infty$  denotes that the system will not terminate, \* denotes that the default memory allocation of Sicstus Prolog was exhausted, - denotes that the experiment was not performed because the runtime required was excessive, ? denotes that the experiment could not be performed. The times recorded refer only to the computation time, and do not include the time for loading the theory.

We begin by addressing the two query-answering implementations.

Comparison of the behaviour of d-Prolog on strict and defeasible versions of the problems in the first group demonstrates the expected overhead of interpretation wrt direct execution. Nevertheless, d-Prolog is substantially more efficient than *Deimos* when there are no disputing rules (as in `chain(n)` and `tree(n,k)`). However, when disputing rules are common (as in `levels-(n)`, `levels(n)` and `teams(n)`) d-Prolog performs badly, with time growing exponentially in the problem size. In the table we only provide the data on this behaviour for `levels-(n)`. The exponential behaviour can

	Problem Size	<i>Deimos</i>	d-Prolog	Delores	Delores (partial)
<b>empty()</b>	0	0.0	0.0	0.18	0.18
<b>chains(n)</b>					
n = 25,000	50,001	3.12	0.10	8.61	0.50
n = 50,000	100,001	6.50	0.19	-	0.82
n = 75,000	150,001	10.47	0.28	-	1.11
n = 100,000	200,001	14.49	0.38	-	1.47
<b>circles(n)</b>					
n = 25,000	50,000	3.32	∞	7.98	0.24
n = 50,000	100,000	7.39	∞	-	0.30
n = 75,000	150,000	10.63	∞	-	0.35
n = 100,000	200,000	14.43	∞	-	0.40
<b>chain(n)</b>					
n = 25,000	50,001	17.54	3.22	6.38	0.24
n = 50,000	100,001	38.48	6.48	62.08	0.30
n = 75,000	150,001	57.28	9.63	-	0.36
n = 100,000	200,001	82.03	12.54	-	0.41
<b>circle(n)</b>					
n = 25,000	50,000	8.55	∞	6.03	0.24
n = 50,000	100,000	17.87	∞	-	0.30
n = 75,000	150,000	27.75	∞	-	0.36
n = 100,000	200,000	42.42	∞	-	0.41
<b>tree(n,k)</b>					
n=8, k=3	19,681	5.24	0.61	0.38	0.24
n=9, k=3	59,047	16.62	1.89	0.81	0.34
n=10, k=3	177,145	55.41	5.19	22.70	0.64
<b>dag(n,k)</b>					
n=3, k=3	43	0.00	0.06	0.19	0.19
n=4, k=4	89	0.05	8.80	0.19	0.19
n=100, k=10	11,021	1.06	*	0.22	0.19
n=1,000, k=10	110,021	11.60	*	0.50	0.20
n=100, k=40	164,041	9.73	*	0.31	0.20

Figure 3. Undisputed inferences

be attributed to a duplication of work – for example, in (2.1) and (2.3.3) of the  $+\partial$  inference rule – that is repeated recursively. *Deimos* avoids this duplication through memoization.

d-Prolog shows its incompleteness when it loops on **circle(n)**. d-Prolog was unable to execute **mix(m,n,k)**, due to an incompatibility with the underlying Prolog system.

For the problems under discussion, *Deimos* exercises all rules. In these and other experiments, when space is not an issue, the time for *Deimos* grows at  $O(N \log N)$ , as expected (the loop-checking contributes the  $\log N$  factor). For some of the problems, like **chain()** and **levels()**, the loop-checking and memoization of *Deimos* has no effect. In these cases, a comparison of executions with and without these features also reveals the  $\log N$  factor. For problems of size about 200,000, memoization increased time by a factor of about 10. In problems **dag()** and **teams()** the use of memoization, without loop-checking resulted in a small, but significant speed-up over the loop-checking implementation. All the same, loop-checking is necessary for completeness and the advantage of memoization has already been seen, so these time overheads are acceptable. The memory require-

	Problem Size	<i>Deimos</i>	d-Prolog	Delores
<b>levels-(n)</b>				
n=10	67	0.01	1.61	0.19
n=11	73	0.01	3.07	0.19
n=12	79	0.01	6.24	0.19
n=13	85	0.01	12.80	0.19
n=14	91	0.01	26.17	0.19
n=15	97	0.01	53.46	0.19
n=20	127	0.01	-	0.19
n=1,000	6,007	1.37	-	0.30
n=5,000	30,007	6.47	-	0.78
n=10,000	60,007	14.40	-	-
n=30,000	180,007	46.44	-	-
<b>levels(n)</b>				
n=10	78	0.01	1.70	0.19
n=1,000	7,008	1.35	-	0.29
n=5,000	35,008	6.48	-	0.87
n=10,000	70,008	14.14	-	-
n=30,000	210,008	48.67	-	-
<b>teams(n)</b>				
n=3	594	0.05	-	0.20
n=4	2,386	0.25	-	0.22
n=5	9,554	1.12	-	0.33
n=6	38,226	4.41	-	2.85
n=7	152,914	21.19	-	-
<b>mix(m,n,k)</b>				
n=10, k=0				
m=100	4200	0.40	?	0.22
m=1000	42000	3.83	?	0.45
m=5000	210000	21.15	?	1.36

Figure 4. Disputed inferences

ment for proofs, over and above theory storage, is  $O(m)$  (measured using a heap profiling version of Hugs [20]) with or without memoization and loop-checking, where  $m$  is the number of sub-goals required for the proof.

Since *Deimos* exercises all rules in the problems we have addressed, its advantage over Delores when responding to a single query in more realistic situations has not been assessed by these experiments. That will be addressed in future work.

We now turn to an assessment of Delores. **empty()** shows that Delores has a small but significant overhead on start-up. This is the initialization of  $S$ , which visits the entire atom table. Above this overhead, the cost of initialization is proportional to the number of distinct atoms in the theory. In the worst case, the initialization calls a memory allocation routine for each atom.

Except for the direct execution of strict rules by Prolog, the partial implementation of Delores is clearly the fastest of the implementations, when it is applicable. Thus the basic engine has excellent performance. The figures support the

claim that its complexity is linear in the size of the input theory. In many of the experiments with full Delores the linear complexity is also apparent.

However, it is apparent that the overhead introduced by the pre-processing transformation varies quite significantly from problem to problem and is sometimes extraordinarily high, well above what would be expected for a transformation that increases the size of the program only by a factor of 12 (see, for example, **tree(10,3)**). The timings of such problems were the only ones to vary significantly when experiments were repeated. It turns out that the initialization of  $S$  consumes the bulk of this time. Furthermore, it is on those problems that contain many different atoms that Delores performs worst. This is evident in comparing the behaviour of Delores on **tree()** and **dag()** problems. It is also apparent when comparing the data for Delores on the problems with undisputed inferences (Figure 3) – where the complexity comes mostly from the number of atoms – and problems with disputed inferences (Figure 4), where the number of different atoms is smaller.

We have not yet properly accounted for Delores's sensitivity to the number of atoms. Certainly the transformation exacerbates the situation by introducing many more atoms. We could partly address the latter problem by implementing defeaters directly, in a similar manner to defeasible rules. This would require only minor extensions to the existing system and would reduce the number of atoms by 1/4. We could also redesign the transformation, trading incrementality for a more parsimonious introduction of new atoms. However, the main problem is the apparent nonlinearity of initializing  $S$  and the source of this behaviour requires further investigation.

Another point to note is that, in contrast to the query answering systems, Delores performs slightly worse on problems with strict rules. The reason is that strict rules are duplicated and so the inferences performed by the system are effectively doubled.

## 7 Conclusion

We have presented two new implementations of defeasible logic, based on substantially different techniques. Our experiments on query-answering implementations have demonstrated that both *Deimos* and the existing d-Prolog system can handle very large rule sets, although d-Prolog is effective on only a narrow range of rule sets. *Deimos* is clearly superior in the more realistic situations when some rules conflict.

We have seen that the complexity of computing consequences in defeasible logic is linear in the size of the input theory. Our experiments with the partial implementation of Delores have confirmed this claim. Indeed the partial implementation of Delores was clearly the faster system in al-

most all experiments on which it could be run. However, the transformation implemented in full Delores did not behave linearly. Since theoretically it is of linear complexity, there is clearly an engineering issue to be addressed here.

In summary, both *Deimos* and Delores show promise as high-speed implementations of defeasible logic, and *Deimos* has already partly fulfilled its promise. Consequently it appears that defeasible logic provides rule prioritization and defeasible reasoning in an efficiently implementable way.

Work is continuing on both systems. For *Deimos*, we are implementing memoization using mutable arrays, instead of a balanced tree, in order to eliminate the  $O(\log n)$  factor. For Delores, we are addressing the problems of initialization and the pre-processing transformation that were exposed by our experimental evaluation, and work is proceeding on a better implementation of the hash tables used when loading a theory.

## Acknowledgements

We thank Scott Brady and Chris Herring for their work on a preliminary all-conclusions system, and Guido Governatori for his work on the transformations used in Delores and for discussions on defeasible logic. This research was supported by the Australian Research Council under grant A49803544.

## References

- [1] G. Antoniou. *Nonmonotonic Reasoning*. MIT Press 1997.
- [2] G. Antoniou, D. Billington, G. Governatori and M.J. Maher. Representation Results for Defeasible Logic. manuscript, 1999.
- [3] G. Antoniou, D. Billington, and M.J. Maher. Normal forms for defeasible logic. In *Proc. 1998 Joint International Conference and Symposium on Logic Programming*, MIT Press 1998, 160–174.
- [4] G. Antoniou, D. Billington and M.J. Maher. On the analysis of regulations using defeasible rules. In *Proc. 32nd Hawaii International Conference on Systems Science* 1999.
- [5] G. Antoniou. On the Role of Rule-Based Nonmonotonic Systems in Electronic Commerce – A Position Statement. In *Proc. 1st Australasian Workshop on AI and Electronic Commerce*, Sydney, 1999.
- [6] G. Antoniou, M.J. Maher and D. Billington. Defeasible Logic versus Logic Programming without Nega-



- tion as Failure. *Journal of Logic Programming* 42(1): 47–57, 2000.
- [7] D. Billington, K. de Coster and D. Nute. A modular translation from defeasible nets to defeasible logic. *Journal of Experimental and Theoretical Artificial Intelligence* 2 (1990): 151–177.
- [8] D. Billington. Defeasible Logic is Stable. *Journal of Logic and Computation* 3 (1993): 370–400.
- [9] P. Cholewinski, V. Marek, A. Mikitiuk and M. Truszczynski. Experimenting with Nonmonotonic Reasoning. In *Proc. International Conference on Logic Programming*, MIT Press, 1995, 267–281.
- [10] M.A. Covington, D. Nute and A. Vellino. *Prolog Programming in Depth*. Prentice Hall 1997.
- [11] Y. Dimopoulos and A. Kakas. Logic Programming without Negation as Failure. In *Proc. 5th International Symposium on Logic Programming*, MIT Press 1995, 369–384.
- [12] W.F. Dowling and J.H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming* 1(3): 267–284, 1984.
- [13] G. Gallo and G. Urbani. Algorithms for Testing the Satisfiability of Propositional Formulae. *Journal of Logic Programming* 7(1): 45–61, 1989.
- [14] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming* 17 (1993): 301–322.
- [15] G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation* 2, 1992, 397–425.
- [16] G. Governatori & M. Maher. An Argumentation-Theoretic Characterization of Defeasible Logic. In *Proc. European Conf. on Artificial Intelligence*, 2000.
- [17] B.N. Groszof. Prioritized Conflict Handling for Logic Programs. In *Proc. Int. Logic Programming Symposium*, J. Maluszynski (Ed.), 197–211, MIT Press 1997.
- [18] B.N. Groszof. DIPLOMAT: Business Rules Interlingua and Conflict Handling, for E-Commerce Agent Applications (Overview of System Demonstration). In *Proc. IJCAI-99 Workshop on Agent-mediated Electronic Commerce (AMEC-99)*, July 1999.
- [19] J.F. Horty, R.H. Thomason and D. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. In *Proc. AAAI-87*, 358–363.
- [20] M.P. Jones et al. The Hugs98 User Manual. <http://www.haskell.org/hugs/>
- [21] H. Kautz and B. Selman. Hard problems for simple default logics. In *Proc. 1st International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann 1989, 189 – 1997.
- [22] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R.B. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31 (1997): 59–84.
- [23] M. Maher, G. Antoniou and D. Billington. A Study of Provability in Defeasible Logic. In *Proc. Australian Joint Conference on Artificial Intelligence*, LNAI 1502, Springer, 1998, 215–226.
- [24] M. Maher and G. Governatori, A Semantic Decomposition of Defeasible Logics, *Proc. American National Conference on Artificial Intelligence (AAAI-99)*, AAAI/MIT Press, 299–305.
- [25] V. Marek and M. Truszczynski. *Nonmonotonic Logic*, Springer 1993.
- [26] L. Morgenstern. Inheritance Comes of Age: Applying Nonmonotonic Techniques to Problems in Industry. *Artificial Intelligence* 103(1998) 1–34.
- [27] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, Springer-Verlag, 420–429.
- [28] D. Nute. Defeasible Reasoning. In *Proc. 20th Hawaii International Conference on Systems Science*, IEEE Press 1987, 470–477.
- [29] D. Nute. Defeasible Logic. In D.M. Gabbay, C.J. Hogger and J.A. Robinson (eds.): *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*, Oxford University Press 1994, 353–395.
- [30] A. Rock and D. Billington. A propositional Plausible Logic implementation in Haskell. In *Proc. Australasian Computer Science Conference*, 2000, 204–210.
- [31] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press 1997.
- [32] L.A. Stein. Resolving Ambiguity in Nonmonotonic Inheritance Hierarchies. *Artificial Intelligence* 55(1992): 259–310.