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 and complete listing 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 long form of this document. The short form omits the details of 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. In this long form of this document, the complete sources for the system are included in section 4.

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 in section 4.19.1. 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`.

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.

3.6.2 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.

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.

4 Implementation

This section, on the implementation of *Deimos*, presents the modules in a bottom-up sequence. Library modules that are not directly concerned with implementing Defeasible logic are presented in a separate document [6].

The sources are compatible with Haskell-98, with the exception that support for multi-parameter type classes is required. Haskell code is presented in `typewriter` font, as are syntax specifying productions. Productions use the `::=` symbol and are commentary material, not formal Haskell code. The source code for the Haskell modules have been written in the literate style, and the following subsections have been produced directly from the Haskell+LaTeX source code.

4.1 Lexical Issues

Various elements of the *Deimos* system parse textual representations of literals, rules, priorities, theories and queries. *Deimos* uses the `Parser` module [6] to implement functions that perform lexical analysis and parsers. The `DefeasibleLexer` module implements the functions for lexical analysis of Defeasible sources.

```
module DefeasibleLexer(lexerL) where

import Char

import ABR.Parser; import ABR.Parser.Lexers
```

4.1.1 Comments

Comments in Defeasible sources follow the Prolog conventions. Comments that start with a percent sign `%`) extend to the end of the line. Comments that start with the sequence `/*` extend to the the next sequence `*/` and may span more than one line.

Formally, the syntax for each type of comment is:

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

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

These comment forms are recognized by these lexer functions.

```
comment1L :: Lexer
comment1L
   = tokenL "%"
     <**> (many (satisfyL (/= '\n') "") *%> "")
     <**> (optional (literalL '\n') *%> "")
     %> " "

comment2L :: Lexer
comment2L
   = tokenL "/*" <**> comment2L' %> " "
     where
     comment2L' :: Lexer
     comment2L'
        =     tokenL "*/"
          <|> (satisfyL (\c -> True) "") <**> comment2L'
```

4.1.2 Names

Literals, rule labels, constants and variables are all instances of names that occur in Defeasible sources. Two types are distinguished: those starting with lower case letters; and those starting with upper-case letters.

Formally, the syntax for each type of name is:

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

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

These name forms are recognized by these lexer functions.

```
name1L :: Lexer
name1L
   = (satisfyL isLower "lower-case letter" <**>
      ((many (satisfyL isName1Char "letter, digit, _"))
       *%> ""))
     %> "name1"
     where
     isName1Char c = isAlpha c || isDigit c || c == '_'

name2L :: Lexer
name2L
   = (satisfyL isUpper "upper-case letter" <**>
      ((many (satisfyL isName2Char "letter, digit, _"))
       *%> ""))
     %> "name2"
     where
     isName2Char c = isAlpha c || isDigit c || c == '_'
```

4.1.3 Symbols and everything else

This function performs the lexical analysis of a Defeasible source. It lists all of the symbols that are special in Defeasible sources.

```
lexerL :: Lexer
lexerL
   = dropWhite $ nofail $ total $ listL [
        comment1L,                 comment2L,
        tokenL ":=" %> "symbol", tokenL ":^" %> "symbol",
        tokenL ":-" %> "symbol", tokenL "->" %> "symbol",
        tokenL "=>" %> "symbol", tokenL "~>" %> "symbol",
        tokenL "+"  %> "symbol", tokenL "-"  %> "symbol",
        tokenL "~"  %> "symbol", tokenL ">"  %> "symbol",
        tokenL "{"  %> "symbol", tokenL "}"  %> "symbol",
        tokenL "("  %> "symbol", tokenL ")"  %> "symbol",
        tokenL "."  %> "symbol", tokenL ","  %> "symbol",
        tokenL ":"  %> "symbol", name1L,
        name2L,                    whitespaceL
     ]
```

4.2 Literals

Literals for the Defeasible and Plausible logic implementations are defined by module `Literal`.

```
{-# LANGUAGE TypeSynonymInstances #-}

module Literal(
      Argument(..), Literal(..), PrologLiteral(..),
      LiteralName, pLiteralP, prologLiteralP, OLiteral,
      LitArray, LitTree, HasLits(..), makeLitTables,
      Negatable(..), IsLiteral(..), HasConstNames(..),
      HasVarNames(..), Subst, Groundable(..)
   ) where

import Array

import ABR.Control.Check; import ABR.Data.BSTree
import ABR.SparseSet; import ABR.Showing; import ABR.List
import ABR.DeepSeq; import ABR.Parser
```

4.2.1 Data type definitions

The primary representation of a literal is a string containing the name of the literal and a tag that indicates positive or negative. Some literals in a theory may have arguments which are either constants or variables to be replaced by constants.

```
type LiteralName  = String
type ConstantName = String
type VariableName = String

data Argument =   Const ConstantName
                | Var   VariableName
                deriving (Eq, Ord)

data Literal =   PosLit  LiteralName
               | PosLit_ LiteralName [Argument]
               | NegLit  LiteralName
               | NegLit_ LiteralName [Argument]
               deriving Eq
```

To mark a literal to be treated as a Prolog literal, for example to select a different syntax for textual output, it should be wrapped by the `PrologLiteral` constructor.

```
newtype PrologLiteral = PrologLiteral Literal
```

After variables have been removed, a literal is just a constant value. Integers will do. A negative literal is negative. Zero is not a valid literal since it can not be negated. Handling integers will be much more rapid and they can be used as array indices. This type represents an optimized literal.

```
type OLiteral = Int
```

4.2.2 Parsers

The syntax for a literal is:

```
argument ::= name1 | name2

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

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

which is implemented with these parsers:

```
argumentP :: Parser Argument
argumentP = nofail' "argument expected" (
                tagP "name1" @> (\(_,n,_) -> Const n)
            <|> tagP "name2" @> (\(_,n,_) -> Var n)
            )

argListP :: Parser [Argument]
argListP = literalP "symbol" "("
           *> argumentP
           <*> many (literalP "symbol" "," *> argumentP)
           <* nofail (literalP "symbol" ")")
           @> cons

pLiteralP :: Parser Literal
pLiteralP = optional (literalP "symbol" "~")
            <*> tagP "name1" <*> optional argListP
            @> (\(ts,((_,n,_),ass)) -> case (ts,ass) of
                  ([],[])     -> PosLit n
                  ([_],[])    -> NegLit n
                  ([],[as])   -> PosLit_ n as
                  ([_],[as])  -> NegLit_ n as
               )
```

An alternate syntax for literals, compatible with d-Prolog, is:

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

which is implemented:

```
prologLiteralP :: Parser Literal
prologLiteralP
   = optional (literalP "name1" "neg")
     <*> tagP "name1" <*> optional argListP
     @> (\(negs,((_,n,_),ass)) -> case (negs,ass) of
           ([],[])     -> PosLit n
           ([_],[])    -> NegLit n
           ([],[as])   -> PosLit_ n as
           ([_],[as])  -> NegLit_ n as
        )
```

4.2.3 Negation

Literals are either positive or negative. The `neg` function converts from positive to negative and *vice versa*. This function can be overloaded as other entities, such as Plausible formulas, can also be negated. The `Negatable` class includes all such entities. The `pos` method forces the anything to be positive.

```
class Negatable a where

   neg :: a -> a

   pos :: a -> a

   isPos :: a -> Bool

instance Negatable Literal where

   neg l
      = case l of
           PosLit  n     -> NegLit  n
           PosLit_ n as  -> NegLit_ n as
           NegLit  n     -> PosLit  n
           NegLit_ n as  -> PosLit_ n as

   pos l
      = case l of
           NegLit  n     -> PosLit  n
           NegLit_ n as  -> PosLit_ n as
           _             -> l

   isPos l
      = case l of
           PosLit  _     -> True
           PosLit_ n as  -> True
           _             -> False

instance Negatable OLiteral where

   neg = negate

   pos = abs

   isPos = (> 0)
```

4.2.4 Literal lookup tables

The `OLiteral` numeric value that represents the literal needs to be mapped to and from the literal. An array lets us map from numbers to literals in $O(1)$ time. A binary search tree lets us map from literals to numbers in $O(\log N)$ time, where N is the number of unique literals.

```
type LitArray = Array OLiteral Literal

type LitTree  = BSTree Literal OLiteral
```

To build these data structures, we must first collect all of the unique literals, without distinguishing positive and negative literals. `getLits thing set` adds all of the literals in `thing` to `set`.

```
class HasLits a where

   getLits :: a -> SparseSet Literal
              -> SparseSet Literal

instance HasLits Literal where

   getLits l = insertSS (pos l)
```

`makeLitTables set` makes the data structures required to quickly map between both representations of literals. `set` is the set of literals accumulated with `getLits`.

```
makeLitTables :: SparseSet Literal -> (LitArray, LitTree)
makeLitTables set =
   let lits = domBST set
       n = length lits
   in (listArray (1,n) lits, pairs2BST (zip lits [1..]))
```

Using look-up tables created above, literals and some formulas can be mapped to and from their numeric equivalents. `toOLiteral tree thing` uses the `tree` to map `thing` to the equivalent optimized literal. `fromOLiteral array ol` uses the `array` to map an optimized literal `ol` to some other thing which is equivalent. `isLiteral thing` returns `True` iff `thing` is equivalent to one literal.

```
class IsLiteral a where

   toOLiteral :: LitTree -> a -> OLiteral

   fromOLiteral :: LitArray -> OLiteral -> a

   isLiteral :: a -> Bool

instance IsLiteral Literal where

   toOLiteral t l = case l of
      PosLit _ -> case lookupBST l t of
         Just n -> n
         Nothing -> error "unknown literal"
      NegLit _ -> case lookupBST (pos l) t of
         Just n -> neg n
         Nothing -> error "unknown literal"
      PosLit_ _ _ -> case lookupBST l t of
         Just n -> n
         Nothing -> error "unknown literal"
      NegLit_ _ _ -> case lookupBST (pos l) t of
         Just n -> neg n
         Nothing -> error "unknown literal"

   fromOLiteral a l
      = let (low, high) = bounds a
            n = abs l
            s = signum l
        in if low <= n && n <= high
              then if s > 0
                      then a ! n
                      else neg (a ! n)
              else error "OLiteral out of range"

   isLiteral l = True
```

4.2.5 Collecting constant names

To ground all of the removable variables we must first collect all of the constant names. We can accumulate them in the same way we can accumulate all of the literal names.

```
class HasConstNames a where

   getConstNames :: a -> SparseSet ConstantName
                    -> SparseSet ConstantName
```

```
instance HasConstNames Argument where

   getConstNames a ns = case a of
      Const n -> insertSS n ns
      _       -> ns

instance HasConstNames Literal where

   getConstNames l ns = case l of
      PosLit_ _ as -> foldr getConstNames ns as
      NegLit_ _ as -> foldr getConstNames ns as
      _            -> ns
```

4.2.6 Collecting variable names

To ground all of the removable variables we must first collect all of the variables names. We can accumulate them in the same way we can accumulate all of the literal names.

```
class HasVarNames a where

   getVarNames :: a -> SparseSet VariableName
                  -> SparseSet VariableName
```

`hasVars x` returns `True` iff `x` contains variables.

```
   hasVars :: a -> Bool
   hasVars x = nullSS $ getVarNames x emptySS
```

`checkNoVars x` is a check that `x` does *not* contain variables.

```
   checkNoVars :: Check a a String
   checkNoVars x = if hasVars x
      then CheckPass x
      else CheckFail "Variables are not permitted."

instance HasVarNames Argument where

   getVarNames a ns = case a of
      Var n -> insertSS n ns
      _     -> ns

instance HasVarNames Literal where

   getVarNames l ns = case l of
      PosLit_ _ as -> foldr getVarNames ns as
      NegLit_ _ as -> foldr getVarNames ns as
      _            -> ns
```

4.2.7 Grounding

A substitution is a function which performs this operation. Substitutions may be composed to handle more than one variable substitution.

```
type Subst a = a -> a
```

To "ground" is to substitute a variable with a constant.

```
class HasVarNames a => Groundable a where
```

`ground v c x` returns the thing `x` with all occurrences of variable `v` replaced by constant `c`.

```
   ground :: VariableName -> ConstantName -> Subst a
```

`groundAll cs x` returns all of the ground instances of `x`, obtained by substituting the constants in `cs` for the variables in `x`.

```
   groundAll :: [ConstantName] -> a -> [a]
   groundAll cs x =
      let vs = flattenSS $ getVarNames x emptySS
          nvs = length vs
      in if nvs == 0 then
            [x]
         else
            [foldl (.) id (zipWith ground vs cs') x
            | cs' <- cartProd (take nvs (repeat cs))]

instance Groundable Argument where

   ground v c a = case a of
      Const c' -> a
      Var   v' -> if v == v' then Const c else a

instance Groundable Literal where

   ground v c l = case l of
      PosLit_ n as -> PosLit_ n (map (ground v c) as)
      NegLit_ n as -> NegLit_ n (map (ground v c) as)
      _          -> l
```

4.2.8 Instance declarations

Textual output of literals is performed with the `show` function, which is a method of class `Show`.

```
instance Show Argument where

   showsPrec p a
      = case a of
           Const n -> showString n
           Var   n -> showString n

instance Show Literal where

   showsPrec p l
      = case l of
           PosLit n ->
              showString n
           NegLit n ->
              showChar '~' . showString n
           PosLit_ n as ->
              showString n . showChar '('
              . showWithSep ", " as . showChar ')'
           NegLit_ n as ->
              showChar '~' . showString n . showChar '('
              . showWithSep ", " as . showChar ')'

instance Show PrologLiteral where

   showsPrec p (PrologLiteral l)
      = case l of
           PosLit n ->
              showString n
           NegLit n ->
              showString "neg " . showString n
           PosLit_ n as ->
              showString n . showChar '('
              . showWithSep ", " as . showChar ')'
           NegLit_ n as ->
              showString "neg " . showString n
              . showChar '(' . showWithSep ", " as
              . showChar ')'

instance Ord Literal where

   compare q q' = case q of
      PosLit a -> case q' of
         PosLit  b   -> compare a b
         PosLit_ b _ -> if a == b
            then LT
            else compare a b
         NegLit  _   -> GT
         NegLit_ _ _ -> GT
      NegLit a -> case q' of
         PosLit  _   -> LT
         PosLit_ _ _ -> LT
         NegLit  b   -> case compare a b of
            GT -> LT
            EQ -> EQ
            LT -> GT
         NegLit_ b _ -> case compare a b of
            GT -> LT
            EQ -> LT
            LT -> GT
      PosLit_ a ps -> case q' of
         PosLit  b    -> if a == b
            then GT
            else compare a b
         PosLit_ b qs -> if a == b
            then compare ps qs
            else compare a b
         NegLit  _    -> GT
         NegLit_ _ _  -> GT
      NegLit_ a ps -> case q' of
         PosLit  _    -> LT
         PosLit_ _ _  -> LT
         NegLit  b    -> case compare a b of
            LT -> GT
            EQ -> EQ
            GT -> LT
         NegLit_ b qs -> if a == b
            then case compare ps qs of
```

```
                LT -> GT
                EQ -> EQ
                GT -> LT
             else case compare a b of
                LT -> GT
                EQ -> EQ
                GT -> LT
```

```
instance DeepSeq Argument where

   deepSeq a x = case a of
      Const n -> deepSeq n x
      Var   n -> deepSeq n x
```

```
instance DeepSeq Literal where

   deepSeq l x = case l of
      PosLit   n     -> deepSeq n x
      PosLit_  n as  -> deepSeq n $ deepSeq as x
      NegLit   n     -> deepSeq n x
      NegLit_  n as  -> deepSeq n $ deepSeq as x
```

4.3 Rules

Module DRule implements a data type for representing rules in Defeasible logic theories.

```
{-# LANGUAGE MultiParamTypeClasses,
             TypeSynonymInstances #-}
```

```
module DRule (
      DRule(..), Rule, PrologRule(..), ruleP, prologRuleP,
      IsRule(..)
   ) where
```

```
import ABR.Parser; import ABR.Showing
```

```
import Literal
```

```
infix 4 :->, :=>, :~>
```

4.3.1 Data type definitions

These data type declarations are suitable for easy manipulation of rules and as parse trees. This definition is parameterized with respect to the type of literal to be used. This makes this code a little more general, and makes possible some fancy stuff with multiparameter type classes later on.

```
data DRule lit =   ![lit] :-> !lit
                 | ![lit] :=> !lit
                 | ![lit] :~> !lit
                 deriving (Eq, Ord)
```

As shorthand, use this type synonym.

```
type Rule = DRule Literal
```

To mark a rule for Prolog output, wrap up in this type.

```
newtype PrologRule = PrologRule Rule
```

4.3.2 Parsers

The syntax for a rule is:

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

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

which is implemented:

```
antecedentP :: Parser [Literal]
antecedentP
   =     literalP "symbol" "{" <*> literalP "symbol" "}"
         #> []
     <|> literalP "symbol" "{"
         *> (pLiteralP <*>
            many (literalP "symbol" "," *> pLiteralP))
         <* nofail (literalP "symbol" "}")
         @> cons
     <|> pLiteralP <*>
         many (literalP "symbol" "," *> pLiteralP)
         @> cons
     <|> epsilonA
         #> []
```

```
ruleP :: Parser Rule
ruleP = antecedentP
        <*> (     literalP "symbol" "->"
              <|> literalP "symbol" "=>"
              <|> literalP "symbol" "~>") <*> pLiteralP
        @> (\(as,((_,arrow,_),c)) -> (case arrow of
                 "->" -> (:->)
                 "=>" -> (:=>)
                 "~>" -> (:~>)
              ) as c)
```

The alternate d-Prolog-compatible syntax for a rule is:

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

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

which is implemented:

```
prologAntecedentP :: Parser [Literal]
prologAntecedentP
   =     literalP "name1" "true"
         #> []
     <|> prologLiteralP
         <*> many (literalP "symbol" ","
                  *> nofail' "literal expected"
                          prologLiteralP)
         @> cons
```

```
prologRuleP :: Parser Rule
prologRuleP
   = prologLiteralP
     <*> (     literalP "symbol" ":-"
           <|> literalP "symbol" ":="
           <|> literalP "symbol" ":^")
     <*> prologAntecedentP
     @> (\(c,((_,arrow,_),as)) -> (case arrow of
             ":-" -> (:->)
             ":=" -> (:=>)
             ":^" -> (:~>)
          ) as c)
```

4.3.3 Properties of rules

The IsRule class collects the properties of rules and rule-like types. is$X$ r returns True iff r is an X. antecedent r returns the list of literals which are the antecedents of rule r. consequent r returns the literal which is the consequent of r. This is a multi-parameter type class, which relies on Haskell extensions.

```
class IsRule rul lit where

   isStrict :: rul lit -> Bool

   isPlausible :: rul lit -> Bool

   isDefeater :: rul lit -> Bool

   antecedent :: rul lit -> [lit]

   consequent :: rul lit -> lit
```

```
instance IsRule DRule Literal where

   isStrict r = case r of
                   _ :-> _ -> True
                   _       -> False

   isPlausible r = case r of
                      _ :=> _ -> True
                      _       -> False

   isDefeater r = case r of
                     _ :~> _ -> True
                     _       -> False

   antecedent r = case r of
                     a :-> _ -> a
                     a :=> _ -> a
                     a :~> _ -> a

   consequent r = case r of
                     _ :-> c -> c
                     _ :=> c -> c
                     _ :~> c -> c
```

4.3.4 Instance declarations

Conversion of rules to printable representations is implemented by declaring these instances of class Show.

```
instance Show Rule where

   showsPrec p rule = case rule of
         (a :-> c) -> showsAntecedent a . showString " -> "
                      . shows c
         (a :=> c) -> showsAntecedent a . showString " => "
                      . shows c
         (a :~> c) -> showsAntecedent a . showString " ~> "
                      . shows c
      where
      showsAntecedent as = showWithSep ", " as
```

```
instance Show PrologRule where

   showsPrec p (PrologRule rule) = case rule of
         (a :-> c) -> shows (PrologLiteral c)
                      . showString " :- "
                      . showsAntecedent a
         (a :=> c) -> shows (PrologLiteral c)
                      . showString " := "
                      . showsAntecedent a
         (a :~> c) -> shows (PrologLiteral c)
                      . showString " :^ "
                      . showsAntecedent a
      where
      showsAntecedent as = case as of
         [] -> showString "true"
         _  -> showWithSep ", " $ map PrologLiteral as
```

Introducing type Rule to class HasLits enables the extraction of all the unique literals in a rule.

```
instance HasLits Rule where

   getLits r t = case r of
      (a :-> c) -> foldr getLits (getLits c t) a
      (a :=> c) -> foldr getLits (getLits c t) a
      (a :~> c) -> foldr getLits (getLits c t) a
```

Extracting constant names.

```
instance HasConstNames Rule where

   getConstNames r t = case r of
      (a :-> c) ->
         foldr getConstNames (getConstNames c t) a
      (a :=> c) ->
         foldr getConstNames (getConstNames c t) a
      (a :~> c) ->
         foldr getConstNames (getConstNames c t) a
```

Extracting variable names.

```
instance HasVarNames Rule where

   getVarNames r t = case r of
      (a :-> c) -> foldr getVarNames (getVarNames c t) a
      (a :=> c) -> foldr getVarNames (getVarNames c t) a
      (a :~> c) -> foldr getVarNames (getVarNames c t) a
```

Grounding.

```
instance Groundable Rule where

   ground v c r = case r of
      qs :-> q -> map (ground v c) qs :-> ground v c q
      qs :=> q -> map (ground v c) qs :=> ground v c q
      qs :~> q -> map (ground v c) qs :~> ground v c q
```

4.4 Labels

Labels are used to tag rules.

```
module Label(
      LabelName, Label(Label), labelP,
      HasLabelNames(getLabelNames)
   ) where

import ABR.SparseSet; import ABR.Parser; import ABR.DeepSeq
```

4.4.1 Data type definition

A Label is just a string with a constructor to tag it as a label.

```
type LabelName = String

newtype Label = Label LabelName
                deriving (Eq, Ord)
```

4.4.2 Parsers

Labels should start with upper case letters. The syntax for a label is:

```
label ::= name2
```

which is implemented:

```
labelP :: Parser Label
labelP = tagP "name2" @> (\(_,n,_) -> Label n)
```

4.4.3 Collecting label names

To extract the set of unique name strings from labels or objects that contains labels, use getLabelNames, which accumulates names in a set.

```
class HasLabelNames a where

   getLabelNames :: a -> SparseSet LabelName
                     -> SparseSet LabelName

instance HasLabelNames Label where

   getLabelNames (Label n) = insertSS n
```

4.4.4 Instance declarations

```
instance Show Label where

   showsPrec p (Label n) = showString n
```

Forced evaluation

```
instance DeepSeq Label where

   deepSeq (Label n) x = deepSeq n x
```

4.5 Priorities

The Priority module defines a data type for representing the superiority relation for Defeasible and Plausible logic.

```
module Priority(
      Priority(( :> )), priorityP, countPriorities, cycles
   ) where

import ABR.Data.BSTree; import ABR.Parser
import ABR.DeepSeq

import Label

infix 4 :>
```

4.5.1 Data type definition

```
data Priority = !Label :> !Label
                deriving (Eq, Ord)
```

4.5.2 Parser

The syntax for a priority declaration is:

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

which is implemented:

```
priorityP :: Parser Priority
priorityP = labelP <*> literalP "symbol" ">"
            *> nofail' "label expected" labelP
            @> uncurry (:>)
```

4.5.3 Testing for cycles

In the Defeasible and Plausible logics, cycles in the priority relation are not permitted. The following is sufficient to detect cycles, but can not identify only those priorities that contribute directly to cycles.

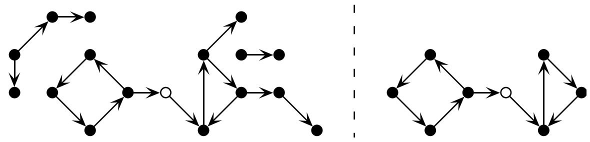

Figure 1: A priority relation represented as directed graphs, before and after cycle detection.

The algorithm is to count the number of times each label is superior and inferior. Then delete any priority where the label at either end has either count equal to zero. Repeat until no progress is made. Then all remaining priorities are either involved in a cycle or involved in a connection between two cycles. For example, figure 1 shows a priority relation before and after the application of `cycles` as directed graphs. The nodes are labels. The edges are priorities. The unfilled node is not involved in any cycle but is not removed.

```
type LCount = Int  -- # times on left of :>
type RCount = Int  -- # times on right of :>
type LRCounts = BSTree Label (LCount,RCount)

countPriorities :: [Priority] -> LRCounts
countPriorities
   = let count1L :: Label -> LRCounts -> LRCounts
         count1L l
            = updateBST (\ _ (l,r) -> (l + 1, r)) l (1,0)
         count1R :: Label -> LRCounts -> LRCounts
         count1R l
            = updateBST (\ _ (l,r) -> (l, r + 1)) l (0,1)
         count1 :: Priority -> LRCounts -> LRCounts
         count1 (lL :> lR) = (count1L lL) . (count1R lR)
     in foldr count1 emptyBST

pruneAcyclicLabels :: [Priority] -> [Priority]
pruneAcyclicLabels ps
   = let counts :: LRCounts
         counts = countPriorities ps
         isCyclic :: Priority -> Bool
         isCyclic (lL :> lR)
            = let Just (nLL,nLR) = lookupBST lL counts
                  Just (nRL,nRR) = lookupBST lR counts
              in nLL /= 0 && nLR /= 0 && nRL /= 0
                 && nRR /= 0
     in filter isCyclic ps
```

`cycles ps` returns the priorities in that may be involved in cycles in `ps`. The empty list is returned iff there are no cycles in `ps`.

```
cycles :: [Priority] -> [Priority]
cycles ps = let ps' = pruneAcyclicLabels ps
            in if length ps' == length ps
                  then ps
                  else cycles ps'
```

4.5.4 Instance declarations

```
instance Show Priority where

   showsPrec p (l :> l')
      = shows l . showString " > " . shows l'

instance HasLabelNames Priority where

   getLabelNames (l :> l')
      = getLabelNames l . getLabelNames l'
```

Forced Evaluation

```
instance DeepSeq Priority where

   deepSeq (l :> l') x = deepSeq l $ deepSeq l' x
```

4.6 Theories

The module `DTheory` defines the Defeasible logic theory data types.

```
{-# LANGUAGE MultiParamTypeClasses,
             TypeSynonymInstances #-}

module DTheory(
      LabeledRule(..), LRule, DTheory(..), Theory,
      PrologTheory(..), DeloresTheory(..), theoryP,
      cyclesCheck, groundCheck
   ) where

import List

import ABR.Data.BSTree; import ABR.SparseSet
import ABR.Parser; import ABR.Showing; import ABR.List
import ABR.Control.Check

import Label; import Literal; import Priority
import DRule
```

4.6.1 Data type definitions

A Defeasible theory consists of a set of facts (literals), a set of rules (some of which may be labeled), and a priority relation. These parameterized type definitions make possible some fancy multi-parameter class definitions later on.

```
data LabeledRule lit = Rule !Label !(DRule lit)
                       deriving (Eq)

data DTheory rul = Theory [Literal] [rul] [Priority]
                   deriving (Eq)
```

For shorthand use:

```
type LRule = LabeledRule Literal
type Theory = DTheory LRule
```

A `Statement` is an intermediate data structure used while parsing.

```
data Statement =   Fact        !Literal
                 | LabeledRule !LRule
                 | Priority    !Priority
                 | Superiority !Rule !Rule
```

The wrapper types `PrologTheory` and `PrologPriority` are used to mark theories and priorities for Prolog syntax output.

```
newtype PrologTheory = PrologTheory Theory

data PrologPriority = !Rule :>> !Rule
```

The wrapper types `DeloresTheory` and `DeloresRule` are used to mark theories and rules for *Delores* syntax output.

```
newtype DeloresTheory = DeloresTheory Theory

newtype DeloresRule = DeloresRule LRule
```

4.6.2 Parser

Syntax:

```
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 "."}
```

Implemented:

```
rule'P :: Parser Rule
rule'P = prologRuleP <|> ruleP
```

```
prologSuperiorityP :: Parser Statement
prologSuperiorityP
   = literalP "name1" "sup"
     *> nofail (literalP "symbol" "(")
     *> nofail (literalP "symbol" "(")
     *> nofail' "rule expected" rule'P
     <*> (nofail (literalP "symbol" ")")
          *> nofail (literalP "symbol" ",")
          *> nofail (literalP "symbol" "(")
          *> nofail' "rule expected" rule'P
          <* nofail (literalP "symbol" ")")
          <* nofail (literalP "symbol" ")"))
     @> (\(r1,r2) -> Superiority r1 r2)

factP :: Parser Literal
factP = prologLiteralP <|> pLiteralP

labeledRuleP :: Parser (LabeledRule Literal)
labeledRuleP = optional (labelP <* literalP "symbol" ":")
               <*> rule'P
               @> (\(ls,r) -> case ls of
                     []  -> Rule (Label "") r
                     [l] -> Rule l r
                  )

statementP :: Parser Statement
statementP =     prologSuperiorityP
           <|> labeledRuleP @> LabeledRule
           <|> factP        @> Fact
           <|> priorityP    @> Priority

theoryP :: Parser Theory
theoryP
   = total (many (statementP <* nofail (
                                  literalP "symbol" ".")))
     @> makeTheory
     where
     makeTheory :: [Statement] -> Theory
     makeTheory = (\(fs,rs,ps) -> Theory fs rs ps)
                  . pass2 0 . pass1
     pass1 :: [Statement]
        -> ([Literal], [LRule], [Priority], [(Rule,Rule)])
     pass1 []
        = ([],[],[],[])
     pass1 (s:ss)
        = case pass1 ss of
             (fs,rs,ps,sups) ->
                case s of
                   Fact f ->
                      ((f : fs), rs, ps, sups)
                   LabeledRule r ->
                      (fs, (r : rs), ps, sups)
                   Priority p ->
                      (fs, rs, (p : ps), sups)
                   Superiority r1 r2 ->
                      (fs, rs, ps, (r1,r2) : sups)
     pass2 :: Int
        -> ([Literal], [LRule], [Priority], [(Rule,Rule)])
        -> ([Literal], [LRule], [Priority])
     pass2 _ (fs, rs, ps, [])
        = (fs, rs, ps)
     pass2 n (fs, rs, ps, ((r1,r2):sups))
        = case findRule r1 rs n of
             (l1, rs', n') ->
                case findRule r2 rs' n' of
                   (l2, rs'', n'') ->
                      pass2 n'' (fs, rs'', (l1 :> l2) : ps,
                                 sups)
     findRule
        :: Rule -> [LRule] -> Int -> (Label, [LRule], Int)
     findRule _ [] _
        = error "rule in sup relation does not exist in \
                \theory."
     findRule r' ((Rule label r):rs) n
        | r' /= r
          = case findRule r' rs n of
               (l, rs', n') ->
                  (l, (Rule label r) : rs', n')
        | otherwise
          = case label of
               Label "" ->
                  let l = Label $ "R__" ++ show n
                  in (l, (Rule l r) : rs, n + 1)
               _ ->
                  (label, (Rule label r) : rs, n)
```

4.6.3 Checking for cycles

`cyclesCheck t` detects cycles in the priority relation of theory `t`. The theory is returned passed or, on failure, the showed list of priorities involved in cycles is returned.

```
cyclesCheck :: Check Theory Theory String
cyclesCheck t@(Theory _ _ ps)
   = case cycles ps of
        []  -> CheckPass t
        ps' -> CheckFail $ show ps'
```

4.6.4 Labeled rule manipulations

`dropLabel lr` converts a labeled rule `lr` to a `Rule`.

```
dropLabel :: LRule -> Rule
dropLabel (Rule _ r) = r
```

4.6.5 Grounding all variables

The `groundCheck` passes a theory if it can replace all facts and rules with ground instances generated from the constants appearing in the theory. If there are variables, but no constants the check fails.

```
groundCheck :: Check Theory Theory String
groundCheck t@(Theory fs rs ps)
   = let cs = flattenSS $ getConstNames t emptySS
         vs = getVarNames t emptySS
         fs' = concat $ map (groundAll cs) fs
         rs' = concat $ map (groundAll cs) rs
         renumber :: Int -> [LRule] -> ([LRule],
            BSTree Label (SparseSet Label))
         renumber n rs = case rs of
            []                        ->
               ([], emptyBST)
            ((Rule (Label "") r) : rs) ->
               let (rs', t) = renumber (n+1) rs
               in (Rule (Label ("R" ++ show n)) r : rs', t)
            ((Rule l r) : rs)         ->
               let (rs', t) = renumber (n+1) rs
                   l' = Label ("R" ++ show n)
                   sl' = insertSS l' emptySS
               in (Rule l' r : rs',
                   updateBST unionSS l sl' t)
         (rs'', lmap) = renumber 0 rs'
         dupPri :: Priority -> [Priority]
         dupPri (l :> l')
            = let Just lS = lookupBST l lmap
                  ls = flattenSS lS
                  Just lS' = lookupBST l' lmap
                  ls' = flattenSS lS'
              in [l :> l' | l <- ls, l' <- ls']
         ps' = concat $ map dupPri ps
     in if nullSS vs && not (null cs) then
           CheckFail "Can't ground variables. \
                     \No constants."
        else
           CheckPass (Theory fs' rs'' ps')
```

4.6.6 Instance declarations

Textual output.

```
instance Show LRule where

   showsPrec p (Rule l r)
      = case l of
           Label "" -> shows r
           _        -> shows l . showString ": " . shows r

instance Show Theory where

   showsPrec p (Theory fs rs ps)
      = showWithTerm ".\n" fs . showWithTerm ".\n" rs
        . showWithTerm ".\n" ps

instance Show PrologPriority where
```

```
  showsPrec p (r1 :>> r2)
     = showString "sup((" . shows (PrologRule r1)
       . showString "), (" . shows (PrologRule r2)
       . showString "))"

instance Show PrologTheory where

  showsPrec p (PrologTheory (Theory fs rs ps))
     = showString header
       . showWithTerm ".\n" (map PrologLiteral fs)
       . showWithTerm ".\n" (map pr rs)
       . showWithTerm ".\n" (map pp ps)
       where
       header :: String
       header = "% declarations needed for Sicstus 3\n\
                \:- multifile (neg)/1, (:=)/2, (:^)/2.\n\
                \:- dynamic (neg)/1, (:=)/2, (:^)/2.\n\n"
       tree :: BSTree Label Rule
       tree = foldr (\(Rule l r) ->
              updateBST (\x _ -> x) l r) emptyBST rs
       pr :: LRule -> PrologRule
       pr = PrologRule . dropLabel
       pp :: Priority -> PrologPriority
       pp (l1 :> l2)
          = case lookupBST l1 tree of
              Just r1 -> case lookupBST l2 tree of
                Just r2 -> r1 :>> r2

instance Show DeloresRule where

  showsPrec p (DeloresRule (Rule l r))
     = case l of
         Label "" -> shows (PrologRule r)
         _        -> shows l . showString ": "
                     . shows (PrologRule r)

instance Show DeloresTheory where

  showsPrec p (DeloresTheory (Theory fs rs ps))
     = showWithTerm ".\n" (map PrologLiteral fs)
       . showWithTerm ".\n" (map DeloresRule rs)
       . showWithTerm ".\n" ps
       . showString "infer.\n"
```

Extracting literal names.

```
instance HasLits LRule where

  getLits (Rule _ r) = getLits r

instance HasLits Theory where

  getLits (Theory fs rs ps) t
     = foldr getLits (foldr getLits t fs) rs
```

Extracting constant names.

```
instance HasConstNames LRule where

  getConstNames (Rule _ r) = getConstNames r

instance HasConstNames Theory where

  getConstNames (Theory fs rs ps) t
     = foldr getConstNames (foldr getConstNames t fs) rs
```

Extracting variable names.

```
instance HasVarNames LRule where

  getVarNames (Rule _ r) = getVarNames r

instance HasVarNames Theory where

  getVarNames (Theory fs rs ps) t
     = foldr getVarNames (foldr getVarNames t fs) rs
```

Extracting Label names.

```
instance HasLabelNames LRule where

  getLabelNames (Rule l _)
     = getLabelNames l

instance HasLabelNames Theory where

  getLabelNames (Theory _ rs ps) t
     = foldr getLabelNames (foldr getLabelNames t rs) ps
```

`LabeledRule`s are still rules:

```
instance IsRule LabeledRule Literal where

  isStrict    = isStrict    . dropLabel
  isPlausible = isPlausible . dropLabel
  isDefeater  = isDefeater  . dropLabel
  antecedent  = antecedent  . dropLabel
  consequent  = consequent  . dropLabel
```

Grounding.

```
instance Groundable LRule where

  ground v c (Rule l r) = Rule l (ground v c r)
```

4.7 DefeasibleParser

See the user's guide (section 3.5) for a description of this module.

```
module Main (main) where

import System

import ABR.Parser; import ABR.Control.Check
import ABR.Parser.Checks

import DefeasibleLexer; import DTheory

main :: IO ()
main = do
  paths <- getArgs
  if null paths
     then do
        source <- getContents
        parse source
     else run paths

run :: [FilePath] -> IO ()
run = mapM_ run1

run1 :: FilePath -> IO ()
run1 path = do
  putStr $ "Theory file name: " ++ path ++ "\n"
  source <- readFile path
  parse source

parse :: String -> IO ()
parse source = do
  case checkParse lexerL (total theoryP) source of
     CheckFail msg  -> putStrLn msg
     CheckPass t -> do
        putStrLn "\nParsed OK.\n"
        putStrLn $ show t
        case cyclesCheck t of
           CheckFail msg ->
              putStrLn $ "\nCycles in priorities: " ++ msg
           CheckPass t'  -> do
              putStrLn "\nNo cyclic priorities. \
                       \Grounding variables:\n"
              case groundCheck t of
                 CheckFail msg -> putStrLn msg
                 CheckPass t''  -> putStr $ show t''
```

4.8 Threaded Tests

The module `ThreadedTest` implements abstractions and combiners that allow the treading of proofs and state through monads; for example the `IO` or `ST` monads.

```
module ThreadedTest(
     ThreadedTest,
     ThreadedResult(mkTest, (&&&), (|||),
                    fA', tE', fA, tE)
  ) where
```

4.8.1 Data types

A test must be performed. We need the result (of type `r`) returned, and a state (of type `s`) may be updated. There may be other side effects, so all of this is threaded through some monad `m`.

```
type ThreadedTest m r s = s -> m (r, s)
```

4.8.2 Combining threaded tests

`mkTest b` promotes some simple Boolean result `b` to a `ThreadedTest`. `&&&` and `|||` conjoin and disjoin two threaded tests. `fA` and `tE` are $\forall$ and $\exists$ respectively.

```
class ThreadedResult r where

   infixr 3 &&&
   infixr 2 |||

   mkTest :: Monad m => Bool -> ThreadedTest m r s

   (&&&), (|||) :: Monad m => ThreadedTest m r s
                           -> ThreadedTest m r s
                           -> ThreadedTest m r s

   fA', tE' :: Monad m => [ThreadedTest m r s]
                       -> ThreadedTest m r s

   fA, tE :: Monad m => [b] -> (b -> ThreadedTest m r s)
                     -> ThreadedTest m r s
   fA xs p = fA' (map p xs)
   tE xs p = tE' (map p xs)
```

4.9 Inference Conditions

Module `DInference` defines the inference conditions for Defeasible logic.

```
{-# LANGUAGE MultiParamTypeClasses #-}

module DInference(
      ProofSymbol(..), Tagged(..), taggedLiteralP,
      DefeasibleLogic(..)
   ) where

import Ix

import ABR.Parser

import Literal; import ThreadedTest
```

4.9.1 Data type definitions

A tagged literal consists of a literal, a symbol to indicate the level of proof required, and a + or − sign to indicate that a proof or proof that it can not be proved is required. The proof symbols are defined by table 7.

```
data ProofSymbol
   =   PS_D | PS_d | PS_da | PS_S | PS_dt
     deriving (Eq, Ord, Ix)

data (Eq a, Show a, Ord a) => Tagged a
   =   Plus  !ProofSymbol !a
     | Minus !ProofSymbol !a
     deriving (Eq, Ord)
```

constructor	*symbol*	*meaning*
`PS_D`	Δ	strict
`PS_d`	∂	defeasible
`PS_dt`	∂_{-t}	defeasible variant without team defeat
`PS_da`	δ	defeasible variant with ambiguity propagation
`PS_S`	$\int$	defeasible variant: support

Table 7: The proof symbols and their Haskell representation and meanings.

4.9.2 Parser

The syntax for a tagged literal is:

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

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

which is implemented:

```
proofSymbolP :: Parser ProofSymbol
proofSymbolP
   =     literalP "name2" "D"    #> PS_D
     <|> literalP "name1" "d"    #> PS_d
     <|> literalP "name1" "da"   #> PS_da
     <|> literalP "name2" "S"    #> PS_S
     <|> literalP "name1" "dt"   #> PS_dt

taggedLiteralP :: Parser (Tagged Literal)
taggedLiteralP
   = (literalP "symbol" "+" <|> literalP "symbol" "-")
     <*> nofail' "proof symbol expected" proofSymbolP
     <*> nofail' "literal expected" pLiteralP
     @> (\((_,c,_),(ps,l)) -> case c of
           "+" -> Plus ps l
           "-" -> Minus ps l)
```

4.9.3 Overloaded functions

Class `DefeasibleLogic` overloads the some functions that the inference conditions are defined in terms of to hide (and generalize) the representation of theories, labels and rules. Then the inference conditions need only be specified once. This class has multiple type parameters, and therefore relies on Hugs and GHC extensions. The parameters `th`, `rul`, and `lit` are the names of the theory, rule, and literal types. The type for rules must be parameterized by the type for literals, and the type for theories must be parameterized by the type for rules.

```
class (Negatable lit, Show lit, Eq lit, Ord lit) =>
   DefeasibleLogic th rul lit where

   infix 6 |--
```

The following methods need to be defined for instances of this class.

`isFactIn q t` is a test whether `q` is a fact in theory `t`. `notFactIn q t` returns the opposite result.

```
   isFactIn, notFactIn :: (Monad m, ThreadedResult r) =>
      lit -> th (rul lit)  -> ThreadedTest m r s
```

`rq t q` returns the list of rules in `t` that have consequent `q`. `rsq t q` returns the list of strict rules in `t` that have consequent `q`. `rsdq t q` returns the list of rules in `t` that have consequent `q` and are strict or defeasible.

```
   rq, rsq, rsdq :: th (rul lit)  -> lit -> [rul lit]
```

`ants t r` returns the list of literals that are the antecedents of rule `r` in theory `t`.

```
   ants :: th (rul lit)  -> rul lit -> [lit]
```

`beats t r1 r2` is a test whether there exists in `t` a priority that asserts that `r1` is superior to `r2`. `notBeats t r1 r2` returns the opposite result.

```
   beats, notBeats :: (Monad m, ThreadedResult r) =>
      th (rul lit) -> rul lit -> rul lit
      -> ThreadedTest m r s
```

4.9.4 Inference Conditions

`t |-- tl (|-)` is a test whether the tagged literal `tl` can be proved from theory `t`. The definition of this function is shown in figure 2 along with the inference conditions it implements. `|-` is the main proof function that is mutually recursive with this one. `|-` handles all state manipulations and/or I/O.

```
   (|--) :: (Monad m, ThreadedResult r) =>
      th (rul lit) -> Tagged lit -> (th (rul lit) ->
      Tagged lit -> ThreadedTest m r s)
      -> ThreadedTest m r s
```

Additional inference conditions for variants of Defeasible logic that feature ambiguity propagation ($\pm\delta$ and $\pm\int$) and variants that do not feature team defeat ($\pm\partial_{-t}$) have also been implemented and are shown in figure 3.

$+\Delta$:
```
(|--) t (Plus PS_D q) (|-)
   = q `isFactIn` t |||
     tE (rsq t q) (\r -> fA (ants t r) (\a -> t |- Plus PS_D a))
```

$-\Delta$:
```
(|--) t (Minus PS_D q) (|-)
   = q `notFactIn` t &&&
     fA (rsq t q) (\r -> tE (ants t r) (\a -> t |- Minus PS_D a))
```

$+\partial$:
```
(|--) 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))
```

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

Figure 2: Inference conditions for defeasible logic.

4.9.5 Instance declarations

Textual output.

```
instance Show ProofSymbol where

   showsPrec p ps
      = case ps of
           PS_D   -> showChar   'D'
           PS_d   -> showChar   'd'
           PS_da  -> showString "da"
           PS_S   -> showChar   'S'
           PS_dt  -> showString "dt"

instance (Show a, Eq a, Ord a) => Show (Tagged a) where

   showsPrec p t = case t of
      Plus ps q  -> showChar '+' . shows ps . showChar ' '
                    . shows q
      Minus ps q -> showChar '-' . shows ps . showChar ' '
                    . shows q
```

Extracting literal names

```
instance (HasLits a, Show a, Eq a, Ord a) =>
   HasLits (Tagged a) where

   getLits t s = case t of
      Plus  _ q -> getLits q s
      Minus _ q -> getLits q s
```

Detecting variable names.

```
instance (HasVarNames a, Show a, Ord a) =>
   HasVarNames (Tagged a) where

   getVarNames tl s = case tl of
      Plus  _ l -> getVarNames l s
      Minus _ l -> getVarNames l s
```

4.10 Histories

The module `History` implements a data structure for storage and recall of prior proof results.

```
module History(
      History, emptyHistory, addProof, getResult,
      retractProof
   ) where

import ABR.Data.BSTree
```

4.10.1 Data types

A history is a record of the result of each proof attempted.

```
type History proof result = BSTree proof result
```

4.10.2 Methods

This is an empty History.

```
emptyHistory :: Ord proof => History proof result
emptyHistory = emptyBST
```

This adds a proof and status to the History.

```
addProof :: Ord proof => History proof result -> proof
            -> result -> History proof result
addProof h p s = updateBST (\x _ -> x) p s h
```

This retrieves a `ProofResult`.

```
getResult :: Ord proof => History proof result -> proof
             -> Maybe result
getResult h p = lookupBST p h
```

`retractProof` h p retracts the result stored in h for p if it exists.

```
retractProof :: Ord proof => History proof result
   -> proof -> History proof result
retractProof h p = deleteBST p h
```

4.11 Proof Results

The module `ProofResult` implements a data type that represents all the possible results on attempting a proof.

```
module ProofResult(
      ProofResult(..), WFResult(..)
   ) where

import ThreadedTest
```

4.11.1 Data type

An attempted proof may at a given point in time, have been definitely proved, definitely not proved, known to loop, or be still in progress.

$+\delta$:
```
(|--) t (Plus PS_da q)  (|-)
   = t |- Plus PS_D q |||
        tE (rsdq t q) (\r -> fA (ants t r) (\a -> t |- Plus PS_da a)) &&&
        t |- Minus PS_D (neg q) &&&
        fA (rq t (neg q)) (\s ->
           tE (ants t s) (\a -> t |- Minus PS_S a) |||
           tE (rsdq t q) (\u ->
              fA (ants t u) (\a -> t |- Plus PS_da a) &&& beats t u s))
```

$-\delta$:
```
(|--) t (Minus PS_da q)  (|-)
   = t |- Minus PS_D q &&& (
        fA (rsdq t q) (\r -> tE (ants t r) (\a -> t |- Minus PS_da a)) |||
        t |- Plus PS_D (neg q) |||
        tE (rq t (neg q)) (\s ->
           fA (ants t s) (\a -> t |- Plus PS_S a) &&&
           fA (rsdq t q) (\u ->
              tE (ants t u) (\a -> t |- Minus PS_da a) ||| notBeats t u s)))
```

$+\int$:
```
(|--) t (Plus PS_S q)  (|-)
   = t |- Plus PS_D q |||
        tE (rsdq t q) (\r ->
           fA (ants t r) (\a -> t |- Plus PS_S a) &&&
           fA (rq t (neg q)) (\s ->
              tE (ants t s) (\a -> t |- Minus PS_da a) ||| notBeats t s r))
```

$-\int$:
```
(|--) t (Minus PS_S q)  (|-)
   = t |- Minus PS_D q &&&
        fA (rsdq t q) (\r ->
           tE (ants t r) (\a -> t |- Minus PS_S a) |||
           tE (rq t (neg q)) (\s ->
              fA (ants t s) (\a -> t |- Plus PS_da a) &&& beats t s r))
```

$+\partial_{-t}$:
```
(|--) t (Plus PS_dt q) (|-)
   = t |- Plus PS_D q |||
        tE (rsdq t q) (\r -> fA (ants t r) (\a -> t |- Plus PS_dt a) &&&
        t |- Minus PS_D (neg q) &&&
        fA (rq t (neg q)) (\s ->
           beats t r s |||
           tE (ants t s) (\a -> t |- Minus PS_dt a)))
```

$-\partial_{-t}$:
```
(|--) t (Minus PS_dt q) (|-)
   = t |- Minus PS_D q &&& (
        fA (rsdq t q) (\r -> tE (ants t r) (\a -> t |- Minus PS_dt a) |||
        t |- Plus PS_D (neg q) |||
        tE (rq t (neg q)) (\s ->
           notBeats t r s &&&
           fA (ants t s) (\a -> t |- Plus PS_dt a))))
```

Figure 3: Inference conditions for variants of defeasible logic.

```
data ProofResult =
     Yes          -- Proved True
   | No           -- Definitely False
   | Bottom       -- Loop detected
   | Pending      -- Still waiting to find out
   | NotAttempted -- Proof never attempted
   deriving (Eq, Ord)
```

4.11.2 Instance declarations

Textual output.

```
instance Show ProofResult where
   showsPrec p Yes          = showString "Proved"
   showsPrec p No           = showString "Not proved"
   showsPrec p Bottom       = showString "Loops"
   showsPrec p Pending      = showString "Pending"
   showsPrec p NotAttempted = showString "Not Attempted"
```

Threading tests.

```
instance ThreadedResult ProofResult where

   mkTest b s = return (if b then Yes else No, s)

   (&&&) 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 &&&"
         NotAttempted -> error "NotAttempted in &&&"

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

```
   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'"
            NotAttempted -> error "NotAttempted in fA'"

   tE' ts s = case ts of
      []         -> return (No, s)
      [t]        -> t s
      (t1:t2:ts) -> do
         (r1,s1) <- t1 s
         case r1 of
            Yes          -> return (r1,s1)
            No           -> tE' (t2:ts) s1
            Bottom       -> tE' (t2:ts) s1
            Pending      -> error "Pending in tE'"
            NotAttempted -> error "NotAttempted in tE'"
```

4.11.3 Well-founded variant

This variant proof result type allows the implementation of well-founded provers. This makes a difference only when loop detection is available. The result bottom (loops) is not propagated and gets changed to not proved.

```
data WFResult =
     WFYes     -- Proved True
   | WFNo      -- Definitely False
   | WFBottom  -- Loop detected
   | WFPending -- Still waiting to find out
   | WFNotAtt  -- Proof never attempted
   deriving (Eq, Ord)

instance Show WFResult where
   showsPrec p WFYes     = showString "Proved"
   showsPrec p WFNo      = showString "Not proved"
   showsPrec p WFBottom  = showString "Loops"
   showsPrec p WFPending = showString "Pending"
   showsPrec p WFNotAtt  = showString "Not Attempted"

instance ThreadedResult WFResult where

   mkTest b s = return (if b then WFYes else WFNo, s)

   (&&&) t1 t2 s = do
      (r1,s1) <- t1 s
      case r1 of
         WFYes     -> do
            (r2,s2) <- t2 s1
            case r2 of
               WFYes     -> return (r2, s2)
               WFNo      -> return (r2, s2)
               WFBottom  -> return (WFNo, s2)
               WFPending -> error "Pending in &&&"
               WFNotAtt  -> error "NotAttempted in &&&"
         WFNo      -> return (r1, s1)
         WFBottom  -> return (WFNo, s1)
         WFPending -> error "Pending in &&&"
         WFNotAtt  -> error "NotAttempted in &&&"

   (|||) t1 t2 s = do
      (r1,s1) <- t1 s
      case r1 of
         WFYes     -> return (r1, s1)
         WFNo      -> do
            (r2,s2) <- t2 s1
            case r2 of
               WFYes     -> return (r2, s2)
               WFNo      -> return (r2, s2)
               WFBottom  -> return (WFNo, s2)
               WFPending -> error "Pending in |||"
               WFNotAtt  -> error "NotAttempted in |||"
         WFBottom  -> do
            (r2,s2) <- t2 s1
            case r2 of
               WFYes     -> return (r2, s2)
               WFNo      -> return (r2, s2)
               WFBottom  -> return (WFNo, s2)
               WFPending -> error "Pending in |||"
               WFNotAtt  -> error "NotAttempted in |||"
         WFPending -> error "Pending in |||"
         WFNotAtt  -> error "NotAttempted in |||"

   fA' ts s  = case ts of
      []         -> return (WFYes, s)
      [t]        -> do
         (r1,s1) <- t s
         case r1 of
            WFYes     -> return (r1,s1)
            WFNo      -> return (r1,s1)
            WFBottom  -> return (WFNo,s1)
            WFPending -> error "Pending in fA'"
            WFNotAtt  -> error "NotAttempted in fA'"
      (t1:t2:ts) -> do
         (r1,s1) <- t1 s
         case r1 of
            WFYes     -> fA' (t2:ts) s1
            WFNo      -> return (r1,s1)
            WFBottom  -> return (WFNo,s1)
            WFPending -> error "Pending in fA'"
            WFNotAtt  -> error "NotAttempted in fA'"

   tE' ts s = case ts of
      []         -> return (WFNo, s)
      [t]        -> do
         (r1,s1) <- t s
         case r1 of
            WFYes     -> return (r1,s1)
            WFNo      -> return (r1,s1)
            WFBottom  -> return (WFNo,s1)
            WFPending -> error "Pending in tE'"
            WFNotAtt  -> error "NotAttempted in tE'"
      (t1:t2:ts) -> do
         (r1,s1) <- t1 s
         case r1 of
            WFYes     -> return (r1,s1)
            WFNo      -> tE' (t2:ts) s1
            WFBottom  -> tE' (t2:ts) s1
            WFPending -> error "Pending in tE'"
            WFNotAtt  -> error "NotAttempted in tE'"
```

4.12 DProve

This module implements provers for Defeasible logic.

```
{-# LANGUAGE MultiParamTypeClasses #-}

module DProve(
      prove, Hist, WFHist
   ) where

import CPUTime

import ABR.Args; import ABR.Data.BSTree

import Literal; import DRule; import Priority
import ThreadedTest; import ProofResult
import History; import DTheory; import DInference
```

4.12.1 Defeasible logic instance

This instance implements the functions required by the inference conditions to use the simple theory type.

```
instance DefeasibleLogic DTheory LabeledRule Literal where

   isFactIn q (Theory fs _ _) = mkTest (q `elem` fs)

   notFactIn q (Theory fs _ _) = mkTest (q `notElem` fs)

   rq (Theory _ rs _) q
      = filter (\r -> consequent r == q) rs

   rsq (Theory _ rs _) q
      = filter (\r -> isStrict r && consequent r == q) rs

   rsdq (Theory _ rs _) q
      = filter (\r -> (isStrict r || isPlausible r)
                && consequent r == q) rs

   ants t r = antecedent r
```

```
beats (Theory _ _ ps) (Rule l _) (Rule l' _)
   = mkTest ((l :> l') `elem` ps)

notBeats (Theory _ _ ps) (Rule l _) (Rule l' _)
   = mkTest ((l :> l') `notElem` ps)
```

4.12.2 Provers

`prove_ t tl ()` returns `(r,())`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`. This is the simplest prover, with no trace, no history and therefore no loop checking, and not well founded.

```
prove_ :: Theory -> Tagged Literal
       -> ThreadedTest Maybe ProofResult ()
prove_ t tl () = (t |-- tl) prove_ ()
```

`prove_n t tl 0` returns `(r,ng)`, where `r` is the result of trying to prove tagged literal `tl` with theory `t` and `ng` is the number of subgoals required to do so.

```
prove_n :: Theory -> Tagged Literal
        -> ThreadedTest Maybe ProofResult Int
prove_n t tl ng = do
   (r, ng') <- (t |-- tl) prove_n ng
   return (r, ng' + 1)
```

`prove_t t tl ""` returns `(r,"")`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`. A trace is printed.

```
prove_t :: Theory -> Tagged Literal
        -> ThreadedTest IO ProofResult String
prove_t t tl indent = do
   putStrLn (indent ++ "To Prove: " ++ show tl)
   (r, _) <-
      (t |-- tl) prove_t (".  " ++ indent)
   putStrLn (indent ++ show r ++ ": " ++ show tl)
   return (r, indent)
```

`prove_nt t tl (0,"")` returns `(r,(ng,""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t` and `ng` is the number of subgoals required to do so. A trace is printed.

```
prove_nt :: Theory -> Tagged Literal
         -> ThreadedTest IO ProofResult (Int,String)
prove_nt t tl (ng,indent) = do
   putStrLn (indent ++ "To Prove: " ++ show tl)
   (r, (ng',_)) <-
      (t |-- tl) prove_nt (ng, ".  " ++ indent)
   putStrLn (indent ++ show r ++ ": " ++ show tl)
   return (r, (ng' + 1, indent))
```

This type is shorthand for the history that maps tagged literals to prior results.

```
type Hist = History (Tagged Literal) ProofResult
```

`prove_nh t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, but does not perform loop checking.

```
prove_nh :: Theory -> Tagged Literal
         -> ThreadedTest Maybe ProofResult (Int, Hist)
prove_nh t tl (ng,h) = case getResult h tl of
   Just r ->
      return (r, (ng,h))
   Nothing -> do
      (r, (ng',h')) <- (t |-- tl) prove_nh (ng,h)
      return (r, (ng' + 1, addProof h' tl r))
```

`prove_nht t tl (0,h,"")` returns `(r,(ng,h',""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, but does not perform loop checking. A trace is printed.

```
prove_nht
   :: Theory -> Tagged Literal
      -> ThreadedTest IO ProofResult (Int,Hist,String)
prove_nht t tl (ng,h,indent) = case getResult h tl of
   Just r -> do
      putStrLn (indent ++ show r ++ " previously: "
                ++ show tl)
      return (r, (ng,h,indent))
   Nothing -> do
      putStrLn (indent ++ "To Prove: " ++ show tl)
      (r, (ng',h',_)) <-
         (t |-- tl) prove_nht (ng, h, ".  " ++ indent)
      putStrLn (indent ++ show r ++ ": " ++ show tl)
      return (r, (ng' + 1, addProof h' tl r, indent))
```

`prove_nhl t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, and performs loop checking.

```
prove_nhl :: Theory -> Tagged Literal
          -> ThreadedTest Maybe ProofResult (Int, Hist)
prove_nhl t tl (ng,h) = case getResult h tl of
   Just Pending ->
      return (Bottom, (ng, addProof h tl Bottom))
   Just r ->
      return (r, (ng, h))
   Nothing -> do
      (r, (ng',h')) <-
          (t |-- tl) prove_nhl (ng, addProof h tl Pending)
      return (r, (ng' + 1, addProof h' tl r))
```

`prove_nhlt t tl (0,h,"")` returns `(r,(ng,h',""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, and performs loop checking. A trace is printed.

```
prove_nhlt
   :: Theory -> Tagged Literal
      -> ThreadedTest IO ProofResult (Int, Hist, String)
prove_nhlt t tl (ng,h,indent) = case getResult h tl of
   Just Pending -> do
      putStrLn (indent ++ "Loop detected: " ++ show tl)
      return (Bottom, (ng, addProof h tl Bottom, indent))
   Just r -> do
      putStrLn (indent ++ show r ++ " previously: "
                ++ show tl)
      return (r, (ng, h, indent))
   Nothing -> do
      putStrLn (indent ++ "To Prove: " ++ show tl)
      (r, (ng',h',_)) <-
         (t |-- tl) prove_nhlt
            (ng, addProof h tl Pending, ".  " ++ indent)
      putStrLn (indent ++ show r ++ ": " ++ show tl)
      let h'' = case r of
             Bottom -> h
             _      -> addProof h' tl r
      return (r, (ng' + 1, h'', indent))
```

4.12.3 Provers with well-founded semantics

This type is shorthand for the history that maps tagged literals to prior well-founded results.

```
type WFHist = History (Tagged Literal) WFResult
```

`prove_nhlw t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, performs loop checking, and has well-founded semantics.

```
prove_nhlw :: Theory -> Tagged Literal
   -> ThreadedTest Maybe WFResult (Int, WFHist)
prove_nhlw t tl (ng,h) = case getResult h tl of
   Just WFPending ->
      return (WFBottom, (ng, addProof h tl WFBottom))
   Just r ->
      return (r, (ng, h))
   Nothing -> do
      (r, (ng',h')) <-
          (t |-- tl) prove_nhlw (ng, addProof h tl WFPending)
      return (r, (ng' + 1, addProof h' tl r))
```

`prove_nhlwt t tl (0,h,"")` returns `(r,(ng,h',""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and h' is the final history. This prover avoids redoing prior proofs, performs loop checking, and has well-founded semantics. A trace is printed.

```
prove_nhlwt :: Theory -> Tagged Literal
   -> ThreadedTest IO WFResult (Int, WFHist, String)
prove_nhlwt t tl (ng,h,indent) = case getResult h tl of
   Just WFPending -> do
      putStrLn (indent ++ "Loop detected: " ++ show tl)
      return (WFBottom, (ng, addProof h tl WFBottom, indent))
   Just r -> do
      putStrLn (indent ++ show r ++ " previously: "
                ++ show tl)
      return (r, (ng, h, indent))
   Nothing -> do
      putStrLn (indent ++ "To Prove: " ++ show tl)
      (r, (ng',h',_)) <-
         (t |-- tl) prove_nhlwt
            (ng, addProof h tl WFPending, ".  " ++ indent)
      putStrLn (indent ++ show r ++ ": " ++ show tl)
      return (r, (ng' + 1, addProof h' tl r, indent))
```

4.12.4 Prover selector

prove t options def tl h uses the prover engine selected by the e option in options, or the default indicated by def if the e option is not present, to prove tl using t. h is a history of prior results. Updated histories and the proof result as a string are returned.

```
prove :: Theory -> Options -> String -> Tagged Literal
         -> Hist -> WFHist -> IO (Hist, WFHist, String)
prove t options def tl h wh = case lookupBST "e" options of
      Nothing -> prove t (updateBST (\x _ -> x) "e"
                      (ParamValue def) options) def tl h wh
      Just (ParamValue cs) -> case cs of
         "-"     -> use_prove_ tl
         "n"     -> use_prove_n tl
         "t"     -> use_prove_t tl
         "nt"    -> use_prove_nt tl
         "nh"    -> use_prove_nh tl
         "nht"   -> use_prove_nht tl
         "nhl"   -> use_prove_nhl tl
         "nhlt"  -> use_prove_nhlt tl
         "nhlw"  -> use_prove_nhlw tl
         "nhlwt" -> use_prove_nhlwt tl
         _       -> do
            putStrLn $ "Error: No such prover as \""
                       ++ cs ++ "\""
            return (h, wh, "")
   where
   use_prove_ tl = do
      time0 <- getCPUTime
      let Just (result,_) = prove_ t tl ()
      putStrLn $ show result ++ "."
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh, show result)
   use_prove_n tl = do
      time0 <- getCPUTime
      let Just (result,ng) = prove_n t tl 0
      putStrLn $ show result ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh, show result)
   use_prove_t tl = do
      time0 <- getCPUTime
      (result,_) <- prove_t t tl ""
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh, show result)
   use_prove_nt tl = do
      time0 <- getCPUTime
      (result,(ng,_)) <- prove_nt t tl (0, "")
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh, show result)
   use_prove_nh tl = do
      time0 <- getCPUTime
      let Just (result,(ng,h')) = prove_nh t tl (0,h)
      putStrLn $ show result ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h', wh, show result)
   use_prove_nht tl = do
      time0 <- getCPUTime
      (result,(ng,h',_)) <- prove_nht t tl (0,h,"")
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h', wh, show result)
   use_prove_nhl tl = do
      time0 <- getCPUTime
      let Just (result,(ng,h')) = prove_nhl t tl (0,h)
      putStrLn $ show result ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h', wh, show result)
   use_prove_nhlt tl = do
      time0 <- getCPUTime
      (result,(ng,h',_)) <- prove_nhlt t tl (0,h,"")
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h', wh, show result)
   use_prove_nhlw tl = do
      time0 <- getCPUTime
      let Just (result,(ng,wh')) = prove_nhlw t tl (0,wh)
          result' = case result of
             WFBottom -> WFNo
             _        -> result
      putStrLn $ show result' ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh', show result)
   use_prove_nhlwt tl = do
      time0 <- getCPUTime
      (result,(ng,wh',_)) <- prove_nhlwt t tl (0,wh,"")
      let result' = case result of
             WFBottom -> WFNo
             _        -> result
      putStrLn $ show result' ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, wh', show result)
```

4.13 Run-files

The DRunFile module defines a data type for representing a generator of test cases with combinations of facts.

```
module DRunFile(
      RInput, RIgnore, ROutput, RunFile, Run, runFileP,
      generateRuns
   ) where

import ABR.Parser; import ABR.List

import Literal; import DefeasibleLexer
import DInference
```

4.13.1 Data type definitions

An RInput is one set of mutually exclusive literals.

```
type RInput = [Literal]
```

An RIgnore is one set of inconsistent literals.

```
type RIgnore = [Literal]
```

An ROutput is one tagged literal.

```
type ROutput = Tagged Literal
```

An RStatement represents one statement from a run-file.

```
data RStatement =   RInput [Literal]
                  | RIgnore [Literal]
                  | ROutput (Tagged Literal)
```

A Runfile represents the whole runfile.

```
type RunFile = ([RInput], [RIgnore], [ROutput])
```

A Run represents one set of generated facts.

```
type Run = [Literal]
```

4.13.2 Parser

The syntax for a runfile is given in section 3.6, and is implemented as follows:

```
runFileP :: Parser RunFile
runFileP = total $ many (
      (     rInputP  @> RInput
        <|> rIgnoreP @> RIgnore
        <|> rOutputP @> ROutput
      ) <* literalP "symbol" "."
   ) @> (foldr (\s (ins,igs,os) -> case s of
           RInput qs  -> (qs : ins, igs,      os     )
           RIgnore qs -> (ins,      qs : igs, os     )
           ROutput tl -> (ins,      igs,      tl : os)
        ) ([],[],[]))

rInputP :: Parser RInput
rInputP =
      literalP "name1" "input"
    *> (     literalP "symbol" "{"
          *> pLiteralP
         <*> many (
                literalP "symbol" ","
             *> pLiteralP
            )
         <*  literalP "symbol" "}"
       ) @> cons

rIgnoreP :: Parser RIgnore
rIgnoreP =
      literalP "name1" "ignore"
    *> (     literalP "symbol" "{"
          *> pLiteralP
         <*> many (
                literalP "symbol" ","
             *> pLiteralP
            )
         <*  literalP "symbol" "}"
       ) @> cons

rOutputP :: Parser ROutput
rOutputP =
      literalP "name1" "output"
    *> (     literalP "symbol" "{"
          *> taggedLiteralP
         <*  literalP "symbol" "}"
       )
```

4.13.3 Generating runs

generateRuns *run-file* returns the list of generated set of facts.

```
generateRuns :: RunFile -> [Run]
generateRuns (ins,igs,_) =
   filter (\qs -> and [or [q' `notElem` qs
                          | q' <- ig] | ig <- igs]) $
   map concat $
   cartProd $
   map (\qs -> case qs of
      [q] -> [[q], [neg q]]
      _   -> (map (\(b,e,a) -> reverse (map neg b) ++ [e] ++
                 map neg a) . fragments) qs) ins
```

4.14 DProver

See the user's guide (section 3.6) for a description of this module.

```
module Main (main) where

import System; import CPUTime; import Char
import List; import IO

import ABR.Parser; import ABR.Args; import ABR.List
import ABR.Text.String; import ABR.Control.Check
import ABR.Data.BSTree; import ABR.Parser.Checks

import DefeasibleLexer; import Literal; import DTheory
import History; import DInference; import DProve
import DRunFile

main :: IO ()
main = do
   args <- getArgs
   run $ unwords args

run :: String -> IO ()
run args = do
   let (options,others) =
          findOpts [ParamS "e", FlagS "t", FlagS "tp",
             FlagS "td", ParamS "r"] (words args)
   case others of
      []   -> getPath options
      p:[] -> openTheory options p Nothing
      p:f  -> openTheory options p (Just (unwords f))

getPath :: Options -> IO ()
getPath options = do
   putStr "Theory file name (or \"q\" to quit): "
   hFlush stdout
   path <- getLine
   let path' = trim path
   case path' of
      []  -> getPath options
      "q" -> quit
      _:_ -> openTheory options path' Nothing

openTheory :: Options -> FilePath -> Maybe String -> IO ()
openTheory options path mtl = do
   source <- catch (readFile path) (\e -> return "\0")
   case source of
      "\0" -> do
         putStrLn $ "Error: File " ++ path ++ " is \
            \empty or could not be read."
         getPath options
      _ -> case (checkParse lexerL (total theoryP)
            &? cyclesCheck  &? groundCheck) source of
         CheckFail msg -> do
            putStrLn msg
            case mtl of
               Nothing -> getPath options
               _       -> quit
         CheckPass t   -> do
            case (lookupBST "t" options,
                  lookupBST "tp" options,
                  lookupBST "td" options,
                  lookupBST "r" options) of
               (Just FlagMinus,_,_,_) ->
                  putStr $ show t
               (_,Just FlagMinus,_,_) ->
                  putStr $ show $ PrologTheory t
               (_,_,Just FlagMinus,_) ->
                  putStr $ show $ DeloresTheory t
               (_,_,_,Just (ParamValue rFile)) ->
                  doRunFile t options rFile
               _ -> case mtl of
                  Nothing    ->
                     interactive t options
                  Just l -> do
                     proveOne t options l emptyHistory
                        emptyHistory
                     return ()

interactive :: Theory -> Options -> IO ()
interactive t options = do
      putStrLn "Type \"?\" for help."
      proofLoop options emptyHistory emptyHistory
```

```
   where
   proofLoop :: Options -> Hist -> WFHist -> IO ()
   proofLoop options h wh = do
      putStr "|- "
      hFlush stdout
      input <- getLine
      let input' = words input
      case input' of
         [] ->
            proofLoop options h wh
         "?" : _ -> do
            showHelp
            proofLoop options h wh
         "q" : _ ->
            quit
         "t" : _ -> do
            putStrLn $ show t
            proofLoop options h wh
         "tp": _ -> do
            putStrLn $ show $ PrologTheory t
            proofLoop options h wh
         "td" : _ -> do
            putStrLn $ show $ DeloresTheory t
            proofLoop options h wh
         "f" : _ -> do
            putStrLn "Those who forget history \
                     \are destined to repeat it."
            proofLoop options emptyHistory emptyHistory
         "e" : css ->
            let cs = unwords css
            in if cs `elem` ["-","n","nt","nh","nht",
                  "nhl","nhlt","nhlw","nhlwt"] then
                  proofLoop (updateBST (\x _ -> x)
                  "e" (ParamValue cs) options) h wh
               else if cs == "" then do
                  putStr "Current prover: "
                  case lookupBST "e" options of
                     Nothing ->
                        putStrLn $ "nhlt"
                     Just (ParamValue p) ->
                        putStrLn p
                  proofLoop options h wh
               else do
                  putStrLn $ "Error: No such prover: "
                           ++ cs
                  proofLoop options h wh
         "r" : rFile : _ -> do
            doRunFile t options rFile
            proofLoop options h wh
         "l" : [] ->
            getPath options
         "l" : p: [] ->
            openTheory options p Nothing
         _ -> do
            (h',wh') <- proveOne t options input h wh
            proofLoop options h' wh'

showHelp :: IO ()
showHelp = putStrLn
   "To prove things: type a tagged literal.\n\
   \Other commands:\n\
   \   ?          = this message\n\
   \   q          = quit\n\
   \   t          = print theory\n\
   \   tp         = print theory in d-Prolog syntax\n\
   \   td         = print theory in Delores syntax\n\
   \   f          = forget history\n\
   \   e          = show current prover engine\n\
   \   e prover   = select prover engine from {-, n, nh, \
                  \ nhl, nhlw, t, nt, nht, nhlt, nhlwt}\
   \   r run-file = run the tests in run-file\n\
   \   l [path]   = read a new theory file\
                  \ [named path].\n"

proveOne :: Theory -> Options -> String  -> Hist -> WFHist
   -> IO (Hist,WFHist)
proveOne t options input h wh
   = case (checkParse lexerL (total taggedLiteralP)
         &? checkNoVars) input of
      CheckFail msg -> do
         putStrLn msg
         return (h,wh)
      CheckPass tl -> do
         (h',wh',_) <- prove t options "nhlt" tl h wh
         return (h', wh')

quit :: IO ()
quit = putStrLn "Goodbye."

doRunFile :: Theory -> Options -> FilePath -> IO ()
doRunFile t@(Theory fs rs ps) options rFile = do
   source <- catch (readFile rFile) (\e -> return "\0")
   case source of
      "\0" -> putStrLn $ "Can't read file: " ++ rFile
      _    -> case checkParse lexerL runFileP source of
         CheckFail msg     -> do
            putStrLn msg
         CheckPass runFile@(_,_,tls) -> do
            let runs :: [Run]
                runs = generateRuns runFile
                run1 :: Run -> Tagged Literal -> IO String
                run1 fs' tl = do
                   (_,_,r) <- prove (Theory (fs' ++ fs)
                      rs ps) options "nhlt" tl
                      emptyHistory emptyHistory
                   case r of
                      "Proved"     -> return "P"
                      "Not proved" -> return "N"
                      "Loops"      -> return "L"
                runRun :: Run -> IO [String]
                runRun run = mapM (run1 run) tls
            rss <- mapM runRun runs
            putStrLn "\nSummary table:"
            let table = if null runs
                   then [["No runs."]]
                   else (map (const "") (head runs) ++
                      map show tls) :
                      zipWith (\run rs ->
                         map show run ++ rs) runs rss
                widths = map ((+2) . maximum) $
                   (map . map) length $
                   transpose table
                spaceOut :: [Int] -> [String] -> String
                spaceOut ws css = concat $
                   zipWith (\w cs -> rJustify w cs) ws css
                table' = map (spaceOut widths) table
            putStr $ unlines $ table'
```

4.15 Optimized Theories

Module `ODTheory` defines a data type for storage of Defeasible logic theories that facilitates faster proofs.

```
{-# LANGUAGE MultiParamTypeClasses,
             TypeSynonymInstances #-}

module ODTheory(
      ORule, OPriorities, OTheory, makeOTheory,
      makeOTL, unmakeOTL, showOTL, OHist, oprove,
      FHist, initPmSyLitHist
   ) where

import Control.Monad.ST; import CPUTime
import Data.Array.ST; import Data.Array

import ABR.Data.BSTree; import ABR.SparseSet; import ABR.Args
import ABR.Graph

import Literal; import DRule; import Label
import Priority; import ThreadedTest
import DInference; import DTheory
import ProofResult; import History
```

4.15.1 Data types

All the facts should be stored in an array that maps each literal to `True` (it's a fact) or `False` (it's not).

```
type OFacts = Array OLiteral Bool
```

All the rules will be stored in parallel arrays of the antecedents and consequents. An `ORule` is the index type for these arrays.

```
data ORuleIndex lit = OR Int
                      deriving (Eq, Ord, Show)
```

```
type ORule = ORuleIndex OLiteral
```

```
type OAnts = Array ORule [OLiteral]
```

```
type OCons = Array ORule OLiteral
```

We can presort the rules by consequent.

```
type ORuleTable = Array OLiteral [ORule]
```

The priorities are a graph.

```
type OPriorities = SGraph ORule
```

A complete theory ready to use:

```
data ODTheory rul = OTheory {
      num2name   :: LitArray,
      name2num   :: LitTree,
      facts      :: OFacts,
      cons       :: OCons,
      antes      :: OAnts,
      plausStart :: ORule,
      defStart   :: ORule,
      priorities :: OPriorities,
      prsq       :: ORuleTable,
      prsdq      :: ORuleTable,
      prq        :: ORuleTable
   }
```

```
type OTheory = ODTheory ORule
```

`plausStart` is the index in the rules arrays where the rules turn from strict to plausible, and `defStart` is the index at which rules start being defeaters.

4.15.2 Building an optimized theory

`makeOTheory s t` builds an optimized theory using the set of literal names `s` and theory `t`.

```
makeOTheory :: SparseSet Literal -> Theory -> OTheory
makeOTheory s t@(Theory fs rs ps) = let
      (num2nam,nam2num) = makeLitTables s
      (_,nLit) = bounds num2nam
      srs = filter isStrict rs
      prs = filter isPlausible rs
      drs = filter isDefeater rs
      rs' = srs ++ prs ++ drs
      n_srs = length srs
      n_prs = length prs
      n_drs = length drs
      n_rs = n_srs + n_prs + n_drs
      cons' =
         listArray (OR 0, OR (n_rs - 1))
         (map (toOLiteral nam2num . consequent) rs')
      labelTable =
         pairs2BST $ filter (not . null . fst)
         $ zip (map (\(Rule (Label l) _) -> l) rs') [0..]
      toRuleIndex :: Label -> ORule
      toRuleIndex (Label l) =
         case lookupBST l labelTable of
            Just i  -> i
            Nothing -> error "toRuleIndex: Label not found"
      crs = map (\(x,y) -> (y,x)) $ assocs cons'
   in OTheory {
      num2name = num2nam,
      name2num = nam2num,
      facts =
         accumArray (\ _ _ -> True) False (-nLit, nLit)
         $ map (\l -> (toOLiteral nam2num l, True)) fs,
      cons = cons',
      antes =
         listArray (OR 0, OR n_rs - 1) (map ((map
         (toOLiteral nam2num)) . antecedent) rs'),
      plausStart = OR n_srs,
      defStart = OR (n_srs + n_prs),
      priorities =
         mkGraph (OR 0) (OR n_rs - 1)
         (map (\(l1 :> l2) ->
            (toRuleIndex l1, toRuleIndex l2))
          ps),
      prsq =
         accumArray (flip (:)) [] (- nLit, nLit)
         $ take n_srs crs,
      prsdq =
         accumArray (flip (:)) [] (- nLit, nLit)
         $ take (n_srs + n_prs) crs,
      prq = accumArray (flip (:)) [] (- nLit, nLit) crs
   }
```

4.15.3 Instance declarations

```
instance Num (ORuleIndex lit) where
   OR a + OR b = OR (a + b)
   OR a - OR b = OR (a - b)
   OR a * OR b = OR (a * b)
   abs (OR a) = OR (abs a)
   signum (OR a) = OR (signum a)
   fromInteger = OR . fromInteger
instance Enum (ORuleIndex lit) where
   toEnum = OR
   fromEnum (OR i) = i
   enumFrom (OR i) = map OR [i..]
instance Ix (ORuleIndex lit) where
   range (a,b) = [a..b]
   index (OR i,_) (OR j) = j - i
   inRange (OR i,OR j) (OR k) = i <= k && k <= j
instance Show OTheory where
   showsPrec p t =
      showString "num2name: "       . shows (num2name t)
      . showString "\nname2num: "   . shows (name2num t)
      . showString "\nfacts: "      . shows (facts t)
      . showString "\ncons: "       . shows (cons t)
      . showString "\nants: "       . shows (antes t)
      . showString "\nplausStart: " . shows (plausStart t)
      . showString "\ndefStart: "   . shows (defStart t)
      . showString "\npriorities: " . shows (priorities t)
      . showString "\nprsq: "       . shows (prsq t)
      . showString "\nprsdq: "      . shows (prsdq t)
      . showString "\nprq: "        . shows (prq t)
instance DefeasibleLogic ODTheory ORuleIndex OLiteral where
   isFactIn q t = mkTest (facts t ! q)
   notFactIn q t = mkTest (not (facts t ! q))
   rq t q = prq t ! q
   rsq t q = prsq t ! q
   rsdq t q = prsdq t ! q
   ants t r = antes t ! r
   beats t r1 r2 = mkTest $ isAdjacent (priorities t) r1 r2
   notBeats t r1 r2
      = mkTest $ not $ isAdjacent (priorities t) r1 r2
```

4.15.4 Optimized tagged literals

`makeOTL ot tl` converts tagged literal `tl` to an optimized tagged literal using the mapping to optimized literals in optimized theory `ot`. `unmakeOTL` performs the reverse operation. `showOTL` uses `unmakeOTL` before showing an optimized literal so that the true name is shown, rather than the number.

```
makeOTL :: OTheory -> Tagged Literal -> Tagged OLiteral
makeOTL ot tl = case tl of
   Plus  ps l -> Plus  ps (toOLiteral (name2num ot) l)
   Minus ps l -> Minus ps (toOLiteral (name2num ot) l)
```

```
unmakeOTL :: OTheory -> Tagged OLiteral -> Tagged Literal
unmakeOTL ot otl = case otl of
   Plus  ps ol -> Plus  ps (fromOLiteral (num2name ot) ol)
   Minus ps ol -> Minus ps (fromOLiteral (num2name ot) ol)
```

```
showOTL :: OTheory -> Tagged OLiteral -> String
showOTL ot = show . unmakeOTL ot
```

4.15.5 Provers without histories

`oprove_ t tl ()` returns `(r,())`, where `r` is the result of trying to `oprove` tagged literal `tl` with theory `t`. This is the simplest prover, with no trace, no history and therefore no loop checking, and not well founded.

```
oprove_ :: OTheory -> Tagged OLiteral
        -> ThreadedTest Maybe ProofResult ()
oprove_ t tl () = (t |-- tl) oprove_ ()
```

`oprove_n t tl 0` returns `(r,ng)`, where `r` is the result of trying to `oprove` tagged literal `tl` with theory `t` and `ng` is the number of subgoals required to do so.

```
oprove_n :: OTheory -> Tagged OLiteral
         -> ThreadedTest Maybe ProofResult Int
oprove_n t tl ng = do
   (r, ng') <- (t |-- tl) oprove_n ng
   return (r, ng' + 1)
```

`oprove_t t tl ""` returns `(r,"")`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`. A trace is printed.

```
oprove_t :: OTheory -> Tagged OLiteral
         -> ThreadedTest IO ProofResult String
oprove_t t tl indent = do
   putStrLn (indent ++ "To Prove: " ++ showOTL t tl)
   (r, _) <-
      (t |-- tl) oprove_t (".  " ++ indent)
   putStrLn (indent ++ show r ++ ": " ++ showOTL t tl)
   return (r, indent)
```

`oprove_nt t tl (0,"")` returns `(r,(ng,""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t` and `ng` is the number of subgoals required to do so. A trace is printed.

```
oprove_nt :: OTheory -> Tagged OLiteral
          -> ThreadedTest IO ProofResult (Int,String)
oprove_nt t tl (ng,indent) = do
   putStrLn (indent ++ "To Prove: " ++ showOTL t tl)
   (r, (ng',_)) <-
      (t |-- tl) oprove_nt (ng, ".  " ++ indent)
   putStrLn (indent ++ show r ++ ": " ++ showOTL t tl)
   return (r, (ng' + 1, indent))
```

4.15.6 Provers with tree histories

This type is shorthand for the history that maps tagged literals to prior results.

```
type OHist = History (Tagged OLiteral) ProofResult
```

`oprove_nh t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, but does not perform loop checking.

```
oprove_nh :: OTheory -> Tagged OLiteral
          -> ThreadedTest Maybe ProofResult (Int, OHist)
oprove_nh t tl (ng,h) = case getResult h tl of
   Just r ->
      return (r, (ng,h))
   Nothing -> do
      (r, (ng',h')) <- (t |-- tl) oprove_nh (ng,h)
      return (r, (ng' + 1, addProof h' tl r))
```

`oprove_nht t tl (0,h,"")` returns `(r,(ng,h',""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, but does not perform loop checking. A trace is printed.

```
oprove_nht
   :: OTheory -> Tagged OLiteral
      -> ThreadedTest IO ProofResult (Int,OHist,String)
oprove_nht t tl (ng,h,indent) = case getResult h tl of
   Just r -> do
      putStrLn (indent ++ show r ++ " previously: "
                ++ showOTL t tl)
      return (r, (ng,h,indent))
   Nothing -> do
      putStrLn (indent ++ "To Prove: " ++ showOTL t tl)
      (r, (ng',h',_)) <-
         (t |-- tl) oprove_nht (ng, h, ".  " ++ indent)
      putStrLn (indent ++ show r ++ ": " ++ showOTL t tl)
      return (r, (ng' + 1, addProof h' tl r, indent))
```

`oprove_nhl t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, and performs loop checking.

```
oprove_nhl :: OTheory -> Tagged OLiteral
           -> ThreadedTest Maybe ProofResult (Int, OHist)
oprove_nhl t tl (ng,h) = case getResult h tl of
   Just Pending ->
      return (Bottom, (ng, addProof h tl Bottom))
   Just r ->
      return (r, (ng, h))
   Nothing -> do
      (r, (ng',h')) <-
          (t |-- tl) oprove_nhl (ng, addProof h tl Pending)
      return (r, (ng' + 1, addProof h' tl r))
```

`oprove_nhlt t tl (0,h,"")` returns `(r,(ng,h',""))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, and performs loop checking.

```
oprove_nhlt
   :: OTheory -> Tagged OLiteral
      -> ThreadedTest IO ProofResult (Int, OHist, String)
oprove_nhlt t tl (ng,h,indent) = case getResult h tl of
   Just Pending -> do
      putStrLn (indent ++ "Loop detected: "
                ++ showOTL t tl)
      return (Bottom, (ng, addProof h tl Bottom, indent))
   Just r -> do
      putStrLn (indent ++ show r ++ " previously: "
                ++ showOTL t tl)
      return (r, (ng, h, indent))
   Nothing -> do
      putStrLn (indent ++ "To Prove: " ++ showOTL t tl)
      (r, (ng',h',_)) <-
         (t |-- tl) oprove_nhlt
            (ng, addProof h tl Pending, ".  " ++ indent)
      putStrLn (indent ++ show r ++ ": " ++ showOTL t tl)
      return (r, (ng' + 1, addProof h' tl r, indent))
```

4.15.7 Provers with array histories

The tree implementation of histories works well, but adds changes the complexity of a proof with N subgoals from $O(N)$ to $O(N \log N)$. This can be avoided by replacing the tree in the history by an array. Accessing and *updating* the array must however be performed in constant time or there will be no speedup. This requires mutable arrays, and therefore the ST monad.

We must record the results for each possible tagged literal. This is essentially a three dimensional structure, $\{+.-\} \times \{\Delta, \partial, \ldots\} \times$ literals.

These declarations define a collection of parallel mutable arrays that hold all possible proof results.

```
type LitHist s = STArray s OLiteral ProofResult

type SyLitHist s = Array ProofSymbol (LitHist s)

type PmSyLitHist s = (SyLitHist s, SyLitHist s) -- (+,-)
```

Between proofs we need frozen (immutable) versions.

```
type FLitHist = Array OLiteral ProofResult

type FSyLitHist = Array ProofSymbol FLitHist

type FPmSyLitHist = (FSyLitHist, FSyLitHist) -- (+,-)

type FHist = FPmSyLitHist -- F = flat and frozen
```

An initial history takes some building.

```
initLitHist :: OTheory -> FLitHist
initLitHist t =
   listArray (bounds $ facts t) (repeat NotAttempted)

initSyLitHist :: FLitHist -> FSyLitHist
initSyLitHist flh =
   listArray (PS_D, PS_dt) (repeat flh)
```

```
initPmSyLitHist :: OTheory -> FPmSyLitHist
initPmSyLitHist t =
   let flh  = initLitHist t
       fslh = initSyLitHist flh
   in (fslh, fslh)
```

`extendPmSyLitHist ot fh` rebuilds the history `fh` as new literals are introduced by a new optimized theory `ot`. For the moment, we'll just reset it.

```
extendPmSyLitHist ::
   OTheory -> FPmSyLitHist -> FPmSyLitHist
extendPmSyLitHist t (p,m) =
   initPmSyLitHist t
```

A the start of a proof, we must thaw the history.

```
thawLitHist :: FLitHist -> ST s (LitHist s)
thawLitHist = thaw

thawSyLitHist :: FSyLitHist -> ST s (SyLitHist s)
thawSyLitHist a = do
   let as = elems a
   as' <- mapM thawLitHist as
   return $ listArray (bounds a) as'

thawPmSyLitHist :: FPmSyLitHist -> ST s (PmSyLitHist s)
thawPmSyLitHist (p,m) = do
   p' <- thawSyLitHist p
   m' <- thawSyLitHist m
   return (p', m')
```

At the end of a proof, we must freeze the history.

```
freezeLitHist :: LitHist s -> ST s FLitHist
freezeLitHist = freeze

freezeSyLitHist :: SyLitHist s -> ST s FSyLitHist
freezeSyLitHist a = do
   let as = elems a
   as' <- mapM freezeLitHist as
   return $ listArray (bounds a) as'

freezePmSyLitHist :: PmSyLitHist s -> ST s FPmSyLitHist
freezePmSyLitHist (p,m) = do
   p' <- freezeSyLitHist p
   m' <- freezeSyLitHist m
   return (p', m')
```

`oprove_nH t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, but does not perform loop checking.

```
oprove_nH :: OTheory -> Tagged OLiteral
   -> ThreadedTest (ST s) ProofResult (Int, PmSyLitHist s)
oprove_nH t tl (ng,(p,m)) = do
   r <- case tl of
      Plus  ps q -> readArray (p ! ps) q
      Minus ps q -> readArray (m ! ps) q
   case r of
      NotAttempted -> do
         (r', (ng',(p',m')))
            <- (t |-- tl) oprove_nH (ng,(p,m))
         case tl of
            Plus  ps q -> writeArray (p' ! ps) q r'
            Minus ps q -> writeArray (m' ! ps) q r'
         return (r', (ng' + 1, (p',m')))
      _ ->
         return (r, (ng, (p, m)))
```

`oprove_nHl t tl (0,h)` returns `(r,(ng,h'))`, where `r` is the result of trying to prove tagged literal `tl` with theory `t`, `ng` is the number of subgoals required to do so, `h` is a history of prior results and `h'` is the final history. This prover avoids redoing prior proofs, and performs loop checking.

```
oprove_nHl :: OTheory -> Tagged OLiteral
   -> ThreadedTest (ST s) ProofResult (Int, PmSyLitHist s)
oprove_nHl t tl (ng,(p,m)) = do
   r <- case tl of
      Plus  ps q -> readArray (p ! ps) q
      Minus ps q -> readArray (m ! ps) q
   case r of
      Pending -> do
         case tl of
            Plus  ps q -> writeArray (p ! ps) q Bottom
            Minus ps q -> writeArray (m ! ps) q Bottom
         return (Bottom, (ng + 1, (p,m)))
      NotAttempted -> do
         (r', (ng',(p',m')))
            <- (t |-- tl) oprove_nHl (ng,(p,m))
         case tl of
            Plus  ps q -> writeArray (p' ! ps) q r'
            Minus ps q -> writeArray (m' ! ps) q r'
         return (r', (ng' + 1, (p',m')))
      _ ->
         return (r, (ng, (p, m)))
```

4.15.8 Prover selector

`oprove ls t ot options def tl h fh` uses the prover selected by the `e` option in `options`, or the default indicated by `def` if the `e` option is not present, to prove `tl` using `ot`. `h` is a tree history of prior results. `fh` is a flat (array) history of prior results. If the literal in `tl` is not defined in the present optimized theory, i.e. not in `ls`, a new one is built to accommodate it. An updated history, literal name set, optimized theory and the proof result as a string are returned.

```
oprove :: SparseSet Literal -> Theory -> OTheory
   -> Options -> String -> Tagged Literal
   -> OHist -> FHist
   -> IO (SparseSet Literal, OTheory, OHist, FHist, String)
oprove ls t ot options def tl h fh = do
      let tls = getLits tl emptySS
          (ls', ot', fh')
             = if tls `isSubSet` ls then
                  (ls, ot, fh)
               else
                  let ns  = tls `unionSS` ls
                      ot' = makeOTheory ns t
                      fh'' = extendPmSyLitHist ot' fh
                  in (ns, ot', fh'')
          otl = makeOTL ot' tl
      case lookupBST "e" options of
         Nothing -> oprove ls' t ot' (updateBST (\x _ -> x)
                       "e" (ParamValue def) options)
                       def tl h fh'
         Just (ParamValue cs) -> do
            (h',fh'',r) <- case cs of
               "-"    -> use_prove_     ot' otl fh'
               "n"    -> use_prove_n    ot' otl fh'
               "t"    -> use_prove_t    ot' otl fh'
               "nt"   -> use_prove_nt   ot' otl fh'
               "nh"   -> use_prove_nh   ot' otl fh'
               "nht"  -> use_prove_nht  ot' otl fh'
               "nhl"  -> use_prove_nhl  ot' otl fh'
               "nhlt" -> use_prove_nhlt ot' otl fh'
               "nH"   -> use_prove_nH   ot' otl fh'
               "nHl"  -> use_prove_nHl  ot' otl fh'
               _      -> do
                  putStrLn $ "Error: No such prover as \""
                             ++ cs ++ "\""
                  return (h, fh', "")
            return (ls', ot', h', fh'', r)
   where
   use_prove_ ot otl fh' = do
      time0 <- getCPUTime
      let Just (result,_) = oprove_ ot otl ()
      putStrLn $ show result ++ "."
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
      return (h, fh', show result)
   use_prove_n ot otl fh' = do
      time0 <- getCPUTime
      let Just (result,ng) = oprove_n ot otl 0
      putStrLn $ show result ++ "."
      putStrLn $ "Number of goals: " ++ show ng
      time1 <- getCPUTime
      putStrLn $ "CPU time for proof (s): "
         ++ show (fromIntegral(time1 - time0) / 1.0e12)
```

```
   return (h, fh', show result)
use_prove_t ot otl fh' = do
   time0 <- getCPUTime
   (result,_) <- oprove_t ot otl ""
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h, fh', show result)
use_prove_nt ot otl fh' = do
   time0 <- getCPUTime
   (result,(ng,_)) <- oprove_nt ot otl (0, "")
   putStrLn $ "Number of goals: " ++ show ng
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h, fh', show result)
use_prove_nh ot otl fh' = do
   time0 <- getCPUTime
   let Just (result,(ng,h')) = oprove_nh ot otl (0,h)
   putStrLn $ show result ++ "."
   putStrLn $ "Number of goals: " ++ show ng
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h', fh', show result)
use_prove_nht ot otl fh' = do
   time0 <- getCPUTime
   (result,(ng,h',_)) <- oprove_nht ot otl (0,h,"")
   putStrLn $ "Number of goals: " ++ show ng
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h', fh', show result)
use_prove_nhl ot otl fh' = do
   time0 <- getCPUTime
   let Just (result,(ng,h')) = oprove_nhl ot otl (0,h)
   putStrLn $ show result ++ "."
   putStrLn $ "Number of goals: " ++ show ng
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h', fh', show result)
use_prove_nhlt ot otl fh' = do
   time0 <- getCPUTime
   (result,(ng,h',_)) <- oprove_nhlt ot otl (0,h,"")
   putStrLn $ "Number of goals: " ++ show ng
   time1 <- getCPUTime
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h', fh', show result)
use_prove_nH ot otl fh' = do
   time0 <- getCPUTime
   let (result,(ng,fh'')) = runST (do
            h <- thawPmSyLitHist fh'
            (result,(ng,h')) <- oprove_nH ot otl (0,h)
            fh''' <- freezePmSyLitHist h'
            return (result,(ng,fh'''))
         )
   putStrLn $ show result ++ "."
   time1 <- getCPUTime
   putStrLn $ "Number of goals: " ++ show ng
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h, fh'', show result)
use_prove_nHl ot otl fh' = do
   time0 <- getCPUTime
   let (result,(ng,fh'')) = runST (do
            h <- thawPmSyLitHist fh'
            (result,(ng,h')) <- oprove_nHl ot otl (0,h)
            fh''' <- freezePmSyLitHist h'
            return (result,(ng,fh'''))
         )
   putStrLn $ show result ++ "."
   time1 <- getCPUTime
   putStrLn $ "Number of goals: " ++ show ng
   putStrLn $ "CPU time for proof (s): "
      ++ show (fromIntegral(time1 - time0) / 1.0e12)
   return (h, fh'', show result)
```

4.16 ODProver

See the user's guide (section 3.7) for a description of this module.

```
module Main (main) where

import System; import CPUTime; import Char

import ABR.Args; import ABR.SparseSet
import ABR.Text.String; import ABR.Parser
import ABR.Control.Check; import ABR.Data.BSTree
import ABR.Parser.Checks

import Literal; import DTheory; import Priority
import DefeasibleLexer; import ODTheory; import History
import DInference

main :: IO ()
main = do
   args <- getArgs
   run $ unwords args

run :: String -> IO ()
run args = do
   let (options,others) =
          findOpts [ParamS "e", FlagS "t", FlagS "td",
             FlagS "tp"] (words args)
   case others of
       []   -> getPath options
       p:[] -> openTheory options p Nothing
       p:l  -> openTheory options p (Just (unwords l))

getPath :: Options -> IO ()
getPath options = do
   putStr "Theory file name (or \"q\" to quit): "
   path <- getLine
   let path' = trim path
   case path' of
      []  -> getPath options
      "q" -> quit
      _:_ -> openTheory options path' Nothing

openTheory :: Options -> FilePath -> Maybe String -> IO ()
openTheory options path mtl = do
   source <- catch (readFile path) (\e -> return "\0")
   case source of
      "\0" -> do
         putStrLn $ "Error: File " ++ path ++ " is \
            \empty or could not be read."
         getPath options
      _ -> case (checkParse lexerL (total theoryP)
                 &? cyclesCheck &? groundCheck) source of
         CheckFail msg -> do
            putStrLn msg
            case mtl of
               Nothing -> getPath options
               _       -> quit
         CheckPass t   -> do
            case (lookupBST "tp" options,
                  lookupBST "td" options) of
               (Just FlagMinus,_) ->
                  putStr $ show $ PrologTheory t
               (_,Just FlagMinus) ->
                  putStr $ show $ DeloresTheory t
               _ -> do
                  let ls = getLits t emptySS
                      ot = makeOTheory ls t
                  case lookupBST "t" options of
                     Just FlagMinus ->
                        putStr $ show ot
                     _ -> case mtl of
                        Nothing    ->
                           interactive ls t ot options
                        Just l -> do
                           proveOne ls t ot options l
                              emptyHistory
                              (initPmSyLitHist ot)
                           return ()

interactive :: SparseSet Literal -> Theory -> OTheory ->
   Options -> IO ()
interactive ls t ot options = do
      putStrLn "Type \"?\" for help."
```

```
      proofLoop ls ot options emptyHistory
         (initPmSyLitHist ot)
   where
   proofLoop :: SparseSet Literal -> OTheory
                -> Options -> OHist -> FHist -> IO ()
   proofLoop ls ot options h fh = do
      putStr "|- "
      input <- getLine
      let input' = words input
      case input' of
         [] ->
            proofLoop ls ot options h fh
         "?" : _ -> do
            showHelp
            proofLoop ls ot options h fh
         "q" : _ ->
            quit
         "t" : _ -> do
            putStrLn $ show ot
            proofLoop ls ot options h fh
         "td" : _ -> do
            putStrLn $ show $ DeloresTheory t
            proofLoop ls ot options h fh
         "tp" : _ -> do
            putStrLn $ show $ PrologTheory t
            proofLoop ls ot options h fh
         "f" : _ -> do
            putStrLn "Those who forget history \
                     \are destined to repeat it."
            proofLoop ls ot options emptyHistory
               (initPmSyLitHist ot)
         "e" : css ->
            let cs = unwords css
            in if cs `elem` ["-","n","nh","nhl","nt",
                  "nht","nhlt", "nH", "nHl"] then
                  proofLoop ls ot (updateBST (\x _ -> x)
                  "e" (ParamValue cs) options) h fh
               else if cs == "" then do
                  putStr "Current prover: "
                  case lookupBST "e" options of
                     Nothing ->
                        putStrLn $ "nhlt"
                     Just (ParamValue p) ->
                        putStrLn p
                  proofLoop ls ot options h fh
               else do
                  putStrLn $ "Error: No such prover: "
                        ++ cs
                  proofLoop ls ot options h fh
         "l" : [] ->
            getPath options
         "l" : p: [] ->
            openTheory options p Nothing
         _ -> do
            (ls', ot', h', fh') <-
               proveOne ls t ot options input h fh
            proofLoop ls' ot' options h' fh'

proveOne :: SparseSet Literal -> Theory -> OTheory
   -> Options -> String  -> OHist -> FHist
   -> IO (SparseSet Literal, OTheory, OHist, FHist)
proveOne ls t ot options input h fh =
   case (checkParse lexerL (total taggedLiteralP)
         &? checkNoVars) input of
      CheckFail msg -> do
         putStrLn msg
         return (ls, ot, h, fh)
      CheckPass tl -> do
         (ls', ot', h', fh', _) <-
            oprove ls t ot options "nhlt" tl h fh
         return (ls', ot', h', fh')

showHelp :: IO ()
showHelp = putStrLn
   "To prove things: type a tagged literal.\n\
   \Other commands:\n\
   \   ?        = this message\n\
   \   q        = quit\n\
   \   t        = print theory\n\
   \   tp       = print theory in d-Prolog syntax\n\
   \   td       = print theory in delores syntax\n\
   \   f        = forget history\n\
   \   e        = show current prover engine\n\
   \   e prover = select prover engine from {-, n, nh,\
                \ nhl, t, nt, nht, nhlt, nH, nHl}\n\
   \   l [path] = read a new theory file\
                \ [named path]."

quit :: IO ()
quit = putStrLn "Goodbye."
```

4.17 Scalable Test Theories

This module defines functions that generate scalable test Defeasible theories and queries to exercise them.

```
{-# LANGUAGE TypeSynonymInstances #-}

module DTestTheories(
      generateTheory, generateTL, generateMetrics
   ) where

import Literal; import DRule; import Label
import Priority; import DTheory; import DInference

infix 7 >>>
```

4.17.1 Shorthand

Scalable theories are usually built with literals of the form a_i. `a i` returns such a literal. `na i` returns the corresponding negative literal $\neg a_i$.

```
a, na :: Int -> Literal
a  i = PosLit ('a' : show i)
na i = NegLit ('a' : show i)
```

Theories are built from (usually) labeled rules. `r i rule` adds a label to `rule`. The label is a capital `R` followed by `i`.

```
r :: Int -> Rule -> LRule
r i = Rule (Label ('R' : show i))
```

Priorities indicate one rule beats another. `r1 >>> r2` returns a priority $r_1 > r_2$. This operator is overloaded. Priorities can be made from label numbers, labels or labeled rules.

```
class MakesPriority a where

   (>>>) :: a -> a -> Priority

instance MakesPriority Int where

   i >>> j
      = (Label ('R' : show i)) :> (Label ('R' : show j))

instance MakesPriority Label where

   (>>>) = (:>)

instance MakesPriority LRule where

   (Rule l1 _) >>> (Rule l2 _) = (l1 :> l2)
```

4.17.2 Chain theories

See section B.1 for a description of chain theories.

`chainTheory n` returns theory **chain**(n). `chainTL n` returns the default tagged literal $+\partial a_n$ the proof of which exercises all of theory **chain**(n).

```
chainTheory :: Int -> Theory
chainTheory n
   = Theory
        [a 0] [r i ([a (i-1)] :=> a i) | i <- [1..n]] []

chainTL :: Int -> Tagged Literal
chainTL n = Plus PS_d (a n)
```

`chainSTheory n` returns theory **chain**$^s(n)$ which is a strict variant of **chain**(n). `chainSTL n` returns the default tagged literal $+\Delta a_n$.

```
chainSTheory :: Int -> Theory
chainSTheory n
  = Theory
      [a 0] [r i ([a (i-1)] :-> a i) | i <- [1..n]] []

chainSTL :: Int -> Tagged Literal
chainSTL n = Plus PS_D (a n)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules in Mac Hugs with default heap and stack.

4.17.3 Circle theories

See section B.2 for a description of circle theories.

`circleTheory n` returns theory **circle**(n). `circleTL n` returns the default tagged literal $+\partial a_0$ the proof of which exercises all of theory **circle**(n).

```
circleTheory :: Int -> Theory
circleTheory n
  = Theory
      [] [r i ([a i] :=> a ((i+1) `mod` n))
         | i <- [0..n-1]] []

circleTL :: Tagged Literal
circleTL = Plus PS_d (a 0)
```

`circleSTheory n` returns theory **circle**$^{\mathrm{s}}(n)$ which is a strict variant of **circle**(n). `circleSTL n` returns the default tagged literal $+\Delta a_0$.

```
circleSTheory :: Int -> Theory
circleSTheory n
  = Theory
      [] [r i ([a i] :-> a ((i+1) `mod` n))
         | i <- [0..n-1]] []

circleSTL :: Tagged Literal
circleSTL = Plus PS_D (a 0)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage).

4.17.4 Levels theories

See section B.3 for a description of levels theories.

`levelsTheory n` returns theory **levels**(n). `levelsTL n` returns the default tagged literal $+\partial a_0$ the proof of which exercises all of theory **levels**(n).

```
levelsTheory :: Int -> Theory
levelsTheory n
  = Theory [] (rules (-1)) (priorities 0)
    where
    rules i
      | i < 0
        =   (r 0 ([] :=> a 0)) : rules (i+1)
      | i <= n
        =   (r (4*i+1) ([a (2*i+1)] :=> na (2*i)))
          : (r (4*i+2) ([]          :=> a (2*i+1)))
          : (r (4*i+3) ([a (2*i+2)] :=> na (2*i+1)))
          : (r (4*i+4) ([]          :=> a (2*i+2)))
          : rules (i + 1)
      | otherwise
        = []
    priorities i
      | i < 0     = priorities (i+1)
      | i <= n    =   (4*i+3) >>> (4*i+2)
                    : priorities (i+1)
      | otherwise = []

levelsTL :: Tagged Literal
levelsTL = Plus PS_d (a 0)
```

`levels_Theory n` returns theory **levels**$^{-}(n)$ which is a variant of **levels**(n) that omits the priorities. `levels_TL n` returns the default tagged literal $+\partial a_0$.

```
levels_Theory :: Int -> Theory
levels_Theory n
  = let Theory fs rs _ = levelsTheory n
    in Theory fs rs []

levels_TL :: Tagged Literal
levels_TL = Plus PS_d (a 0)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules in Mac Hugs with default heap and stack.

4.17.5 Teams theories

See section B.4 for a description of teams theories, **teams**(n).

```
teamsTheory :: Int -> Theory
teamsTheory n = Theory [] (rules 0 0) (priorities 0 0)
  where
  rules :: Int -> Int -> [LRule]
  rules i t  -- i = level, t = # rules in prior levels
     | i < n     = tRules 0
     | otherwise = bRules 0
     where
     k = 4 ^ (i + 1) -- # rules at level i
     bRules j        -- bottom level rules
        | j < k     =   r (t + j) ([] :=> c j)
                      : bRules (j + 1)
        | otherwise = []
     tRules j        -- top and middle level rules
        | j < k     =   r (t + j) ([a j] :=> c j)
                      : tRules (j + 1)
        | otherwise = rules (i + 1) (t + k)
     c j = (if j `mod` 4 < 2 then PosLit else NegLit)
           ('a' : show ((t + j) `div` 4))
     a j = PosLit ('a' : show (t + 1 + j))
  priorities :: Int -> Int -> [Priority]
  priorities i t
     | i < n     = tPriors 0
     | otherwise = bPriors 0
     where
     k = 4 ^ (i + 1)
     bPriors j
        | j < k     =   (t + j)     >>> (t + j + 2)
                      : (t + j + 1) >>> (t + j + 3)
                      : bPriors (j + 4)
        | otherwise = []
     tPriors j
        | j < k     =   (t + j)     >>> (t + j + 2)
                      : (t + j + 1) >>> (t + j + 3)
                      : tPriors (j + 4)
        | otherwise = priorities (i + 1) (t + k)

teamsTL :: Tagged Literal
teamsTL  = Plus PS_d (a 0)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules ($n = 9$) in Mac Hugs.

4.17.6 Tree theories

See section B.5 for a description of tree theories, **tree**(n, k).

```
treeTheory :: Int -> Int -> Theory
treeTheory n k = Theory facts rules []
  where
  facts = [a i | let above = sum [k^j | j <- [0..n-1]],
                 i <- [above .. above + k * k^(n-1) - 1]]
  rules = [r i (as :=> a i) | d <- [0..n-1],
                w <- [0..k^d-1],
                let above = sum [k^j | j <- [0..d-1]]
                    i = above + w
                    below = above + k^d + w * k
                    as = [a j | j <- [below .. below + k - 1]]]

treeTL  :: Tagged Literal
treeTL = Plus PS_d (a 0)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules ($n = 12, k = 3$) in Mac Hugs.

4.17.7 Directed acyclic graph theories

See section B.6 for a description of directed acyclic graph theories, **dag**(n, k).

```
dagTheory :: Int -> Int -> Theory
dagTheory n k = Theory facts rules []
   where
   facts = [a i | i <- [k*n+1..k*n+k]]
   rules = [r i (as :=> a i) | i <- [0..k*n],
            let as = [a (i+j) | j <- [1..k]]]

dagTL :: Tagged Literal
dagTL= Plus PS_d (a 0)
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules in Mac Hugs.

4.17.8 Mix theories

See section B.7 for a description of directed mix theories, $\textbf{mix}(m, n, k)$.

```
mixTheory :: Int -> Int -> Int -> Theory
mixTheory m n k
   = Theory facts rules []
     where
     p = PosLit "p"
     np = NegLit "p"
     a i j = PosLit $ "a" ++ show i ++ "_" ++ show j
     b i j k = PosLit $ "b" ++ show i ++ "_" ++ show j
               ++ "_" ++ show k
     facts
        | k == 0
          = [a i j | i <- [1..2*m], j <- [1..n]]
        | otherwise
          = [b i j 1 | i <- [1..2*m], j <- [1..n]]
     rules
        = rules' 1 1 1 0
     rules' i j k' l
        | i > 2 * m
          = []
        | j > n
          = (if i <= m
                then (r l ([a i j | j <- [1..n]] :=> p))
                else (r l ([a i j | j <- [1..n]] :~> np))
            )
            : rules' (i+1) 1 1 (l+1)
        | k' > k
          = rules' i (j+1) 1 l
        | k' == k
          = (r l ([b i j k] :-> a i j))
            : rules' i j (k'+1) (l+1)
        | otherwise
          = (r l ([b i j k'] :-> b i j (k'+1)))
            : rules' i j (k'+1) (l+1)

mixTL :: Tagged Literal
mixTL = Plus PS_d (PosLit "p")
```

Testing note: Space friendly, $O(1)$ stack, $O(1)$ heap (over theory storage). Can generate 10^6 rules in Mac Hugs.

4.17.9 Selectors

`generateTheory name sizes` returns the named theory. `sizes` is a list of size parameters to select the size of the theory.

```
generateTheory :: String -> [Int] -> Maybe Theory
generateTheory name sizes
   | name == "chain"
      = if head sizes >= 0
           then Just $ chainTheory $ head sizes
           else Nothing
   | name == "chains"
      = if head sizes >= 0
           then Just $ chainSTheory $ head sizes
           else Nothing
   | name == "circle"
      = if head sizes >= 0
           then Just $ circleTheory $ head sizes
           else Nothing
   | name == "circles"
      = if head sizes >= 0
           then Just $ circleSTheory $ head sizes
           else Nothing
   | name == "levels"
      = if head sizes >= 0
           then Just $ levelsTheory $ head sizes
           else Nothing
   | name == "levels-"
      = if head sizes >= 0
           then Just $ levels_Theory $ head sizes
           else Nothing
   | name == "teams"
      = if head sizes >= 0
           then Just $ teamsTheory $ head sizes
           else Nothing
   | name == "tree"
      = if length sizes == 2 && and (map (> 0) sizes)
           then let [n,k] = sizes
                in Just $ treeTheory n k
           else Nothing
   | name == "dag"
      = if length sizes == 2 && and (map (> 0) sizes)
           then let [n,p] = sizes
                in Just $ dagTheory n p
           else Nothing
   | name == "mix"
      = if length sizes == 3 && and (map (>= 0) sizes)
           then let [m,n,k] = sizes
                in Just $ mixTheory m n k
           else Nothing
   | otherwise
      = Nothing
```

`generateTL name sizes` returns the suggested tagged literal to prove for the named theory. `sizes` is a list of size parameters to select the size of the theory.

```
generateTL :: String -> [Int] -> Maybe (Tagged Literal)
generateTL name sizes
   | name == "chain"   = Just $ chainTL $ head sizes
   | name == "chains"  = Just $ chainSTL $ head sizes
   | name == "circle"  = Just $ circleTL
   | name == "circles" = Just $ circleSTL
   | name == "levels"  = Just levelsTL
   | name == "levels-" = Just levels_TL
   | name == "teams"   = Just teamsTL
   | name == "tree"    = Just treeTL
   | name == "dag"     = Just dagTL
   | name == "mix"     = Just mixTL
   | otherwise         = Nothing
```

`generateMetrics name sizes` computes the tuple $(facts, rules, priorities, size)$ which contains the metrics computed for the `name`d theory with the given `sizes`.

```
generateMetrics :: String -> [Int]
   -> Maybe (Int, Int, Int, Int)
generateMetrics name sizes
   | name == "chain" =
      let n = head sizes
      in Just (
         1,
         n,
         0,
         2 * n + 1
      )
   | name == "chains" =
      let n = head sizes
      in Just (
         1,
         n,
         0,
         2 * n + 1
      )
   | name == "circle" =
      let n = head sizes
      in Just (
         0,
         n,
         0,
         2 * n
      )
   | name == "circles" =
```

```
      let n = head sizes
      in Just (
         0,
         n,
         0,
         2 * n
      )
   | name == "levels" =
      let n = head sizes
      in Just (
         0,
         4 * n + 5,
         n + 1,
         7 * n + 8
      )
   | name == "levels-" =
      let n = head sizes
      in Just (
         0,
         4 * n + 5,
         0,
         6 * n + 7
      )
   | name == "teams" =
      let n = head sizes
      in Just (
         0,
         4 * sum [4 ^ i | i <- [0..n]],
         2 * sum [4 ^ i | i <- [0..n]],
         10 * sum [4 ^ i | i <- [0..n-1]] + 6 * (4^n)
      )
   | name == "tree" =
      let n = head sizes
          k = sizes !! 1
      in Just (
         k^n,
         sum [k ^ i | i <- [0..n-1]],
         0,
         (k+1) * sum [k ^ i | i <- [0..n-1]] + k^n
      )
   | name == "dag" =
      let n = head sizes
          k = sizes !! 1
      in Just (
         k,
         n * k + 1,
         0,
         n * (k^2) + (n+2) * k + 1
      )
   | name == "mix" =
      let m = head sizes
          n = sizes !! 1
          k = sizes !! 2
      in Just (
         2 * m * n,
         2 * m + 2 * m * n * k,
         0,
         2 * m + 4 * m * n + 4 * m * n * k
      )
   | otherwise = Nothing
```

4.18 DTScale

See the user's guide (section 3.8) for a description of this module.

```
module Main (main) where

import System; import CPUTime

import ABR.Args; import ABR.SparseSet
import ABR.Data.BSTree

import Literal; import DRule; import DTheory
import History; import DTestTheories; import DInference
import DProve; import ODTheory

main :: IO ()
main = do
   args <- getArgs
   run' args

run :: String -> IO () -- Hugs entry point
run = run' . words

run' :: [String] -> IO ()
run' args =
   let (options,thName:sizes) = findOpts [ParamS "e",
          FlagS "t", FlagS "tp", FlagS "td", FlagS "o",
          FlagS "m"]
          args
       sizes' = map read sizes
       th = generateTheory thName sizes'
       tl = generateTL thName sizes'
   in case (th, tl) of
      (Nothing, _) ->
         putStrLn ("ERROR: no such theory: " ++
            thName ++ " " ++ unwords sizes)
      (_, Nothing) ->
         putStrLn ("ERROR: no such tagged literal: " ++
            thName ++ " " ++ unwords sizes)
      (Just th, Just tl) -> case (lookupBST "t" options,
         lookupBST "tp" options, lookupBST "td" options) of
         (Just FlagMinus,_,_) ->
            putStr $ show th
         (_,Just FlagMinus,_) ->
            putStr $ show $ PrologTheory th
         (_,_,Just FlagMinus) ->
            putStr $ show $ DeloresTheory th
         _ -> do
            case lookupBST "m" options of
               Just FlagMinus -> do
                  let Just (f,r,p,s) =
                         generateMetrics thName sizes'
                  putStrLn $
                     "Computed metrics:"
                     ++ "\n   # facts =      " ++ show f
                     ++ "\n   # rules =      " ++ show r
                     ++ "\n   # priorities = " ++ show p
                     ++ "\n   size =         " ++ show s
                     ++ "\n"
               _ -> return ()
            let Theory fs rs ps = th
                nfs = length fs
                nrs = length rs
                nps = length ps
                nls = sum $ map (length . antecedent) rs
            putStrLn $ "\n\n# facts:      " ++ show nfs
            putStrLn $ "# rules:      " ++ show nrs
            putStrLn $ "# priorities: " ++ show nps
            putStrLn $ "# literals in all bodies: "
                       ++ show nls
            putStrLn $ "### total size = "
                       ++ show (nfs + nrs + nps + nls)
                       ++ "\n"
            case lookupBST "o" options of
               Just FlagMinus -> do
                  _ <- prove th options "nhl" tl
                          emptyHistory emptyHistory
                  return ()
               _ -> do
                  let ls = getLits th emptySS
                      ot = makeOTheory ls th
                      fh = initPmSyLitHist ot
                      dummy = Plus PS_D (PosLit "bogus")
                  putStrLn $ show $ length $ show ot
                  putStrLn "Dummy Run"
                  (ls', ot', h', fh', _) <-
                     oprove ls th ot options "nHl"
                        dummy emptyHistory fh
                  putStrLn "Real Run"
                  _ <- oprove ls th ot options "nHl"
                          tl h' fh'
                  return ()
```

4.19 CGI Tool

This module implements the CGI tool that provides a web interface for the *Deimos* system. Section 3.9 describes its use.

```
module Main (main) where
```

```
import Directory; import List; import Char

import ABR.CGI; import ABR.Control.Check
import ABR.Data.BSTree; import ABR.SparseSet
import ABR.Parser hiding (cons); import ABR.Parser.Checks
import ABR.Text.Markup

import Literal; import DTheory; import DefeasibleLexer
import History; import DInference; import ODTheory
```

4.19.1 Paths

These constants will require modification to set *Deimos* up on new web servers. Use new values of `installWhere` to select the right values. `infoDir` is a file path to a directory containing some texts to be included in the output. `theoryDir` is the path to the sample theories. `infoURL` is the URL that gets to same directory pointed to by `infoDir`. `theoryURL` is the URL that gets to same directory pointed to by `theoryDir`.

```
installWhere :: String
installWhere = "kurango"

infoDir, theoryDir ::  FilePath
infoDir = if installWhere == "hunchentoot"
   then
      "/Program Files/Apache Group/Apache/htdocs/def-info/"
   else if installWhere == "kurango" then
      "doc/"
   else
      "doc/"
theoryDir = if installWhere == "hunchentoot"
   then
      "/Program Files/Apache Group/Apache/htdocs/\
      \def-theories/"
   else if installWhere == "kurango" then
      "theories/"
   else
      "theories/"

infoURL, theoryURL :: String
infoURL = if installWhere == "hunchentoot"
   then
      "http://localhost/def-info/"
   else if installWhere == "kurango" then
      "doc/"
   else
      "doc/"
theoryURL = if installWhere == "hunchentoot"
   then
      "http://localhost/def-theories/"
   else if installWhere == "kurango" then
      "theories/"
   else
      "theories/"
```

`subs text` prints `text` replacing all occurrences of `###I###` with the value of `infoURL`, `###T###` by `theoryURL`, and `###C###` by the CGI tool URL. This permits included HTML documents to refer back to the tool and information directories.

```
subs :: String -> IO ()
subs css =
   if css == "" then
     return ()
   else if take 7 css == "###I###" then do
      putStr infoURL
      subs (drop 7 css)
   else if take 7 css == "###T###" then do
      putStr theoryURL
      subs (drop 7 css)
   else if take 7 css == "###C###" then do
      script <- getSCRIPT_NAME
      putStr script
      subs (drop 7 css)
   else do
      putChar $ head css
      subs (tail css)
```

4.19.2 Main entry point

```
main :: IO ()
main = do
   printMimeHeader
   queryString <- getQUERY_STRING
   case queryString of
      ""              -> doWelcome
      "new-theory"    -> doNewTheory
      "theory"        -> doTheory
      "proof"         -> doProof
      "syntax"        -> doSyntax
      "proof-help"    -> doProofHelp
      _               -> doBadQuery
```

4.19.3 Common cosmetic bits

`wrap title rows` prints the HTML code common to every page. The content of each page must be a sequence of table rows. Each page has a title.

```
wrap :: String -> [IO ()] -> IO ()
wrap title rows = htmlN (do
      headN $ titleN $ put $ "Deimos: " ++ title
      bodyE [("background",
              infoURL ++ "background.jpg")] (do
            centerN $ h1N $ fontE [("color", "FFFFFF")] (do
                  iN $ put "Deimos"
                  put ": "
                  put $ title
               )
            tableE [("cellpadding","10"),
                    ("cellspacing","10"),
                    ("width","100%")] $ sequence_ rows
            imgE_ [("src", infoURL ++ "logo.jpg")]
         )
   )
```

`row color item` prints item in a table data element in a row with the given background colour. `norm item` displays an item in a row with the normal background colour. `high` displays the item in a highlight background color. `oops item` displays the item in a row with an error-indicating background colour. `oops'` displays a plain text message in a PRE element. `whoops` does all that and puts it in a complete document with a title.

```
row :: String -> IO () -> IO ()
row colour item = trE [("bgcolor",colour)] $ tdN item

norm, high, oops :: IO () -> IO ()
norm = row "FFFFFF"
high = row "FFFF99"
oops = row "FF9999"

oops' :: String -> IO ()
oops' = oops . preN . put

whoops :: String -> String -> IO ()
whoops title = wrap title . (: []) . oops'
```

`form query items` produces a form with `query` as the URL query string and containing the form elements in `items`

```
form :: String -> IO () -> IO ()
form query items = do
   script <- getSCRIPT_NAME
   formE [("method","post"),
          ("action",script ++ "?" ++ query)] items
```

`link query text` produces a hyperlink back to this CGI tool with `query` as the URL query string and containing the `text`.

```
link :: String -> IO () -> IO ()
link query text = do
   script <- getSCRIPT_NAME
   aE [("href",script ++ "?" ++ query)] text
```

This item is displayed when the query string in the URL is not understood.

```
doBadQuery :: IO ()
doBadQuery = wrap "Unknown Query String" [
      oops $ pN $ put "The query string in the URL is \
                      \unknown."
   ]
```

4.19.4 Welcome

doWelcome shows the entry page for the system, which includes context information, a selection of example theories, and a link to a page where new theories may be entered.

```
doWelcome :: IO ()
doWelcome = wrap
   "Query Answering Defeasible Logic System" [
      high introMsg,
      high pickATheory,
      high newTheory
   ]

introMsg :: IO ()
introMsg = do
   text <- readFile $ infoDir ++ "intro.html"
   subs text

pickATheory :: IO ()
pickATheory = form "theory" (do
      h2N $ put "Select an Example Defeasible Theory"
      fileNames <- getDirectoryContents theoryDir
      let fileNames' = sort $ filter ((== 't') . head
                        . reverse) fileNames
      pN $ put "Click on an example:"
      pN $ selectE [("name","theory"), ("size","20")]
         $ mapM_ theoryOption fileNames'
      pN $ inputE_ [("name","origin"), ("type","hidden"),
                   ("value","file")]
      pN $ inputE_ [("name","submit"), ("type","submit"),
                   ("value","Open Theory")]
   )

theoryOption :: FilePath -> IO ()
theoryOption file = do
   contents <- readFile $ theoryDir ++ "/" ++ file
   optionE [("value",file)] $ put $ trim $ drop 1
      $ trim $ head $ lines $ contents

trim :: String -> String
trim = dropWhile isSpace . reverse . dropWhile isSpace
       . reverse

newTheory :: IO ()
newTheory = do
   h2N $ put "Create a New Defeasible Theory"
   pN (do
      put "Click "
      link "new-theory" $ put "here"
      put " to create a new defeasible theory."
     )
```

4.19.5 New theory

doNewTheory displays the page with the form where new theories may be typed in.

```
doNewTheory :: IO ()
doNewTheory = wrap "New Theory" [high $ form "theory" (do
      pN (do
            put "Type in your new theory. (The syntax \
                \for theories is defined "
            link "syntax" $ put "here"
            put ".)"
            inputE_ [("name","origin"),
                     ("type","hidden"),("value", "form")]
         )
      pN $ textareaE [("name","theory"), ("cols","80"),
        ("rows","15"), ("wrap","off")] $ return ()
      pN $ inputE_ [("type","submit"), ("name","submit"),
         ("value","Go Prove Things")]
   )]
```

4.19.6 Theory

doTheory displays the page where the theory is displayed and queries are prompted for.

```
doTheory :: IO ()
doTheory = do
   let title = "Defeasible Theory"
   formData <- getFormData
   lookupGuard  formData ["origin","theory"]
      (\ cs -> whoops title $ "Missing " ++ cs ++ ".")
      (\ [origin,theory] -> do
         source <- if origin == "file" then
                       readFile $ theoryDir ++ theory
                    else
                       return theory
         wrap title [
               high $ showTheory source,
               case (emptyCheck "theory"
                  &? checkParse lexerL (total theoryP)
                  &? cyclesCheck) source  of
                  CheckFail msg -> oops' msg
                  CheckPass th  -> high $ queryForm source
            ]
      )

emptyCheck :: String -> Check String String String
emptyCheck item content =
   if and $ map isSpace content then
      CheckFail $ "The " ++ item ++ " is empty."
   else
      CheckPass content

showTheory :: String -> IO ()
showTheory t = do
   h2N $ put "Defeasible Theory"
   tableE [("cellpadding","10"), ("cellspacing","10"),
           ("width","100%")] $ norm $ preN $ put t

queryForm :: String -> IO ()
queryForm th = form "proof" (do
      h2N $ put "Do a Proof"
      inputE_ [("name","theory"), ("type","hidden"),
         ("value", makeHTMLSafe th)]
      pN (do
            put "What do you want to prove? "
            inputE_ [("name","taggedliteral"),
                     ("type","text"), ("size","15")]
            put " ("
            link "proof-help" $ put "What do I type here?"
            put ")"
         )
      pN (do
            put "Select a prover with: "
            selectE [("name","prover")] (do
                  opt "-"    "no extras"
                  opt "n"    g
                  opt "nh"   (g +++ h)
                  opt "nhl"  (g +++ h +++ l)
                  opt "t"    t
                  opt "nt"   (g +++ t)
                  opt "nht"  (g +++ h +++ t)
                  optionE [("value","nhlt"),
                     ("selected","")] $ put
                     (g +++ h +++ l +++ t)
                  opt "nH"   (g +++ h')
                  opt "nHl"  (g +++ h' +++ l)
               )
         )
      pN $ inputE_ [("type","submit"), ("name","submit"),
         ("value","Prove it")]
   )
   where
   opt name name'
      = optionE [("value",name)] $ put name'
   g = "goal counting"
   h = "history keeping"
   h' = "faster history keeping"
   l = "loop detection"
   t = "tracing"
   (+++) a b = a ++ ", " ++ b
```

4.19.7 Proof

doProof displays the page containing the results of a query.

```
doProof :: IO ()
doProof = do
   let title = "Proof"
```

```
   formData <- getFormData
   lookupGuard formData ["theory", "taggedliteral",
      "prover"]
      (\ cs -> whoops title $ "Missing " ++ cs ++ ".")
      (\ [t,tl, p] -> wrap title
                    $ proveIt (noSemicolons t) tl p)

noSemicolons :: String -> String
noSemicolons cs = case cs of
   []                     -> []
   (';':';':';':';':';':cs) -> '\n' : noSemicolons cs
   (c:cs)                 -> c : noSemicolons cs

proveIt :: String -> String -> String -> [IO ()]
proveIt source q prover =
   high (showTheory source) :
   case (emptyCheck "theory"
      &? checkParse lexerL (nofail theoryP)
      &? cyclesCheck &? groundCheck) source  of
      CheckFail msg -> [oops' msg]
      CheckPass t ->
         showQuery q :
         case (emptyCheck "query" &? checkParse lexerL
            (total taggedLiteralP) &? checkNoVars) q of
            CheckFail msg -> [oops' msg]
            CheckPass tl  -> [high (do
                  h2N $ put "Proof"
                  proveIt' t tl prover
               )]

showQuery :: String -> IO ()
showQuery q = high (do
      h2N $ put "Query (Tagged Literal)"
      tableE [("cellpadding","10"), ("cellspacing","10"),
              ("width","100%")] $ norm $ preN $ put q
   )

proveIt' :: Theory -> Tagged Literal -> String -> IO ()
proveIt' t tl prover = do
   let s = getLits tl (getLits t emptySS)
       ot = makeOTheory s t
   tableE [("cellpadding","10"), ("cellspacing","10"),
          ("width","100%")] $ norm (do
         putStr "<pre>"
         (_,_,_,_,r) <- oprove s t ot emptyBST prover tl
                emptyHistory (initPmSyLitHist ot)
         putStr "</pre>"
         h3N $ put r
      )
```

4.19.8 Help pages

doSyntax and doProofHelp display the help pages for this CGI tool.

```
doSyntax :: IO ()
doSyntax = wrap "Syntax" [norm (do
      text <- readFile $ infoDir ++ "syntax.html"
      subs text
   )]

doProofHelp :: IO ()
doProofHelp = wrap "Proof Help" [norm (do
      text <- readFile $ infoDir ++ "proof-help.html"
      subs text
   )]
```

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 $\mathbf{chain}(n)$ 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.

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

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

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

The implementation of functions that generate chain theories is given in section 4.17.2.

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 $\mathbf{circle}^{\mathbf{s}}(n)$ uses only strict rules.

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

The implementation of functions that generate circle theories is given in section 4.17.3.

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}.

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

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

The implementation of functions that generate levels theories is given in section 4.17.4.

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:

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

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

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

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

The implementation of functions that generate teams theories is given in section 4.17.5.

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:

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

and:

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

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

The implementation of functions that generate tree theories is given in section 4.17.6.

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.

The implementation of functions that generate directed acyclic graph theories is given in section 4.17.7.

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.

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

where:

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

or, if $k > 0$:

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

The implementation of functions that generate mix theories is given in section 4.17.8.

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 8 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

[6] Andrew Rock. ABR Haskell libraries. Technical report, (continually) in preparation. 4, 4.1

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 8: Sizes of scalable test theories