Deimos: A Query Answering Defeasible Logic System

Andrew Rock
School of Computing and Information Technology
Griffith University
Nathan, Queensland, 4111, Australia
a.rock@griffith.edu.au

Abstract

This document is a description of *Deimos*, a query answering Defeasible logic system. *Deimos* is a complete implementation of propositional Defeasible logic and some variants. System components include command-line-driven theorem provers and a web-accessible theorem prover. The system has been implemented in Haskell.

This is the short form of this document. The long form includes details about the implementation.

Contents

1 Introduction

Deimos [1] is a system that implements Defeasible logic [2, 3]. The procedures for installation of the *Deimos* system are described in section 2. Section 3 is a guide for users of the system.

The *Phobos* system implements an extension to Defeasible logic, Plausible logic [4], and is described in a separate document [5].

The symbol `$` appears is command examples to represent the shell command line prompt. Milti-line commands are continued with the UNIX escape character, `\`. The Hugs command line prompt is shown as `Hugs>`.

2 Installation

2.1 Downloading

The *Deimos* system and this documentation can be downloaded from:
`http://www.cit.gu.edu.au/~arock/defeasible/Defeasible.cgi`

2.2 Unpacking and compiling Deimos

Compiling the system requires a Haskell compiler. Haskell compilers are available from `http://www.haskell.org/`. The compiler requires extensions to the Haskell-98 standard, specifically support for multi-parameter type classes. The Haskell Interpreter, Hugs, is capable of running *Deimos* albeit more slowly and for smaller theories.

To unpack:

```
$ gunzip Deimos.tar.gz
$ tar -xf Deimos.tar.gz
```

To unpack on Windows, use the free tool, PowerArchiver.

Change directory to `Deimos/src`.

```
$ cd Deimos/src
```

To compile all of the *Deimos* tools, type:

```
$ make bin
```

On windows, where binaries end in `.EXE`, and if you have make, try:

```
$ make pc_bin
```

Only the CGI tool (section 3.9) is sensitive to its location for installation and the location of its resources. The Haskell source will require modification to adjust the file and directory names referred to. Most users will not want to install the CGI tool.

2.3 Compiling without make

If you are wishing to compile the *Deimos* tools without `make`, for instance if you are using Windows, you can use GHC's `--make` option to compile the modules in the correct order to satisfy their dependencies. The following are the commands required to compile each tool.

```
$ ghc --make -O DefeasibleParser.lhs \
-o ../bin/DefeasibleParser
```

```
$ ghc --make -O DProver.lhs -o ../bin/DProver
```

```
$ ghc --make -O ODProver.lhs -o ../bin/ODProver
```

```
$ ghc --make -O DTScale.lhs -o ../bin/DTScale
```

```
$ ghc --make -O Defeasible.cgi.lhs -o ../bin/Defeasible.cgi
```

3 User's Guide

This user's guide begins in section 3.1 with a description of the syntax that *Deimos* will recognize for defeasible theories. Section 3.2 describes the syntax of the queries the system will respond to. Sections 3.3 and 3.4 describe how to use the two most popular Haskell runtime systems to execute the tools that make up *Deimos*. The remaining subsections of section 3 give usage instructions for each of those tools.

3.1 Theories

Defeasible theories are entered into components of *Deimos* in textual form. The syntax for theories is summarized in appendix A.

3.1.1 Whitespace and comments

Any amount of whitespace is permitted before and after any symbol. Comments are treated as whitespace. There are two types:

- Comments that begin with a `%` extend to the end of the line.
- Comments that begin with `/*` extend to the next `*/` and may extend across many lines.

3.1.2 Atoms

Atoms are names made up of letters of either case, digits and underscores (`_`), but must start with a lower case letter.

Phobos extends defeasible theories by permitting arguments in atoms. Arguments may be either:

constants – names that begin with lower case letters; or

variables – names that begin with upper case letters.

Arguments are enclosed in parentheses and are comma separated. A "grounded" object contains no variables, only constants. Example atoms:

```
p               p(a,b,C)
proposition_13  proposition14(const1,const2,Var_1)
```

3.1.3 Literals

A literal is an atom p or its negation $\neg p$. *Deimos* uses `~` for $\neg$. Example literals:

```
p   ~p   p(a,b,C)   ~p(a,b,C)
```

3.1.4 Facts

Facts are literals that are asserted as true.

3.1.5 Rules

There are three types of rules permitted in *Deimos* thories:

Strict rules consist of an antecedent (a set of literals), the strict arrow `->` (for $\rightarrow$) and a consequent (a literal).

Defeasible rules consist of an antecedent, the plausible arrow `=>` (for $\Rightarrow$) and a consequent.

Defeater rules consist of an antecedent, the defeater arrow `~>` (for $\rightsquigarrow$) and a consequent.

The set braces may be omitted from antecedents. Example rules:

formal	*Deimos*
$\{\} \Rightarrow p$	`{} => p`
$\{a,b,c\} \rightsquigarrow \neg d$	`{a, b, c} ~> ~d`
$\{a,b,c\} \Rightarrow d$	`{a, b, c} => d`
$p \rightarrow q$	`p -> q`

3.1.6 Labelled rules

Labels are names that start with an upper case letter. Rules in defeasible theories are usually preceded by a unique label and a colon.

3.1.7 Priority assertions

A priority assertion consists of two labels separated by `>`. Example:

```
R1 > R2
```

In this example we assert that the rule labelled `R1` "beats" the rule labelled `R2`.

3.1.8 Theories

A defeasible theory is a triple $T = (F, R, >)$, where F is a set of facts, R is a set of rules, some of which are labelled, and $>$ is the priority relation on the labelled rules.

The syntax preferred for *Deimos* theories is demonstrated with these two examples. The first example is purely propositional.

```
% A test defeasible theory in Deimos syntax

      emu.
      emu   => heavy.
      emu   -> bird.
R1:   bird  => flies.
R2:   heavy ~> ~flies.
      R2 > R1.
```

This second example uses removable variables. The example shows only one argument for each literal, but more are permitted and must be comma separated.

```
% A test defeasible theory in Deimos syntax,
% with removable variables

      emu(tweety).
      emu(X)    => heavy(X).
      emu(X)    -> bird(X).
R1:   bird(X)  => flies(X).
R2:   heavy(X) ~> ~flies(X).
      R2 > R1.
```

Deimos can also parse theories expressed in d-Prolog syntax. d-Prolog does not use rule labels, and must therefore explicitly restate the rules in priority (`sup`) declarations. Example:

```
% A test defeasible theory in d-Prolog syntax,
% with removable variables

emu(tweety).
bird(X) :- emu(X).
heavy(X) := emu(X).
flies(X) := bird(X).
neg flies(X) :^ heavy(X).
sup((neg flies(X) :^ heavy(X)), (flies(X) := bird(X))).
```

Deimos syntax and d-Prolog syntax can be mixed to some extent, as in the syntax accepted by the *Delores* [1] system. Here the rules are stated using d-Prolog syntax, but priorities are declared using rule labels. Example:

```
% A test defeasible theory in a mix of Deimos and
% d-Prolog syntax, with removable variables

      emu(tweety).
      heavy(X)     := emu(X).
      bird(X)      :- emu(X).
R1:   flies(X)     := bird(X).
R2:   neg flies(X) :^ heavy(X).
      R2 > R1.
```

3.2 Tagged Literals

The queries that the prover components of *Deimos* respond to are tagged literals. The syntax for tagged literals is:

```
proof_symbol   ::= "D" | "d" | "da" | "S" | "dt"

tagged_literal ::= ("+" | "-") proof_symbol literal
```

At present the literal in a tagged literal must be grounded, that is, contain no variables. Examples:

`+D emu    -d flies(tweety)`

The meaning of each proof symbol is listed in table 1.

symbol	*meaning*
D	Δ: strict
d	∂: defeasible
dt	∂_{-t}: defeasible variant without team defeat
da	δ: defeasible variant with ambiguity propagation
S	$\int$: defeasible variant – support

Table 1: The proof symbols.

3.2.1 Standard inference conditions

The following are the inference rules that are used to prove a given tagged literal. A formal proof or derivation $P = (P(1), \ldots, P(|P|))$ of is a finite sequence of tagged literals $\pm\alpha q$ where $\alpha \in \{\Delta, \partial, \partial_{-t}, \delta, \int\}$, and q is a literal. In these rules q is a literal, $A(r)$ is the antecedent of rule r, $R[q]$ is the set of rules with consequent q, $R_s[q]$ is the set of strict rules with consequent q, $R_{sd}[q]$ is the set of strict and defeasible with consequent q, $r > s$ means that a rule r beats rule s, and $r \not> s$ means that a rule r does not beat rule s.

$+\Delta$: If $P(i+1) = +\Delta q$ then either
  $q \in F$ or
    $\exists r \in R_s[q]\ \forall a \in A(r) : +\Delta a \in P(1..i)$

$-\Delta$: If $P(i+1) = -\Delta q$ then
  $q \notin F$ and
    $\forall r \in R_s[q]\ \exists a \in A(r) : -\Delta a \in P(1..i)$

$+\partial$: If $P(i+1) = +\partial q$ then either
  $+\Delta q \in P(1..i)$ or
    $\exists r \in R_{sd}[q] \forall a \in A(r) : +\partial a \in P(1..i)$ and
    $-\Delta \sim q \in P(1..i)$ and
    $\forall s \in R[\sim q]$ either
      $\exists a \in A(s) : -\partial a \in P(1..i)$ or
      $\exists t \in R_{sd}[q]$ such that
        $\forall a \in A(t) : +\partial a \in P(1..i)$ and $t > s$

$-\partial$: If $P(i+1) = -\partial q$ then
  $-\Delta q \in P(1..i)$ and either
    $\forall r \in R_{sd}[q] \exists a \in A(r) : -\partial a \in P(1..i)$ or
    $+\Delta \sim q \in P(1..i)$ or
    $\exists s \in R[\sim q]$ such that
      $\forall a \in A(s) : +\partial a \in P(1..i)$ and
      $\forall t \in R_{sd}[q]$
        $\exists a \in A(t) : -\partial a \in P(1..i)$ or $t \not> s$

3.2.2 Variant inference conditions

$+\partial_{-t}$: If $P(i+1) = +\partial_{-t} q$ then
  $+\Delta q \in P(1..i)$ or
    $\exists r \in R_{sd}[q] \forall a \in A(r) : +\partial_{-t} a \in P(1..i)$ and
    $-\Delta \sim q \in P(1..i)$ and
    $\forall s \in R[\sim q]$ either
      $r > s$ or
      $\exists a \in A(s) : -\partial_{-t} a \in P(1..i)$

$-\partial_{-t}$: If $P(i+1) = -\partial_{-t} q$ then
  $-\Delta q \in P(1..i)$ and
    $\forall r \in R_{sd}[q] \exists a \in A(r) : -\partial_{-t} a \in P(1..i)$ or
    $+\Delta \sim q \in P(1..i)$ or
    $\exists s \in R[\sim q]$ either
      $r \not> s$ or
      $\forall a \in A(s) : +\partial_{-t} a \in P(1..i)$

$+\delta$: If $P(i+1) = +\delta q$ then either
  $+\Delta q \in P(1..i)$ or
    $\exists r \in R_{sd}[q] \forall a \in A(r) : +\delta a \in P(1..i)$ and
    $-\Delta \sim q \in P(1..i)$ and
    $\forall s \in R[\sim q]$ either
      $\exists a \in A(s) : -\int a \in P(1..i)$ or
      $\exists t \in R_{sd}[q]$ such that
        $\forall a \in A(t) : +\delta a \in P(1..i)$ and $t > s$

$-\delta$: If $P(i+1) = -\delta q$ then
  $-\Delta q \in P(1..i)$ and either
    $\forall r \in R_{sd}[q] \exists a \in A(r) : -\delta a \in P(1..i)$ or
    $+\Delta \sim q \in P(1..i)$ or
    $\exists s \in R[\sim q]$ such that
      $\forall a \in A(s) : +\int a \in P(1..i)$ and
      $\forall t \in R_{sd}[q]$
        $\exists a \in A(t) : -\delta a \in P(1..i)$ or not $(t > s)$

$+\int$: If $P(i+1) = +\int q$ then either
  $+\Delta q \in P(1..i)$ or
    $\exists r \in R_{sd}[q]$ such that
      $\forall a \in A(r) : +\int a \in P(1..i)$ and
      $\forall s \in R[\sim q]$ either
        $\exists a \in A(s) : -\delta a \in P(1..i)$ or $s \not> r$

$-\int$: If $P(i+1) = -\int q$ then either
  $-\Delta q \in P(1..i)$ and
    $\forall r \in R_{sd}[q]$ such that
      $\exists a \in A(r) : -\int a \in P(1..i)$ or
      $\exists s \in R[\sim q]$ either
        $\forall a \in A(s) : +\delta a \in P(1..i)$ and $s > r$

3.3 Just enough Hugs

The Haskell programming language has been used to implement *Deimos*. There are several Haskell implementations. The most widely used are the interpreter, Hugs, and the (glorious) Glasgow Haskell Compiler, GHC. Compiling *Deimos* with GHC is described in section 2. While compiling with GHC is the only way to install the web-based components of *Deimos* and the compiled provers will significantly out-perform the interpreted ones, for many users running the provers with the interpreter is quite sufficient. There are advantages: Hugs has been ported to more platforms than GHC; and installing Hugs is much easier than installing GHC. Here is just enough information to get and use Hugs to run *Deimos*.

The latest version of Hugs and installation instructions for all platforms can be always be obtained from http://www.haskell.org/.

Deimos uses Haskell language features that are not included in the Haskell-98 standard, and also demands a large heap for compilation and execution, so hugs should be launched with the options `-98` and `-h10000000` or more.

Also hugs needs to know where to load the modules from. Use the `-P` option when launching hugs to specify the locations of the library and *Deimos* modules. For example:

```
$ hugs -98 -h10000000 -P"ABRHLibs:Deimos/src:"
```

Defining a shell alias for this complicated command is recommended.

Once Hugs is installed and launched, *Deimos* programs can be loaded by typing the command:

```
Hugs> :l <program-name>
```

where `<program-name>` is the filename of the main module of the *Deimos* program. The file name extension `.lhs` may be omitted.

To run the program, in most cases, type the expression:

```
Hugs> main
```

To kill any Haskell program type a control-C, or command-. on a Macintosh (prior to Mac OS X).

To quit Hugs, type the command:

```
Hugs> :q
```

3.4 Running compiled tools

Once compiled with GHC (section 2), the *Deimos* tools can be executed directly from a command line shell.

The command to type is the name of the program. Each of the following sections covers one program. The options and other command line arguments that can be specified in addition to the program name are described there.

For very large theories, the default memory allocations may be insufficient. The program may fail because either the heap or stack space limits are exceeded. In each case, the error message that results specified which limit was exceeded. Performance can be less than optimal if the program spends too much time garbage collecting. The following options are available to control memory usage. These options control the Haskell run-time system.

Run-time system command line options are separated from the command line options passed to the program, by the delimiting options `+RTS` and `-RTS`. Example:

```
$ program opt1 opt2 +RTS opt3 opt4 -RTS opt5 opt6
```

In this example: `program` is the name of the program, `opt1`, `opt2`, `opt5`, and `opt6` are options passed to the program; and `opt3` and `opt4` are options passed to the Haskell run-time system.

The stack limit can be set with the option `-K`#, where # is the number of bytes. # can be specified as with the suffix `M` (megabytes). For example, `-K10M` limits the stack 10 ten megabytes.

The maximum heap size is similarly set with the option `-M`#. The heap will grow slowly towards this limit. The run-time system always tries to reclaim memory with the garbage collector before extending the heap. This has a big impact on performance. To avoid this make the initial heap size bigger with the option `-H`#.

This is an example command line that gives the run-time system plenty of room.

```
$ program opt1 opt2 +RTS -K20M -M100M -H50M
```

3.5 DefeasibleParser

The program `DefeasibleParser` is a test program that exercises the lexers and parsers required to parse a defeasible theory. It can be used as a quick syntax checker for defeasible theory files. This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.5.1 Usage (GHC)

Run the program with the command

```
$ DefeasibleParser path1 path2 ...
```

where *path*, *path2*, ... are the paths to each of the theory files to be parsed. For each file the program will display the name of the file and either a syntax error message or, if the file parsed correctly, the regenerated theory. A check for cycles in the priority relation is performed. If there are cycles, the priorities involved are printed. If there are no cycles an attempt is made to remove all variables by generating ground instances of them using all of the constants appearing in the theory. The grounded theory is printed.

If no paths are supplied on the command line, then standard input will be read and parsed.

3.5.2 Usage (Hugs)

Load the script `DefeasibleParser.lhs` into the Hugs interpreter. To test the parser on one description file, type the expression

```
Hugs> run1 "path"
```

where *path* is the path to the theory file. To test the parser on a list of files, type the expression

```
Hugs> run ["path1", "path2", ... ]
```

Standard input will not be parsed if that list is empty, otherwise the program will then behave as described for GHC.

3.6 DProver

The program `DProver` is the query answering prover with the simplest (and slowest) implementation. This program is maintained as a test-bed for new features as it is simpler and quicker to modify than the other prover programs constituting *Deimos*. Current features available to this prover, but not to others, include:

- provers with well-founded semantics; and
- run-files.

This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.6.1 Usage (GHC)

Run the program by typing a command of the form:

`$ DProver` *options* `[`*theory-file-name* `[`*tagged-literal*`]]`

where the options are:

`-t` Print the theory in *Deimos* syntax and terminate.

`-tp` Print the theory in d-Prolog syntax and terminate.

`-td` Print the theory in *Delores* syntax and terminate.

`-e` *prover* Use the named *prover* engine. See table 2 for the names of the prover engines that are available. The default prover engine is `nhlt`.

`-r` *run-file* Use the named *run-file* to generate a truth table and terminate.

If a theory file name is supplied on the command line, that theory will be loaded. Otherwise when the program starts it will prompt for the name of a theory file to load. If there is a tagged literal supplied on the command line, then that proof will be attempted and the program will terminate upon its completion. If the `-r` option is specified and a *run-file* name is supplied, then all the proofs specified by the runfile are attempted, and then a truth table will be printed. Otherwise the program will prompt for and handle commands.

When a theory is loaded it is parsed and checked for consistency. If these checks fail an error message will be printed and another file name promped for.

When a theory has been loaded successfully, the program prompts for commands with `|-`. The following commands are accepted:

`?` Print the list of commands.

`q` Quit the program.

`t` Print the theory in *Deimos* syntax.

`tp` Print the theory in d-Prolog syntax.

`td` Print the theory in *Delores* syntax.

`f` Forget the history of subgoals accumulated so far.

`e` Identify the current prover engine.

`e` *engine* Select a prover *engine*.

`l [`*file-name*`]` Load a new theory file [named *file-name*].

tagged-literal Answer *tagged-literal* by attempting a proof.

`r [`*run-file*`]` Run the named run-file, printing a table of results.

Tagged literals are described in section 3.2. The prover engines that can be selected with the `e` command are listed in table 2. The different provers feature combinations of goal counting, avoiding recomputation by maintaining a history of prior results, loop detection, well-founded semantics, and trace printing. The default prover is `nhlt`.

3.6.2 Usage (Hugs)

Load the script `DProver.lhs` into the Hugs interpreter. At the Hugs prompt, type the expression

```
Hugs> run "options [theory-file-name [tagged-literal]]"
```

The program then behaves as descibed for GHC.

prover name	*counts goals*	*keeps history*	*detects loops*	*well-founded*	*prints trace*
-					
n	•				
nh	•	•			
nhl	•	•	•		
nhlw	•	•	•	•	
t					•
nt	•				•
nht	•	•			•
nhlt	•	•	•		•
nhlwt	•	•	•	•	•

Table 2: DProver provers.

3.6.3 Run-files

A theory may be tested by augmentation by combinations of extra facts, generating a summary table of results. `DProver` reads a file, a *run-file* to specify the combinations of facts to test with and the proofs to attempt.

A run-file consists of a sequence of statements that specify the literals to assert as facts, the combinations of literals to ignore, and the proofs to attempt for each combination of inputs.

The syntax of a run-file is summarized as follows.

```
run-file ::= {(input | ignore | output) "." }

input ::= "input" "{" literal {"," literal} "}"

ignore ::= "ignore"  "{" literal {"," literal} "}"

output ::= "output" "{" taggedLiteral "}"
```

All literals in a run-file must be grounded. Comments are permitted, with the same syntax as for theory files.

An `input` statement usually contains one literals. If two or more literals are present in a single input statement, then they are mutually exclusive. Examples are shown in table 3. An `ignore` statement rules out specific combinations of facts. An example is shown in table 3. An `output` statement specifies a proof to attempt for each combination of literals. A run-file will produce a summary table of results. The results will be abbreviated as shown in table 4.

statements	*facts generated*

`input{a}. input{b}.`	`a. b.`
	`a. ~b.`
	`~a. b.`
	`~a. ~b.`

`input{a, b}.`	`a. ~b.` `~a. b.`
`input{a, ~b}.`	`a. b.` `~a. ~b.`

`input{a}. input{b}. ignore{a, ~b}.`	`a. b.`
	`~a. b.`
	`~a. ~b.`

Table 3: Example input and ignore statements and the combinations of facts generated.

Result	*abbreviation*
Proved	`P`
Not Proved	`N`
Loops	`L`

Table 4: Abbreviated proof results.

3.7 ODProver

The program `ODProver` is a query answering prover with an improved (faster) implementation.

This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.7.1 Usage (GHC)

Run the program by typing a command of the form:

`$ ODProver` *options* [*theory-file-name* [*tagged-literal*]]

The program options, commands and behavior are the same as described for `DProver` in section 3.6, with the following exceptions:

- Prover engines with well-founded sematics are not available.
- Some additional provers with an array-based history for improved speed are provided.
- Run-files are not implemented. Consequently there is no `-r` command line option or `r` command.

The available provers are listed in table 5.

prover name	*counts goals*	*keeps history*	*detects loops*	*well-founded*	*prints trace*
-					
n	•				
nh	•	•			
nhl	•	•	•		
t					•
nt	•				•
nht	•	•			•
nhlt	•	•	•		•
nH	•	•			
nHl	•	•	•		

Table 5: ODProver provers.

3.7.2 Usage (Hugs)

Load the script `ODProver.lhs` into the Hugs interpreter. The program should be invoked and used the same way as `DProver`.

3.8 DTScale

The program `DTScale` is used for the generation of scalable test theories and for measuring the time required for proofs using them.

This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell. Execution time measurement is only possible using the GHC compiled version of this program.

3.8.1 Usage (GHC)

Compile the program by typing `make DTScale`. Run the program by typing a command of the form:

`$ DTScale` *options theory-name size...*

where the options are:

`-t` Print the theory in *Deimos* syntax and terminate without attempting a proof.

`-tp` Print the theory in d-Prolog syntax and terminate without attempting a proof.

`-td` Print the theory in *Delores* syntax and terminate without attempting a proof.

`-m` Print the computed metrics (defined in section B.8) for the theory before proving it.

`-e` *prover* Use the named *prover* engine. See tables 2 and 5 for the names of the provers that are available. The default prover is `nHl`.

`-o` Don't use the faster array-based theory representation.

Example:

```
$ DTScale -t mix 100 10 5
```

When a proof is requested, statistics about the size of the theory, the number of goals and the time required for proof are printed.

The theory and the tagged literal to use are specified by *theory-name* and *size*. The mapping from name to theory is given in table 6. The scalable test theories are described in detail in appendix B.

theory	*theory name*	*smallest size*
chain(n)	`chain`	`0`
chain$^{\mathbf{s}}(n)$	`chains`	`0`
circle(n)	`circle`	`1`
circle$^{\mathbf{s}}(n)$	`circles`	`1`
levels(n)	`levels`	`0`
levels$^{-}(n)$	`levels-`	`0`
teams(n)	`teams`	`0`
tree(n, k)	`tree`	`1 1`
dag(n, k)	`dag`	`1 1`
mix(m, n, k)	`mix`	`1 0 0`

Table 6: Names for specifying scalable test theories, and the smallest size parameters permitted for each theory.

3.8.2 Usage (Hugs)

Load the script `DTScale.lhs` into the Hugs interpreter. At the Hugs prompt, type the expression `run` *args*, where *args* is a string containing the command line arguments as described above for the compiled version. Example:

```
Hugs> run "-p nhlt tree 5 3"
```

3.9 CGI Tool

The program `Defeasible.cgi` is a Common Gateway Interface program which provides a world wide web interface to *Deimos*. The program should be accessed with a WWW browser with the URL: `http://your.www.site/Defeasible.cgi`.
For our WWW site, this is:
`http://www.cit.gu.edu.au/~arock/defeasible/Defeasible.cgi`

This opens the starting page for the system, containing pointers to information about Defeasible logic and *Deimos*. A form allows the user to select an example Defeasible theory to work with, or to open a page where a new theory can be entered.

With a theory selected or entered, the user can enter queries in the form of tagged literals. The form for entry of the queries has a menu that selects the prover to use. The choices available are equivalent to those offered by `ODProver` and summarized in table 5.

The CGI tool is stateless. All information about a session is maintained within the HTML data returned to the user's browser.

A Syntax Summary

This is a summary description the syntax accepted by this implementation of Defeasible Logic.

A.1 Comments

Before or after any token can be any amount of whitespace. Comments are treated as whitespace.

```
comment1  ::= "%" {anything-not-"\n"} ("\n" | end-of-file)

comment2  ::= "/*" comment2'
comment2' ::=   "*/"
              | any-character comment2'
```

A.2 Identifiers

```
name1 ::= lower-case-letter {letter | digit | "_"}

name2 ::= upper-case-letter {letter | digit | "_"}
```

A.3 Literals

```
argument ::= name1 | name2

argList ::= "(" argument {"," argument} ")"

literal ::= ["~"] name1 [argList]

prolog_literal ::= ["neg"] name1 [argList]
```

A.4 Rules

```
antecedent ::=   "{" "}"
               | "{" literal {"," literal} "}"
               | literal {"," literal}
               | epsilon

rule ::= antecedent ("->" | "=>" | "~>") literal

prolog_antecedent
   ::=   "true"
       | prolog_literal {"," prolog_literal}

prolog_rule ::= prolog_literal (":-" | ":=" | ":^")
                prolog_antecedent
```

A.5 Labels and Priorities

```
label ::= name2

priority ::= label ">" label
```

A.6 Theories

```
fact ::= prolog_literal | literal

rule' ::= prolog_rule | rule

prolog_superiority
   ::= "sup" "(" "("  rule' ")" "," "(" rule' ")" ")"

labeled_rule ::= [label ":"] rule'

statement ::=   prolog_superiority
              | labeled_rule
              | fact
              | priority

theory ::= {statement "."}
```

A.7 Tagged Literals

A query to this system is a tagged literal; a literal to be proved, tagged by the level of proof required.

```
proof_symbol   ::= "D" | "d" | "da" | "S" | "dt"

tagged_literal ::= ("+" | "-") proof_symbol literal
```

B Scalable Test Theories

This appendix specifies the scalable test theories used to test the performance of *Deimos* system components.

B.1 Chain Theories

Chain theories **chain**(n) start with fact a_0 and continue with a chain of n defeasible rules of the form $a_{i-1} \Rightarrow a_i$. A proof of $+\partial a_n$ will use all of the rules and the fact.

$$\mathbf{chain}(n) = \left\{ \begin{array}{lcl} & & a_0 \\ r_1 : a_0 & \Rightarrow & a_1 \\ r_2 : a_1 & \Rightarrow & a_2 \\ & \vdots & \\ r_n : a_{n-1} & \Rightarrow & a_n \end{array} \right.$$

A variant **chain**$^{\mathbf{s}}(n)$ uses only strict rules.

$$
\mathbf{chain^s}(n) = \begin{cases} & a_0 \\ r_1: a_0 & \rightarrow a_1 \\ r_2: a_1 & \rightarrow a_2 \\ & \vdots \\ r_n: a_{n-1} & \rightarrow a_n \end{cases}
$$

B.2 Circle Theories

Circle theories $\mathbf{circle}(n)$ consist of n defeasible rules $a_i \Rightarrow a_{(i+1) \bmod n}$.

$$
\mathbf{circle}(n) = \begin{cases} r_0: a_0 & \Rightarrow a_1 \\ r_1: a_1 & \Rightarrow a_2 \\ & \vdots \\ r_{n-1}: a_{n-1} & \Rightarrow a_0 \end{cases}
$$

Any proof of $+\partial a_i$ will loop. A variant $\mathbf{circle^s}(n)$ uses only strict rules.

$$
\mathbf{circle^s}(n) = \begin{cases} r_0: a_0 & \rightarrow a_1 \\ r_1: a_1 & \rightarrow a_2 \\ & \vdots \\ r_{n-1}: a_{n-1} & \rightarrow a_0 \end{cases}
$$

B.3 Levels Theories

Levels theories $\mathbf{levels}(n)$ consist of a cascade of $2n+2$ disputed conclusions a_i, $i \in [0..2n+1]$. For each i, there are rules $\Rightarrow a_i$ and $a_{i+1} \Rightarrow \neg a_i$. For each odd i a priority asserts that the latter rule is superior. A final rule $\Rightarrow a_{2n+2}$ gives uncontested support for a_{2n+2}.

$$
\mathbf{levels}(n) = \begin{cases} r_0: \{\} \Rightarrow a_0 \\ r_1: a_1 \Rightarrow \neg a_0 \\ \hline r_2: \{\} \Rightarrow a_1 \\ r_3: a_2 \Rightarrow \neg a_1 \\ r_3 > r_2 \\ \hline r_4: \{\} \Rightarrow a_2 \\ r_5: a_3 \Rightarrow \neg a_2 \\ \hline \vdots \\ \hline r_{4n+2}: \{\} \Rightarrow a_{2n+1} \\ r_{4n+3}: a_{2n+2} \Rightarrow \neg a_{2n+1} \\ r_{4n+3} > r_{4n+2} \\ \hline r_{4n+4}: \{\} \Rightarrow a_{2n+2} \end{cases}
$$

A proof of $+\partial a_0$ will use every rule and priority. A variant $\mathbf{levels}^-(n)$ omits the priorities.

B.4 Teams Theories

Teams theories $\mathbf{teams}(n)$ consist of conclusions a_i which are supported by a team two defeasible rules and attacked by another team of two defeasible rules. Priorities ensure that each attacking rule is beaten by one of the supporting rules. The antecedents of these rules are in turn supported and attacked by cascades of teams of rules.

$$
\mathbf{teams}(n) = \mathbf{block}(a_0, n)
$$

where, if p is a literal, and $r_1, \ldots, r_4$ are new unique labels:

$$
\mathbf{block}(p, 0) = \begin{cases} r_1: \{\} \Rightarrow p \\ r_2: \{\} \Rightarrow p \\ r_3: \{\} \Rightarrow \neg p \\ r_4: \{\} \Rightarrow \neg p \\ r_1 > r_3 \\ r_2 > r_4 \end{cases}
$$

and, if $n > 0$, $a_1, \ldots, a_4$ are new unique literals, and $r_1, \ldots, r_4$ are new unique labels:

$$
\mathbf{block}(p, n) = \begin{cases} r_1: a_1 \Rightarrow p \\ r_2: a_2 \Rightarrow p \\ r_3: a_3 \Rightarrow \neg p \\ r_4: a_4 \Rightarrow \neg p \\ r_1 > r_3 \\ r_2 > r_4 \\ \mathbf{block}(a_1, n-1) \\ \mathbf{block}(a_2, n-1) \\ \mathbf{block}(a_3, n-1) \\ \mathbf{block}(a_4, n-1) \end{cases}
$$

A proof of $+\partial a_0$ will use every rule and priority.

B.5 Tree Theories

In tree theories $\mathbf{tree}(n, k)$ a_0 is at the root of a k-branching tree of depth n in which every literal occurs once.

$$
\mathbf{tree}(n, k) = \mathbf{block}(a_0, n, k)
$$

where, if p is a literal, $n > 0$, r is a new unique label, and $a_1, a_2, \ldots, a_k$ are new unique literals:

$$
\mathbf{block}(p, n, k) = \begin{cases} r: a_1, a_2, \ldots, a_k \Rightarrow p \\ \mathbf{block}(a_1, n-1, k) \\ \mathbf{block}(a_2, n-1, k) \\ \vdots \\ \mathbf{block}(a_k, n-1, k) \end{cases}
$$

and:

$$
\mathbf{block}(p, 0, k) = \{p
$$

A proof of $+\partial a_0$ will use every rule and fact.

B.6 Directed Acyclic Graph Theories

In directed acyclic graph theories $\mathbf{dag}(n, k)$, a_0 is at the root of a k-branching tree of depth n in which every literal occurs k times.

$$
\mathbf{dag}(n, k) = \begin{cases} & & & & a_{kn+1} \\ & & & & a_{kn+2} \\ & & \vdots \\ & & & & a_{kn+k} \\ r_0: a_1, & a_2, & \ldots, & a_k & \Rightarrow a_0 \\ r_1: a_2, & a_3, & \ldots, & a_{k+1} & \Rightarrow a_1 \\ & & \vdots \\ r_{nk}: a_{nk+1}, & a_{nk+2}, & \ldots, & a_{nk+k} & \Rightarrow a_{nk} \end{cases}
$$

A proof of $+\partial a_0$ will use every rule and fact.

B.7 Mix Theories

In mix theories $\mathbf{mix}(m, n, k)$ there are m defeasible rules for conclusion p and m defeaters against p, where each rule has n unique literals as antecedents. Each antecedent literal can be strictly established by a chain of strict rules of length k. A proof of $+\partial p$ uses all the rules and facts.

$$
\mathbf{mix}(m, n, k) = \begin{cases} r_1: a_{1,1}, & a_{1,2}, & \ldots, & a_{1,n} & \Rightarrow p \\ r_2: a_{2,1}, & a_{2,2}, & \ldots, & a_{2,n} & \Rightarrow p \\ & \vdots \\ r_m: a_{m,1}, & a_{m,2}, & \ldots, & a_{m,n} & \Rightarrow p \\ r_{m+1}: a_{m+1,1}, & a_{m+1,2}, & \ldots, & a_{m+1,n} & \rightsquigarrow \neg p \\ r_{m+2}: a_{m+2,1}, & a_{m+2,2}, & \ldots, & a_{m+2,n} & \rightsquigarrow \neg p \\ & \vdots \\ r_{2m}: a_{2m,1}, & a_{2m,2}, & \ldots, & a_{2m,n} & \rightsquigarrow \neg p \\ & \mathbf{strictChain}(a_{1,1}, k) \\ & \vdots \\ & \mathbf{strictChain}(a_{2m,n}, k) \end{cases}
$$

where:

$$
\mathbf{strictChain}(a_{i,j}, 0) = \{ a_{i,j}
$$

or, if $k > 0$:

$$
\mathbf{strictChain}(a_{i,j}, k) = \begin{cases} & b_{i,j,1} \\ r_{i,j,1}: b_{i,j,1} & \Rightarrow b_{i,j,2} \\ r_{i,j,2}: b_{i,j,2} & \Rightarrow b_{i,j,3} \\ & \vdots \\ r_{i,j,k-1}: b_{i,j,k-1} & \Rightarrow b_{i,j,k} \\ r_{i,j,k}: b_{i,j,k} & \Rightarrow a_{i,j} \end{cases}
$$

theory	*facts*	*rules*	*priorities*	*size*
chain(n)	1	n	0	$2n+1$
chain$^{\mathbf{s}}(n)$	1	n	0	$2n+1$
circle(n)	0	n	0	$2n$
circle$^{\mathbf{s}}(n)$	0	n	0	$2n$
levels(n)	0	$4n+5$	$n+1$	$7n+8$
levels$^{-}(n)$	0	$4n+5$	0	$6n+7$
teams(n)	0	$4\sum_{i=0}^{n} 4^i$	$2\sum_{i=0}^{n} 4^i$	$10\sum_{i=0}^{n-1} 4^i + 6(4^n)$
tree(n,k)	k^n	$\sum_{i=0}^{n-1} k^i$	0	$(k+1)\sum_{i=0}^{n-1} k^i + k^n$
dag(n,k)	k	$nk+1$	0	$nk^2+(n+2)k+1$
mix(m,n,k)	$2mn$	$2m+2mnk$	0	$2m+4mn+4mnk$

Table 7: Sizes of scalable test theories

B.8 Theory Sizes

A *Deimos* theory can be characterized by various metrics that give an indication of the size or complexity of the theory. These metrics might be used to estimate the memory required to store a theory or estimate the time taken to respond to queries to them.

Table 7 lists the formulae that predict these metrics for the scalable test theories described above. The metrics reported are:

facts the number of facts in the theory;

rules the number of rules in the theory;

priorities the number of priorities in the theory; and

size the overall "size" of the theory, defined as the sum of the numbers of facts, rules, priorities and literals in the bodies of all rules.

References

[1] M.J. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. Efficient defeasible reasoning systems. In *12th IEEE International Conference on Tools with Artificial Intelligence*, pages 384–392. IEEE, 2000. 1, 3.1.8

[2] D. Nute. Defeasible logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 353–395. Oxford University Press, 1994. 1

[3] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1997. 1

[4] Andrew Rock and David Billington. An implementation of propositional plausible logic. In Jenny Edwards, editor, *23rd Australasian Computer Science Conference*, volume 22(1) of *Australian Computer Science Communications*, pages 204–210, Canberra, January 2000. IEEE Computer Society, Los Alamitos. 1

[5] Andrew Rock. *Phobos*: A query answering Plausible logic system. Technical report, (continually) in preparation. 1