Phobos (Version 2): A Query Answering Plausible Logic System

Andrew Rock
School of Computing and Information Technology
Griffith University
Nathan, Queensland, 4111, Australia
arock@cit.gu.edu.au

Abstract
This document is a description and complete listing of Phobos, a query answering plausible logic system. Phobos is a complete implementation of propositional plausible logic. 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 2
2 Installation 2
  2.1 Downloading 2
  2.2 Unpacking and compiling ABRHLibs 2
  2.3 Unpacking and compiling Deimos 2
  2.4 Unpacking and compiling Phobos 2
  2.5 Compiling without make 2
3 User’s Guide 2
  3.1 Descriptions 2
  3.1.1 Whitespace and comments 3
  3.1.2 Proposition symbols 3
  3.1.3 Literals 3
  3.1.4 Formulas 3
  3.1.5 Rules 3
  3.1.6 Descriptions 3
  3.2 Theories 3
  3.2.1 Labelled rules 3
  3.2.2 Priority assertions 3
  3.2.3 Theories 3
  3.3 Tagged Formulas 4
  3.4 Inference Rules 4
  3.4.1 Conjunction 4
  3.4.2 Disjunction 4
  3.4.3 Define 4
  3.4.4 Likely 4
  3.4.5 Plausible 4
  3.4.6 Support 4
  3.5 Just enough Hugs 5
  3.6 Running compiled tools 5
  3.7 DescriptionParser 5
  3.7.1 Usage (GHC) 5
  3.7.2 Usage (Hugs) 5
  3.8 Description2Theory 5
  3.8.1 Usage (GHC) 5
  3.8.2 Usage (Hugs) 5
  3.9 TheoryParser 5
  3.9.1 Usage (GHC) 6
  3.9.2 Usage (Hugs) 6
  3.10 Prover 6
  3.10.1 Usage (GHC) 6
  3.10.2 Usage (Hugs) 6
  3.11 OProver 6
  3.11.1 Usage (GHC) 6
  3.11.2 Usage (Hugs) 6
  3.12 TScale 6
  3.12.1 Usage (GHC) 6
  3.12.2 Usage (Hugs) 7
4 Implementation 7
  4.1 Lexical Issues 7
  4.1.1 Comments 7
  4.1.2 Names 7
  4.1.3 Symbols and everything else 7
  4.2 Formulas 8
  4.2.1 Data type definitions 8
  4.2.2 Parcers 8
  4.2.3 Operations on formulas 8
  4.2.4 Instance declarations 9
  4.3 Rules 10
  4.3.1 Data type definitions 10
  4.3.2 Parcers 10
  4.3.3 Properties of rules 11
  4.3.4 Instance declarations 11
  4.4 Descriptions 11
  4.4.1 Data type definitions 11
  4.4.2 Parser 11
  4.4.3 Normalizing descriptions 12
  4.4.4 Instance declarations 12
  4.5 DescriptionParser 12
  4.6 Theories 12
  4.6.1 Data type definitions 12
  4.6.2 Making theories from descriptions 13
  4.6.3 Parser 13
  4.6.4 Labeled rule operations 13
  4.6.5 Consistency checks 13
  4.6.6 Grouping all variables 14
  4.6.7 Theory operations 14
  4.6.8 Instance declarations 14
  4.7 Description2Theory 15
  4.8 TheoryParser 15
  4.9 Inference Conditions 15
  4.9.1 Data type definitions 15
  4.9.2 Parser 16
  4.9.3 Overloaded functions 16
  4.9.4 Inference Conditions 16
  4.9.5 Instance declarations 16
  4.10 Prove 16
  4.10.1 Histories 17
  4.10.2 Inconsistent Literals 17
  4.10.3 Precomputed stuff 19
  4.10.4 Plausible logic instance 19
  4.10.5 Provers 19
  4.10.6 Prover selector 20
  4.11 Prover 21
  4.12 Optimized Theories 22
  4.12.1 Data types 22
  4.12.2 Building an optimized theory 22
  4.12.3 Instance declarations 23
  4.13 OProve 23
  4.13.1 Optimized tagged formulas 23
  4.13.2 Histories 24
  4.13.3 Inconsistent Literals 24
  4.13.4 Precomputed stuff 24
  4.13.5 Plausible logic instance 24
  4.13.6 Provers 24
  4.13.7 Prover selector 25
  4.14 OProve 26
  4.15 Scalable Test Theories 27
1 Introduction

Phobos is a system that implements plausible logic [1, 2]. The procedures for installation of the Phobos 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 reuses some modules developed for the Deimos system. They are not reprinted here. The Deimos system [3] implements Defeasible logic [4, 5], and is described in a separate document [6].

This is the second major version of Phobos. Changes:

- This version implements the revised formal description of Plausible logic.
- The command line tools have improved usability.
- The installation process is better defined.
- Numerous defects have been rectified.

2 Installation

2.1 Downloading

The Phobos system and this documentation can be downloaded from:

http://www.cit.gu.edu.au/~arock/Phobos2/Plausible.cgi


2.2 Unpacking and compiling ABRHLibs

The documentation bundled with the libraries describes the procedure for unpacking and compiling them. The result will be a directory called ABRHLibs containing the object (.o) and interface (.hi) files required by Deimos. Deimos and Phobos require only those modules compatible with Windows and not those requiring UNIX or MySQL.

2.3 Unpacking and compiling Deimos

Refer to the documentation bundled with Deimos.

2.4 Unpacking and compiling Phobos

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 Phobos2.tar.gz
$ tar -xf Phobos2.tar.gz

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

Change directory to Phobos2/src.

$ cd Phobos2/src

To compile all of the Deimos tools, type:

$ make bin

Only the CGI tool (section 3.13) 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 4.17.1. Most users will not want to install the CGI tool.

2.5 Compiling without make

If you are wishing to compile the Phobos 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.

$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/DescriptionParser.lhs -o ../bin/DescriptionParser
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/Description2Theory.lhs -o ../bin/Description2Theory
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/TheoryParser.lhs -o ../bin/TheoryParser
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/Prover.lhs -o ../bin/Prover
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/OProver.lhs -o ../bin/OProver
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/TScale.lhs -o ../bin/TScale
$ gcc --make --fglasgow-exts -o -package lang -i./ABRHLibs -i./Deimos/src/Plausible.cgi.lhs -o ../bin/Plausible.cgi

3 User’s Guide

This user’s guide begins with section 3.1 which details the syntax that Phobos will recognize for plausible descriptions. Section 3.2 follows this with a description of the syntax for plausible theories. Section 3.3 describes the syntax of the queries the system will respond to. Sections 3.5 and 3.6 describe how to use the two most popular Haskell runtime systems to execute the tools that make up Phobos. The remaining subsections of section 3 give usage instructions for each of those tools.

3.1 Descriptions

Plausible theories are not directly entered by the user. The user creates plausible descriptions which are automatically transformed into plausible theories.

Plausible descriptions are entered into components of Phobos in textual form. The syntax for descriptions 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 of comments, as per Prolog:

- 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 Proposition symbols

Proposition symbols may consist of letters of either case, digits and underscores (_), but must start with a lower case letter. Phobos extends plausible descriptions by permitting arguments to proposition symbols. 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. Examples:

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

3.1.3 Literals

A literal is a proposition p or its negation ¬p. Phobos uses `~` for ¬. Examples:

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

3.1.4 Formulas

A formula is formed from literals and the connectives (in descending order of precedence): ¬ (not), ∧ (and), ∨ (or), → (strict implies,⇒) and a consequent. The normal order of precedence may be overridden with parentheses. Formula sets are enclosed in braces and contain zero or more comma separated formulas. Formulas may be formed by preceding a formula set by \(\land\) (and, ∨) or \(\lor\) (or, ∨).

Example formulas:

```
p
p & q
p | (~q & r)
/p, q, r/
```

3.1.5 Rules

There are two types of rules permitted in Phobos descriptions:

- **Plausible rules** consist of an antecedent (a set of formulas), the plausible arrow ⇒ (for ⇒) and a consequent (a formula).
- **Defeater rules** consist of an antecedent, the defeater arrow \(\neg\) (for \(\neg\)) and a consequent.

If the antecedent is a singleton set, the set braces may be omitted. Example rules (legal within descriptions):

```
{} ⇒ p
{a, b, c} \neg \neg d
{a, b | c} ⇒ g & f
p ⇒ q
```

3.1.6 Descriptions

A plausible description is a sequence of formulas, plausible rules and defeater rules. Each formula and rule must be terminated by a period. This first example is purely propositional. Note that the lines containing ⇒ are formulas, not rules.

```
p ⇒ b. % Penguins are birds.
b ⇒ f. % Birds usually fly.
p ⇒ ~f. % Penguins usually do not fly.
gap. % We have a genetically altered penguin.
gap ⇒ p. % A genetically altered penguin is still a % penguin.
gap \neg f. % It is too risky to suppose that a %
```

% genetically altered penguin cannot fly.

This example uses variables:

```
nsd(h). % Hans is a native speaker of
nsd(x) ⇒ bp(x). % Native speakers of Pennsylvania-
bp(x) ⇒ bec(x). % People born in Pennsylvania are
bec(x) ⇒ busa(x). % People born near the USA east coast.
nsd(x) ⇒ nsg(x). % Native speakers of Pennsylvania-
nsd(x) ⇒ ~nsg(x). % Dutch are native speakers of
ngs(x) ⇒ ~busa(x). % Native speakers of German are
```

% Is Hans born in the USA?

In the formal description of plausible logic, the facts are defined as a set of clauses. To enter a clause in a Phobos description use the form \(\lor\) (literal, literal, ...). Also, the rules must only be simple plausible or defeater rules. Phobos will transform input descriptions to meet these requirements. Use the tool Description2Theory to view the result of this transformation.

3.2 Theories

Phobos users do not usually create plausible theories directly, rather they create plausible descriptions which are mechanically transformed into theories, using the Description2Theory tool. Plausible theories consist of labelled rules and priority assertions.

3.2.1 Labelled rules

Labels are names that start with an upper case letter. Rules in plausible theories are usually preceded by a unique label and a colon. In a theory there may be three kinds of rules: plausible (⇒) defeater (\(\neg\)) and strict (⇒).

3.2.2 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.

The priority assertions are added to a mechanically generated theory by the user.

3.2.3 Theories

A theory consists of a sequence of rules and priority assertions. Each rule and priority assertion must be terminated with a period.

The rules in the following examples have been mechanically generated using the Description2Theory tool, but the comments and priorities have been added by hand.

Example 1 (genetically modified penguins):

```
R1: (\{} ⇒ gap. % We have a genetically altered % penguin.
R2: (gap) ⇒ p. % A genetically altered penguin % is still a penguin.
R3: (p) ⇒ b. % Penguins are birds.
R4: (b) ⇒ p. % Things which are not birds are % not penguins.
R5: (~p) ⇒ gap. % Things which are not penguins % are not genetically altered % penguins.
R6: (b) ⇒ f. % Birds usually fly.
R7: (p) ⇒ ~f. % Penguins usually do not fly.
R8: (gap) ⇒ f. % Genetically altered penguins % might fly.
R9: (b | gap) ⇒ f. % Birds or genetically altered % penguins might fly.
```
3.3 Tagged Formulas

The queries that the proper components of Phobos respond to are tagged formulas. The syntax for tagged formulas is:

\[
\text{proof\_symbol ::= "d" | "1" | "p" | "s"}
\]

\[
\text{tagged\_formula ::= ("*p" | "=*") proof\_symbol formula}
\]

At present all literals in a tagged formula must be grounded, that is, contain no variables. Examples:

\[
\text{\# symbol meaning}
\]

\[
\text{d: definite}
\]

\[
\text{1: likely (plausible with ambiguity propagation)}
\]

\[
p: plausible
\]

\[
s: support
\]

Table 1: The proof symbols.

3.4 Inference Rules

The following are the inference rules that are used to prove a given tagged formula. A formal proof or derivation \( P = (P(1), \ldots, P(|P|)) \) is a finite sequence of ordered pairs \((T, \pm\alpha F)\) where \( T = (R, \alpha, f) \) is a plausible theory, \( \alpha \in \{d, 1, p, s\} \), and \( f \) is a cnf-formula. In these rules \( F \) is a finite set of clauses, \( q \) is a literal, \( A(r) \) is the antecedent of rule \( \alpha \), \( P(1) \) is the set of nonempty subsets of \( F \), \( R[q] \) is the set of rules with consequent \( q \), \( R_p[q] \) is the set of plausible rules with consequent \( q \), and \( R_p[q; s] = \{ t \in R_p[q] : t > s \} \) is the set of plausible rules with consequent \( q \) that beat some other rule \( s \).

3.4.1 Conjunction

\[
\bigwedge \text{If } P(i+1) = (T, +\alpha \bigwedge F) \text{ then } \forall f \in F, (T, +\alpha F) \in P[1..i].
\]

\[
\bigwedge \text{If } P(i+1) = (T, -\alpha F) \text{ then } \exists f \in F, (T, -\alpha F) \in P[1..i].
\]

3.4.2 Disjunction

\[
\bigvee \text{If } P(i+1) = (T, +\alpha \bigvee F) \text{ and } F \text{ is a finite set of literals then } \exists F' \in P_{1..i} \forall f \in F', ((R \cup \{(\rightarrow q : q \in (F' - \{f\})) \}}, +\alpha f) \in P[1..i].
\]

\[
\bigvee \text{If } P(i+1) = (T, -\alpha \bigvee F) \text{ and } F \text{ is a finite set of literals then } \forall F' \in P_{1..i} \exists f \in F', ((R \cup \{(\rightarrow q : q \in (F' - \{f\})) \}}, +\alpha f) \in P[1..i].
\]

3.4.3 Definite

\[
\begin{align*}
+\delta & \text{If } P(i+1) = (T, +\delta q) \text{ then } \exists r \in R_p[q], (T, +\delta A(r)) \in P[1..i]. \\
-\delta & \text{If } P(i+1) = (T, -\delta q) \text{ then } \forall r \in R_p[q], (T, -\delta A(r)) \in P[1..i]. 
\end{align*}
\]

3.4.4 Likely

\[
\begin{align*}
+\lambda & \text{If } P(i+1) = (T, +\lambda q) \text{ then either } \\
& .1) \exists r \in R_p[q], (T, +\lambda A(r)) \in P[1..i], \text{ or } \\text{or}\text{ the following two conditions hold.} \\
& .2) \text{the following two conditions hold.} \\
& .1) \exists r \in R_p[q], (T, +\lambda A(r)) \in P[1..i], \text{ and } \\
& .2) \forall C \in Inc(R_p, q) \exists c \in C \forall s \in R[c] \text{ such that } \\
& .1) (T, -\sigma A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, -\sigma A(t)) \in P[1..i]. \\
-\lambda & \text{If } P(i+1) = (T, -\lambda q) \text{ then } \\
& .1) \forall r \in R_p[q], (T, -\lambda A(r)) \in P[1..i], \text{ and } \\
& .2) \text{or either} \\
& .1) \forall r \in R_p[q], (T, -\lambda A(r)) \in P[1..i], \text{ or } \\text{the following two conditions hold.} \\
& .2) \exists C \in Inc(R_p, q) \exists c \in C \exists s \in R[c] \text{ such that } \\
& .1) (T, -\pi A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, -\pi A(t)) \in P[1..i]. \\
\end{align*}
\]

3.4.5 Plausible

\[
\begin{align*}
+\pi & \text{If } P(i+1) = (T, +\pi q) \text{ then either } \\
& .1) \exists r \in R_p[q], (T, +\pi A(r)) \in P[1..i], \text{ or } \\text{or the following two conditions hold.} \\
& .2) \forall C \in Inc(R_p, q) \exists c \in C \forall s \in R[c] \text{ such that } \\
& .1) (T, -\pi A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, -\pi A(t)) \in P[1..i]. \\
-\pi & \text{If } P(i+1) = (T, -\pi q) \text{ then } \\
& .1) \forall r \in R_p[q], (T, -\pi A(r)) \in P[1..i], \text{ and } \\
& .2) \text{or either} \\
& .1) \forall r \in R_p[q], (T, -\pi A(r)) \in P[1..i], \text{ or } \\text{the following two conditions hold.} \\
& .2) \exists C \in Inc(R_p, q) \exists c \in C \exists s \in R[c] \text{ such that } \\
& .1) (T, +\pi A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, +\pi A(t)) \in P[1..i]. \\
\end{align*}
\]

3.4.6 Support

\[
\begin{align*}
+\sigma & \text{If } P(i+1) = (T, +\sigma q) \text{ then either } \\
& .1) \exists r \in R_p[q], (T, +\sigma A(r)) \in P[1..i], \text{ or } \\text{the following two conditions hold.} \\
& .2) \forall C \in Inc(R_p, q) \exists c \in C \exists s \in R[c] \text{ such that } \\
& .1) (T, +\sigma A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, +\sigma A(t)) \in P[1..i]. \\
-\sigma & \text{If } P(i+1) = (T, -\sigma q) \text{ then } \\
& .1) \forall r \in R_p[q], (T, -\sigma A(r)) \in P[1..i], \text{ and } \\
& .2) \forall r \in R_p[q] \text{ either} \\
& .1) (T, -\sigma A(r)) \in P[1..i], \text{ or } \\text{the following two conditions hold.} \\
& .2) \exists C \in Inc(R_p, q) \exists c \in C \exists s \in R[c] \text{ such that } \\
& .1) (T, -\sigma A(s)) \in P[1..i], \text{ and } \\
& .2) \exists t \in R_p[q; s], (T, -\sigma A(t)) \in P[1..i]. \\
\end{align*}
\]

March 22, 2004
3.5 Just enough Hugs

The Haskell programming language has been used to implement Phobos. There are several Haskell implementations. The most widely used are the interpreter, Hugs, and the (glorious) Glasgow Haskell Compiler, GHC. Compiling Phobos with GHC is described in section 2. While compiling with GHC is the only way to install the web-based components of Phobos 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. (I do most development and testing using Hugs on a Macintosh and do most performance testing using GHC on a UNIX box.) Here is just enough information to get and use Hugs to run Phobos.

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

Phobos 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 stubs, library, Desmos and Phobos modules. For example:

hugs -98 -h10000000 
-P"std-lib-stubs:hlibs:Deimos:src:Phobos2:src:"

Defining a shell alias for this complicated command is recommended.

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

:1 <program-name>

where <program-name> is the name of the main module of the Phobos program. The file name extension .lhs may be omitted.

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

main

To kill any Haskell program type a control-C, or command- on a Macintosh.

To quit Hugs, type the command:

:\q

3.6 Running compiled tools

Once compiled with GHC (section 2), the Phobos 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 arguments that can be specified in addition to the program name plus the suffix ".hs". The program will then behave as described for GHC.

3.7 DescriptionParser

The program DescriptionParser is a test program that exercises the lexers and parsers required to parse a plausible description. It can be used as a quick syntax checker for plausible description files. 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 with the command

DescriptionParser path1 path2 ...

where path, path2, ... are the paths to each of the description 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 description in its original and normalized forms. If no paths are supplied on the command line, then standard input will be read and parsed.

3.7.2 Usage (Hugs)

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

run1 "path"

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

run ["path1", "path2", ... ]

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

3.8 Description2Theory

The program Description2Theory is the program that transforms plausible descriptions to plausible theories. This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.8.1 Usage (GHC)

Run the program with the command

Description2Theory path1 path2 ...

where path, path2, ... are the paths to each of the files. The program will display any syntax errors detected in each file, or if it parsed correctly, the generated theory will be written to a new file with the same name as the description source file plus the suffix ".t". If no paths are supplied on the command line, then standard input will be read and parsed and the generated theory will be written to standard output.

3.8.2 Usage (Hugs)

Load the script Description2Theory.lhs into the Hugs interpreter. To transform one description file, type the expression

run1 "path"

where path is the path to the description file. To transform a list of files, type the expression

run ["path1", "path2", ... ]

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

3.9 TheoryParser

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

Run the program with the command

```
TheoryParser path1 path2 ...
```

where `path1`, `path2`, ... are the paths to each of the files. The program will display the name each file and either a syntax error message or, if the file parsed correctly and it passes a sequence of consistency checks, the regenerated theory in its original and grounded forms. The grounded form has all variables replaced by constants. If no paths are supplied on the command line, then standard input will be read and parsed.

3.9.2 Usage (Hugs)

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

```
run "path"
```

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

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

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

3.10 Prover

The program `Prover` is the query answering prover with the simplest (and slowest) implementation. This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.10.1 Usage (GHC)

Run the program by typing a command of the form:

```
Prover options [theory-file-name [tagged-formula]]
```

where the options are:

- `-t` Print the theory and terminate.
- `-e engine` Use the named prover `engine`. See table 2 for the names of the prover engines that are available. The default prover engine is `nht`.

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 formula supplied on the command line, then that proof will be attempted and the program will terminate upon its completion. 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 prompted for.

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

- `?` Print the list of commands.
- `q` Quit the program.
- `t` Print the theory.
- `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]].
- `i` Print the intermediate results used to compute Inc(R,q), tagged-formula Answer tagged-formula by attempting a proof.

Tagged formulas are described in section 3.3. 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, and trace printing. The default prover is `nht`.

<table>
<thead>
<tr>
<th>engine name</th>
<th>counts</th>
<th>keeps history</th>
<th>detects loops</th>
<th>prints trace</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td><code>nh</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>nhl</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>t</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>nt</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>nht</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><code>nhlt</code></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 2: Prover engines.

3.10.2 Usage (Hugs)

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

```
run "options [theory-file-name [tagged-formula]]"
```

The program then behave as described for GHC.

3.11 OProver

The program `OProver` is a query answering prover with a faster implementation. This program can be run using the Hugs interpreter, or compiled with GHC and run directly from the shell.

3.11.1 Usage (GHC)

Run the program by typing a command of the form:

```
OProver options [theory-file-name [tagged-formula]]
```

where the options and commands accepted are the same as for `Prover`, with the exception that the `i` command is not available.

3.11.2 Usage (Hugs)

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

3.12 TScale

The program `TScale` 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.

3.12.1 Usage (GHC)

Run the program by typing a command of the form:

```
TScale options theory-name size...
```

where the options are:

- `-t` Print the theory and terminate without attempting a proof.
- `=-m` Print the computed metrics (defined in section B.4) for the theory before proving it.
- `-e engine` Use the named prover `engine`. See table 2 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:

```
TScale -t levels 100
```

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 3. The scalable test theories are described in detail in appendix B.
### 3.12.2 Usage (Hugs)

Load the script `TScale.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:

```haskell
run "$p nhlt comp 5"
```

### 3.13 CGI Tool

The program `Plausible.cgi` is a Common Gateway Interface program which provides a world wide web interface to Phobos. The program should be accessed with a WWW browser with the URL: http://your.www.site/Plausible.cgi

For our WWW site, this is: http://www.cit.gu.edu.au/~arock/Phobos2/Plausible.cgi

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

The description can then be transformed into a plausible theory, which can be augmented by appropriate priorities.

With a theory suitably augmented, the user can enter queries in the form of tagged formulas. 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 OProver and summarized in table 2.

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 Phobos, presents the modules in a bottom-up sequence. Phobos reuses modules that are part of and documented with Demos [6]. They are not reproduced here. They are modules: Literal, Label, Priority, ProofResult, History, and ThreadedTest. Library modules that are not directly concerned with implementing plausible logic are presented in a separate document [?].

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+TEX source code.

## 4.1 Lexical Issues

Various elements of the Phobos system parse textual representations of literals, formulas, rules, descriptions, priorities, theories and queries. Phobos uses the Parser module [?] to implement functions that perform lexical analysis and parsers. The PlausibleLexer module implements the functions for lexical analysis of plausible sources.

### 4.1.1 Comments

Comments in plausible 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\n" | end-of-file)
- `comment2 ::= "/\" comment2'
- `comment2' ::= "\" | any-character comment2'

These comment forms are recognized by these lexer functions.

```haskell
module PlausibleLexer{lex} where
import Char; import Parser

theory theory name smallest size
levels(n) levels 0
levels("n") levels 0
comp(n) comp 2
null null not applicable

<table>
<thead>
<tr>
<th>theory name</th>
<th>smallest size</th>
</tr>
</thead>
<tbody>
<tr>
<td>levels(n)</td>
<td>levels 0</td>
</tr>
<tr>
<td>levels(&quot;n&quot;)</td>
<td>levels 0</td>
</tr>
<tr>
<td>comp(n)</td>
<td>comp 2</td>
</tr>
<tr>
<td>null</td>
<td>not applicable</td>
</tr>
</tbody>
</table>
```

This opens the starting page for the system, containing pointers to information about plausible logic and Phobos. A form allows the user to select an example plausible description to work with, or to open a page where a new description can be entered. With a theory suitably augmented, the user can enter queries in the form of tagged formulas. 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 OProver and summarized in table 2.

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

### 4.1.2 Names

Literals, rule labels, constants and variables are all instances of names that occur in plausible 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.

```haskell
lexerL = (tokenL "!" %> "symbol", name1L, name2L)
```

### 4.1.3 Symbols and everything else

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

```haskell
lexL = dropWhites $ nofail $ total $ listL [lexerL]
```
4.2 Formulas
Formulas in this plausible logic implementation are defined by module Formula.

module Formula
  where

import List; import Parser; import Showing

4.2.1 Data type definitions
A formula is either a literal or formulas combined by negation, conjunction, disjunction or implication.

data PFormula lit = FLiteral lit |
  | FNot Formula |
  | Formula :+: Formula |
  | Formula :->: Formula |
  | FAnd [Formula] |
  | FOr [Formula] deriving (Eq, Ord)

Normally this synonym is used.
type Formula = PFormula Literal

4.2.2 Parsers
The syntax for a formula is:

  formula ::= formula "->" consequent |
  consequent |
  consequent "|" disjunct |
  disjunct |
  disjunct "&" conjunct |
  conjunct |

4.2.3 Operations on formulas
Class Formulaic defines operations that can be performed on formulas.
class Formulaic form where
toLit q converts formula q back to a single literal.

toLit :: form -> Literal

isAnd f returns True iff f = \( \bigvee \) s. isOr f iff True iff f = \( \bigvee \) s.
isOr, isAnd :: form -> Bool
dropAndOr f drops the \( \bigvee \) or \( \bigvee \) from formula f, returning the enclosed list of formulas.
dropAndOr :: form -> [form]
simplify f returns f simplified if possible. An alternate version simplify' is tuned for use by cnf and dnf. Use simplify if you want it to work harder.
simplify, simplify' :: form -> form
cnf f transforms f into conjunctive normal form. dnf f transforms f into disjunctive normal form.
cnf, dnf :: form -> form

if cnf f = \( \bigwedge \) c | s then cl f = \( \bigvee \) s | c and dcl f = \( \bigwedge \) s | c. If dnf f = \( \bigvee \) d | s then dcl f = \( \bigwedge \) d | s and dcl f = \( \bigwedge \) d | s.

March 22, 2004
countLiterals \( f \) returns the number of literals in \( f \).

instance Formulaic Formula where

\begin{align*}
\text{toLit} \ f &= \text{case simplify'} \ f \ of \\
& \quad \text{FLiteral} \ q \rightarrow q \\
& \quad \rightarrow \text{error "toLit: not literal"}
\end{align*}

isOr \( f \) = case \( f \) of

\begin{align*}
\text{FOr} \_ &= \text{True} \\
\_ &= \text{False}
\end{align*}

isAnd \( f \) = case \( f \) of

\begin{align*}
\text{FAnd} \_ &= \text{True} \\
\_ &= \text{False}
\end{align*}

dropAndOr \( f \) = case \( f \) of

\begin{align*}
\text{FAnd} \ fs &= \ text{fs} \\
\text{FOr} \ fs &= \ text{fs} \\
f &= \ [f]
\end{align*}

simplify \( f \) = case \( f \) of

\begin{align*}
\text{FNot} \ (\text{FLiteral} \ (\text{PosLit} \ n)) &= \ \text{FLiteral} \ (\text{NegLit} \ n) \\
\text{FNot} \ (\text{FLiteral} \ (\text{NegLit} \ n)) &= \ \text{FLiteral} \ (\text{PosLit} \ n) \\
\text{FNot} \ (\text{FLiteral} \ (\text{PosLit} \_ n \ as)) &= \ \text{FLiteral} \ (\text{NegLit} \_ n \ as) \\
\text{FNot} \ (\text{FLiteral} \ (\text{NegLit} \_ n \ as)) &= \ \text{FLiteral} \ (\text{PosLit} \_ n \ as) \\
\text{FNot} \ (\text{FNot} \ f) &= \ \text{simplify'} \ f \\
\text{FNot} \ (\text{FAnd} \ fs) &= \ \text{FAnd} \ \text{map} \ \text{neg} \ \text{fs} \\
\text{FNot} \ (\text{FOr} \ f) &= \ \text{simplify'} \ f \\
\text{FOr} \ fs &= \ \text{let} \ fs' = \ \text{snub} \ \text{fs} \\
in \ \text{if} \ \text{and} \ \text{(map} \ \text{isAnd} \ \text{fs'}) \ \text{then} \\
\text{simplify'} \ \text{FAnd} \ \text{map} \ \text{concat} \ \text{map} \ \text{dropAndOr} \\
\text{$fs'$} \\
\text{else} \\
\text{FAnd} \ \text{map} \ \text{simplify'} \ \text{fs'}
\end{align*}

forceAndOrs forces all of the Venus formula (or other entity) into \( f \land \ldots \) and \( f \lor \ldots \) form. This is usually more readable.

class HasAndOrs a where

\begin{align*}
\text{forceAndOrs} \ a &= \ a \\
\text{forceAndOrs} \ f &= \ \text{case} \ (\text{f}) \ \text{of} \\
\text{FLiteral} \ l &= \ \text{FLiteral} \ l \\
\text{FNot} \ f &= \ \text{FNot} \ (\text{forceAndOrs} \ f) \\
\text{FAnd} \ f &= \ \text{FAnd} \ (\text{forceAndOrs} \ f) \\
\text{FOr} \ f &= \ \text{FOr} \ (\text{forceAndOrs} \ f)
\end{align*}

4.2.4 Instance declarations

Textual output
instance Show Formula where

showsPrec p formula =
  case formula of
    FLiteral l -> shows l
    FNot f -> showString "~ " . showsParen 9 f
    f :&: f' -> showsParen 7 f . showString " & " . showsParen 7 f'
    f :|: f' -> showsParen 6 f . showString " | " . showsParen 6 f'
    f :->: f' -> showsParen 5 f . showString " -> " . showsParen 5 f'
    FAnd fs -> showString "/\{" . showWithSep ", " fs . showChar '}'
    FOr fs -> showString "\/{" . showWithSep ", " fs . showChar '}

where

showsParen precedence formula =
  if getPrecedence formula < precedence then
    showChar '(' . shows formula . showChar ')' else
    shows formula

getPrecedence formula =
  case formula of
    FLiteral _ -> 10
    FNot _ -> 9
    FAnd _ -> 9
    FOr _ -> 9
    _ :&: _ -> 7
    _ :|: _ -> 6
    _ :->: _ -> 5

instance HasLits Formula where

getLits f t =
  case f of
    FLiteral l -> getLits l t
    FNot f -> getLits f t
    f :&: f' -> getLits f $ getLits f' t
    f :|: f' -> getLits f $ getLits f' t
    f :->: f' -> getLits f $ getLits f' t
    FAnd fs -> foldr getLits t fs
    FOr fs -> foldr getLits t fs

instance HasConstNames Formula where

getConstNames f t =
  case f of
    FLiteral l -> getConstNames l t
    FNot f -> getConstNames f t
    f :&: f' -> getConstNames f $ getConstNames f' t
    f :|: f' -> getConstNames f $ getConstNames f' t
    f :->: f' -> getConstNames f $ getConstNames f' t
    FAnd fs -> foldr getConstNames t fs
    FOr fs -> foldr getConstNames t fs

instance HasVarNames Formula where

getVarNames f t =
  case f of
    FLiteral l -> getVarNames l t
    FNot f -> getVarNames f t
    f :&: f' -> getVarNames f $ getVarNames f' t
    f :|: f' -> getVarNames f $ getVarNames f' t
    f :->: f' -> getVarNames f $ getVarNames f' t
    FAnd fs -> foldr getVarNames t fs
    FOr fs -> foldr getVarNames t fs

instance Negatable Formula where

neg = simplify' . FNot

isPos f = case simplify' f of
  FLiteral q -> isPos q
  _ -> error "isPos of non-literal"

instance IsLiteral Formula where

toLiteral t f =
  case f of
    FLiteral l -> toLiteral l t
    _ -> error "not a literal"

fromLiteral a = FLiteral . fromLiteral a

instance Groundable Formula where

ground v c f =
  case f of
    FLiteral l -> ground v c l
    FNot f -> FNot (ground v c f)
    f :&: f' -> (ground v c f) :&: (ground v c f')
    f :|: f' -> (ground v c f) :|: (ground v c f')
    f :->: f' -> (ground v c f) :->: (ground v c f')
    FAnd fs -> FAnd (map (ground v c) fs)
    FOr fs -> FOr (map (ground v c) fs)

Forced evaluation

instance DeepSeq Formula where

deepSeq f x =
  case f of
    FLiteral l -> deepSeq l x
    FNot f -> deepSeq f x
    f :&: f' -> deepSeq f $ deepSeq f' x
    f :|: f' -> deepSeq f $ deepSeq f' x
    f :->: f' -> deepSeq f $ deepSeq f' x
    FAnd fs -> deepSeq fs x
    FOr fs -> deepSeq fs x

4.3 Rules

Module Rule implements a data type for representing rules in plausible logic descriptions and theories.

module Rule(
  PRule(..), Rule, ruleP, IsRule(..)
) where

import Parser; import Showing; import Formula
import DeepSeq; import Literal

infix 4 :->, :=>, :-\n
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 formula to be used. This makes this code a little more general, and makes possible some fancy stuff with multi-parameter type classes later on.

data PRule form = ![form] :-> !form
  | !form :=> !form
  | !form :-\ !form

As shorthand, use this type synonym.

type Rule = PRule Formula

4.3.2 Parsers

The syntax for a rule is:

antecedent ::= formulaSet
  | consequent
  | epsilon

March 22, 2004
which is implemented:

\[
\text{antecedentP} :: \text{Parser } [\text{Formula}]
\]

\[
\text{antecedentP} = \text{formulaSetP} <|> \text{consequentP}
\]

\[
\text{consequentP} = \text{epsilonA} #> \text{[]}
\]

\[
\text{ruleP} :: \text{Parser Rule}
\]

\[
\text{ruleP} = \text{antecedentP} <*> (\text{literalP} "symbol" "-\" <|> \text{literalP} "symbol" "\"\"") <*> \text{formulaP}
\]

\[
\text{IsRule} \text{ class collects the properties of rules and rule-like types.}
\]

\[
\text{isStrict :: rule form } \rightarrow \text{Bool}
\]

\[
\text{isPlausible :: rule form } \rightarrow \text{Bool}
\]

\[
\text{isDefeater :: rule form } \rightarrow \text{Bool}
\]

\[
\text{antecedent :: rule form } \rightarrow \text{[form]}
\]

\[
\text{consequent :: rule form } \rightarrow \text{form}
\]

\[
\text{instance IsRule PRule Formula where}
\]

\[
\text{isStrict r = case r of}
\]

\[
_ :-> _ -> True
\]

\[
_ -> False
\]

\[
\text{isPlausible r = case r of}
\]

\[
_ :=> _ -> True
\]

\[
_ -> False
\]

\[
\text{isDefeater r = case r of}
\]

\[
_ :-\_ _ -> True
\]

\[
_ -> False
\]

\[
\text{antecedent r = case r of}
\]

\[
a :-> _ _ -> a
\]

\[
a :=> _ _ -> a
\]

\[
a :-\_ _ -> a
\]

\[
\text{consequent r = case r of}
\]

\[
_ :-> c _ -> c
\]

\[
_ :=> c _ -> c
\]

\[
_ :-\_ c _ -> c
\]

\[
\]

\[
\text{4.3.3 Properties of rules}
\]

\[
The \text{IsRule} \text{ class collects the properties of rules and rule-like types.}
\]

\[
\text{isX r } \text{returns True iff r is an X. antecedent r returns the list of}
\]

\[
\text{formulas which are the antecedents of rule r. consequent r returns the}
\]

\[
\text{formula which is the consequent of r. This is a multi-parameter}
\]

\[
\text{type class, which relies on Haskell extensions.}
\]

\[
\text{class IsRule rul form where}
\]

\[
\text{isStrict :: rul form } \rightarrow \text{Bool}
\]

\[
\text{isPlausible :: rul form } \rightarrow \text{Bool}
\]

\[
\text{isDefeater :: rul form } \rightarrow \text{Bool}
\]

\[
\text{antecedent :: rul form } \rightarrow \text{[form]}
\]

\[
\text{consequent :: rul form } \rightarrow \text{form}
\]

\[
\text{instance IsRule PRule Formula where}
\]

\[
\text{isStrict r = case r of}
\]

\[
_ :-> _ _ -> True
\]

\[
_ -> False
\]

\[
\text{isPlausible r = case r of}
\]

\[
_ :=> _ _ -> True
\]

\[
_ -> False
\]

\[
\text{isDefeater r = case r of}
\]

\[
_ :-\_ _ -> True
\]

\[
_ -> False
\]

\[
\text{antecedent r = case r of}
\]

\[
a :-> _ _ -> a
\]

\[
a :=> _ _ -> a
\]

\[
a :-\_ _ -> a
\]

\[
\text{consequent r = case r of}
\]

\[
_ :-> c _ -> c
\]

\[
_ :=> c _ -> c
\]

\[
_ :-\_ c _ -> c
\]

\[
\]

\[
\text{4.3.4 Instance declarations}
\]

\[
\text{Textual output}
\]

\[
\text{instance Show Rule where}
\]

\[
\text{showsPrec p rule = case rule of}
\]

\[
(a :-> c) -> \text{showsAntecedent a} . \text{showString } " -> " . \text{shows c}
\]

\[
(a :=> c) -> \text{showsAntecedent a} . \text{showString } " => " . \text{shows c}
\]

\[
(a :-\_ c) -> \text{showsAntecedent a} . \text{showString } " -\" " . \text{shows c}
\]

\[
\text{where}
\]

\[
\text{showsAntecedent fs} = \text{showChar } '{' . \text{showWithSep } ", " . \text{showChar '}'
\]

\[
\text{Extracting literal names}
\]

\[
\text{instance HasLits Rule where}
\]

\[
\text{getLits r t } = \text{case r of}
\]

\[
(a :-> c) -> \text{foldr getLits } (\text{getLits c t}) a
\]

\[
(a :=> c) -> \text{foldr getLits } (\text{getLits c t}) a
\]

\[
(a :-\_ c) -> \text{foldr getLits } (\text{getLits c t}) a
\]

\[
\text{Simplifying formulas}
\]

\[
\text{instance HasAndOrs Rule where}
\]

\[
\text{forceAndOrs r } = \text{case r of}
\]

\[
\text{as :-> c } \rightarrow \text{map forceAndOrs as :-> forceAndOrs c}
\]

\[
\text{as :=> c } \rightarrow \text{map forceAndOrs as :=> forceAndOrs c}
\]

\[
\text{as :-\_ c } \rightarrow \text{map forceAndOrs as :-\_ forceAndOrs c}
\]

\[
\text{Extracting constant names}
\]

\[
\text{instance HasConstNames Rule where}
\]

\[
\text{getConstNames r t } = \text{case r of}
\]

\[
(a :-> c) \rightarrow \text{foldr getConstNames } (\text{getConstNames c t}) a
\]

\[
(a :=> c) \rightarrow \text{foldr getConstNames } (\text{getConstNames c t}) a
\]

\[
(a :-\_ c) \rightarrow \text{foldr getConstNames } (\text{getConstNames c t}) a
\]

\[
\text{Extracting variable names}
\]

\[
\text{instance HasVarNames Rule where}
\]

\[
\text{getVarNames r t } = \text{case r of}
\]

\[
(a :-> c) \rightarrow \text{foldr getVarNames } (\text{getVarNames c t}) a
\]

\[
(a :=> c) \rightarrow \text{foldr getVarNames } (\text{getVarNames c t}) a
\]

\[
(a :-\_ c) \rightarrow \text{foldr getVarNames } (\text{getVarNames c t}) a
\]

\[
\text{Grounding}
\]

\[
\text{instance Groundable Rule where}
\]

\[
\text{ground v c r } = \text{case r of}
\]

\[
\text{as :-> co } \rightarrow \text{map } (\text{ground v c}) \text{ as :-> ground v c co}
\]

\[
\text{as :=> co } \rightarrow \text{map } (\text{ground v c}) \text{ as :=> ground v c co}
\]

\[
\text{as :-\_ co } \rightarrow \text{map } (\text{ground v c}) \text{ as :-\_ ground v c co}
\]

\[
\text{Forced evaluation}
\]

\[
\text{instance DeepSeq Rule where}
\]

\[
\text{deepSeq r x } = \text{case r of}
\]

\[
\text{as :-> c } \rightarrow \text{deepSeq as } \&\& \text{deepSeq c x}
\]

\[
\text{as :=> c } \rightarrow \text{deepSeq as } \&\& \text{deepSeq c x}
\]

\[
\text{as :-\_ c } \rightarrow \text{deepSeq as } \&\& \text{deepSeq c x}
\]

\[
\text{4.4 Descriptions}
\]

\[
\text{module Description(}
\]

\[
\text{PDescription(..), Description, descriptionP,}
\]

\[
\text{normalizeDescription}
\]

\[
\text{) where}
\]

\[
\text{import Parser; import Showing; import Rule; import List}
\]

\[
\text{import ABLListUtils; import Formula; import Literal}
\]

\[
\]

\[
\text{4.4.1 Data type definitions}
\]

\[
\text{A plausible description consists of a set of facts (formulas), a set of}
\]

\[
\text{plausible rules and a set of defeaters. These parameterized type def-}
\]

\[
\text{initions make possible some fancy multi-parameter class definitions}
\]

\[
\text{later on.}
\]

\[
\text{data PDescription form = Description [form] [PRule form] [PRule form]}
\]

\[
\text{deriving Eq}
\]

\[
\text{For shorthand use:}
\]

\[
\text{type Description = PDescription Formula}
\]

March 22, 2004 11
A `dStatement` is an intermediate data structure used while parsing.

```haskell
data DStatement = Fact Formula
  | PlausibleRule Rule
  | Defeater Rule
```

### 4.4.2 Parser

**Syntax:**

```haskell
dStatement ::= plausible-rule | defeater-rule | formula
```

**Implemented:**

```haskell
dStatementP :: Parser DStatement
  = dataSatisfies ruleP (not . isStrict)
    @> ( -> if isPlausible r
      then PlausibleRule r
      else Defeater r)
  <|> formulaP @> Fact
```

```haskell
descriptionP :: Parser Description
  = total (many (dStatementP
  <*> nofail (literalP "symbol" ".")))
  @> makeDescription
where
makeDescription ss = case ss of
  [] -> Description [] [] []
  (s:ss) ->
    let Description fs ps ds = makeDescription ss
    in case s of
      Fact f -> Description (f:fs) ps ds
      PlausibleRule p -> Description fs (p:ps) ds
      Defeater d -> Description fs ps (d:ds)
```

### 4.4.3 Normalizing descriptions

```haskell
normalizeDescription :: Description -> Description
normalizeDescription (Description fs ps ds) =
  Description fs' ps' ds'
where
  fs' = cl $ FAnd fs
  ps' = [snub (b ++ map neg (c \ [q])) ::=> q |
    p <- ps, c <- cll (consequent p),
    q <- c, b <- dcll (FAnd (antecedent p))]
  ds' = [snub (b ++ map neg (c \ [q])) ::=q |
    d <- ds, c <- cll (consequent d), q <- c,
    b <- dcll (FAnd (antecedent d))]
```

### 4.4.4 Instance declarations

**Textual output**

```haskell
instance Show Description where
  showsPrec p (Description fs ps ds) =
    showString "\% facts:"
    . showWithTerm ".\n" fs
    . showString "\% plausible rules:"
    . showWithTerm ".\n" ps
    . showString "\% defeaters:"
    . showWithTerm ".\n" ds
```

**Extracting literal names**

```haskell
instance HasLits Description where
  getLits (Description fs ps ds) t
    = foldr getLits (fold getLits (fold getLits t fs) ps) ds
```

### 4.5 DescriptionParser

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

```haskell
module Main (main) where
import Parser; import PlausibleLexer; import Description
import System; import Check
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 $ "Description file name: " ++ path ++ \
    "\n"
  source <- readFile path
  parse source
parse :: String -> IO ()
parse source = do
  case checkParse lexerL descriptionP source of
    CheckFail msg -> putStrLn msg
    CheckPass desc -> do
      putStrLn "\nParsed:"
      putStrLn $ show desc
      putStrLn "\nNormalized:"
      putStrLn $ show $ normalizeDescription desc
```

### 4.6 Theories

The module `Theory` defines the plausible logic theory data types.

```haskell
module Theory(
    LabeledRule(..), LRule, Precomp(..), PTheory(..),
    Theory, makeTheory, checkParseTheory, groundCheck,
    Theoretical(..), dropLabel
) where
  import List; import ABRListUtils; import BSTree
  import SparseSet; import Parser; import Showing
  import PlausibleLexer; import Description; import Label
  import Priority; import DeepSeq; import Check
  import Rule; import Literal; import Formula
```

#### 4.6.1 Data type definitions

A plausible theory consists of a set of rules (some of which may be labeled), a priority relation, a set of literals which tag the theory as it is augmented with additional rules, and possibly some precomputed information to support proofs. These apparently redundantly parameterized type definitions make possible some fancy multi-parameter class definitions later on.

```haskell
data LabeledRule form = Rule !Label !(PRule form)
  deriving (Eq)
```

```haskell
data Precomp = Precomp {
  incr :: [[Literal]],
  incrq :: BSTree Literal [[Literal]]
}
```

```haskell
data PTheory rul =
  Theory [rul] [Priority] [Literal] (Maybe Precomp)
```

For shorthand use:

```haskell
type LRule = LabeledRule Formula
type Theory = PTheory LRule
```

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

```haskell
data Statement = LabeledRule LRule
  | Priority Priority
```

March 22, 2004 12
4.6.2 Making theories from descriptions

```
makeTheory :: Description -> Theory
makeTheory d = Theory
  where
    cs = the clauses derived from the facts in d.
    srs_s = the set of simple strict rules derived from those facts.
    srs_p = the set of simple plausible rules.
    srs_d = the set of simple defeasible rules.
    srs = the set of all simple rules.
    srs_s ++ srs_p ++ srs_d

cr_s q returns the compound strict rule for q if there is one. cr_p q
returns the compound plausible rule for q if there is one. cr_d q
returns the compound defeasible rule for q if there is one.
```

4.6.3 Parser

Syntax:

```
statement ::= rule
  | label "." * rule
  | priority

theory ::= (statement "." | statement "$")
```

4.6.4 Labeled rule operations

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

```
lookupRule l rs
  lookupRule _ [] = error "lookupRule: Labeled rule just ain't there."
  lookupRule l (Rule l' r : rs)
    | label == label' = r
    | otherwise = lookupRule l rs
```

4.6.5 Consistency checks

```
nonexistentLabels t returns the list of labels in t that occur in
priorities, but don't occur in rules. checkLabelsNotUsed is a check
that either passes the theory or returns an error message including
the nonexistent labels.
```

```
nonexistentLabels :: Theory -> [Label]
nonexistentLabels (Theory rs ps _ _ _) = (domBST . countPriorities) ps \n\n(| Rule l (Rule _ r) > r |
lookUpRule l rs returns the rule in rs that has label l.
```

```
lookUpRule :: Label -> [LRule] -> Rule
```

A Theory statement usually ends in ".", but some (for research
hacking purposes) need to bypass the usual consistency checks.
These end with with "!". The theory parser does not just return the
Theory. It also returns the lists of forced rules and priorities, those
that are marked with "!".

```
theoryP :: Parser (Theory, [LRule], [Priority])
theoryP =
  (total $ many $ statementP <*> (Priority p, (_,"!",_))
  <*> makeTheory
  where
    makeTheory ss = case ss of
      [] -> (Theory [] [] [] Nothing, [], [])
      s:ss ->
        let (Theory rs ps qs mp, rs', ps') = makeTheory ss
        in case s of
          Nothing -> (Theory rs ps qs mp, rs', ps')
        otherwise = lookupRule label rs

checkLabelsExist :: Check Theory Theory String
checkLabelsExist :: Theory String
checkLabelsExist (Theory rs ps _ _ _)
```

March 22, 2004
irregularPriorities t returns the list of priorities in t that involve
the wrong kinds of rules. checkIrregularPriorities is a check that
either passes the theory or returns an error message including the
offending priorities.

irregularPriorities :: Theory -> [Priority]
irregularPriorities (Theory rs ps _ _) = cp ps
where
cp [] = []
cp ((l1 :> l2) : ps)
| isStrict (lookupRule l2 rs) = (l1 :> l2) : cp ps
| otherwise = cp ps

checkIrregularPriorities :: Check Theory Theory String
checkIrregularPriorities t =
  case irregularPriorities t of
    [] -> CheckPass t
    ps -> CheckFail $ "Error: Irregular priorities: " ++ show ps

checkCycles is a check that either passes the theory or returns an
error message including the priorities involved in or connected to
cycles.

checkCycles :: Check Theory Theory String
checkCycles t0 (Theory _ ps _ _)
  = case cycles ps of
    [] -> CheckPass t0
    ps' -> CheckFail $ "Error: Cyclic priorities: " ++ show ps'

This sequences the lexing, parsing and checking.

checkParseTheory :: Check String Theory String
checkParseTheory source =
  case checkParse lexerL (total theoryP) source of
    CheckFail msg -> CheckFail msg
    CheckPass (t0 -> Theory rs ps qs mp),rs',ps') ->
      case (checkLabelsExist &? checkLabelsNotReused &? checkIrregularPriorities &? checkCycles) t of
        CheckFail msg -> CheckFail msg
        CheckPass _ ->
          case mp of
            Nothing -> id
            Just pre -> shows pre

       Theory (rs' ++ rs) (ps' ++ ps)

4.6.6 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 t0 (Theory rs ps fs mp)
  = let cs = flattenSS $ getConstNames t emptySS
      vs = getVarNames t emptySS
      rs' = concat $ map (groundAll cs) rs
      renewer :: Int -> ([LRule] -> ([LRule], BSTree Label (SparseSet Label)))
      renewer n rs = case rs of
        [] -> ([], emptyBST)
        ((Rule l r) : rs) ->
          let (rs', t) = renewer (n+1) rs
          in (Rule l (forceAndOrs r), t)
        ((Rule (Label l) r) : rs) ->
          let (rs', t) = renewer (n+1) rs
          in (Rule (Label l) (forceAndOrs r), t)
        ((Rule l r) : rs) ->
          let (rs', t) = renewer (n+1) rs
          in (Rule l (forceAndOrs r), t)
      (rs'', lmap) = renewer 0 rs'
      dupPri :: Priority -> [Priority]
      dupPri (1 :> l)
        = let Just lS = lookupBST 1 lmap
            in [lS ++ lmap]
        dupPri (1 :> l)
          = let Just lS = lookupBST 1 lmap
              in [lS ++ lmap]
        dupPri (1 :> l)
          = let Just lS = lookupBST 1 lmap
              in [lS ++ lmap]
        dupPri (1 :> l)
          = let Just lS = lookupBST 1 lmap
              in [lS ++ lmap]
      in if null cs && not (nullSS vs) then
          CheckFail "Can't ground variables. \n  \nNo constants."
        else
          CheckPass (Theory rs'' ps' fs' mp)

4.6.7 Theory operations
class Theoretical t where
tTheoryIdent t produces a string that identifies a theory and how it
has been augmented.

theoryIdent :: t -> String

4.6.8 Instance declarations

Theory operations

instance Theoretical Theory where
tTheoryIdent (Theory _ _ qs _) = case qs of
  [] -> "T"
  _ -> "T U " ++ show qs

Textual output
instance Show LRule where
  showsPrec p (Rule l r)
    = case l of
      Label "" -> shows r
      _ -> shows l . showString ": " . shows r

instance Show Precomp where
  showsPrec p precomp = showString "Inc(R) = " .
  shows (incr precomp) . showChar '\n' .
  shows "Inc(R,q) = " .
  shows (flattenBST (incrq precomp))

instance Show Theory where
  showsPrec p t
    = let Theory rs ps qs mp = forceAndOrs t
        showTheory "% Theory: 
          . showString (theoryIdent t) .
          . showChar '\n' .
          . showWithTerm "\n" rs
          . showWithTerm "\n" ps
          . (case mp of
            Nothing -> id
            Just pre -> shows pre
          )

Extracting literal names

instance HasLits LRule where
  getLits (Rule _ r) = getLits r
instance HasLits Theory where
  getLits (Theory rs ps qs _) t
    = foldr getLits (foldr getLits t qs) rs

Changing ands and ors

instance HasAndOrs LRule where
  forceAndOrs (Rule l r) = Rule l (forceAndOrs r)
inforceAndOrs (Theory rs' ps' qs' mp) = Theory (map forceAndOrs rs') ps' qs' mp

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

Rule properties

instance IsRule LabeledRule Formula where
  isStrict = isStrict . dropLabel
  isPlausible = isPlausible . dropLabel
  isDefeater = isDefeater . dropLabel
  antecedent = antecedent . dropLabel
  consequent = consequent . dropLabel

Grounding

instance HasConstNames LRule where
  getConstNames (Rule _ r) = getConstNames r

instance HasVarNames LRule where
  getVarNames (Rule _ r) = getVarNames r

instance Groundable LRule where
  ground v c (Rule l r) = Rule l (ground v c r)

instance HasConstNames Theory where
  getConstNames (Theory rs _ _ _ _) t = foldr getConstNames t rs

instance HasVarNames Theory where
  getVarNames (Theory rs _ _ _ _) t = foldr getVarNames t rs

Forced evaluation

instance DeepSeq LRule where
  (Rule l r) 'deepSeq' x = deepSeq l $ deepSeq r x

instance DeepSeq Precomp where
  pre 'deepSeq' x = deepSeq (incr pre) $ deepSeq (incrq pre) x

instance DeepSeq Theory where
  (Theory rs ps qs pre) 'deepSeq' x = deepSeq rs $ deepSeq ps $ deepSeq qs $ deepSeq pre x

4.7 Description2Theory

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

module Main (main) where
import System; import Parser; import PlausibleLexer
import Theory; import Check; import Description
import Formula
main :: IO ()
main = do
  paths <- getArgs
  if null paths
    then do
      source <- getContents
      transform Nothing source
    else run paths
  run :: [FilePath] -> IO ()
  run = mapM_ run1
  run1 :: FilePath -> IO ()
  run1 path = do
    source <- readFile path
    parse source

  parse :: String -> IO ()
  parse source = do
    case checkParse source of
      CheckFail msg -> putStrLn msg
      CheckPass t -> do
        putStrLn "Parsed theory:
        putStrLn $ show t
        case groundCheck t of
          CheckFail msg -> putStrLn msg
          CheckPass t' -> do
            putStrLn "Grounded theory:
            putStrLn $ show t'

4.8 TheoryParser

See the user’s guide (section 3.9) for a description of this module.

module Main (main) where
import System; import Parser; import Theory
import Check
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
    putStrLn $ "Theory file name: " ++ path ++ ".t"
    source <- readFile path
    parse source

  parse :: String -> IO ()
  parse source = do
    case checkParse source of
      CheckFail msg -> putStrLn msg
      CheckPass t -> do
        putStrLn "Parsed theory:
        putStrLn $ show t
        case groundCheck t of
          CheckFail msg -> putStrLn msg
          CheckPass t' -> do
            putStrLn "Grounded theory:
            putStrLn $ show t'

4.9 Inference Conditions

Module Inference defines the inference conditions for plausible logic.

module Inference (ProofSymbol(..), Tagged(..), taggedFormulaP, PlausibleLogic(..)) where
import Parser; import ABRListUtils; import Formula
import ThreadedTest; import DeepSeq; import Literal

4.9.1 Data type definitions
A tagged formula consists of a formula, 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. The proof symbols are defined by table 4.

data ProofSymbol = PS_s | PS_p | PS_l | PS_d
  deriving (Eq, Ord)

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

Before proofs are possible, a formula must be converted to CNF form. Get rid of all \( \wedge \) and \( \vee \) to express all formulas as nested lists of literals.
<table>
<thead>
<tr>
<th>constructor</th>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS_d</td>
<td>δ</td>
<td>definite</td>
</tr>
<tr>
<td>PS_l</td>
<td>λ</td>
<td>likely (plausible with ambiguity propagation)</td>
</tr>
<tr>
<td>PS_p</td>
<td>π</td>
<td>plausible</td>
</tr>
<tr>
<td>PS_s</td>
<td>σ</td>
<td>support</td>
</tr>
</tbody>
</table>

Table 4: The proof symbols and their Haskell representation and meanings.

4.9.2 Parser

The syntax for a tagged formula is:

```
proof_symbol ::= "d" | "l" | "p" | "s"
tagged_formula ::= ("+" | "-") proof_symbol formula
```

which is implemented:

```
proofSymbolP :: Parser ProofSymbol
proofSymbolP =
    literalP "name1" "d" #> PS_d
    <|> literalP "name1" "l" #> PS_l
    <|> literalP "name1" "p" #> PS_p
    <|> literalP "name1" "s" #> PS_s

taggedFormulaP :: Parser (Tagged Formula)
taggedFormulaP =
    (literalP "symbol" "+" <|> literalP "symbol" "-") proofSymbolP
    <|> nofail' "proof symbol expected" proofSymbolP
    <|> nofail' "formula expected" formulaP
    <*> nofail' "theory expected" theoryP

TheoryP :: Parser Theory
TheoryP =
    nameP "name1" <*> (taggedFormulaP <|> literalP "symbol" "d" |>
    PS_d <*> literalP "symbol" "l" |> PS_l <*> literalP "symbol" "p" |> PS_p
    <*> literalP "symbol" "s" |> PS_s
```

4.9.3 Overloaded functions

Class PlausibleLogic overloads the same 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 form are the names of the theory, rule, and formula types. The type for rules must be parameterized by the type for formulas, and the type for theories must be parameterized by the type for rules.

```
class (Ord lit, Show lit) => (HasLits a, Show a, Eq a, Ord a) =>
    HasLits (Tagged a) where
    getLits t s = case t of
        "+" -> Plus ps f
        "-" -> Minus ps f
```

4.9.4 Inference Conditions

\[ T \vdash tf \vdash \] is a test whether the tagged formula \( tf \) can be proved from theory \( T \). The argument \( t \) is the user-supplied, auxiliary proof function that is mutually recursive with this one and handles all state manipulations and/or I/O.

\[
(\vdash \vdash) :: (Monad m, ThreadedResult r) \Rightarrow
    \theta (\text{rul (form lit)}) \Rightarrow \text{Tagged } [[[\text{lit}]]]
    \Rightarrow \text{ThreadTest } m r s \Rightarrow \text{ThreadTest } m r s
\]

The definition of this function consists of the inference conditions that are displayed in figures 1 and 2.

4.9.5 Instance declarations

Textual output

```
instance Show ProofSymbol where
    showsPrec p ps = case ps of
        PS_d -> showChar 'd'
        PS_l -> showChar 'l'
        PS_p -> showChar 'p'
        PS_s -> showChar 's'
```

```
instance (HasLits a, Show a, Ord a) =>
    HasLits (Tagged a) where
    getLits t s = case t of
        Plus ps q -> getLits ps . showChar '+' . shows q
        Minus ps q -> getLits ps . showChar '-' . shows q
```

Extracting literal names

```
instance (HasLits a, Show a, Ord a) =>
    HasLits (Tagged a) where
    getLits t s = case t of
        Plus ps q -> getLits ps . showChar '+' . shows q
        Minus ps q -> getLits ps . showChar '-' . shows q
```

Mapping

```
instance Functor Tagged where
    fmap f t = case t of
        Plus ps x -> Plus ps (f x)
        Minus ps x -> Minus ps (f x)
```

Forcing evaluation

```
instance (DeepSeq a) =>
    DeepSeq (Tagged a) where
    deepSeq f x = case f of
        Plus ps f -> deepSeq ps $ deepSeq f x
        Minus ps f -> deepSeq ps $ deepSeq f x
```

instance DeepSeq ProofSymbol where ()

4.10 Prove

This module implements provers for plausible logic using only the simple theory data type defined in module Theory (section 4.6).

module Prove(
    addPrecomp, resolution, fct, pr, incR, incRq, Hist, prove
) where

import List; import ABRListUtil; import CFUTime
import Args; import ProofResult; import History
import Theory; import Inference; import Maybe
import BSTree; import Literal; import Rule
import Formula; import Priority
import ThreadTest; import Label
import ABRStringUtil
\(+\Lambda\):  
\((\vdash)\ t \ (\text{Plus} \ a \ []) \ (\vdash) = \)  
mkTest True

\(-\Lambda\):  
\((\vdash)\ t \ (\text{Minus} \ a \ []) \ (\vdash) = \)  
mkTest False

\(+\Lambda\):  
\((\vdash)\ t \ (\text{Plus} \ fs@(_:_:_)) \ (\vdash) = \)  
fA \(fs \ \forall \ x \rightarrow t \ \vdash \text{Plus} \ a \ [f]\)

\(-\Lambda\):  
\((\vdash)\ t \ (\text{Minus} \ a \ fs@(_:_:_)) \ (\vdash) = \)  
tE \(fs \ \forall \ x \rightarrow t \ \vdash \text{Minus} \ a \ [f]\)

\(+\forall\):  
\((\vdash)\ t \ (\text{Plus} \ []) \ (\vdash) = \)  
mkTest False

\(-\forall\):  
\((\vdash)\ t \ (\text{Minus} \ []) \ (\vdash) = \)  
mkTest True

\(+\forall\):  
\((\vdash)\ t \ (\text{Plus} \ [fs@(_:_:_)]) \ (\vdash) = \)  
fA \(\text{powSet\_ge1}'\ fs \ \forall \ fs' \rightarrow fA \ fs' \ (\forall \ x \rightarrow\)  
orgment \(t \ fs' \ f \ \vdash \text{Plus} \ a \ [f]\)

\(-\forall\):  
\((\vdash)\ t \ (\text{Minus} \ [fs@(_:_:_)]) \ (\vdash) = \)  
fA \(\text{powSet\_ge1}'\ fs \ \forall \ fs' \rightarrow tE \ fs' \ (\forall \ x \rightarrow\)  
orgment \(t \ fs' \ f \ \vdash \text{Minus} \ a \ [f]\)

\(+\delta\):  
\((\vdash)\ t \ (\text{Plus} \ PS_d \ [[q]]) \ (\vdash) = \)  
tE \(\text{rsq} \ t \ q \ \forall \ r \rightarrow t \ \vdash \text{Plus} \ PS_d \ (\text{ants} \ t \ r))

\(-\delta\):  
\((\vdash)\ t \ (\text{Minus} \ PS_d \ [[q]]) \ (\vdash) = \)  
fA \(\text{rsq} \ t \ q \ \forall \ r \rightarrow t \ \vdash \text{Minus} \ PS_d \ (\text{ants} \ t \ r))

\(+\lambda\):  
\((\vdash)\ t \ (\text{Plus} \ PS_l \ [[q]]) \ (\vdash) = \)  
tE \(\text{rsq} \ t \ q \ \forall \ r \rightarrow t \ \vdash \text{Plus} \ PS_l \ (\text{ants} \ t \ r)) || |\  
tE \(\text{rpq} \ t \ q \ s \ \forall \ u \rightarrow t \ \vdash \text{Plus} \ PS_l \ (\text{ants} \ t \ u))

\(-\lambda\):  
\((\vdash)\ t \ (\text{Minus} \ PS_l \ [[q]]) \ (\vdash) = \)  
fA \(\text{rsq} \ t \ q \ s \ \forall \ u \rightarrow t \ \vdash \text{Minus} \ PS_l \ (\text{ants} \ t \ u)) || |\  
\text{resolve} \ cs \ ds \ returns \ Just \ a \ clause \ that \ is \ a \ resolvent \ of \ cs \ and \ ds, \ or \ Nothing. \ Precondition: \ cs \ and \ ds \ must \ be \ in \ strictly \ ascending \ order; \ e.g. \ [\nega, \negb, \negc, \negd, \negf, \negg]. \ Postcondition: \ If \ there \ is \ a \ resolvent, \ it \ is \ returned \ with \ the \ same \ ordering.\n
\text{resolve} :: (Ord \ a, \ Negatable \ a) \Rightarrow \ [a] \rightarrow [a] \rightarrow \ Maybe \ [a]\n
\text{resolve} \ cs \ ds = \text{resolve'} \ cs \ (\text{reverse} \ ds) \ \square \ \square \ \

\text{Figure 1: Inference conditions for plausible logic.}\n
\subsection{Histories}\n
This type is shorthand for the history that maps tagged formulas to prior results. A history is keyed, not only by the tagged formula, but also by the list of literals which indicate for which augmented theory the result was obtained.

\text{type Hist} = \text{History} ([\text{Literal}], \text{Tagged} \ [[\text{Literal}]]) \ \text{ProofResult}\n
\subsection{Inconsistent Literals}\n
In this section we compute the objects required for all proof levels other than \(\delta\) with plausible theory \(T = (R, >)\).
\[\pi: (\langle\cdot\rangle t (Plus \ PS_p \ [[q]]) \langle\cdot\rangle) =
\begin{align*}
&\text{tE (rsq t q) } (\forall r \rightarrow t \vdash Plus \ PS_p \ (\text{ants } t \ r)) \\
&\text{tE (rpq t q) } (\forall r \rightarrow t \vdash Minus \ PS_p \ (\text{ants } t \ r)) \\
&\text{fa (inc t q) } (\forall c \rightarrow tE \ cs \ (\forall s \rightarrow t \vdash Minus \ PS_p \ (\text{ants } t \ s)))
\end{align*}
\]

\[\pi: (\langle\cdot\rangle t (Minus \ PS_p \ [[q]]) \langle\cdot\rangle) =
\begin{align*}
&\text{fa (rsq t q) } (\forall r \rightarrow t \vdash Minus \ PS_p \ (\text{ants } t \ r)) \\
&\text{fa (rpq t q) } (\forall r \rightarrow t \vdash Minus \ PS_p \ (\text{ants } t \ r)) \\
&\text{tE (inc t q) } (\forall c \rightarrow tE \ cs \ (\forall s \rightarrow t \vdash Plus \ PS_p \ (\text{ants } t \ s)))
\end{align*}
\]

\[\sigma: (\langle\cdot\rangle t (Plus \ PS_s \ [[q]]) \langle\cdot\rangle) =
\begin{align*}
&\text{tE (rsq t q) } (\forall r \rightarrow t \vdash Plus \ PS_s \ (\text{ants } t \ r)) \\
&\text{tE (rpq t q) } (\forall r \rightarrow t \vdash Plus \ PS_s \ (\text{ants } t \ r)) \\
&\text{fA (inc t q) } (\forall cs \rightarrow tE \ cs \ (\forall c \rightarrow fA \ (rsq_rqr t c r) \ (\forall s \rightarrow t \vdash Minus \ PS_l \ (\text{ants } t \ s))))
\end{align*}
\]

\[\sigma: (\langle\cdot\rangle t (Minus \ PS_s \ [[q]]) \langle\cdot\rangle) =
\begin{align*}
&\text{fA (rsq t q) } (\forall r \rightarrow t \vdash Minus \ PS_s \ (\text{ants } t \ r)) \\
&\text{fA (rpq t q) } (\forall r \rightarrow t \vdash Minus \ PS_s \ (\text{ants } t \ r)) \\
&\text{tE (inc t q) } (\forall cs \rightarrow tE \ cs \ (\forall c \rightarrow fA \ (rsq_rqr t c r) \ (\forall s \rightarrow t \vdash Plus \ PS_l \ (\text{ants } t \ s))))
\end{align*}
\]

Figure 2: More inference conditions for plausible logic.
4.10.3 Precomputed stuff

\[
\text{addPrecomp} :: \text{Theory} \to \text{Theory}
\]

\[
\text{adds the precomputed stuff to theory } T.
\]

\[
\text{addPrecomp} \ t \equiv \text{addPrecomp stuff to } T.
\]

\[
\text{addPrecomp} \ t \equiv \text{addPrecomp stuff to } T.
\]

4.10.4 Plausible logic instance

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

\[
\text{PlausibleLogic PTheory LabeledRule PFormula Literal}
\]

\[
\text{where}
\]

\[
\text{rq} \quad \text{(Theory } \ rs \ \_ \ \_ \ \_) \ q = \text{filter } \{(== q) \cdot \text{toLit } \cdot \text{consequent} \} \ \text{rs}
\]

\[
\text{rsq} \quad \text{(Theory } \ rs \ \_ \ \_ \ \_) \ q = \text{filter } \{(\forall r \rightarrow \text{isStrict r } \&\& \text{toLit } \text{(consequent r) }== q) \} \ \text{rs}
\]

\[
\text{rpq} \quad \text{(Theory } \ rs \ \_ \ \_ \ \_) \ q = \text{filter } \{(\forall r \rightarrow \text{isPlausible r } \&\& \text{toLit } \text{(consequent r) }== q) \} \ \text{rs}
\]

\[
\text{rpqr} \quad \text{(Theory } \ rs \ \_ \ \_ \ \_) \ q \ \text{lr0 } \text{(Rule } l \ \_ \ \_ r) = \text{filter } \{(\forall r' \rightarrow \text{isPlausible r' } \&\& \text{toLit } \text{(consequent r') }== q) \&\& (r' > 1) \text{ 'elem' ps) rs}
\]

\[
\text{ants t r }= \text{map } \text{(map toLit ) . concat . cll) } \&\& \text{antecedent r}
\]

\[
\text{orgment } \text{(Theory } \ rs \ \_ \ \_ \ q) \text{ fs f} = \text{let fs' }= \text{map neg } \text{(snd fs) \ (add fst toLit) } \ \text{qs} \text{ fs' = [Rule (Label "") [] := FLiteral q] }\text{ q < fs'} \text{ qs' = snub qs fs' in Theory (rs ++ rs') ps qs' Nothing}
\]

\[
\text{inc } \text{(Theory } \_ \ \_ \ \_ \ \_ \ \_ \ \_ ) = \text{let Just c }= \text{lookupBST q (incr precomp) in c}
\]

4.10.5 Provers

\[
\text{prove } T \ tf () \equiv \text{return } (r, ()) \text{ where r is the result of trying to prove tagged formula } tf \text{ with theory } T. \text{ This is the simplest prover, with no trace, no history no loop checking, and not well founded.}
\]

\[
\text{prove } T \ tf () \equiv \text{return } (r, ()) \text{ where r is the result of trying to prove tagged formula } tf \text{ with theory } T. \text{ This is the simplest prover, with no trace, no history no loop checking, and not well founded.}
\]

\[
\text{prove } T \ tf () \equiv \text{return } (r, ()) \text{ where r is the result of trying to prove tagged formula } tf \text{ with theory } T. \text{ This is the simplest prover, with no trace, no history no loop checking, and not well founded.}
\]

\[
\text{prove } T \ tf () \equiv \text{return } (r, ()) \text{ where r is the result of trying to prove tagged formula } tf \text{ with theory } T. \text{ This is the simplest prover, with no trace, no history no loop checking, and not well founded.}
\]

\[
\text{prove } T \ tf () \equiv \text{return } (r, ()) \text{ where r is the result of trying to prove tagged formula } tf \text{ with theory } T. \text{ This is the simplest prover, with no trace, no history no loop checking, and not well founded.}
\]
prove_nht $T \tau f (0, h, \"\") returns $(r, (ng, h, \"\"))$, where $r$ is the result of trying to prove tagged formula $f$ 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 ID ProofResult (Int, Hist, String)
prove_nht $t@\tau (Theory _ _ qs _) tf (ng, h, indent) =
let \(t' = addPrecomp t\)
in case getResult h (qs, tf) of
\( Just r -> do \)
goal indent "To Prove" t' \( tf \)
return (r, (ng, h, indent))
Nothing -> do
goal indent "To Prove" t' \( tf \)
\( \{t' \|\-\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\![...\]
4.11 Prover

See the user’s guide (section 3.10) for a description of this module.
putStrLn "Fct:"
putStrLn $ show $ fct rs
putStrLn "pr(Res(Fct)):":
putStrLn $ show $ resolution $ fct rs
putStrLn "Inc(R):":
putStrLn $ show $ incR rs

proofLoop options h
  _ -> do
    h' <- proveOne t options input h
    proofLoop options h'

showHelp :: IO ()
showHelp = putStrLn
  "To prove things: type a tagged formula.
 Other commands:
  ? = this message
  q = quit
  t = print theory
  f = forget history
  e = show current prover
  e engine = select prover engine from {-, n,
    nh, nhl, t, nt, nht, nhl} \n  l [path] = read a new theory file
    [named path].
  i = show intermediates in Inc(R,q)"

proveOne :: Theory -> Options -> String -> Hist
  -> IO Hist
proveOne t options input h = case checkParse lexerL (total taggedFormulaP) input of
  CheckFail msg -> do putStrLn msg
                      return h
  CheckPass tf -> prove t options "nhlt" tf h

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

4.12 Optimized Theories
Module OTheory defines a data type for storage of plausible logic theories that facilitates faster proofs.

module OTheory (OPFormula(..), OFormula, ORuleIndex(..), ORule,
  OPrecomp(..), OPTheory(..), OTheory,
  makeOTheory) where

import Array; import BSTree; import SparseSet
import Theory; import DeepSeq; import Literal
import Formula; import Rule; import Label
import Priority

4.12.1 Data types
This is a dummy type used to parameterize the rule type.
data OPFormula lit = OFormula [[lit]]
type OFormula = OPFormula OLiteral

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

newtype ORuleIndex form = OR Int
  deriving (Eq, Ord, Show)

A complete theory ready to use consists of the following components:
num2lit a mapping from optimized literals back to their original forms;
lit2num a mapping from literals forward to their optimized from;
pcons the consequents of all rules;
pants the antecedents of all rules;
plausStart the index of the first plausible rule;
defStart the index of the first defeater rule;
prq the indices of all rules with consequent q, R[q];
prsq the indices of all strict rules with consequent q, Rs[q];
prpq the indices of all plausible rules with consequent q, Rp[q];
priorities stores, for all rules, the indices of the rules that beat that rule;
qs the list of literals used to tag augmented theories; and
mp the precomputed information.

data OTheory rul = OTheory {
  num2lit :: LitArray,
  lit2num :: LitTree,
  pcons :: Array ORule OLiteral,
  pants :: Array ORule [[OLiteral]],
  plausStart :: ORule,
  defStart :: ORule,
  prq :: Array OLiteral [ORule],
  prsq :: Array OLiteral [ORule],
  pripq :: Array OLiteral [ORule],
  priorities :: Array ORule [ORule],
  qs :: [OLiteral],
  mp :: Maybe OPrecomp
}

A convenient synonym.
type OTheory = OTheory ORule

4.12.2 Building an optimized theory
makeOTheory $ T builds an optimized theory using the set of literal names S and theory T.

makeOTheory :: SparseSet Literal -> Theory -> OTheory
makeOTheory s t0(Theory rs ps _ _) = let
Build the literal lookup tables.
  (num2nam, nam2num) = makeLitTables s
  (_,nLit) = bounds num2nam
Sort and count the rules by type. nLit is the greatest positive OLiteral.
  srs = filter isStrict rs
  prs = filter isPlausible rs
  drs = filter isDefeater rs
  srs' = srs ++ prs ++ drs
  n_srs = length srs
  n_prs = length prs
  n_drs = length drs
  n_rs = n_srs + n_prs + n_drs
Accumulate arrays of consequents and antecedents.
  pcons' = listArray (OR 0, OR(n_rs - 1))
    (map (toOLiteral nam2num . consequent) rs')
toOAntecedent :: [OLiteral] -> [OLiteral]
toOAncedent fs = let f :: Formula
    f = FAnd fs
    qss :: [Formula]
    qss = cll f
    in map (map (toOLiteral nam2num)) qss
  pants' = listArray (OR 0, OR (n.rs - 1))
    (map (toOAncedent . antecedent) rs')

Accumulate arrays of rules, presorted by kind and consequent.
Accumulate arrays of rules, sorted by kind and consequent and what beats what. \texttt{labelTable} is a mapping from label names to the rule with that label. \texttt{toRuleIndex \_i} returns the number of the rule with label \_i. \texttt{priorities} is an array that maps each rule to the set of rules it beats. \texttt{beats r r' \_r} returns \texttt{r > r'}.

\texttt{labelTable} = pairs2BST \n filter (not . null . fst) \n $ \texttt{zip (map \texttt{(Rule (Label l) \_)} -> l) \texttt{rs'}} \n \texttt{[0..]} \n$ \texttt{map \texttt{\_1 \_2 \texttt{\_}}} \n \texttt{(toRuleIndex 12, toRuleIndex 11)) ps}

Build the optimized theory.

\texttt{in OTheory \{} \n \texttt{num2lit} = num2nam, \n \texttt{lit2num} = num2num, \n \texttt{pcons} = pcons\', \n \texttt{pants} = pants\', \n \texttt{plausStart} = OR \texttt{n_srs} + \texttt{n_prs}, \n \texttt{prq} = prq', \n \texttt{prsq} = prsq', \n \texttt{prpq} = prpq', \n \texttt{priorities} = listArray (bounds \texttt{priorities'}) (map flattenSS (elems \texttt{priorities'})), \n \texttt{qs} = [], \n \texttt{mp} = Nothing \n\}

\textbf{4.12.3 Instance declarations}

\textbf{Theory operations}

\texttt{instance Theoretical OTheory where}
\texttt{theoryIdent \_t = case \texttt{qs \_t of}}
\texttt{[] -> "T"}
\texttt{-> let \texttt{qs' :: [Literal]} \n \texttt{qs' = map (fromLiteral (num2lit \_t)) \texttt{(qs \_t)}} \n \texttt{in \"T U \" ++ show \texttt{qs'}}

\textbf{Indexing}

\texttt{instance Num \texttt{(ORuleIndex form) where}}
\texttt{range (a,b) = [a..b]} \n\texttt{index (OR \texttt{i}i,\texttt{j}) (OR \texttt{j}j) = \texttt{j} - \texttt{i}} \n\texttt{inRange (OR \texttt{i}i,OR \texttt{j}j) (OR \texttt{k}k) = \texttt{i} <= k \&\& k <= \texttt{j}}

\textbf{Showing}

\texttt{instance Show \texttt{OPrecomp where}}
\texttt{showsPrec \_p \texttt{precomp} = showString \"\texttt{\_pInc(R)} = \" .}
\texttt{shows \texttt{(oInc precomp)} . showString \"\texttt{\_pInc(R,q)} = \" .}
\texttt{shows \texttt{(oIncq precomp)}}

\texttt{instance Show \texttt{OTheory where}}
\texttt{showsPrec \_p \_t =}
\texttt{\begin{verbatim}
  showString \"num2lit: " . shows (num2lit \_t)
  . showString \"lit2num: " . shows (lit2num \_t)
  . showString \"pcons: " . shows (pcons \_t)
  . showString \"pants: " . shows (pants \_t)
  . showString \"plausStart: "
  . shows (plausStart \_t)
  . showString \"ndefStart: " . shows (ndefStart \_t)
  . showString \"npstart: " . shows (npstart \_t)
  . showString \"nplausStart: " . shows (nplausStart \_t)
  . showString \"npriorities: "
  . shows (npriorities \_t)
  . showString \"nqs: " . shows (nqs \_t)
  . showChar \"\n\" . (case \texttt{mp \_t of}}
  \texttt{Nothing -> id}
  \texttt{Just \texttt{pre} -> shows \texttt{pre}}
\end{verbatim}}

\textbf{Forced evaluation}

\texttt{instance DeepSeq OTheory where}
\texttt{deepSeq \_t \_x = deepSeq (num2lit \_t) \_f}
\texttt{deepSeq (lit2num \_t) \_f}
\texttt{deepSeq (pcons \_t) \_f}
\texttt{deepSeq (pants \_t) \_f}
\texttt{deepSeq (plausStart \_t) \_f}
\texttt{deepSeq (ndefStart \_t) \_f}
\texttt{deepSeq (npstart \_t) \_f}
\texttt{deepSeq (nplausStart \_t) \_f}
\texttt{deepSeq (npriorities \_t) \_f}
\texttt{deepSeq (nqs \_t) \_f}
\texttt{deepSeq (mp \_t) \_f}

\texttt{instance \texttt{DeepSeq \_\_r \texttt{ORule where}}}
\texttt{deepSeq \_r \_x = deepSeq (num2lit \_r) \_f}
\texttt{deepSeq (lit2num \_r) \_f}
\texttt{deepSeq (pcons \_r) \_f}
\texttt{deepSeq (pants \_r) \_f}
\texttt{deepSeq (plausStart \_r) \_f}
\texttt{deepSeq (ndefStart \_r) \_f}
\texttt{deepSeq (npstart \_r) \_f}
\texttt{deepSeq (nplausStart \_r) \_f}
\texttt{deepSeq (npriorities \_r) \_f}
\texttt{deepSeq (nqs \_r) \_f}
\texttt{deepSeq (mp \_r) \_f}

\textbf{4.13 \texttt{OProve}}

This module implements provers for plausible logic using the optimized array-based theory data type defined in module \texttt{OTheory} (section 4.12).

\texttt{module OProve(}}
\texttt{  OTF, makeOTF, unmakeOTF, Hist, prove, addPrecomp}
\texttt{ ) where}
\texttt{import List; import Array; import CPUTime; import Args}
\texttt{import ABRListUtils; import SparseSet; import Inference}
\texttt{import OTheory; import History; import ProofResult}
\texttt{import qualified Prove; import Literal; import Formula}
\texttt{import ThreadedTest; import Theory; import BSTree}

\textbf{4.13.1 Optimized tagged formulas}

A CNF formula is represented by [[\texttt{GLiteral}]]\. This type synonym defines the optimized representation of a tagged formula. \texttt{type OTF = Tagged [[\texttt{GLiteral}]]}
4.13.5 Plausible logic instance

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

instance
PlausibleLogic OPTheory ORuleIndex OPFormula OLiteral

where
rq t q = prq t ! q
rsq t q = FAnd (FOr $ map (FLiteral .$ fromOLiteral (num2lit t))) tf

4.13.6 Provers

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

prove_ :: OTheory -> OTF

-> ThreadedTest Maybe ProofResult ()

prove_ t tf () =

let t' = addPrecomp t

in (t' |-> tf) prove_ ()

prove_n T tf () returns (r, ng), where r is the result of trying to prove tagged formula tf with theory T and ng is the number of subgoals required to do so.

prove_n :: OTheory -> OTF

-> ThreadedTest Maybe ProofResult Int

prove_n t tf ng = do

let t' = addPrecomp t

(r, ng') <- (t' |-> tf) prove_n ng

return (r, ng' + 1)

goal indent msg T tf prints the indentation, the message the theory identifier and the tagged formula, making a line from the trace.

goal :: String -> String -> OTheory -> OTF -> IO ()

goal indent msg t tf = do

putStrLn indent

putStrLn msg

putStrLn $ theoryIdent t

putStrLn $ FAnd $ map (FLiteral . fromOLiteral (num2lit t))) tf

putStrLn $ show t

putStrLnLn "")
perform_t T tf "" returns (r,""), where r is the result of trying to prove tagged formula tf with theory T. A trace is printed.

prove_t :: OTheory -> OTF
  -> ThreadedTest IO ProofResult String
prove_t t tf indent = do
  let t' = addPrecomp t
  goal indent "To Prove" t' tf
      (r, (_,_) <- (t' |-- tf) prove_t t tf)
  goal indent (show r) t' tf
  return (r, indent)
prove_nt T tf (0,"") returns r,(ng,""), where r is the result of proving that the formula has been proved.

prove_nt :: OTheory -> OTF
  -> ThreadedTest IO ProofResult (Int, Hist)
prove_nt t tf (ng,h) =
    let t' = addPrecomp t
    in case getResult h (qs t, tf) of
        Nothing -> do
          tf' = fmap (map (map toLit) . cll) tf
          let ls' = tfls 'unionSS' ls
          if tfls 'isSubSet' ls then
            Just (ng, addProof h (qs t, tf) Botton)
          else
            Just Pending
          Nothing -> do
            tf' = fmap (map (map toLit) . cll) tf
            let ls' = tfls 'unionSS' ls
            if tfls 'isSubSet' ls then
              Just (ng, addProof h (qs t, tf) Botton)
            else
              Just Pending
        Just r' -> do
          tf' = fmap (map (map toLit) . cll) tf
          let ls' = tfls 'unionSS' ls
          return (r, (ng',h'))
      return (r, (ng,h))
prove_nh :: OTheory -> OTF
  -> ThreadedTest IO ProofResult (Int, Hist)
prove_nh t tf (ng,h,indent) =
  let t' = addPrecomp t
  in case getResult h (qs t, tf) of
      Nothing -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        if tfls 'isSubSet' ls then
          Just (ng, addProof h (qs t, tf) Botton)
        else
          Just Pending
      Just r' -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        return (r, (ng',h'))
    return (r, (ng,h,indent))
prove_nhlt :: OTheory -> OTF
  -> ThreadedTest IO ProofResult (Int, Hist)
prove_nhlt t tf (ng,h) =
  let t' = addPrecomp t
  in case getResult h (qs t, tf) of
      Nothing -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        if tfls 'isSubSet' ls then
          Just (ng, addProof h (qs t, tf) Botton)
        else
          Just Pending
      Just r' -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        return (r, (ng',h'))
    return (r, (ng,h,indent))
prove_nhl :: OTheory -> OTF
  -> ThreadedTest IO ProofResult (Int, Hist)
prove_nhl t tf (ng,h) =
  let t' = addPrecomp t
  in case getResult h (qs t, tf) of
      Nothing -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        if tfls 'isSubSet' ls then
          Just (ng, addProof h (qs t, tf) Botton)
        else
          Just Pending
      Just r' -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        return (r, (ng',h'))
    return (r, (ng,h,indent))
prove_nht :: OTheory -> OTF
  -> ThreadedTest IO ProofResult (Int, Hist)
prove_nht t tf (ng,h) =
  let t' = addPrecomp t
  in case getResult h (qs t, tf) of
      Nothing -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        if tfls 'isSubSet' ls then
          Just (ng, addProof h (qs t, tf) Botton)
        else
          Just Pending
      Just r' -> do
        tf' = fmap (map (map toLit) . cll) tf
        let ls' = tfls 'unionSS' ls
        return (r, (ng',h'))
    return (r, (ng,h,indent))
4.14 OProver

See the user's guide (section 3.11) for a description of this module.
March 22, 2004

4.15 Scalable Test Theories

Module TestTheories generates plausible theories for performance testing. These theories are described in detail in appendix B.

```haskell
module TestTheories (generateTheory, generateTF, generateMetrics) where
import Theory; import Inference; import Formula
import Literal; import Rule; import Label
import Priority; import Description

infix 7 >>>

4.15.1 Shorthand
Scalable theories are usually built with literals of the form \( a_i \). \( a_i \) returns such a literal. \( \overline{a} \) returns the corresponding negation.

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

Theories are built from (usually) labeled rules. \( r_i \) adds a label \( R_i \) to rule \( r \). Priorities indicate one rule beats another. \( r_1 >> r_2 \) returns a priority \( r_1 > r_2 \).

```haskell
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)
```

4.15.2 Levels theories

levelsTheory \( n \) returns theory \( \text{levels}(n) \). levelsTF returns the default tagged formula +\( \pi_{a_0} \) the proof of which exercises all of theory \( \text{levels}(n) \).

```haskell
levelsTheory :: Int -> Theory
levelsTheory n = Theory (rules (-1)) (priorities 0) [] Nothing
    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+1)))
                 | otherwise = (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)
```

showHelp :: IO ()
showHelp = putStrLn
    "To prove things: type a tagged formula.\n\nOther commands:\n\n? = this message\nq = quit\nt = print theory\nf = print history\np = show current prover\ne = select prover engine from \{a, n, \n\n\n1 [path] = read a new theory file\n\n\n\nproveOne :: SparseSet Literal -> Theory -> OTheory ->
        Options -> String -> Hist -> IO (SparseSet Literal, OTheory, Hist)
proveOne ls t ot options input h = case checkParse lexerL (totalTaggedFormulaP) input of
    CheckFail msg -> do putStrLn msg
    return (ls, ot, h)
    CheckPass tf -> prove ls t ot options "nhlt" tf h
    quit :: IO ()
    quit = putStrLn "Goodbye."

4.15.1 Shorthand
Scalable theories are usually built with literals of the form \( a_i \). \( a_i \) returns such a literal. \( \overline{a} \) returns the corresponding negation.

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

Theories are built from (usually) labeled rules. \( r_i \) adds a label \( R_i \) to rule \( r \). Priorities indicate one rule beats another. \( r_1 >> r_2 \) returns a priority \( r_1 > r_2 \).

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)

4.15.2 Levels theories

levelsTheory \( n \) returns theory \( \text{levels}(n) \). levelsTF returns the default tagged formula +\( \pi_{a_0} \) the proof of which exercises all of theory \( \text{levels}(n) \).

```haskell
levelsTheory :: Int -> Theory
levelsTheory n = Theory (rules (-1)) (priorities 0) [] Nothing
    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+1)))
                 | otherwise = (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)
```
4.15.3 Competitor theories

compDesc n returns description compD(n). compTheory n returns the corresponding theory comp(n). compTheory' n returns a faster, direct computation of that theory. compTF returns the default tagged formula +πa₀ the proof of which exercises all of theory comp(n).

| compDesc :: Int -> Description
| compDesc n = Description (facts 2) (rules 1) []
| where
| a_1 = a 1
| na_1 = na 1
| facts i |
| i <= n |
| = (a i :->: a_1) : facts (i + 1)
| otherwise = []
| rules i |
| i == 1 |
| = ([] :=> a 0)
| : ([a_1] :=> na 0)
| : ([] :=> na_1)
| : rules (i + 1)
| i <= n |
| = ([] :=> a i)
| : rules (i + 1)
| otherwise = []

compTheory :: Int -> Theory
compTheory = makeTheory . compDesc

compTheory' :: Int -> Theory
compTheory' n = Theory (rules 1) [] Nothing
where
a_1 = a 1
na_1 = na 1
rules i |
| i == 1 |
| = r 0 ([] :=> a 0)
| : r 1 ([a_1] :=> na 0)
| : r 2 ([] :=> na_1)
| : rules (i + 1)
| i <= n |
| = r (3*i+3) ([a_i] :=> a_1)
| : r (3*i+2) ([na_1] :=> na i)
| : r (3*i+1) ([] :=> a i)
| : rules (i + 1)
| otherwise = [r (3*n) (foldl1 (:) [a j | j <- [2..n]]) :-> a_1]]

4.15.4 Null theories

nullTheory returns theory null. nullTF n returns the default tagged formula +πV{a₁, ..., aₙ}.

nullTheory :: Theory
nullTheory = Theory [] [] Nothing

nullTF :: Int -> Tagged Formula
nullTF n = Plus PS_p (foldl1 (:) [a i | i <- [1..n]])
import System; import CPUTime; import SparseSet
import Args; import Theory; import TestTheories
import OTheory; import History; import DeepSeq
import qualified OProve; import qualified Prove
import BSTree; import Literal; import Rule
import Formula

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

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

run' :: [String] -> IO ()
run' args = do
  let (options, thName: sizes) = findOpts [FlagS "t", FlagS "o", ParamS "e", FlagS "m"] args
  sizes' = map read sizes
  time0 <- getCPUTime
  th = time0 'seq' generateTheory thName sizes'
  tf = generateTF thName sizes'
  time1 <- (th, tf) 'deepSeq' getCPUTime
  putStrLn $ "Time to build theory (s) = " ++ show (fromIntegral (time1 - time0) / 1.0e12)
  case (th, tf) of
    (Nothing, _) ->
      putStrLn ("ERROR: no such theory: " ++ thName ++ unwords sizes)
    (_, Nothing) ->
      putStrLn ("ERROR: no such tagged formula: " ++ thName ++ unwords sizes)
    (Just th, Just tf) -> case lookupBST "t" options of
      Just FlagMinus ->
        putStrLn $ "Computed metrics:
        # rules = " ++ show r
        ++ " priorities = " ++ show p
        ++ " size = " ++ show s
        ++ " lines = " ++ show l
        _ -> return ()
    let Theory rs ps _ _ = th
      nrs = length rs
      nps = length ps
      lqs = sum $ map (sum . map countLiterals . antecedent) rs
      putStrLn $ "# rules: " ++ show nrs
      putStrLn $ "# priorities: " ++ show nps
      putStrLn $ "### total size = " ++ show (nrs + nps + lqs)
    case lookupBST "o" options of
    let Just FLAGMinus = do
      time0 <- getCPUTime
      th'@(Theory _ _ _ mp) =
        timel <- mp 'deepSeq' getCPUTime
        putStrLn $ "Time to precomp (s) = " ++ show (fromIntegral (timel - time0) / 1.0e12)
        time0 <- getCPUTime
        ot = timel 'seq'
    Prove.prove th options "nhl" tf emptyHistory
    putStrLn $ "Time to precomp (s) = " ++ show (fromIntegral (timel - time0) / 1.0e12)
    Prove.prove s th '
    options "nhl" tf emptyHistory
    return ()
  execute

4.17 CGI Tool

This module implements the CGI tool that provides a web interface for the Phobos system. Section 3.13 describes its use.

module Main (main) where

import Directory; import List; import Char; import Check
import Parser hiding (cons); import BSTree; import CGI
import PlausibleLexer; import Description; import Theory
import OTheory; import Inference; import SparseSet
import OProve; import Literal; import Formula
import History

4.17.1 Paths

These constants will require modification to set Phobos 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 descriptions and 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/plaus-info/"
else if installWhere == "kurango" then
  "doc/"
else
  "doc/"
theoryDir = if installWhere == "hunchentoot" then
  "/Program Files/Apache Group/Apache/htdocs/plaus-theories/"
else if installWhere == "kurango" then
  "theories/"
else
  "theories/"

infoURL, theoryURL :: String
infoURL = if installWhere == "hunchentoot" then
  "http://localhost/plaus-info/"
else if installWhere == "kurango" then
  "doc/"
else
  "doc/"
theoryURL = if installWhere == "hunchentoot" then
  "http://localhost/plaus-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.
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)

4.17.2 Main entry point
main :: IO ()
main = do
printMimeHeader
queryString <- getQUERY_STRING
case queryString of
"" -> doWelcome
"new-description" -> doNewDescription
"theory" -> doTheory
"proof" -> doProof
"syntax" -> doSyntax
"proof-help" -> doProofHelp
_ -> doBadQuery

4.17.3 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 $ "Phobos: " ++ title
bodyE [("background", infoURL ++ "background.jpg")]
let i = 1
in [do
put "$"
put "Phobos"
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 color. norm item displays an item in a row with the normal background color. high item displays the item in a highlight background color. oops item displays the item in a row with an error-indicating background color. oops' message displays a plain text message in a PÆE element. whoops title message 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 . pref . 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.

4.17.4 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 Plausible Logic System" [
  high introMsg,
  high pickADescription,
  high newDescription
]

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

pickADescription :: IO ()
pickADescription = form "theory" (do
  h2N $ put "Select an Example Plausible Description"
  fileNames <- getDirectoryContents theoryDir
  let fileNames' = sort $ filter ((== 'd') . head)
  reverse fileNames
  do in $ put "Phobos"
  put "="
  put $ title
  tableE [["cellpadding","10"],
  ["cellspacing","10"],
  ["width","100%"] $ sequence_ rows
  imgE_ ["src", infoURL ++ "logo.jpg"]
)

descriptionOption :: FilePath -> IO ()
descriptionOption file = do
  contents <- readFile $ theoryDir ++ file
  optionE ["value",filename] $ put $ trim $ lines $ contents
trim :: String -> String
trim = dropWhile isSpace . reverse . dropWhile isSpace . reverse

newDescription :: IO ()
newDescription = do
  h2N $ put "Create a New Plausible Description"
  put "Click "
  link "new-description" $ put "here"
  put " to create a new plausible description."

4.17.5 New description
doNewDescription displays the page with the form where new descriptions may be typed in.

doNewDescription :: IO ()
doNewDescription = wrap "New Description" [high $ form "theory" (do
  pN $ put "Create a New Plausible Description"
  h2N $ put "Create a New Plausible Description"
  put "Click "
  put "new-description" $ put "here"
  put " to create a new plausible description."
)]
4.17.6 Theory

doTheory displays the page where the description and theory are displayed and tagged formulas are prompted for.

doTheory :: IO ()
doTheory = do
  let title = "Plausible Description and Theory"
  formData <- getFormData
  lookupGuard formData ["origin","description"]
    \ cs -> whoops title $ "Missing " ++ cs ++ "."
  source <- if origin == "file" then
    readFile $ theoryDir ++ description
  else
    return description
  let cd = (emptyCheck "description" &? checkParse lexerL descriptionP) source
  (theory, ct) <- case cd of
    CheckFail _ -> return $ ("", CheckFail "")
    CheckPass des -> if origin == "form" then
      let th = makeTheory des
      in return (show th, CheckPass th)
    else do
      let path = theoryDir ++ reverse ("t" : tail (reverse description))
      exists <- doesFileExist path
      if exists then do
        source <- readFile path
        return (source, (emptyCheck "theory" &? checkParseTheory) source)
      else do
        let th = makeTheory des
        return (show th, CheckPass th)
  wrap title ()
    (high $ showTheory "Description" source)
    : (case cd of
      CheckFail msg -> [oops' msg]
      CheckPass des -> (high $ showTheory "Theory" theory)
    )
    : [ case ct of
      CheckPass _ ->
      high $ showTheory "Theory" CheckPass mg ->
      [oops' mg]
    ]
  )

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 -> String -> IO ()
showTheory title t = do
  h2N $ put "Plausible " ++ title
  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","Plausible theory"] $ return ()

4.17.7 Proof

doProof displays the page containing the results of a query.

doProof :: IO ()
doProof = do
  let title = "Proof"
  formData <- getFormData
  lookupGuard formData ["theory", "priorities", "tagged formula", "prover"]
    \ cs -> whoops title $ "Missing " ++ cs ++ "."
  [t,pr,tf,p] <- case cd of
    CheckFail msg -> [oops' msg]
    CheckPass vals ->
      let source = readFile $ theoryDir ++ vals
      in return (source, (emptyCheck "theory" &? checkParseTheory) source)
  wrap title
    (high $ showQuery "Tagged Formula"
    source)
    : [ case (emptyCheck "tagged formula" &? checkParseL (total taggedFormulaP))
    of CheckFail msg -> [oops' msg]
      CheckPass tf -> (high $ h2N $ put "Proof"
                    proveIt' t tf proveIt)
    ]
  )

showQuery :: String -> IO ()
showQuery tf = high (do
  h2N $ put "Tagged Formula"
A Syntax Summary

This is a summary description the syntax accepted by this implementation of plausible 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' ::= "\n"
    | any-character comment2'
```

A.2 Identifiers

```
namel ::= lower-case-letter {letter | digit | "."}
name2 ::= upper-case-letter {letter | digit | "."}
```

A.3 Literals

```
argument ::= namel | name2
argList ::= (" argument ", argument ")
literal ::= ["\"] namel [argList]
```

A.4 Formulas

```
formula ::= formula "->" consequent
    | consequent
consequent ::= consequent "|" disjunct
    | disjunct
disjunct ::= disjunct "&" conjunct
    | conjunct
conjunct ::= literal
    | (" formula ")
    | "\" formulaSet
    | "/" formulaSet
formulaSet ::= {" "}
    | {" formula (" formula ") "}
```

A.5 Rules

```
anteCEDent ::= formulaSet
    | consequent
rule ::= antecedent ("->" | "\") consequent
```

A.6 Descriptions

```
dStatement ::= plausible-rule | defeater-rule | formula
description ::= (dStatement "," )
```

A.7 Labels and Priorities

```
label ::= name2
priority ::= label ":" rule
```

A.8 Theories

```
statement ::= rule
    | label ":" rule
    | priority
theory ::= {statement "," | statement ")"
```

A.9 Tagged Formulas

A query to this system is a tagged formula; a formula to be proved, tagged by the level of proof required. Formulae to be proved must be grounded.

```
proof_symbol ::= "d" | "l" | "p" | "s"
tagged_formula ::= ("\n" | ")" proof_symbol formula
```

B Scalable Test Theories

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

B.1 Levels Theories

Description

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

```
levels(n) = {
    r_0: {} ⇒ a_0
    r_1: {a_1} ⇒ ~a_0
    r_2: {} ⇒ a_1
    r_3: {a_2} ⇒ ~a_1
    r_4: {} ⇒ a_2
    r_5: {a_3} ⇒ ~a_2
    ...
    r_{2n+2}: {} ⇒ a_{2n+1}
    r_{2n+3}: {a_{2n+2}} ⇒ ~a_{2n+1}
    r_{2n+4} > r_{2n+2}
    r_{2n+4} > r_{2n+3}
    ...
    r_{2n+2} ⇒ a_{2n+2}
```

A proof of +πa_0 will use every rule and priority. A variant levels~(n) omits the priorities.

Alternate description

Levels theories levels(n) = (R, >) where R = ∪ {{r_{2i} : {} ⇒ a_i, r_{2i+1} : {a_{i+1}} ⇒ ~a_i} | i ∈ [0, 2n + 1]} ∪ {{r_{2n+3} ⇒ a_2} and > = {r_{2n+4} > r_{2n+2} | i ∈ [1, 3..2n + 1]}. The implementation of functions that generate levels theories is given in section 4.15.2.

March 22, 2004
B.2 Competitor theories

Description

Competitor theory \( \text{comp}(n) \) is derived from the description \( \text{comp}_D(n) \), which consists of rules \( \{ \} \Rightarrow a_0, \{ a_1 \} \Rightarrow \sim a_0 \) and \( \{ \} \Rightarrow \sim a_1 \) followed by a cascade of facts \( a_i \rightarrow a_1 \) and rules \( \{ \} \Rightarrow a_i \) for \( i \in [2..n] \).

\[
\text{comp}_D(n) = \begin{cases} 
\{ \} \Rightarrow a_0 \\
\{ a_1 \} \Rightarrow \sim a_0 \\
\{ \} \Rightarrow \sim a_1 \\
a_2 \rightarrow a_1 \\
\vdots \\
a_0 \rightarrow a_1 \\
\{ \} \Rightarrow a_n 
\end{cases}
\]

The theory \( \text{comp}(n) \) includes all of the plausible rules from \( \text{comp}_D(n) \), the simple strict rules \( \{ a_1 \} \rightarrow a_1 \) and \( \{ \sim a_1 \} \rightarrow \sim a_1 \), and the non-simple strict rule \( \{ a_2 \lor \ldots \lor a_n \} \rightarrow a_1 \).

\[
\text{comp}(n) = \begin{cases} 
\text{rule } 0: \{ \} \Rightarrow a_0 \\
\text{rule } 1: \{ a_1 \} \Rightarrow \sim a_0 \\
\text{rule } 2: \{ a_2 \} \Rightarrow \sim a_1 \\
\text{rule } 3: \{ \sim a_1 \} \rightarrow a_1 \\
\text{rule } 4: \{ \} \Rightarrow a_2 \\
\vdots \\
\text{rule } 3n-3: \{ a_0 \} \rightarrow a_1 \\
\text{rule } 3n-2: \{ \sim a_1 \} \rightarrow \sim a_n \\
\text{rule } 3n-1: \{ \} \Rightarrow a_n \\
\text{rule } 3n: \{ a_2 \lor \ldots \lor a_n \} \rightarrow a_1 
\end{cases}
\]

A proof of \( +\pi a_0 \) exercises all these rules and for \( n > 2 \) is only possible if loops are detected, and the \( +\lor \) inference condition is used to its fullest extent.

The implementation of functions that generate competitor theories is given in section 4.15.3.

B.3 Null theories

Null theory, \( \text{null} \), is used to exercise the \( +\lor \) inference condition. The theory is empty. The query to prove (unsuccessfully) is the disjunction of \( n \) literals, none of which are supported.

The implementation of functions that generate null theories is given in section 4.15.4.

B.4 Theory Sizes

A Phobos 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 5 lists the formulae that predict these metrics for the scalable test theories described above. The metrics reported are:

- 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 rules, priorities and literals in the bodies of all rules.

<table>
<thead>
<tr>
<th>theory</th>
<th>rules</th>
<th>priorities</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>levels(n)</td>
<td>( 4n + 5 )</td>
<td>( n + 1 )</td>
<td>( 7n + 8 )</td>
</tr>
<tr>
<td>levels(^{-}(n))</td>
<td>( 4n + 5 )</td>
<td>( 0 )</td>
<td>( 6n + 7 )</td>
</tr>
<tr>
<td>comp(n)</td>
<td>( 3n + 1 )</td>
<td>( 0 )</td>
<td>( 6n - 1 )</td>
</tr>
<tr>
<td>null</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Table 5: Sizes of scalable test theories

References