Abstract

This is a Haskell implementation of Decisive Plausible Logic. Decisive Plausible Logic is a new variant of Plausible Logic that handles the looping that complicated previous implementations explicitly, and is defined in terms of a proof function with a three-valued result like the previous implementations, though these essential attributes were not reflected in the formal definition of the logic.

Contents

1 Introduction ................................. 2
2 Definition ................................. 2
  2.1 The language of Decisive Plausible Logic .... 2
  2.2 Decisive Plausible Logic ................. 3
  2.3 Summary ................................ 3
3 Installation .............................. 3
  3.1 Obtaining this system ................. 3
  3.2 Compiling DPL ....................... 4
  3.3 Compiling without make ............... 4
4 Using DPL ................................ 4
  4.1 The DPL tool ......................... 4
  4.2 Description file syntax .............. 4
  4.3 Invoking the DPL tool ............... 12
5 Implementation .......................... 12
  5.1 DPLLexer ............................... 12
    5.1.1 Comments ......................... 12
    5.1.2 Names ............................ 12
    5.1.3 Strings ......................... 13
    5.1.4 Symbols and everything else .... 13
  5.2 Constants .............................. 14
    5.2.1 Data type ....................... 14
    5.2.2 Parsers .......................... 14
    5.2.3 Collecting constants .......... 14
    5.2.4 Instance declarations ......... 14
  5.3 Variables ................................ 14
    5.3.1 Data type ....................... 14
    5.3.2 Parsers .......................... 14
    5.3.3 Collecting variables .......... 14
    5.3.4 Grounding ....................... 15
    5.3.5 Instance declarations ......... 15
  5.4 Arguments .............................. 15
    5.4.1 Data type ....................... 15
    5.4.2 Parsers .......................... 15
    5.4.3 Instance declarations .......... 15
  5.5 Atoms ................................ 15
    5.5.1 Data type ....................... 15
    5.5.2 Parsers .......................... 16
    5.5.3 Collecting atoms ............... 16
    5.5.4 Instance declarations ........ 16
  5.6 OAtoms ................................ 16
    5.6.1 Data types ...................... 17
    5.6.2 Building atom maps ............ 17
    5.6.3 Looking up atoms ............... 17
  5.7 Literals ................................ 17
    5.7.1 Data type ....................... 17
    5.7.2 Parser ........................... 17
    5.7.3 Negation ........................ 17
    5.7.4 Instance declarations ........ 17
  5.8 OLiteralst ............................ 18
    5.8.1 Data types ...................... 18
    5.8.2 Looking up literals .......... 18
    5.8.3 Instance declarations ........ 18
  5.9 Types ................................ 18
    5.9.1 Data types ...................... 18
    5.9.2 Parsers .......................... 18
    5.9.3 Evaluating the named types ... 19
    5.9.4 Evaluating a type ............. 19
    5.9.5 Collecting DOrderings .......... 20
    5.9.6 Instance declarations ......... 20
  5.10 TypeDecs .............................. 20
    5.10.1 Data types ...................... 20
    5.10.2 Parsers .......................... 21
    5.10.3 Instance declarations ....... 21
  5.11 LitSets ................................ 22
    5.11.1 Data types ...................... 22
    5.11.2 Parsers .......................... 22
    5.11.3 Methods ........................ 22
    5.11.4 Flattening ..................... 23
    5.11.5 Instance declarations ....... 23
  5.12 Formulas .............................. 23
    5.12.1 Data types ...................... 24
    5.12.2 Parsers .......................... 24
    5.12.3 Tautologies .................... 24
    5.12.4 Resolution ..................... 24
    5.12.5 Instance declarations ....... 25
  5.13 Priorities ............................. 25
    5.13.1 Data type definition .......... 25
    5.13.2 Parser ........................... 25
    5.13.3 Instance declarations ........ 26
  5.14 Rules ................................ 26
    5.14.1 Data type definitions ....... 26
    5.14.2 Parsers .......................... 26
    5.14.3 Interrogating rules .......... 26
    5.14.4 Instance declarations ....... 26
  5.15 Tags ................................ 27
    5.15.1 Data types ...................... 27
    5.15.2 Parsers .......................... 27
    5.15.3 Interrogating orderings ...... 27
  5.16 Descriptions .......................... 28
    5.16.1 Data type definitions ....... 28
    5.16.2 Parser ........................... 28
    5.16.3 Semantic checks .............. 30
    5.16.4 Asserting Orderings .......... 30

1 DPL – an implementation of Decisive Plausible Logic. Copyright (C) 2005, Andrew Rock
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as pub-
lished by the Free Software Foundation; either version 2 of the Li-
cense, or (at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied war-
ranty of MERCHANTABILITY or FITNESS FOR A PARTICU-
LAR PURPOSE. See the GNU General Public License for more
details.
You should have received a copy of the GNU General Public Li-
cense along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-
1301 USA

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

Plausible Logic [1, 2] is an improvement on Defeasible Logic [3, 4]. Defeasible Logic is a new variant of Plausible Logic that handles the looping that complicated previous implementations explicitly, and is defined in terms of a proof function with a three-valued result like the previous implementations, though these essential attributes were not reflected in the formal definition of the logic. This document describes an implementation of Defeasible Plausible Logic (Deimos [5, 6, 7]) and Plausible Logic (Phobos [1, 2, 8], BPL [9], and CPL [10]).

Section 2 is a reformattting of the Decisive Plausible Logic definition by David Billington. My only contribution is the LaTeX markup and some slight editing. Section 3 describes how to obtain and build the DPL program. Section 4 is the user's guide to the DPL program, detailing input and command line formats. Section 5 is a complete listing of the Haskell sources for this system.

2 Definition

This section is a transcription into LaTeX of the definition of Decisive Plausible Logic by David Billington.

2.1 The language of Decisive Plausible Logic

We often abbreviate "if and only if" by "iff." X is a subset of Y if X ⊆ Y; the notation X ⊊ Y means X ⊆ Y and X ≠ Y, and denotes that X is a proper subset of Y. The empty set is denoted by \( \emptyset \), and the set of all integers by \( \mathbb{Z} \). If m and n are integers then we define \([m..n]\) = \{i \in \mathbb{Z} : m \leq i \leq n\}. Let S be any set. The cardinality of S is denoted by |S|.

Our **alphabet** is the union of the following four pairwise disjoint sets of symbols: a non-empty finite set, Atoms, of (propositional) atoms; the set \( \{ \neg, \land, \lor, \Rightarrow, \Leftrightarrow, \Rightarrow \} \) of connectives; the set \( \{ µ, α, π, β, δ \} \) of tags denoting various proof algorithms; and the set of punctuation marks consisting of the comma and both braces. By a **literal** we mean any atom, α, or its negation, \( \neg α \). A **clause**, \( L \), is the disjunction of a finite set, \( L \), of literals. \( \emptyset \) is the empty clause or *falsum* and is thought of as always being false. If \( l \) is a literal then we regard \( V\{l\} \) as another notation for \( l \) and so each literal is a clause. A clause \( V \) is a **tautology** iff both an atom and its negation are in \( L \). A **contingent clause** is a clause which is not empty and not a tautology.

A **cnf-formula**, \( AC \), is the conjunction of a finite set, \( C \), of clauses. A **dnf-formula**, \( VD \), is the disjunction of a finite set, \( D \), of dual-clauses. If \( c \) is a clause then we regard \( \{c\} \) as another notation for \( c \). If \( d \) is a dual-clause then \( \{d\} \) as another notation for \( d \). Thus \( \{l\} = l \lor \emptyset \). Neither the verum nor the falsum are literals. The verum is not a clause, and the falsum is not a dual-clause. A **cnf-formula**, \( AC \), is the conjunction of a finite set, \( C \), of clauses. A **dnf-formula**, \( VD \), is the disjunction of a finite set, \( D \), of dual-clauses.

Let \( r \rightarrow c \) (read \( r \) implies \( c \) or \( r \) causes \( c \)) or \( c \) is a consequence of \( r \) be a rule iff \( (A(r), \uparrow(c), c(r)) \) where \( A(r) \) is a finite set of literals called the **antecedent** of \( r \), \( \downarrow(c(r)) \) is a literal called the **consequent** of \( r \). A rule \( r \) which contains the **strict rule**, \( \rightarrow \), is called a **strict rule** and is usually written \( A(r) \Rightarrow c(r) \). A rule \( r \) which contains the **plausible arrow**, \( \Rightarrow \), is called a **plausible rule** and is usually written \( A(r) \Rightarrow c(r) \). The antecedent of a rule can be the empty set. The set of all rules is denoted by \( \mathcal{R} \); the set of all clauses is denoted by \( \mathcal{C} \); the set of all dual-clauses is denoted by \( \mathcal{D} \); the set of all cnf-formulas is denoted by \( \mathcal{Cnf} \); the set of all dnf-formulas is denoted by \( \mathcal{Dnf} \).

We define the **complement**, \( \sim f \), of a formula \( f \) and the complement, \( \sim F \), of a set of formulas \( F \) as follows. If \( f \) is an atom then \( \sim f = \neg f \); and \( \sim f = f \). If \( L \) is a set of literals then \( \sim L = \{ \sim l : l \in L \} \). If \( V \) is a clause then \( \sim V = \sim L \lor L \). If \( \mathcal{A} \) is a dual-clause then \( \sim \mathcal{A} = \lor \mathcal{A} \). If \( V \) is a set of clauses or a set of dual-clauses then \( \sim E = \{ \sim c : c \in E \} \). If \( AC \) is a cnf-formula then \( \sim AC = \lor AC \). If \( VD \) is a dnf-formula then \( \sim VD = \land VD \). If \( VD \) is a set of formulas then \( \sim F = \{ \sim f : f \in F \} \).

Define \( r \) to be a rule iff \( (A(r), \uparrow(c), c(r)) \) where \( A(r) \) is a finite set of literals called the **antecedent** of \( r \), \( \downarrow(c(r)) \) is a literal called the **consequent** of \( r \). A rule \( r \) which contains the **strict rule**, \( \rightarrow \), is called a **strict rule** and is usually written \( A(r) \Rightarrow c(r) \). A rule \( r \) which contains the **plausible arrow**, \( \Rightarrow \), is called a **plausible rule** and is usually written \( A(r) \Rightarrow c(r) \). The antecedent of a rule can be the empty set. The set of all rules is denoted by \( \mathcal{R} \).

Let \( R \) be any set of rules. The set of antecedents of \( R \) is denoted by \( A(R) \); that is \( A(R) = \{ A(r) : r \in R \} \). The set of consequents of \( R \) is denoted by \( c(R) \); that is \( c(R) = \{ c(r) : r \in R \} \). We denote the set of strict rules in \( R \) by \( R_\star \), the set of plausible rules in \( R \) by \( R_\Rightarrow \), and the set of defeaters in \( R \) by \( R_\downarrow \). Also we define \( R_\setminus \) \( = R_\Rightarrow \cup R_\downarrow \) and \( R_\setminus \) \( = R_\Rightarrow \cup R_\downarrow \).

Let \( l \) be any literal. If \( C \) is any set of clauses define \( C[l] = \{ VL \in C : l \in L \} \) to be the set of all clauses in \( C \) which contain \( l \). If \( R \) is any set of rules and \( L \) is any set of literals then define \( R[l] \) \( = \{ r \in R : l \in c(r) \} \) to be the set of all rules in \( R \) which end with \( l \); and \( R[L] \) \( = \{ r \in R : c(r) \in L \} \) to be the set of all rules in \( R \) which have a consequent in \( L \).

Any binary relation, \( > \), on any set \( S \) is **cyclic** iff there exists a sequence, \( (r_1, r_2, \ldots, r_n) \) where \( n \geq 1 \), of elements of \( S \) such that \( r_1 > r_2 > \cdots > r_n > r_1 \). A relation is **acyclic** iff it is not cyclic.

November 25, 2013
If \( R \) is a set of rules then \( R \) is a priority relation on \( R \) if \( R \) is an acyclic binary relation on \( R \) such that \( \forall s > t \) is a subset of \( R_p \times R_{pd} \). We read \( r_1 > r_2 \) as \( r_1 \) beats \( r_2 \), or \( r_2 \) is beaten by \( r_1 \). Let \( R[\langle t \rangle] = \{ t \in R[\langle i \rangle] : t > s \} \) be the set of all rules in \( R \) with consequent \( l \) that beats.

A plausible description of a situation is a 4-tuple \( PD = (Ax, R_p, R_{pd}, >) \) such that PD1, PD2, PD3, and PD4 all hold.

PD1. \( Ax \) is a set of contingent clauses.

PD2. \( R_p \) is a set of plausible rules.

PD3. \( R_{pd} \) is a set of defeater rules.

PD4. \( > \) is a priority relation on \( R_{pd} \).

The clauses in \( Ax \), called axioms, characterise the aspects of the situation that are certain.

Let \( S \) be a set of clauses. A clause \( C_n \) is resolution-derivable from \( S \) iff there is a finite sequence of clauses \( C_1, \ldots, C_n \) such that for each \( i \in \{1, \ldots, n\} \), either \( C_i \in S \) or \( C_i \) is the resolvent of two preceding clauses. The sequence \( C_1, \ldots, C_n \) is called a resolution-derivation of \( C_n \) from \( S \). The set of all clauses which are resolution-derivable from \( S \) is denoted by \( \text{Res}(S) \). So \( S \subseteq \text{Res}(S) \).

Define \( \text{Res}(S) = \text{Res}(S) - \{ \{ \} \} \) to be the set of all non-empty clauses in \( \text{Res}(S) \). Define \( \text{Min}(S) = \{ VL \in S : \text{if } K \subseteq L \text{ then } \forall K \notin S \} \). Then \( \text{Min}(S) \) is the minimal class of clauses in \( S \).

Let \( PD = (Ax, R_p, R_{pd}, >) \) be a plausible description. Define the set of minimal contingent facts \( \text{Fct}(Ax) \), generated from \( Ax \) by \( \text{Fct}(Ax) = \text{Min}(\text{Res}(Ax)) - \{ VL : \forall L \in S \} \). The set of strict rules, \( R_s \), generated from \( \text{Fct}(Ax) \) is defined by \( R_s = \{ (\langle l \rangle - \{ l \} : l \in L \} \cup \{ VL \in \text{Fct}(Ax) \} \). Define \( R = R_s \cup R_p \cup R_{pd} \) to be the set of rules generated from \( PD \). The ordered pair \( (R \rangle \) is called a plausible theory.

Let \( T \) = \( (R \rangle \) be a plausible theory. The set of axioms in the plausible description from which \( T \) was generated is denoted by \( Ax(T) \). Define \( \text{Tct}(T) = \{\exists(\forall c(r) \cup \forall \neg a(r)) : r \in R_s \} \). Then \( \text{Tct}(T) \) is the set from which \( R_s \) was generated and so \( \text{Tct}(T) = \text{Fct}(Ax(T)) \). If \( L \) is a set of literals then define \( \text{Tct}(T, L) = \{ \forall K \in \text{Tct}(T) : \{ k \in L \} \} \). Define \( \text{Incl}(T, l) = \{ l \} \in \text{Tct}(T) \}. Each member of \( \text{Incl}(T, l) \) is a minimal set of literals which is inconsistent with \( l \).

2.2 Decisive Plausible Logic

Given a plausible theory \( T \) = \( (R \rangle \) we define the following ten functions \( \text{P}, \text{Strict}, \text{Plaus}, \text{For}, \text{Nullified}, \text{Disabled}, \text{Defeated}, \text{Beatn}, \text{Inappl} \) all of which depend on \( T \). \( P \) is called the proof function of \( T \). The other nine functions merely assist in the definition of \( P \). Each function’s domain and range is specified as follows. \( P \) takes any tagged cnf-formula \( \lambda f \) and any set \( B \) of literals and returns a result in \{+1, 0, +1\}. \text{Strict} takes any tagged literal \( \lambda l \) and any set \( B \) of literals such that \( l \notin B \) and returns a result in \{−1, 0, +1\}. \text{Plaus} takes any tagged literal \( \lambda l \) such that \( \lambda \neq \mu \) and any set \( B \) of literals such that \( l \notin B \) and returns a result in \{−1, 0, +1\}. For and \text{Nullified} have the same domain and range as \text{Plaus}. \text{Disabled} takes any tagged literal \( \lambda l \) such that \( \lambda \neq \mu \) and any set \( B \) of literals such that \( l \notin B \) and returns a result in \{−1, 0, +1\}. \text{Defeated} takes any tagged literal \( \lambda l \) such that \( \lambda \neq \mu \) and any set \( B \) of literals such that \( l \notin B \) and returns a result in \{−1, 0, +1\}. \text{Beatn} and \text{Inappl} have the same domain and range as \text{Defeated}. In the following definition of the “rules” for each function, \( C \) is a set of clauses, \( B \) is a set of literals, \( L \) is a set of literals, \( l \) is a literal, \( \lambda \in \{\mu, \alpha, \pi, \beta, \delta\} \), and we define \( \min\{\} = +1 \), and max\{\} = −1.

\[ P(\lambda l, B) = \min(P(\lambda l, B) : c \in C) \]

∀l (i) If \( VL \) is a tautology then \( P(\forall l, B) = +1 \).
3.2 Compiling DPL

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 Glasgow Haskell Compiler (GHC) is recommended. The Haskell Interpreter, Hugs, is capable of running DPL, albeit more slowly and for smaller theories.

To prepare for compilation on a UNIX-like system, change directory to DPL/src.

$ cd DPL/src

To compile the DPL tool, type:

$ make bin

The executable binary will be saved in DPL/bin you can simply copy it wherever you like. By default:

$ make install

copies the binary to ~/.bin. Edit the definition of symbol INSTALLDIR in the Makefile to change that destination.

3.3 Compiling without make

If you are wishing to compile the DPL tool 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 is the command required to compile DPL.

$ ghc --make -fglasgow-exts -O -fallow-overlapping-instances DPL.lhs -o ../bin/DPL

4 Using DPL

4.1 The DPL tool

The DPL tool is not interactive. Its actions are determined by command line arguments and directives embedded in the input description files. Depending on those, the DPL tool:

- reads a plausible description from a text file;
- prints it;
- parses it;
- prints a regenerated plausible description (so that the parsing may be verified);
- instantiates the full description by removing variables and obviating facts and the rules that use them;
- prints the instantiated, obviated description;
- prints the theory that is derived from that description;
- attempts the proofs requested in the text file with the plausible description;
- prints a summary table of the proof results;
- prints C expressions that summarise the proof results;
- exports the theory as a C data structure; and/or
- exports a Haskell glue module that exports functions requesting proofs.

• directives that define:
  - types (sets of constants that may be bound to variable arguments to atoms);
  - the types of the arguments of specific atoms;
  - default facts that only operate in the absence of an explicit alternative;
  - inputs (additional axioms to asserted, alternately positively and negatively, when performing proofs);
  - combinations of inputs to ignore (that is, to not perform proofs for);
  - outputs (the tagged formulas to attempt proof of for all combinations of inputs); and
  - hints for code generation.

The rest of this section defines the syntax of a description file, working from the lowest levels, up. It end with some example description files.

Whitespace and comments

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

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

```
comment1 ::= "\n" {<any character$ ! "\n">} ("\n" | $end of file$);
level="lexical".
```

```
comment2 ::= "/*" comment2end;
level="lexical".
```

```
comment2end ::= "*/" | $any character$ comment2end;
level="lexical".
```

Names

Atoms and rule labels in a description file are names that start with letters and may include digits and underscores. Names that start with upper and lower case letters are distinguished for different purposes.

```
lName ::= $lower case letter$ {$letter$ | $digit$ | "_"};
level="lexical".
```

```
wName ::= $upper case letter$ {$letter$ | $digit$ | "_"};
level="lexical".
```
Strings
Atoms may also be represented as double-quote-delimited strings. To embed a double-quote in a string without prematurely terminating the string, repeat it.

```
string ::= "\" {"\"\" | <$any character$ ! "\"> } "\"; level="lexical".
```

Symbols
These are the special symbols on the language.

```
symbol ::= separator | operator; level="lexical".
```

Constants and variables as arguments
Constants and variables may optionally appear as arguments to atoms. Constants always start with upper case letters and variables always start with lower case letters.

```
constant ::= uName; level="grammar".
```

```
variable ::= lName; level="grammar".
```

Atoms
Atoms are usually proposition symbols with optional arguments that may be constants or removable variables. They are also (rarely) arbitrary strings. To support orderings in enumerated types, there are special literals formed with the infix operators <, <= and ==.

```
separator ::= ":" | "," | ":" | "," | "[" | "]" | "{" | "}" | "{" | "}" | "(" | ")" | "|" | "."; level="lexical".
```

The complete lexical syntax
A DPL source consists of zero or more of the following kinds of tokens.

```
lexer ::= {comment1 | comment2 | symbol | lName | uName | cardinal | string | whitespace}; level="lexical".
```

November 25, 2013
atomName ::= lName | uName;
level="grammar".

argList ::= "(" argument ("," argument ")";
level="grammar".

atom ::= atomName [argList] | string | specialAtom;
level="grammar".

specialAtom ::= argument ("<" | "<=" | "==") argument;
level="grammar".

Example atoms:
p p(A,B,c)
proposition_13 Proposition14(Const1,Const2,var_1)
"x > 0" "command.equals(\"sit\")"

Note that in Decisive Plausible Logic is only a propositional logic.
Any variables that appear in atoms in a declaration will cause that declaration to be instantiated with all possible combinations of constants drawn from the set of all constants appearing in a description. Use the type system to constrain those combinations to only the meaningful ones.

Literals

Literals are atoms or their negation. A tilde (~) is used for negation (~).

literal ::= atom | "~" literal | "(" literal ");"
level="grammar".

Example literals:
p ~p ~(~p) p(A,B,c) ~p(A,B,c) "x > 3"

Types

A type is a set of constants. Types are used to constrain the instantiation of declarations containing variables.

In general, a type is an expression which forms a set of arguments.

A type may be formed by enumerating comma-separated arguments between braces, by reference to a named type, and by forming the union (+), intersection (\^) and difference (\-) of types.

Example atoms:

Example literals:

Declaring named types

A type’s name always starts with an upper case letter.

newTypeDec ::= "type" typeName "=" type;
level="grammar".

Examples:
type Animal = {Cat, Dog, Human}.
type Vegetable = {Grass, Carrot}.
type Organism = Animal \^ Vegetable \^ {Mushroom}.
type Beast = Animal \- {Human}.

The Universe

The predefined type Universe is the type that contains all known constants.

Asserting atom argument types

Where an atom occurs in a fact or rule, with arguments that are variables, that fact or rule will be multiply instantiated with all combinations of constants appearing anywhere in the description.

To instantiate using only those constants that are deemed appropriate, assert with a declaration of the following form the types of each argument of an atom.
atomTypeDec ::= "type" atomName \\
(" varGen ("," varGen) ");
level="grammar".

varGen ::= variable "<~" type;
level="grammar".

Examples:
type eats(a <- Animal, o <- Organism).
type enemy(a1 <- Animal, a2 <- Animal - {a1}).
type boss(a1 <- Animal, a2 <- Animal, boss <- {a1, a2}).
type likes(a <- Animal, x <- Universe).

Note that variables declared within these declarations may appear in the types of other variables that appear to the right, but never to the left. This rule avoids the possibility of cyclic types.

Obviation and default facts
If a rule has a literal in its antecedent that is asserted as a fact, then its strict provability is assured and it can be omitted from the antecedent. If the negation of the literal in an antecedent has been asserted as a fact, then that rule is useless and can be omitted. This process is termed obviation. To assist obviation, extra facts, not to be used in the construction of a theory may be declared as follows.

default ::= "default" literal;
level="grammar".

default
 literal
default

The following default facts are automatically asserted:
default "(x < y).
default "(x <= y).
default "(x = y).

Literal sets
Sets of literals appear in clauses and as antecedents to rules. A literal set may be formed by enumerating comma-separated literals between braces, by comprehensive specification, and by forming the union (+), intersection (^) and difference (-) of literal sets.

litSet ::= litSetTerm litSetEnd;
level="grammar".

Example literal sets:
{} 
{p, q, r} 
{p} + {q, r} 
{p, q, r} - {q} 
{(p) + {q, r} } - {q, r} 
{see(x)} + {~see(y) | y <- Thing - {x}} 
{see(x)} + {~see(y) | y <- Universe - {x}}

Note that x is a free variable that will be instantiated to all of the constants in the Universe unless constrained by an atom type assertion.

Formulas
A clause is the disjunction (∨ for ⋁) of a set of literals. A literal is also a clause, as ∨{l} = l. ∨{} (for ∨{ }) is the empty clause, which is false.

cnfFormula ::= clause | \ "\" {}

A cnf-formula is the conjunction (∧ for ⋀) of a set of clauses. A clause is a cnf-formula, as ∨{c} = c.

cnfFormula ::= clause | \ "\" {}

Example clauses and cnf-formulas:
\ /\ {} /\ {p} /\ {p} /\ {a, b, ~c} /\ {\ /\ {a, b}, ~c}

Rules
Strict rules (→ for →) do not appear in plausible descriptions. Only plausible rules (⇒ for ⇒) and defeater rules (¬ for ¬) are parsed. A rule has an antecedent set of literals, an arrow, and a consequent literal. Optionally, a rule may be preceded by a label and a colon, so that the rule may be referred to by priority assertions. If the antecedent is a singleton or empty set, the set braces may be omitted.

label ::= lName | uName;
level="grammar".

Example labels and cnf-formulas:
lName p p \ /\ (p) \ /\ (a, b), ~c
uName
antecedent ::= litSet | literal | $\epsilon$;
level="grammar".

rule ::= [label "="] antecedent ("\text{=>}" | "\text{-}\") literal;
level="grammar".

Example rules:

\begin{verbatim}
formal DPL
\{
\} => p
\{ \} => p
\{ a \} \rightarrow \neg b
\{ a \} \rightarrow \neg b
\{ a, b, c \} \rightarrow \neg d
\end{verbatim}

Priorities
A priority assertion asserts that one rule beats another. The rules are identified by their labels.

cardinality ::= label "\text{>}
label;
level="grammar".

Example:

\begin{verbatim}
R1 > R2
\end{verbatim}

Input and ignore specifications
An input specification specifies a literal, that is asserted as an axiom on one set of proofs and then again, negated, in another set of proofs. If multiple literals are specified in a single input specification, they are treated as mutually exclusive.

\begin{verbatim}
someLits ::= \{" literal \{," literal \} "};
level="grammar".
\end{verbatim}

Input:

\begin{verbatim}
input ::= "input" someLits;
level="grammar".
\end{verbatim}

An ignore specification specifies combinations of literals, that would be generated by the input specifications, but should be skipped.

\begin{verbatim}
ignore ::= "ignore" [$\text{cardinal}$ "of"] someLits;
level="grammar".
\end{verbatim}

Examples:

Tagged cnf-formulas
A tagged cnf-formula consists of: a proof level ($\mu$ for $\mu$, $\alpha$ for $\alpha$, $\pi$ for $\pi$, $\beta$ for $\beta$, $\delta$ for $\delta$); and a cnf-formula.

\begin{verbatim}
tag ::= "m" | "a" | "p" | "b" | "d";
level="grammar".
taggedFormula ::= tag cnfFormula;
level="grammar".
\end{verbatim}

Examples:

\begin{verbatim}
d \land \{\} \land p \land p \land \{p\} \land a \land \{\land\{a,b\},\neg c\}
\end{verbatim}

Assertions
As an alternative to letting DPL generate combinations of positively and negatively asserted inputs automatically, they may be specified manually. An assert declaration must assert all of the inputs positively or negatively. There may be multiple assert declarations, each specifying a combination with which to perform proofs.

\begin{verbatim}
assert ::= "assert" someLits;
level="grammar".
\end{verbatim}

Assert declarations will be useful for testing descriptions with very many inputs, leading to too many combinations to perform proofs for. Their use will likely be temporary, it is recommended that they be included using the -a filename option.

Output specifications
An output specification specifies a tagged cnf-formula for which a proof should be attempted for each combination of inputs. Optionally it includes the string to be the name of a C macro to be expanded as the computed C expression. If the C macro string is supplied, then the tagged cnf-formula may not contain variables.

\begin{verbatim}
output ::= "output" \{" taggedFormula \{," string \} "};
level="grammar".
\end{verbatim}
Name declaration

Code generation hints

The DPL tool is sometimes used to generate code in other languages. These declarations are hints to help those processes:

- A description may have a declared name, used to uniquely identify it in contexts where there may be more than one description in play. It might be used as the generated module name or as a part of all generated global definitions.

\[
\text{nameDec ::= "name" "\{ uName \}";}
\]

Example: name{Nanook}

- The following declares that something else needs to be declared as an import to the generated module.

\[
\text{importDec ::= "import" "\{ string \}";}
\]

Example: import{"package.Class"}

- The following declares that some token is assumed to be predefined in the target language.

\[
\text{predefDec ::= "predefined" "\{ string \}";}
\]

Example predefined{"elem"}

Description file

A description file is a sequence of type declarations, clauses, plausible rules, defeater rules, priority assertions, input specifications, ignore specifications, output specifications, and optionally assert declarations for testing, and/or a name declaration and import and predefined token declarations for code generation. Each must be terminated with a period.

\[
\text{statement ::= ( newTypeDec | atomTypeDec | clause | default | rule | priority | input | ignore | assert | output | nameDec | importDec | predefDec ) \".\";}
\]

Example:

```plaintext
/*
 ** file: empty.d
 ** purpose: The empty description file for testing.
*/
```

```plaintext
/*
 ** file: cooler.d
 ** purpose: The air cooler example
*/
```

```plaintext
% tag this description with a name.
name{COOL}.

% Normally don't cool.
R1: => ~cool.
R1 > R2.

% If power is low and not(T > 27C) then don't cool.
R3: {lowPower, ~tempGT27} => ~cool.
R3 > R2.

% If we have only recently stopped cooling
% then don't cool.
R4: justOff => ~cool.
R4 > R2.

% the cases to consider:
input{tempGT25}.
input{tempGT27}.
input{lowPower}.
input{justOff}.
ignore{tempGT27, ~tempGT25}.
% T > 27C and T < 25C

% the requested proofs:
output{m cool}.
output{m ~cool}.
```

November 25, 2013
output{p cool}.
output{p "cool"}.
output{b cool}.
output{b "cool"}.
output{d cool}.
output{d "cool"}.

/*
** file: cooler.a
** purpose: For the air cooler example, an extra
** file containing asserts to test specific
** cases only.
** use: DPL cooler.d -a cooler.a
*/
assert{tempGT25, tempGT27, ~lowPower, ~justOff}.
assert{tempGT25, ~tempGT27, lowPower, ~justOff}.

/*
** file: dogs.d
** purpose: Siberian huskies and other mutts
** (uses variables!)
*/
% Rover is a mutt.
m(R).

% Nanook is a Siberian husky.
sh(N).

% Mutts are dogs (m(x) -> d(x)).
\forall(x)(m(x), d(x)).
% Siberian huskies are dogs (sh(x) -> d(x)).
\forall(x)(sh(x), d(x)).
% Dogs usually bark.
R1: d(x) => b(x).
% Siberian huskies usually do not bark,
% even though they are dogs.
R2: sh(x) => ~b(x).
R2 > R1.
% Siberian huskies are usually huge.
sh(x) => h(x).
% Mutts are not usually huge.
m(x) => ~h(x).
% Most things fear barking dogs.
R3: \{d(y), b(y)\} => f(x, y).
% Most things fear huge dogs.
(d(y), h(y)) => f(x, y).
% Huge dogs don’t usually fear anything.
R4: \{h(x), d(x)\} => ~f(x, y).
R4 > R3.
% the requested proofs:
output{m b(N)}.
output{m "b(N)"}.
output{a b(N)}.
output{a "b(N)"}.
output{p b(N)}.
output{p "b(N)"}.
output{b b(N)}.
output{b "b(N)"}.

/*
** small.d
**
** Small example suitable for testing C expressions.
*/
% Note that no inputs are mutually exclusive and that
% there are no ignore directives.
input{"eye.seesBall()"}.
input{"nose.smellsRat()"}.
input{c}.
input{d}.
R1: "eye.seesBall()" => E.
R2: "nose.smellsRat()" => ~E.
R3: "eye.seesBall()" => F.
R4: "nose.smellsRat()" => ~F.
R3 > R4.
R5: c => G.
R6: E => G.
R7: F => ~G.
R8: d => G.
R8 > R7.
output{m E, "M_E"}.
output{p E, "P_E"}.
output{p F, "P_F"}.
output{p G, "P_G"}.

/*
** file: Janken.d
** purpose: The rules for the Janken (paper, scissors, rock) game.
** This version can be used with DPL to output a
** table of proof results.
*/
% The 3 hand signs.
type Sign = {Paper, Scissors, Rock}.
% The two players.
type Player = {A, B}.

November 25, 2013
% \text{win}(x) \text{ means } x \text{ has won a game.}
\text{type \text{win}(x \leftarrow \text{Player}).}

% \text{shows} \ (x, \ y) \text{ means player } x \text{ is showing sign } y.
\text{type \text{shows}(x \leftarrow \text{Player}, y \leftarrow \text{Sign}).}

% Equality.
\text{default \~eq(x,y).}
\text{eq(x,x).}

% What beats what.
\text{default \~beats(x, y).}
\text{beats(Paper, Rock).}
\text{beats(Rock, Scissors).}
\text{beats(Scissors, Paper).}

% By default, a player has not won.
\text{DefaultWin: } \{} \Rightarrow \sim \text{win}(x).

% By default, there is no tie.
\text{DefaultTie: } \{} \Rightarrow \sim \text{tie}.

% A player wins if the sign she shows beats the sign that
% the other player shows.
\text{Win: \{shows(w, x), shows(y, z), \~eq(w,y), \text{beats(x,z)} \} \Rightarrow \text{win}(w).}
\text{Win > DefaultWin.}

% If noone wins, it is a tie.
\text{Tie: \{\sim \text{win}(x) | x \leftarrow \text{Player}\} \Rightarrow \text{tie}.}
\text{Tie > DefaultTie.}

% The inputs are assertions that each player is showing
% one sign or another, but never 2 at the same time.
\text{input\{shows(A, Paper), shows(A, Scissors), shows(A, Rock)\}.}
\text{input\{shows(B, Paper), shows(B, Scissors), shows(B, Rock)\}.}

% We want to know who has won, or if it is a tie.
\text{output\{p win(A)\}.}
\text{output\{p win(B)\}.}
\text{output\{p tie\}.}

The last example description is indented to be used to generate a
Haskell glue module to invoke proofs. Here are the additional
Haskell modules to make a complete test program.

This module defines the constants refered to in the \text{JankenRules.d}
description file.

\begin{verbatim}
module JankenConstants where

The hand signs.
data Sign = Paper | Scissors | Rock
    deriving (Eq, Enum, Show)

the players.
data Player = A | B
    deriving (Eq, Enum, Show)

\end{verbatim}

This module is a test program that prints a table of results for all
possible Janken games of two players.

\begin{verbatim}
module Main (main) where

import Data.List
import ABR.Check
import ABR.String
import DPL.Descriptions
import JankenConstants
import JankenRules

main :: IO ()
main = do
    cd <- loadDescription "JankenRules.d"
    case cd of
        CheckFail msg -> putStrLn msg
        CheckPass d -> do
            let headings = ["A", "B", "A wins?", "B wins?", "Tie?"]
            results = [
                [show a, show b, show (p_win_A_ shows d),
                show (p_win_B_ shows d),
                show (p_tie shows d)] |
                a <- [Paper .. Rock],
                b <- [Paper .. Rock],
                let shows (x,y) =
                    if x == A then y == a else y == b]
            putStrLn $ makeTableL ' ' $ transpose $ (headings :) $ results
    putStrLn "The program prints:

November 25, 2013"
\end{verbatim}
4.3 Invoking the DPL tool

Type the command:

\[ \text{DPL } \text{<option> } \text{<fileNames>} \]

where the command line arguments include: \text{fileNames} – the names of the description files to process; and \text{options} – as follows:

- \text{-v} – be verbose, show progress information, and proof traces (the default);
- \text{-n} – don’t be verbose;
- \text{-a \text{filename}} – append the contents of the named file to the descriptions before all processing (May be specified multiple times to append multiple files);
- \text{-i} – show the combinations of input axioms with which proofs will be attempted (default if verbose, overrides \text{-v});
- \text{-i} – don’t show the combinations of input axioms with which proofs will be attempted (default if not verbose, overrides \text{-v});
- \text{-p} – do proofs (the default);
- \text{-p} – don’t do proofs (overrides \text{-t} and \text{-c});
- \text{-q} – do proofs quickly using optimised theory data structures (overrides \text{-v});
- \text{-q} – do proofs slowly using simple theory data structures (the default);
- \text{-t} – show the proof result summary table (the default);
- \text{-t} – don’t show the proof result summary table;
- \text{-c} – convert results to C expressions;
- \text{-c} – don’t convert results to C expressions (the default);
- \text{-s} – simplify the C expressions (the default);
- \text{-s} – don’t simplify the C expressions;
- \text{-c} – export the theory as C data structures;
- \text{-c} – don’t export the theory as C data structures (the default).

5 Implementation

This section, on the implementation of Decisive Plausible Logic presents the Haskell modules in a bottom-up sequence. Library modules that are not directly concerned with implementing plausible logic are presented in a separate document [11]. The sources are compatible with Haskell-98, with the exception that support for multi-parameter type classes is required. Haskell code is presented in \text{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.

5.1 DPLLexer

Various elements of the DPL system parse textual representations of atoms, literals, formulas, rules and descriptions, etc. Module \text{DPLLexer} implements the functions for lexical analysis of plausible sources.

module DPL.DPLLexer (lexerL) where
import Data.Char
import ABR.Parser
import ABR.Parser.Lexers

5.1.1 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 next sequence */ and may span more than one line. Formally, the syntax for each type of comment is:

\[
\text{comment1} ::= \text{"%" } (\text{<$any character$} \text{"\n"})\text{ <"*/"} \text{<any character}$ comment2end; level="lexical".
\]

\[
\text{comment2} ::= \text{"/*" } \text{<any character$ comment2end; level="lexical".
\]

\[
\text{comment2end} ::= \text{"*/" | } \text{<any character$ comment2end; level="lexical".
\]

5.1.2 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:

\[
\text{1Name} ::= \text{"$lower case letter$ } (\text{<$letter$} | \text{<"_"}; level="lexical".
\]

\[
\text{1NameL} ::= \text{Lexer}
\]

\[
\text{1NameL} =
\text{(satisfyL isLower "lower-case letter" } \text{<"_"}) \text{ <"*/" (many (satisfyL (\text{'c' } \text{-> True}) "") \text{<"*/"}}
\text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

\[
\text{1NameL} \text{<"*/"}}
\]

November 25, 2013 12
5.1.3 Strings

Atoms may be defined as arbitrary strings.

```
string ::= "\" {"\"" | <$any character$ ! "\"">}" \"";
level="lexical".
```

5.1.4 Symbols and everything else

```
symbol ::= separator | operator;
level="lexical".
```

lexerL performs the lexical analysis of a plausible source. Its definition lists all of the symbols that are special in plausible sources.

```
lexerL ::= {comment1 | comment2 | symbol | lName | uName | cardinal | string | whitespace};
level="lexical".
```
5.2 Constants

Module `Constants` implements constants for Decisive Plausible Logic.

5.2.1 Data type

An `Constant` is a fixed token which may appear as an argument to an atom.

```
newtype Constant = Constant String
  deriving (Eq, Ord)
```

5.2.2 Parsers

A constant is a name that starts with an upper case letter.

```
constant ::= uName;
level="grammar".
```

which is implemented by `constantP`.

```
constantP :: Parser Constant
constantP = tagP "uName"
  @@ (\(\_,\_,n\) -> Constant n)
```

5.2.3 Collecting constants

It is required for various purposes to identify all of the distinct constants that occur in an object. Constants can be collected from instances of class `HasConstants`.

```
class HasConstants a where
  getConstants :: a -> S.Set Constant
```

```
getConstants :: a -> S.Set Constant
getConstants a = not $ S.null $ getConstants a $ S.empty
```

5.2.4 Instance declarations

Showing

instance Show Constant where

```
showsPrec _ (Constant n) = showString n
```

Collecting Constants

instance HasConstants Constant where

```
getConstants = S.insert
```

NFData (for deepseq)

instance NFData Constant where

```
rnf (Constant c) = c 'deepseq' ()
```

5.3 Variables

Module `Variables` implements variables for Decisive Plausible Logic.

5.3.1 Data type

An `Variable` is a token which may appear as an argument to an atom, to be instatiated with constants.

```
newtype Variable = Variable String
  deriving (Eq, Ord)
```

5.3.2 Parsers

A variable is a name that starts with a lower case letter.

```
variable ::= lName;
level="grammar".
```

which is implemented by `variableP`.

```
variableP :: Parser Variable
variableP = tagP "lName"
  @@ (\(\_,\_,n\) -> Variable n)
```

5.3.3 Collecting variables

It is required for various purposes to identify all of the distinct variables that occur in an object. Variables can be collected from instances of class `HasVariables`.

```
class HasVariables a where
  getVariables :: a -> S.Set Variable
```

```
getVariables :: a -> S.Set Variable
getVariables a = not $ S.null $ getVariables a $ S.empty
```

```
5.3.4 Grounding

To ground is to substitute a variable with a constant. A substitution $v :\rightarrow c$ replaces a variable $v$ with a constant $c$. Substitutions may be composed. $s_1 :\rightarrow s_2$ first performs $s_1$ and then $s_2$. $\text{NullSub}$ is the null substitution that does nothing.

data Substitution = NullSub | Variable :-> Constant | Substitution :>> Substitution

deriving (Eq, Ord, Show)

Anything groundable should be an instance of class Groundable.

class Groundable a where

  ground1 :: Variable -> Constant -> a -> a
  ground :: Substitution -> a -> a

5.3.5 Instance declarations

Showing

instance Show Argument where

  showsPrec _ a = case a of
  Const c -> shows c
  Var v -> shows v

Collecting constants

instance HasConstants Argument where

  getConstants a cs = case a of
  Const c -> getConstants c cs
  Var _ -> cs

Collecting variables

instance HasVariables Argument where

  getVariables a vs = case a of
  Const _ -> vs
  Var v -> getVariables v vs

Grounding

instance Groundable Argument where

  ground1 :: Variable -> Constant -> a -> a
  ground :: Substitution -> a -> a

5.4 Arguments

Module Arguments implements arguments for Decisive Plausible Logic.

module DPL.Arguments (Argument(..), argumentP)

where

import Control.DeepSeq
import ABR.Parser
import ABR.Text.Showing
import ABR.Data.List
import DPL.Constants
import DPL.Variables
import DPL.Arguments

5.4.1 Data type

An Argument of an atom may be either:

- a constant (Const);
- a variable (Var).

data Argument = Const Constant | Var Variable

deriving (Eq, Ord)
5.5.1 Data type
An Atom is a proposition symbol, Prop. An atom may have a list of arguments.

```haskell
data Atom = Prop String [Argument] deriving (Eq, Ord)
```

5.5.2 Parsers
Atom names start with letters of either case, and are parsed by atomNameP.

```haskell
atomName ::= lName | uName;
level="grammar".

atomName
  lName
  uName

atomNameP :: Parser String
  atomNameP = (tagP "lName" <|> tagP "uName")
  @> (\(_,n,_) -> n)
```

Argument lists consist of parentheses containing one or more comma separated arguments.

```haskell
argList ::= ("(" argument {"," argument")")
  level="grammar".

argList
  argument
  argument

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

Atoms are names followed optionally by an argument list, or (more rarely) an arbitrary string.

```haskell
atom ::= atomName [argList] | string | specialAtom;
  level="grammar".

atom
  atomName
  argList
  string
  specialAtom

specialAtom ::= argument ("<" | "<=" | "+=") argument;
  level="grammar".

specialAtom
  argument
  argument

which is implemented by atomP.

atomP :: Parser Atom
atomP =
  (argumentP <<<
    (literalP "symbol" "<"
    <|>
    literalP "symbol" "+="
    <|>
    nofail argumentP)
  @> \((a1,(..,a2)) -> Prop [a1,a2])
  <|> optional argListP
  @> \((n,ass) -> case ass of

5.5.3 Collecting atoms
It is required for various purposes to identify all of the distinct atoms that occur in an object. Atoms can be collected from instances of class HasAtoms.

```haskell
class HasAtoms a where
  getAtoms :: a -> S.Set Atom
```

Collecting Atoms

```haskell
instance Show Atom where
  showsPrec _ (Prop "<" [a1,a2]) = shows a1 . showString " < " . shows a2
  showsPrec _ (Prop "<=" [a1,a2]) =
    shows a1 . showString " <= " . shows a2
  showsPrec _ (Prop "==" [a1,a2]) =
    shows a1 . showString " == " . shows a2
  showsPrec _ (Prop n as) = case as of
    [] -> showString n
    [as] -> showString n . showChar '(' . showWithSep "," as . showChar ')'

instance HasAtoms Atom where
  getAtoms = S.insert

Collecting constants

instance HasConstants Atom where
  getConstants (Prop _ as) cs = foldr getConstants cs as

Collecting variables

instance HasVariables Atom where
  getVariables (Prop _ as) vs = foldr getVariables vs as

Grounding

instance Groundable Atom where
  ground1 v c (Prop n as) = Prop n $ map (ground1 v c) as
  rename v v' (Prop n as) = Prop n $ map (rename v v') as

instance NFData (for deepseq) Atom where
  rnf (Prop n as) = n 'deepseq' (as 'deepseq' ())
```

5.5.4 Instance declarations

```haskell
5.6 OAtoms
Module OAtoms implements optimised atoms for Decisive Plausible Logic.

module DPL.OAtoms (OAtom, AtomMap, OAtomMap, mkAtomMaps, toOAtom, toAtom)
```

November 25, 2013
5.6.1 Data types

An OAtom is a positive integer that uniquely identifies a corresponding Atom.

type OAtom = Int

An AtomMap maps Atoms to their corresponding OAtoms.

type AtomMap = M.Map Atom OAtom

An OAtomMap maps OAtoms to their corresponding Atoms.

type OAtomMap = Array Int Atom

5.6.2 Building atom maps

\[ \text{mkAtomMaps} x = (N, am, oam) \]

where \( x \) is something that contains \( N \) distinct Atoms, and for each of them, \( am \) maps them to a corresponding OAtom, and \( oam \) maps them back.

\[ \text{mkAtomMaps} :: \text{HasAtoms}\ a \Rightarrow a \rightarrow (\text{Int}, \text{AtomMap}, \text{OAtomMap}) \]

\[ \text{mkAtomMaps} x = \]

\[ \begin{align*}
& \text{let as} = \text{toList}\ S.\emptyset, \text{getAtoms}\ x\ S.\emptyset \text{length}\ as \n = \text{length}\ as \\
& \text{am} = \text{fromList}\ \$\ \text{zip}\ \text{as} [1..n] \\
& \text{oam} = \text{array}\ (1, n)\ \$\ \text{zip}\ [1..n]\ \text{as} \\
& \text{in}\ (n, am, oam) 
\end{align*} \]

5.6.3 Looking up atoms

\[ \text{toOAtom} \]

\[ \text{am} a \]

returns \( A \)'s corresponding OAtom.

\[ \text{toOAtom} :: \text{AtomMap} \rightarrow \text{Atom} \rightarrow \text{OAtom} \]

\[ \text{toAtom} \]

\[ \text{oam} oa \]

returns \( O \)'s corresponding Atom.

\[ \text{toAtom} :: \text{OAtomMap} \rightarrow \text{OAtom} \rightarrow \text{Atom} \]

\[ \text{toAtom} = (!) \]

5.7 Literals

Module \text{Literals} implements literals for Decisive Plausible Logic.

\{-# LANGUAGE FlexibleInstances #-\}

module DPL.Literals (Literal(..), pLiteralP, Negatable(..)) where

import Control.DeepSeq
import ABR.Parser
import DPL.Constants
import DPL.Variables
import DPL.Atoms

5.7.1 Data type

A Literal is any atom \( a \) (Pos) or its negation \( \neg a \) (Neg).

data Literal = Pos Atom | Neg Atom deriving (Eq)

5.7.2 Parser

The syntax for a literal is:

\[ \text{literal ::= atom | "" literal | "(" literal ")"} \]

which shows that a literal may be multiply negated and embedded in parentheses, and is implemented by \text{pLiteralP}.

5.7.3 Negation

Class Negatable includes types that may be logically negated.

class Negatable a where

\[ \text{neg} \]

\[ a \rightarrow a \]

\[ \text{pos} \]

\[ a \rightarrow \neg a = a \]

5.7.4 Instance declarations

Ordering

instance Ord Literal where

\begin{align*}
& \text{compare}\ l\ l' = \text{case}\ l\ of \\
& \text{Pos}\ a \rightarrow \text{case}\ l'\ of \\
& \text{Pos}\ a' \rightarrow \text{compare}\ a\ a' \\
& \text{Neg}\ _ \rightarrow \text{GT} \\
& \text{Neg}\ a \rightarrow \text{case}\ l'\ of \\
& \text{Pos}\ _ \rightarrow \text{LT} \\
& \text{Neg}\ a' \rightarrow \text{compare}\ a'\ a
\end{align*} \]

Showing

instance Show Literal where

\begin{align*}
& \text{showsPrec}\ p\ l = \text{case}\ l\ of \\
& \text{Pos}\ a \rightarrow \text{shows}\ a \\
& \text{Neg}\ a \rightarrow \text{showChar}\ '"' \cdot \text{shows}\ a
\end{align*} \]

Negation

instance Negatable Literal where

\begin{align*}
& \text{neg}\ l = \text{case}\ l\ of \\
& \text{Pos}\ a \rightarrow \text{Neg}\ a \\
& \text{Neg}\ a \rightarrow \text{Pos}\ a
\end{align*} \]

instance HasAtoms [Literal] where

\begin{align*}
& \text{neg} = \text{map}\ \text{neg} \\
& \text{pos} = \text{map}\ \text{pos}
\end{align*} \]

Collecting Atoms

instance HasAtoms Literal where

\begin{align*}
& \text{getAtoms}\ l = \text{case}\ l\ of \\
& \text{Pos}\ a \rightarrow \text{getAtoms}\ a \\
& \text{Neg}\ a \rightarrow \text{getAtoms}\ a
\end{align*} \]

Collecting constants

instance HasConstants [Literal] where

\begin{align*}
& \text{getConstants}\ c s = \text{case}\ c\ of \\
& \text{Pos}\ a \rightarrow \text{getConstants}\ a\ c \\
& \text{Neg}\ a \rightarrow \text{getConstants}\ a\ c
\end{align*} \]

Collecting variables

instance HasVariables Literal where
5.8 OLiterals

Module **OLiterals** implements optimised literals for Decisive Plausible Logic.

{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}

module DPL.OLiterals (OLiteral, toOLiteral, toLiteral) where

import DPL.Atoms
import DPL.OAtoms
import DPL.Literals

5.8.1 Data types

An **OLiteral** is a positive or negative integer (never 0) that uniquely identifies a corresponding **Literal**.

type OLiteral = Int

5.8.2 Looking up literals

toOLiteral :: AtomMap -> Literal -> OLiteral
toOLiteral = case l of
  Pos a -> toOAtom am a
  Neg a -> negate $ toOAtom am a

toLiteral :: OAtomMap -> OLiteral -> Literal
toLiteral = case ol of
  Pos ol -> Pos $ toAtom oam ol
  Neg ol -> Neg $ toAtom oam ol

5.8.3 Instance declarations

Negation

instance Negatable OLiteral where
  neg = negate
  pos = abs

instance Negatable [OLiteral] where
  neg = map neg
  pos = map pos

5.9 Types

Module **Types** implements types for Decisive Plausible Logic.

module DPL.Types (TypeName(..), Type(..), TypeTable, typeNameP, typeP, evalTypes, evalType, evalType', HasTypes(..) ) where

import Data.List
import qualified Data.Map.Strict as M
import qualified Data.Set as S
import ABR.Parser
import ABR.Text.Showing
import DPL.Constants
import DPL.Variables
import DPL.Arguments

5.9.1 Data types

A **TypeName** is a string.

newtype TypeName = TypeName String
  deriving (Eq, Ord)

A **Type** is either

- an enumerated set of arguments (**TEnum**);
- a named type (**TName**);
- the union of two types (**:+**);
- the intersection of two types (**:^**); or
- the difference between two types (**:-**)

data Type = TEnum (S.Set Argument)
  | TOrd [Constant]
  | TName TypeName
  | Type :+: Type
  | Type :^ Type
  | Type :- Type

  deriving (Eq, Ord)

A **TypeTable** is a mapping from named types to their values.

type TypeTable = M.Map TypeName Type

5.9.2 Parsers

A type’s name must start with an upper case letter and are parsed by **typeNameP**.

typeName ::= uName;

level="grammar".

typeName

typeName :: Parser TypeName
  = tagP "uName" @> (\(_,n,_) -> TypeName n)

Types are parsed by **typeP**.

type ::= typeTerm typeEnd;

level="grammar".

type
typeEnd
typeTerm
typeP :: Parser Type
  = typeTermP <*> typeEndP @> (\(t,f) -> f t)

typeEnd ::= :+: typeTerm typeEnd
  | ^ typeTerm typeEnd
  | -: typeTerm typeEnd
  | $epsilon$;
  level="grammar".

November 25, 2013 18
5.9.3 Evaluating the named types

A description will contain zero or more declarations of named types. These types may be defined in terms of each other. Before anything else happens these named types must be fully evaluated.

```haskell
evalTypes :: [(TypeName, Type)] -> Either TypeTable String
```
evalTypes ns = let ns' = map fst ns
dups = [n | (n:ns') <- tails ns, n 'elem' ns']
     tt = M.fromList ns'
in if not (null dups) then Right $ unlines $ map (
     -> "Type \(\text{a} \) is multiply defined.") dups
     else eval ns tt
     where
eval ns tt = case ns of
     [] -> Left tt
     (n:ns') -> case evaln n tt [] of
      Left tt' -> eval tt' ns' tt'
      Right msg -> Right msg
evaln :: TypeName -> TypeTable -> Either TypeTable String
     evaln n tt ns | n 'elem' ns = Right $ "Definition of type " ++ show n ++ " is cyclic."
     | otherwise = case M.lookup n tt of
      Nothing -> Right $ "Type " ++ show n ++ " is used but not declared."
      Just t -> case evalt t tt (n:ns) of
       Left t' -> Left tt' ++ show t ++ " in the \(\text{a} \) declaration of a named type contains \(\text{a} \) variables."
      Right msg -> Right msg
evalt :: Type -> TypeTable -> Either (Type, TypeTable) String
     evalt t tt ns = case t of
      TEnum as ->
       let as' = S.toList as
        cs = [Const c | Const c <- as']
        vs = [Var v | Var v <- as']
in if not (null vs) then Right $ "Type " ++ show t ++ " is used but not declared."
        Just t -> case evalt t tt (n:ns) of
         Left t' -> case evalt t tt (n:ns) of
          Left t'' -> Left tt'' ++ show t ++ " \(\text{a} \) in previous type declaration of \(\text{a} \) contains \(\text{a} \) variables."
          Right msg -> Right msg
        Right msg -> Right msg
      TOrd cs ->
       let cs' = [Var v | Var v <- as']
in if not (null cs) then Right $ "Type \(\text{a} \) is used but not declared."
       Just t -> case evalt t tt (n:ns) of
        Left t' -> case evalt t tt (n:ns) of
         Left t'' -> Left tt'' ++ show t ++ " \(\text{a} \) in previous type declaration of \(\text{a} \) contains \(\text{a} \) variables."
         Right msg -> Right msg
       Right msg -> Right msg
      TName n ->
       let nn = case evaln n tt ns of
        Nothing -> Nothing
        Just t -> TEnum as
         Left tt' -> Left tt' ++ show t ++ " \(\text{a} \) in previous type declaration of \(\text{a} \) contains \(\text{a} \) variables."
        Right msg -> Right msg
eval :: Type -> Either TypeTable String
     eval :: Type -> Either TypeTable String
```
5.9.4 Evaluating a type

At the time an object is instantiated, it should be possible to fully evaluate its type. `evalType tt t` returns the simplified type `t`, where `tt` is the table of named types.

```haskell
evalType :: TypeTable -> Type -> Type
evalType tt t = case t of
  TEnum cs -> TEnum cs
  TOrd cs -> TEnum $ S.fromList $ map Const cs
  TName n -> case M.lookup n tt of
    Nothing -> error $ "evalType: Type " ++ show n ++ " is undefined."
    Just t' -> evalType tt t'
  t1 :+: t2 -> let TEnum t1' = evalType tt t1
                TEnum t2' = evalType tt t2
                in TEnum $ S.union t1' t2'
  t1 :^: t2 -> let TEnum t1' = evalType tt t1
                 TEnum t2' = evalType tt t2
                 in TEnum $ S.intersection t1' t2'
  t1 :-: t2 -> let TEnum t1' = evalType tt t1
                TEnum t2' = evalType tt t2
                in TEnum $ S.difference t1' t2'

evalType' :: TypeTable -> Type -> [Constant]
evalType' tt t = let TEnum as = evalType tt t
                 cs = [c | Const c <- S.toList as]
                 vs = [v | Var v <- S.toList as]
                 in if null vs then cs
                    else error "evalType': variables left in type."
```

5.9.5 Collecting Orderings

Class `HasTypes` overloads things do with data structures that contain Types.

```haskell
class HasTypes a where
  getOrdered os x adds any orderings in x to os.

getOrdered :: [[Constant]] -> a -> [[Constant]]
getOrdered os _ = os
```

5.9.6 Instance declarations

Showing

```haskell
instance Show TypeName where
    showsPrec _ (TypeName n) = showString n
instance Show Type where
    showsPrec _ t = case t of
        TEnum as -> showChar '{' .
        showWithSep ", " (S.toList as) . showChar '}'
        TOrd cs -> showChar '[' .
        showWithSep ", " (cs . showChar ']' )
        TName n -> shows n
        t1 :+: t2 -> showChar '+' .
        showWithSep ", " ( shows t1 . showChar ' ')
        showWithSep ", " ( shows t2 . showChar ' ')
        t1 :^: t2 -> showChar ':' .
        showWithSep ", " ( shows t1 . showChar ' ')
        showWithSep ", " ( shows t2 . showChar ' ')
        t1 :-: t2 -> showChar '-' .
        showWithSep ", " ( shows t1 . showChar ' ')
        showWithSep ", " ( shows t2 . showChar ' ')

getConstants t cs = case t of
    TEnum as -> foldr getConstants cs $ S.toList as
    TOrd cs -> foldr getConstants cs as
    TName n -> cs
    t1 :+: t2 -> getConstants t2 $ getConstants t1 cs
    t1 :^: t2 -> getConstants t2 $ getConstants t1 cs
    t1 :-: t2 -> getConstants t2 $ getConstants t1 cs

getOrderings os t = case t of
    TEnum as -> os
    TOrd cs -> cs ++ os
    TName n -> os
    t1 :+: t2 -> getOrderings (getOrderings os t1) t2
    t1 :^: t2 -> getOrderings (getOrderings os t1) t2
    t1 :-: t2 -> getOrderings (getOrderings os t1) t2
```

5.10 TypeDecs

Module `TypeDecs` implements type declarations for Decisive Plausible Logic.

```haskell
module DPL.TypeDecs ( NewTypeDec(..), AtomTypeDec(..), VarGen(..),
                      Default(..), newTypeDecP, atomTypeDecP, varGenP,
                      defaultP ) where
import ABR.Parser
import ABR.Text.Showing
import DPL.Constants
import DPL.Variables
import DPL.Types
import DPL.Literals
```

5.10.1 Data types

A `NewTypeDec` binds a type name to a type with `:=`.

```haskell
data NewTypeDec = TypeName := Type
  deriving (Eq, Ord)
```
An `AtomTypeDec` asserts the type of every argument to an atom

```haskell
data AtomTypeDec = AtomTypeDec String [VarGen]
  deriving (Eq, Ord)
```

where a `VarGen` binds a Variable to its type with `:=-`.

```haskell
data VarGen = Variable :<- Type
  deriving (Eq, Ord)
```

A `Default` fact is not really a type declaration, but its purpose is purely to control instantiation, so this is as good a place to define it as anywhere. A default fact declaration is a literal to be instantiated to obviate rules.

```haskell
newtype Default = Default Literal
  deriving (Eq, Ord)
```

### 5.10.2 Parsers

New type declarations are parsed by `newTypeDecP`.

```haskell
newTypeDec ::= "type" typeName "=" type;
newTypeDec

newTypeDecP :: Parser NewTypeDec
newTypeDecP =
  literalP "lName" "type"
<*> typeNameP
<*> literalP "symbol" "="
<*> typeP
@> uncurry (:=)
```

Atom type declarations are parsed by `atomTypeDecP`.

```haskell
atomTypeDec ::= "type" atomName \n  "(" varGen {"," varGen} ");"

atomTypeDec

atomTypeDecP :: Parser AtomTypeDec
atomTypeDecP =
  literalP "lName" "type"
<*> atomNameP
<*> literalP "symbol" "="
<*> varGenP
<*> many (literalP "symbol" "," 
  @> nofail' "variable generator expected" varGenP)
<*> nofail" "expected" (literalP "symbol" ")")
@> ((n,(vg,vgs)) -> AtomTypeDec n (vg : vgs))
```

Variable generators are parsed by `varGenP`.

```haskell
varGen ::= variable "<-" type;

varGen

varGenP :: Parser VarGen
varGenP = variableP
<*> literalP "symbol" "<-"
<*> typeP
@> uncurry (:-)
```

Default facts are parsed by `defaultP`.

```haskell
default ::= "default" literal;

default

defaultP :: Parser Default
defaultP =
  literalP "lName" "default"
  @> pLiteralP
  @> Default
```

### 5.10.3 Instance declarations

#### Showing

```haskell
instance Show NewTypeDec where
  showsPrec _ (n := t) = showString "type " . shows n .
  showString " = " . shows t

instance Show AtomTypeDec where
  showsPrec _ (AtomTypeDec n vgs) = showString "type " .
  showString n . showChar '(' . showWithSep "," vgs .
  showChar ')'

instance Show VarGen where
  showsPrec _ (v :<- t) = shows v . showString " <- " .
  shows t

instance Show Default where
  showsPrec _ (Default l) = showString "default " .
  show l
```

#### Collecting constants

```haskell
instance HasConstants NewTypeDec where
  getConstants (_ := t) cs = getConstants t cs

instance HasConstants AtomTypeDec where
  getConstants (AtomTypeDec _ vgs) cs =
    foldr getConstants cs vgs

instance HasConstants VarGen where
  getConstants (v :<- t) cs = getConstants t cs

instance HasConstants Default where
  getConstants (Default l) cs = getConstants l cs
```

#### Collecting variables

```haskell
instance HasVariables NewTypeDec where
  getVariables (_ := t) vs = getVariables t vs

instance HasVariables AtomTypeDec where
  getVariables (AtomTypeDec _ vgs) vs =
    foldr getVariables vs vgs

instance HasVariables VarGen where
  getVariables (v :<- t) vs = getVariables v $
    getVariables t vs

instance HasVariables Default where
  getVariables (Default l) vs = getVariables l vs
```

#### Grounding

```haskell
instance Groundable VarGen where
  ground1 v c (v' :<- t) = v' :<- ground1 v c t
  rename v v' (v' :<- t) | v' = v => v' :<- rename v v' t
  otherwise = v' :<- rename v v' t

instance Groundable Default where
  ground1 v c (Default l) = Default (ground1 v c l)
  rename v v' (Default l) = Default (rename v v' l)
```

November 25, 2013 21
Collecting Orderings

instance HasTypes NewTypeDec where
  getOrderings os (_ := t) = getOrderings os t
instance HasTypes VarGen where
  getOrderings os (_ ::<- t) = getOrderings os t
instance HasTypes AtomTypeDec where
  getOrderings os (AtomTypeDec _ vgs) = foldl1 getOrderings os vgs

5.11 LitSets

Module `LitSets` implements sets of and literals for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module DPL.LitSets (LitSet(..), litSetP, mkLitSet, HasLitSets(..)) where
import qualified Data.Set as S
import ABR.Text.Showing
import ABR.Parser
import ABR.Control.Check
import DPL.Constants
import DPL.Variables
import DPL.Atoms
import DPL.Literals
import DPL.Types
import DPL.TypeDecs

5.11.1 Data types

A `LitSet` is either

• an enumerated set of literals (`LEnum`);
• the union of two sets of literals (`::+`);
• the intersection between two types (`::^`);
• the difference between two types (`::-`); or
• a comprehensive specification (`LComp`).

data LitSet = LEnum (S.Set Literal)
  | LitSet ::+ LitSet
  | LitSet ::^ LitSet
  | LitSet ::- LitSet
  | LComp Literal [VarGen]
  deriving (Eq, Ord)

5.11.2 Parsers

Types are parsed by `litSetP`.

litSet ::= litSetTerm litSetEnd;
level="grammar".

litSetTerm

| "(" litSet ")"
| 
| "{" [literal {"," literal }] "}"
| 
| "{" literal |
| varGen {"," varGen} 

| "(" litSet ")"
| "{" [literal {"," literal }] 
| 
| "{" literal |
| varGen {"," varGen} 

| literal |
| varGen

litSetTermP :: Parser LitSet
litSetTermP =
  literalP "symbol" "(" 
  <$> nofail' "literal set expected" litSetTermP 
  <$> nofail (literalAP "symbol" ")")
  <|> literalP "symbol" "|
  <$> pLiteralP
  <$> literalAP "symbol" |
  <$> nofail1 "variable generator expected" varGenP
  <$> many ( 
    literalAP "symbol" |
    <$> nofail1 "variable generator expected" varGenP)
  <$> nofail (literalAP "symbol" ")")
  <$> (\(l,(vg,vgs)) -> 
    let vgs' = vg : vgs
    f = foldl1 (.) [rename (Variable v) (Variable ("_VG_" ++ v)) 
      | Variable v :- _ <- vgs']
    f' = foldl1 (.) [rename (Variable v) (Variable ("_VG_" ++ v)) 
      | Variable v :- _ <- vgs']
    in LComp (f l) (map f' vgs') 
  )
  <$> nofail (literalAP "symbol" ")")
  <$> (\(lss) -> LEnum (case lss of 
    [] -> S.empty 
    [l] -> S.fromList (l : l))

5.11.3 Methods

mkLitSet l makes a singleton literal set from a literal l.
mkLitSet :: Literal -> LitSet
mkLitSet = LEnum . S.singleton

5.11.4 Flattening

Class HasLitSets includes all those types which contain literal sets that must be flattened. The class is parameterized over both the unflattened and flattened types.

class HasLitSets a b where
  flatten :: TypeTable -> Check a b String
  flatten' tt x flattens x or fails with a fatal error.
  tt is the table of named types.
  flatten' :: TypeTable -> a -> b
  flatten' tt x = case flatten tt x of
  CheckPass x' -> x'
  CheckFail msg -> error msg

5.11.5 Instance declarations

Showing

instance Show LitSet where
  showsPrec _ ls = case ls of
    LEnum ls' -> showChar '{' .
     showWithSep " , " (S.toList ls') . showChar '}'
    ls1 ::+ ls2 -> showChar '(' . shows ls1 .
                 showString " + " . shows ls2 . showChar ')' .
    ls1 ::^ ls2 -> showChar '(' . shows ls1 .
                 showString " ^ " . shows ls2 . showChar ')' .
    ls1 ::- ls2 -> showChar '(' . shows ls1 .
                 showString " - " . shows ls2 . showChar ')' .
    LComp l vgs -> showChar '{' . shows l .
                 showString " | " . showWithSep " , " vgs .
                 showChar '}'

Collecting constants

instance HasConstants LitSet where
  getConstants ls cs = case ls of
    LEnum ls' -> foldr getConstants cs $ S.toList ls'
    ls1 ::+ ls2 -> getConstants ls1 $ getConstants ls2 cs
    ls1 ::^ ls2 -> getConstants ls1 $ getConstants ls2 cs
    ls1 ::- ls2 -> getConstants ls1 $ getConstants ls2 cs
    LComp l vgs -> getConstants l as
    foldr getConstants as cs vgs

Collecting variables

instance HasVariables LitSet where
  getVariables ls vs = case ls of
    LEnum ls' -> foldr getVariables vs $ S.toList ls'
    ls1 ::+ ls2 -> getVariables ls1 $ getVariables ls2 vs
    ls1 ::^ ls2 -> getVariables ls1 $ getVariables ls2 vs
    ls1 ::- ls2 -> getVariables ls1 $ getVariables ls2 vs
    LComp l vgs -> getVariables l $ foldr getVariables vs vgs

Collecting Atoms

instance HasAtoms LitSet where
  getAtoms ls as = case ls of
    LEnum ls' -> foldr getAtoms as $ S.toList ls'
    ls1 ::+ ls2 -> getAtoms ls1 $ getAtoms ls2 as
    ls1 ::^ ls2 -> getAtoms ls1 $ getAtoms ls2 as
    ls1 ::- ls2 -> getAtoms ls1 $ getAtoms ls2 as
    LComp l vgs -> getAtoms l as

Grounding

Precondition: The variable being grounded must be relatively free.

instance Groundable LitSet where
  ground1 v c ls = case ls of
    LEnum ls' -> LEnum $ S.fromList $ map (ground1 v c) $ S.toList ls'
    ls1 ::+ ls2 -> ground1 v c ls1 ::+ ground1 v c ls2
    ls1 ::^ ls2 -> ground1 v c ls1 ::^ ground1 v c ls2
    ls1 ::- ls2 -> ground1 v c ls1 ::- ground1 v c ls2
    LComp l vgs ->
     LComp (ground1 v c l) (ground1 v c vgs)

renames v' v ls = case ls of
  LEnum ls' -> LEnum $ S.fromList $ map (rename v' v) $ S.toList ls'
  ls1 ::+ ls2 -> rename v' ls1 ::+ rename v' ls2
  ls1 ::^ ls2 -> rename v' ls1 ::^ rename v' ls2
  ls1 ::- ls2 -> rename v' ls1 ::- rename v' ls2
  LComp l vgs -> LComp (rename v' v) vgs

5.12 Formulas

Module Formulas implements clauses and formulas for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module DPL.Formulas (Clause(..), CnfFormula(..), clauseP, cnfFormulaP,
                      MaybeTautology(..), res, rsn) where

November 25, 2013

23
5.12.1 Data types

A **Clause** is the disjunction of a set of literals, expressed either:

- comprehensively (**Clause**); or
- as a list ([**Or**](#)).

```haskell
data Clause = Clause LitSet
  | Or [Literal]
deriving (Eq, Ord)
```

A **CnfFormula** is the conjunction (CNF) of a set of clauses.

```haskell
newtype CnfFormula = CNF [[Literal]]
```

5.12.2 Parsers

Clauses are based on sets of literals, they are initially parsed as comprehension, by **clauseP**, but ultimately must be transformed to lists.

```haskell
parse clause
  as literal | "\\" litSet;
level=grammar.
```

```haskell
clauseP :: Parser Clause
clauseP =
  pLiteralP @> Clause . mkLitSet
  <|> literalP "symbol" "\\"
  *> nofail' "clause set expected" clauseSetP
  @> CNF . map ((\(Or ls) -> ls) . flatten' M.empty)
```

**cnfFormulaP** parses cnf-formulas.

```haskell
cnfFormulaP :: Parser CnfFormula
cnfFormulaP =
  clauseP
  @> (\(Or ls) -> CNF [ls]) . flatten' M.empty
  <|> literalP "symbol" "\\"
  *> nofail' "clause set expected" clauseSetP
  @> CNF . map ((\(Or ls) -> ls) . flatten' M.empty)
```

5.12.3 Tautologies

A **MaybeTautology a** where

```haskell
class MaybeTautology a where
  isTautology :: a -> Bool
```

5.12.4 Resolution

**resolve c d** returns a clause that is a resolvent of clauses **c** and **d**, or Nothing. Precondition: **c** and **d** must be in strictly ascending order; e.g. **Or[~e,~b,a,c]**. Postcondition: If there is a resolvent, it is returned with the same ordering.

```haskell
resolve :: Clause -> Clause -> Maybe Clause
resolve (Or cs) (Or ds) = resolve' cs (reverse ds) [] []
  where
    resolve', resolve'' :: [Literal] -> [Literal] ->
      Maybe Clause
    resolve' [] [] = Nothing
    resolve' [] _ = Nothing
    resolve' (q:qs) (r:rs) qs' rs' = case q of
      Pos qa -> case r of
        Pos _ -> resolve' (q:qs) rs qs' (r:rs')
        Neg ra -> if qa == ra then
          resolve'' qs rs qs' rs'
          else if qa < ra then
            resolve' qs (r:qs) (q:qs') rs'
            else
            resolve (q:qs) rs q' (r:rs')
      Neg qa -> case r of
        Pos ra -> if qa == ra then
          resolve'' qs rs qs' rs'
          else if qa < ra then
            resolve' qs (r:qs) (q:qs') rs'
            else
            resolve'' qs rs q' (r:rs')
      Neg _ ->
        resolve' qs (r:qs) (q:qs') rs'
        resolve'' (q:qs) (r:rs) qs' (r:rs')
        Just $ Or $ snub (reverse qs' ++ qs) rs'

    resolve'' [] rs qs' rs' =
      Just $ Or $ snub (reverse qs') (reverse rs ++ rs')
      resolve'' (q:qs) (r:rs) qs' rs' = case q of
        Pos qa -> if qa == ra then
          Nothing
          else if qa < ra then
            resolve'' qs (r:rs) (q:qs') rs'
            else
            resolve (q:qs) rs q' (r:rs')
        Neg _ ->
          resolve' qs (r:qs) (q:qs') rs'
          resolve'' (q:qs) (r:rs) qs' (r:rs')
          Just $ Or $ snub (reverse qs' ++ qs) rs'
```

**res** returns **Res(S)**. Precondition: **S** and all elements of **S** are in strictly ascending order.

```haskell
res :: [Clause] -> [Clause]
res s =
```
let res' = snub \$ catMaybes [resolve c d | c <- s, d <- s]
res'' = res' \ s
res''' = snub \$ s ++ res''
in if null res'' then
  res'''
else
  res res'''

rsn S returns Rsn(S). Precondition: S and all elements of S are
in strictly ascending order.
rsn :: [Clause] -> [Clause]
rsn = (\ (Or [])) . res

5.12.5 Instance declarations

Showing

instance Show Clause where
  showsPrec p c = case c of
    Clause ls -> showString "/\" . shows ls
    Or ls -> case ls of
      [l] -> show l
      ls -> showString "/\{" .
        showWithSep "," ls . showChar ‘}’

instance Show CnfFormula where
  showsPrec p (CNF cs) = case cs of
    [ls] -> shows (Or ls)
    cs -> showString "/\{" .
      showWithSep "," (map Or cs) . showChar ‘}’

Collecting Atoms

instance HasAtoms Clause where
  getAtoms (Or ls) as = foldr getAtoms as ls
instance HasAtoms CnfFormula where
  getAtoms (CNF lss) as =
    foldr getAtoms as (concat lss)

Collecting constants

instance HasConstants Clause where
  getConstants c cs = case c of
    Clause ls -> getConstants ls cs
    Or ls -> foldr getConstants cs ls
instance HasConstants CnfFormula where
  getConstants (CNF lss) cs =
    foldl (foldr getConstants) cs lss

Collecting variables

instance HasVariables Clause where
  getVariables c vs = case c of
    Clause ls -> getVariables ls vs
    Or ls -> foldr getVariables vs ls
instance HasVariables CnfFormula where
  getVariables (CNF lss) vs =
    foldl (foldr getVariables) vs lss

Grounding

instance Groundable Clause where
  ground1 v c cl = case cl of
    Clause ls -> (CNF lss) =
      CNF (map (map (ground1 v c)) lss)
    rename v v' (CNF lss) =
      CNF (map (map (rename v v')) lss)

Flattening

instance HasLitSets Clause Clause where
  flatten tt c = case c of
    Clause ls -> case flatten tt ls of
      CheckPass ls' -> CheckPass $ Or ls'
      CheckFail msg -> CheckFail msg
    Or ls' -> CheckPass $ Or ls'

Tautologies

instance MaybeTautology Clause where
  isTautology (Or ls) =
    or [neg l == l' | l : ls' <- tails ls, l' <- ls’]
instance (Eq a, Negatable a) => MaybeTautology [a] where
  isTautology ls =
    or [neg l == l' | l : ls' <- tails ls, l' <- ls’]

Collecting Orderings

instance HasTypes Clause where
  getOrderings os c = case c of
    Clause ls -> getOrderings os ls
    _ -> os

5.13 Priorities

Module Priorities implements priorities for Decisive Plausible
Logic.

module DPL.Priorities (Label, Priority(..), labelP, priorityP)
  where
import ABR.Parser

infix 4 :>

5.13.2 Parser

label ::= lName | uName;
level="grammar".

5.13.2 Parser

labelP parses labels.

label :: Parser Label

labelP :: Parser Label
labelP =
  (tagP "lName" <|> tagP "uName")
  *> (\(.,_,,_n,_) -> n)
  *> (priorityP parses priorities.

priority :: label ">" label;
level="grammar".

November 25, 2013 25
5.13.3 Instance declarations

Showing

instance Show Priority where
  showsPrec p (l :> a) =  
  showString l . showString " > " . showString a
5.14 Rules

Module Rules implements rules for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, TypeSynonymInstances #-}
module DPL.Rules (PRule(..), Rule, ruleP, IsRule(..)) where
  import qualified Data.Set as S
  import ABR.Parser
  import ABR.Text.Showing
  import ABR.Control.Check
  import DPL.Constants
  import DPL.Variables
  import DPL.Literals
  import DPL.LitSets
  import DPL.Formulas
  import DPL.Priorities
  import DPL.Types

5.14.1 Data type definitions

A PRule is either a Strict, Plausible or a Defeater. Each has an antecedent set of literals (rant) and a consequent literal (rcon). Plausible and Defeater rules have a string label (rlbl). Rules are initially parsed with the LitSet representation of antecedents, but must be transformed to the list representation.

data PRule l = Strict {rcon :: l, rant :: Either LitSet [l]}
          | Plaus {rlbl :: Label, rcon :: l, rant :: Either LitSet [l]}
          | Defeat {rlbl :: Label, rcon :: l, rant :: Either LitSet [l]}
  deriving (Eq, Ord)

Rule is the preferred shorthand.

type Rule = PRule Literal

5.14.2 Parsers

The syntax for a rule is:

antecedent ::= litSet | literal | $epsilon$

rule ::= [label ::] antecedent ("=>" | "-\") literal;

which is implemented by ruleP.

5.14.3 Interrogating rules

Class IsRule provides the properties of rules required for proofs.

class IsRule r l where
  ant, negant :: r l -> [[l]]

Precondition: The rule’s antecedent has been flattened.

ant, negant :: r l -> [[l]]

5.14.4 Instance declarations

Showing

instance Show Rule where
  showsPrec p rule = case rule of
    Strict c a -> showA a . showString " -> " . shows c
    Plaus l c a -> (if null l then id
                   else showString l . showString ": ") .
                   showA a . showString " => " . shows c
    Defeat l c a -> (if null l then id
                     else showString l . showString ": ") .
                     showA a . showString " -\ " . shows c
               where
                  showA a = case a of
                    Left ls -> shows ls
                    Right 1s -> showChar '(' .
                                  showWithSep ", " ls . showChar ')'

Collecting constants

instance HasConstants Rule where
  getConstants r cs =
    let cs1 = getConstants (rcon r) cs
        cs2 = case rant r of
                   Left 1s -> getConstants 1s cs1
                   Right 1s -> foldr getConstants 1s 1s
    in cs2

Collecting variables

instance HasVariables Rule where
  getVariables r vs =
    let vs1 = getVariables (rcon r) vs
        vs2 = case rant r of
                   Left 1s -> getVariables 1s vs1
                   Right 1s -> foldr getVariables 1s 1s
    in vs2
Collecting Atoms

instance HasAtoms Rule where
  getInstanceAtoms r as =
    let as1 = getInstanceAtoms (rcon r) as
    as2 = case rant r of
      Left ls -> Left $ getInstanceAtoms ls as1
      Right ls -> foldr getInstanceAtoms as1 ls
    in as2

Grounding

instance Groundable Rule where
  ground1 v c r = r {
    rcon = ground1 v c $ rcon r,
    rant = case rant r of
      Left ls -> Left $ ground1 v c ls
      Right ls -> Right $ map (ground1 v c) ls
  }

rename v v' r = r {
  rcon = rename v v' $ rcon r,
  rant = case rant r of
    Left ls -> Left $ rename v v' ls
    Right ls -> Right $ map (rename v v') ls
}

Flattening

instance HasLitSets Rule Rule where
  flatten tt r = case rant r of
    Left ls -> case flatten tt ls of
      CheckPass ls' -> CheckPass r {rant = Right ls' }
      CheckFail msg -> CheckFail msg
    Right ls -> CheckPass r

Interrogating rules

instance IsRule PRule Literal where
  ant r = map (::[]) (case rant r of
    Right ls -> error $ "ant: rule not flattened\n     ": " ++ show r
  )

negant r = [neg (case rant r of
    Right ls -> ls
    _   -> error $ "negant: rule not flattened\n      \ned: " ++ show r
  )]

Collecting Orderings

instance HasTypes (PRule l) where
  getOrderings os r = case rant r of
    Left ls -> getOrderings os ls
    Right _ -> os

5.15 Tags

Module Tags implements tagged cnf-formulas for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
module DPL.Tags (Tag(..), TaggedCnfFormula(..), taggedFormulaP ) where
import ABR.Parser hiding (Tag)
import DPL.Constants
import DPL.Variables
import DPL.Atoms
import DPL.Literals
import DPL.Formulas
import DPL.OLiterals

5.15.1 Data type definitions

A Tag is one of Mu (µ), Alpha (α), Pi (π), Beta (β), and Delta (δ).

data Tag = Mu | Alpha | Pi | Beta | Delta
  deriving (Eq, Ord)

A TaggedCnfFormula is the target for proofs. Constructor Tag associates a tag and a cnf-formula.

data TaggedCnfFormula l = Tag Tag (P)[l]
  deriving (Eq, Ord)

5.15.2 Parsers

pTagP parses a tag.

tag ::= "m" | "a" | "p" | "b" | "d";
level="grammar".

pTagP :: Parser Tag
pTagP =
  literalP "lName" "m" #> Mu
<|> literalP "lName" "a" #> Alpha
<|> literalP "lName" "p" #> Pi
<|> literalP "lName" "b" #> Beta
<|> literalP "lName" "d" #> Delta

(taggedFormulaP :: Parser (TaggedCnfFormula Literal)
  taggedFormulaP =
    pTagP
<*> nofail' "cnf-formula expected" cnfFormulaP
  @> (t, CNF lss) -> Tag t lss)

5.15.3 Instance declarations

Showing

instance Show Tag where
  showsPrec p t = case t of
    Mu -> showString "m "
    Alpha -> showString "a "
    Pi -> showString "p "
    Beta -> showString "b "
    Delta -> showString "d "

instance Show (TaggedCnfFormula Literal) where
  showsPrec p (Tag t lss) =
    show t . shows (CNF lss)

instance Show (TaggedCnfFormula OLiteral) where
  showsPrec p (Tag t lss) =
    show t . shows lss

Collecting Constants

instance HasConstants l =>
  HasConstants (TaggedCnfFormula l) where
5.16 Descriptions

Module `Descriptions` implements descriptions for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, TypeSynonymInstances, OverlappingInstances #-}

module DPL.Descriptions (Description(..), descriptionP, labelCheck, groundCheck, obviateCheck, generateRuns, generateAxioms, loadDescription, assertCheck) where

import Data.List
import Data.Maybe
import qualified Data.Map as M
import qualified Data.Set as S
import ABR.Parser hiding (Tag)
import ABR.Parser.Checks
import ABR.Text.Showing
import ABR.Control.Check
import ABR.Data.List
import ABR.Parser.Lexers
import DPL.Constants
import DPL.Variables
import DPL.Arguments
import DPL.Literals
import DPL.LitSets
import DPL.Formulas
import DPL.Rules
import DPL.Priorities
import DPL.Tags
import DPL.TypeDecs
import DPL.DPLLexer

5.16.1 Data type definitions

A plausible Description consists of:

- a set of tagged cnf-formulas to attempt proofs of (dout);
- a name to uniquely identify descriptions and theories (dnam);
- some modules to import (dimp); and
- some predefined tokens in some target language (dpre).

data Description = Description {
  dnt :: [NewTypeDec],
  dtt :: TypeTable,
  dat :: [AtomTypeDec],
  dax :: [Clause],
  ddef :: [Default],
  drp :: [Rule],
  drd :: [Rule],
  dpri :: [Priority],
  din :: [[Literal]],
  dig :: [[Literal]],
  das :: [[Literal]],
  dout :: [[TaggedCnfFormula Literal, Maybe String]],
  dnam :: String,
  dimp :: [String],
  dpre :: [String]
}  

5.16.2 Parser

Gode generation hints

```
nameDec ::= "name" "{" uName "}";
level="grammar".
```

```
nameDec
  name uName
```

```
nameDecP :: Parser String
nameDecP =
  literalP "lName" "name"
  ==> literalP "symbol" "("
  ==> tagP "uName"
  <= nofail (literalP "symbol" "))")
  0> (\(l,n\) -> n)

importDec ::= "import" "{" string "}";
level="grammar".

```
importDec
  import string
```

```
importDecP :: Parser String
importDecP =
  literalP "lName" "import"
  ==> literalP "symbol" "("
  ==> tagP "string"
  <= nofail (literalP "symbol" "))")
  0> (\(l,c,s\) -> cs)

predefDec ::= "predefined" "{" string "}";
level="grammar".

```
predefDec
  predefined string
```
**predefDec**

```
predefDecP :: Parser String
predefDecP =
  literalP "lName" "predefined"
  *> literalP "symbol" "("
  *> tagP "string"
  <* nofail (literalP "symbol" ")")
  @(_,cs,_) -> cs
```

**Inputs**

```
someLits ::= "{ literal "," literal "}";
level="grammar".
```

```
someLitsP :: Parser [Literal]
someLitsP =
  literalP "symbol" "{"
  *> pLiteralP
  <*> many (  literalP "symbol" "",
                  *> nofail pLiteralP )
  <* nofail (literalP "symbol" ")")
  @(ns,ls) -> case ns of
    [] -> Nothing
    [(_,s,_)] -> Just s
```

```
input ::= "input" someLits;
level="grammar".
```

```
inputP :: Parser [Literal]
inputP =
  literalP "lName" "input"
  *> someLitsP
```

**Ignores**

```
ignore ::= "ignore" ["cardinal" "of"] someLits;
level="grammar".
```

```
ignoreP :: Parser [[Literal]]
ignoreP =
  literalP "lName" "ignore"
  *> optional (  tagP "cardinal"
                  <* nofail (literalP "lName" "of") )
  <*> someLitsP
  @(ns,ls) -> case ns of
    [] -> [ls]
    [(_,n,(l,_))] ->
      let n' = read n
          in if n' > length ls then
            error $ "Not enough literals for combinations at line " ++ show (l + 1)
          else
            combinations n' ls
```

**Outputs**

```
output ::= "output" "{ taggedFormula ["," string "}";
level="grammar".
```

```
outputP :: Parser (TaggedCnfFormula Literal, Maybe String)
outputP =
  (  literalP "lName" "output"
      *> (  literalP "symbol" "{
          *> nofail taggedFormulaP
          <* optional (  literalP "symbol" ","
                          *> tagP "string"
                          <* nofail (literalP "symbol" ")")
          )
      )
  )
  @(tf,ss) -> case ss of
    [] -> Nothing
    [(_,s,_) -> Just s
```

**Descriptions**

```
descriptionP parses a description and the input, ignore and output specifications.
```

```
statement ::= ( newTypeDec
                        | atomTypeDec
                        | clause
                        | default
                        | rule
                        | priority
                        | input
                        | ignore
                        | assert
                        | output
                        | nameDec
                        | importDec
                        | predefDec
                  ) ".";
level="grammar".
```

```
assertP :: Parser [Literal]
assertP =
  literalP "lName" "assert"
  *> someLitsP
```

```
assertP :: Parser (Literal, Maybe String)
assertP =
  literalP "lName" "assert"
  *> someLitsP
```

```
assertP :: Parser [Literal]
assertP =
  literalP "lName" "assert"
  *> someLitsP
```
statementP :: Parser Statement
statementP = (newTypeDecP <|> NT <
|> atomTypeDecP <|> AT <
|> defaultP <|> Def <
|> nameDecP <|> Nam <
|> importDecP <|> Imp <
|> predefDecP <|> Pre <
|> inputP <|> In <
|> ignoreP <|> Ig <
|> assertP <|> As <
|> outputP <|> Out <
|> ruleP <|> Rul <
|> priorityP <|> Pri <
|> clauseP <|> Ax ) <* nofail (literalP "symbol" ".")

description ::= {statement};
level="grammar".

descriptionP :: Parser Description
descriptionP =
many statementP @> fix . (foldl addStatement (Description [] M.empty []) defaultDefaults [] [] [] [] ['/DEFAULT/'] [])

where
defaultDefaults = [Default (Neg (Prop "<" [Var (Variable "x"), Var (Variable "y")]),
Default (Neg (Prop "<=" [Var (Variable "x"), Var (Variable "y")]),
Default (Neg (Prop "==" [Var (Variable "x"), Var (Variable "y")]),]

addStatement :: Description -> Statement -> Description
addStatement d s = case s of
NT nt -> d {dnt = nt : dnt d}
AT at -> d {dat = at : dat d}
Ax c -> d {dax = c : dax d}
Rul r -> case r of
Plaus _ _ _ -> d {drp = r : drp d}
Defeat _ _ _ -> d {drd = r : drd d}
Pri p -> d {dpri = p : dpri d}
In ls -> d {din = ls : din d}
Ig ls -> d {dig = ls ++ dig d}
As ls -> d {das = reverse $ das d}
out ts -> d {dout = ts : dout d}
def df -> d {ddf = df : ddf d}
Nam n -> d {dnam = n}
imp cs -> d {imp = cs : imp d}
ruleP P rul -> d {drul = rul : drul d}
clauseP P ax -> d {dax = ax : dax d}

fix :: Description -> Description
fix d =
let cs :: S.Set Constant
    cs = getConstants d S.empty
    u :: NewTypeDec
    -- u = TypeName "Universe" := TEnum (S.fromList
    --   (map Const (S.toList cs))
    os = concat [
        fold1 getOrderings [] (dat d),
        fold1 getOrderings [] (dat d),
        fold1 getOrderings [] (dat d),
        fold1 getOrderings [] (drd d)
    ]

d' :: Description
d' = d {
    dnt = snub $ u : dat d ,
    dat = snub $ dat d,
    dax = snub $ dax d ++ assertOrderings os,
    drp = snub $ drp d,
    drd = snub $ drd d,
    dpri = snub $ dpri d,
    din = snub $ din d,
    dig = snub $ dig d,
    das = reverse $ das d,
    dout = snub $ dout d
}
in d' {
    dtt = case evalTypes (map ((n -> t) -> (n,t)) (dtt d)) of
    Left tt -> tt
    Right msg -> error msg
}

5.16.3 Semantic checks

labelCheck is a Check that all of the labels occurring in priorities also occur in rules. It passes the description through if OK, or returns an error message.

labelCheck :: Check Description Description String
labelCheck d =
let rls = snub $ map rlbl $ drp d ++ drd d
    pls = snub $ concatMap (instances (dtt d) (dat d))
in case undefs of
    [] -> CheckPass d
    _ -> CheckFail $ "Undefined labels: " ++ unwords undefs

5.16.4 Asserting Orderings

assertOrderings :: [Clause]
assertOrderings = concatMap ao
where
ao :: [Clause]
ao = case cs of
    [] -> []
    c : cs -> eq c : le c c : map (lt c) cs ++ map (le c) cs ++ ao cs
eq c = mc $ Pos $ Prop "==" [Const c, Const c]
le c c' = mc $ Pos $ Prop "<=" [Const c, Const c']
lt c c' = mc $ Pos $ Prop "<" [Const c, Const c']
mc = Clause . LEnum . S.singleton

5.16.5 Grounding all variables

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

groundCheck :: Check Description Description String
groundCheck d |
    not (hasVariables d) = CheckPass d
    not (hasConstants d) = CheckFail "Can not ground desc" \
    "\n    \". There are variables, but there are no cons\n    \n    \"
    otherwise = CheckPass d {nax = nub $ 
        concatMap (instances (dtt d) (dat d))
        (dax d),
        dmp = nub $ 
        concatMap (instances (dtt d) (dat d))
        (drp d),
        dfr = nub $ 
        concatMap (instances (dtt d) (dat d))
        (drd d),
        ddef = nub $ 
        concatMap (instances (dtt d) (dat d))
        (ddf d),
        din = nub $ 
        concatMap (instances (dtt d) (dat d))
        (din d),
        dig = nub $ 
        concatMap (instances (dtt d) (dat d))
        (dig d),
        das = nub $ map (nub .

November 25, 2013 30
5.16.6 Checking the asserts against the inputs

The `assertCheck` passes a description if every assert (after grounding) completely covers all of the declared inputs.

```haskell
assertCheck :: Check Description Description String
null (dat d) = CheckFail "Assert does not match the \ngrounded description. Please inspect the\nextra axioms given the inputs." in CheckPass d

let facts = [l | Or [l] <- dax d]
let ls = map pos r1,
ls' <- map (map pos) (das d)
then CheckPass d
else CheckFail "Assert does not match the\n\ndelimited inputs. Hint: Inspect the\n\nground description to fix this."

5.16.7 Removing strictly useless rules

If a rule has a literal in its antecedent that is asserted as a fact, then its strict provability is assured and it can be omitted from the antecedent. If the negation of the literal in an antecedent has been knocked out because they contain \( \neg l \) or \( \neg l \) they contain 32.8, rules, the `obviateCheck` passes a description after removing such obviously true literals from antecedents and such obviously useless rules.

```haskell
obviateCheck :: Check Description Description String
each rule.

```

5.16.8 Generating runs

```haskell
generateRuns d returns the list of extra axioms given the inputs and ignores in \( d \). If there are asserts, they are used instead of the inputs/ignores.

```haskell
generateRuns : Description -> [Literal]
generateRuns d | null (dat d) = gen (din d) (dig d)
| otherwise = dat d
gen inss igss returns the lists of extra axioms generated from remaining sets of mutually exclusive inputs \( inss \) and the remaining ignore sets \( igss \). This does not use the asserts.

gens : [[Literal]] -> [[Literal]]
gen igss = case inss of
[] -> []
ins : ins' ->
let qss = case igss of
[qs] -> [qs, [neg qs]]
otherwise = map (map neg b) ++ [s] ++ map neg a)
applyIgnores l igss returns either: Nothing, when the literal \( l \) is knocked out by an element of igss that equals \([\]) or Just igss', where igss' contains any elements of igss not knocked out because they contain \( \neg l \).

each rule.

```

5.16.9 Loading a description

```haskell
loadDescription path reads the description file from path and processes it ready for subsequent proofs, for example with Inference.doProof. It either returns the description or an error message.

```haskell
loadDescription : FilePath -> IO (CheckResult Description String)
loadDescription path = do
source <- readFile path
case (checkParse (dropWhite . nobad . total) lexerL) (total descriptionP) & labelCheck) source of
CheckFail msg -> return $ CheckFail msg
CheckPass d -> return $ (groundCheck & flatten (dtt d) & obviateCheck) d
5.16.10 Instance declarations

Showing

instance Show Description where

  showsPrec p d =
    showString "\% new type declarations:\n".
    showWithTerm ".\n" (dnt d).
    showString "\% evaluated types:\n".
    showWithTerm ".\n" (map (
      (n,t) -> n := t)
    (M.toList (dtt d))).
    showString "\% atom type assertions:\n".
    showWithTerm ".\n" (dat d).
    showString "\% axioms:\n".
    showWithTerm ".\n" (dax d).
    showString "\% default facts:\n".
    showWithTerm ".\n" (ddef d).
    showString "\% plausible rules:\n".
    showWithTerm ".\n" (drp d).
    showString "\% defeaters:\n".
    showWithTerm ".\n" (drd d).
    showString "\% priorities:\n".
    showWithTerm ".\n" (dpri d).
    showString "\% inputs:\n".
    showWithTerm ".\n" (din d).
    showString "\% ignores:\n".
    showWithTerm ".\n" (dig d).
    showString "\% asserts:\n".
    showWithTerm ".\n" (das d).
    showString "\% outputs:\n".
    showWithTerm ".\n" (dout d).
    showString "\% name: ".
    showString (dnam d).
    showString "\% imports:\n".
    shows (dimp d).
    showString "\% predefined tokens:\n".
    shows (dpre d).

Collecting constants

instance HasConstants Description where

  getConstants d cs =
    let cs1 = foldr getConstants cs $ dnt d
    cs2 = foldr getConstants cs1 $ M.elems $ dtt d
    cs3 = foldr getConstants cs2 $ dat d
    cs4 = foldr getConstants cs3 $ dax d
    cs5 = foldr getConstants cs4 $ drp d
    cs6 = foldr getConstants cs5 $ drd d
    cs7 = foldr getConstants cs6 $ concat $ din d
    cs8 = foldr getConstants cs7 $ concat $ dig d
    cs9 = foldr getConstants cs8 $ map fst $ dout d
    cs10 = foldr getConstants cs9 $ ddef d
    in cs10

Collecting variables

instance HasVariables Description where

  getVariables d vs =
    let vs1 = foldr getVariables vs $ dnt d
    vs2 = foldr getVariables vs1 $ M.elems $ dtt d
    vs3 = foldr getVariables vs2 $ dat d
    vs4 = foldr getVariables vs3 $ dax d
    vs5 = foldr getVariables vs4 $ drp d
    vs6 = foldr getVariables vs5 $ drd d
    vs7 = foldr getVariables vs6 $ concat $ din d
    vs8 = foldr getVariables vs7 $ concat $ dig d
    vs9 = foldr getVariables vs8 $ map fst $ dout d
    vs10 = foldr getVariables vs9 $ ddef d
    -- Don't count the variables in defaults as they can falsely indicate a description needs grounding when it really doesn't.
    in vs9

Collecting Atoms

instance HasAtoms Description where

  getAtoms d as =
    let as1 = foldr getAtoms as $ dax d
    as2 = foldr getAtoms as1 $ drp d
    as3 = foldr getAtoms as2 $ drd d
    as4 = foldr getAtoms as3 $ concat $ din d
    as5 = foldr getAtoms as4 $ concat $ dig d
    as6 = foldr getAtoms as5 $ map fst $ dout d
    in as6

Flattening

instance HasLitSets Description Description where

  flatten tt d =
    let chax = map (flatten tt) $ dax d
    chrp = map (flatten tt) $ drp d
    chrd = map (flatten tt) $ drd d
    fs = 
      [msg | CheckFail msg <- chax]
      ++
      [msg | CheckFail msg <- chrp]
      ++
      [msg | CheckFail msg <- chrd]
    pax = [ax | CheckPass ax <- chax]
    prp = [rp | CheckPass rp <- chrp]
    prd = [rd | CheckPass rd <- chrd]
    in if null fs then
      CheckPass d {dax = pax,
        drp = prp,
        drd = prd}
    else
      CheckFail $ unlines fs

Instantiating

instance Instantiable Argument where

  instantiate tt ats a = [(a, NullSub)]

instance Instantiable Type where

  instantiate tt ats t = case t of
    TEnum as' ->
      [(TEnum (S.fromList as'), s) <- instantiate tt ats (S.toList as)]
    TName n -> [(TName n, NullSub)]
    t1 :+: t2 ->
      [(t1' :+: t2', s1 :>-> s2) <-
        (t1', s1) <- instantiate tt ats t1,
        (t2', s2) <- instantiate tt ats (ground s1 t2)]
    t1 :-: t2 ->
      [(t1' :-: t2', s1 :>-> s2) <-
        (t1', s1) <- instantiate tt ats t1,
        (t2', s2) <- instantiate tt ats (ground s1 t2)]
    t1 ^^ t2 ->
      [(t1' ^^ t2', s1 :>-> s2) <-
        (t1', s1) <- instantiate tt ats t1,
        (t2', s2) <- instantiate tt ats (ground s1 t2)]
    t1 :<- t2 ->
      [(t1' :<- t2', s1 :>-> s2) <-
        (t1', s1) <- instantiate tt ats t1,
        (t2', s2) <- instantiate tt ats (ground s1 t2)]

instance Instantiable VarGen where

  instantiate tt ats (v :<- t) =
    [(v :<- t', s) <-
      (v :<- t', s) <- instantiate tt ats t]

instance Instantiable Default where

  instantiate tt ats (Default l) =
    [(Default l', s) <-
      (1', s) <- instantiate tt ats l]

instance Instantiable [(Argument, VarGen)] where

  instantiate tt ats avgs = case avgs of
    [] -> [((), NullSub)]
    (a, vg@(v :<- t)) : avgs' -> case a of
      Const c ->
        [(a, vg) : avgs, s'] |
        c 'elem' evalType' tt t,
let s = v :->- c,
  (avgs, s') <- instantiate tt ats
[(a, ground s vg) | (a,vg) <- avgs']

instance Instantiable Atom where
  instantiate tt ats (Prop n as) =
    let len :: Int
        len = length as
        tss :: [[VarGen]]
        tss = [ts | AtomTypeDec n' ts <- ats,
                   n' == n, length ts == len]
        ts :: [VarGen]
        ts = case tss of
            [] -> ([Variable "D_" ++ show i) :<- u |
                       let u = TName (TypeName "Universe"),
                           i <- [0 .. len - 1]
                   ]
            _ -> ts
    in (Prop n as', s)
      | (avgs,s) <- instantiate tt ats (zip as ts),
         let as' = map fst avgs

instance Instantiable Literal where
  instantiate tt ats a =
    case a of
      Pos a' -> [(Pos a', s) |
                   (a', s) <- instantiate tt ats a]
      Neg a' -> [(Neg a', s) |
                   (a', s) <- instantiate tt ats a]

instance (Instantiable a, Groundable a) =>
  Instantiable [a] where
  instantiate tt ats xs = case xs of
    [] -> [(], NullSub)
    x : xs ->
      [(x', s') |
          (x', s) <- instantiate tt ats x,
          (xs', s') <- instantiate tt ats (map (ground s) xs)]

instance Instantiable LitSet where
  instantiate tt ats ls = case ls of
    LEnum ls ->
      [LEnum (S.fromList ls'), s]
      | (ls', s) <- instantiate tt ats (S.toList ls)
[(l1', s') <- instantiate tt ats l1,
  (vgs', s') <- instantiate tt ats (ground s vgs)]

instance Instantiable Clause where
  instantiate tt ats c = case c of
    Or ls ->
      [(Or ls', s) |
          (ls', s) <- instantiate tt ats ls]
    Clause ls' s ->
      [(Clause ls', s) |
          (ls', s) <- instantiate tt ats ls]
    Strict c a -> case a of
      Left ls ->
        [(Strict c' (Left ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]
      Right ls ->
        [(Strict c' (Right ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]
    Plaus l c a -> case a of
      Left ls ->
        [(Plaus l c' (Left ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]
      Right ls ->
        [(Plaus l c' (Right ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]
    Defeat l c a -> case a of
      Left ls ->
        [(Defeat l c' (Left ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]
      Right ls ->
        [(Defeat l c' (Right ls'), s :>-> s') |
            (ls', s) <- instantiate tt ats ls,
            (c', s') <- instantiate tt ats (ground s c)]

instance Instantiable (TaggedCnfFormula Literal,
                         Maybe String) where
  instantiate tt ats (Tag tag lss, ms) = case ms of
    Just _ -> error "Can't instantiate output with\"string"
    Nothing -> [(Tag tag ls', Nothing), s]
      | (ls',s) <- instantiate tt ats ls]

5.17 Theories

Module Theories implements theories for Decisive Plausible Logic.
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances,
          TypeSynonymInstances, OverlappingInstances #-}
module DPL.Theories (PTheory(..), Theory, makeTheory, IsTheory(..)
                      ) where
  import Data.List
  import qualified Data.Map.Strict as M
  import qualified Data.Set as S
  import ABR.Data.List
  import ABR.Text.Showing
  import DPL.Atoms
  import DPL.Literals
  import DPL.Rules

November 25, 2013 33
5.17.1 Data type definitions

A plausible PTheory, constructor Theory, consists of a set of rules \( R_s \cup R_p \cup R_d \) and the priority relation \( > \). These are stored in fields: \( \text{trs} \) for \( R_s \), \( \text{trp} \) for \( R_p \), \( \text{trd} \) for \( R_d \) and \( \text{tpri} \) for \( > \). Additionally to perform proofs we store \( \text{tfc}t \) for \( \text{Fct}(T) \), \( \text{tinc} \) for \( \text{Inc}(T) \) and \( \text{tinc1} \) for \( \text{Inc}(T,l) \). For diagnostics we store \( \text{tax} \) for \( \text{Ax} \), and \( \text{trsn} \) for \( \text{Rsn}(A_z) \).

data PTheory r l = Theory {
    \text{trs} :: [r l], 
    \text{trp} :: [r l], 
    \text{trd} :: [r l], 
    \text{tpri} :: [Priority], 
    \text{tinc} :: [[[l]]], 
    \text{tinc1} :: M.Map 1 [[[l]]] 
} 

isTheory t r l = \text{tfc}t \neq \text{null} 

5.17.2 Making theories from descriptions

\( m\text{m}in \) \( S \) returns \( \text{Mn}(S) = \{ VL \in S : k \cap L \neq \emptyset \} \).

\( m\text{m}in :: [\text{Clause}] \rightarrow [\text{Clause}] \)
\( s = \text{map Or} \$ \text{noSuperSets} \{ I \mid \text{Or} l \leftarrow s \} \)
\( m\text{m}in :: [\text{Clause}] \rightarrow [\text{Clause}] \)

\( \text{fct} Ax = \text{filter} \ (\text{not} \ . \ \text{isTautology}) \$ \ \text{mmin} \$ \ \text{rsn} \ ax \)
\( \text{fct} :: [\text{Clause}] \rightarrow [\text{Clause}] \)

\( \text{makeTheory} \ d \) transforms description \( d \) into a theory.

\( \text{makeTheory} :: \text{Description} \rightarrow \text{Theory} \)

\( \text{makeTheory} \ d = \text{let} \)
\( \text{Rs} = \{ l \{ - \{ l \} \} \rightarrow I : l \in L \land \forall VL \in \text{Fct}(Ax) \}. \)
\( \text{tax}' = \text{dax} \ d \)
\( \text{trsn}' = \text{rsn} \ \text{tax}' \)
\( \text{tfc}' = \text{fct}(\text{tax}') \)
\( \text{trs}' = \text{[Special} 1 \ (\text{Right} \ (\text{map neg} \ (ls \ \text{\textbackslash} \ [1]l))) \]
\( \mid \text{Or} ls \leftarrow \text{tfc}' \ , \ l \leq ls \}
\( \ \text{Rs} = \text{Rs} \cup \text{Rps} \cup \text{Rdt} \)
\( \text{trp}' = \text{dpr} \ d \)
\( \text{trd}' = \text{drd} \ d \)
\( \text{r} = \text{trs}' \ + \text{trp}' \ + \text{trd}' \)
\( \text{Inc}(T) = \{ l \{ - \{ l \} \} : l \in \text{Inc}(T) \land l \in I \}. \)
\( \text{allAtoms} = \text{S.toList} \$ \ \text{getAtoms} \ d \ \text{S.empty} \)
\( \text{tinc} = \text{[map neg} \ (ls \ \text{\textbackslash} \ [1]) \mid \text{Or} ls \leftarrow \text{null} \ (\text{tfc'} \ + \text{[Or} \ \text{[Neg a, Pos a]} \mid a \leq \text{allAtoms}]) \)
\( \text{Inc}(T,l) = \{ l \{ - \{ l \} \} : l \in \text{Inc}(T) \land l \in I \}. \)
\( \text{allLits} = \text{snub} \$ \ \text{concat} \ \text{tinc} \)
\( \text{tinc'} :: \text{Literal} \rightarrow [[[\text{Literal}]]] \)
\( \text{tinc1} :: \text{Literal} \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] \rightarrow [[[\text{Literal}]]] 
\)

5.17.3 Interrogating theories

Class IsTheory provides support for performing proofs with a theory data type.

\( \text{class IsTheory} t r l \) where
\( \text{tfctl} T L \) returns \( \text{Fct}(T; L) = \{ \forall K \in \text{Fct}(T) : |K \cap L| \geq 2 \}. \)
\( \text{tfctl} :: t r l \rightarrow \{ [1] \} \rightarrow \{ [1] \} \)
\( \text{inc} :: t r l \rightarrow \{ [1] \} \rightarrow \{ [1] \} \)
\( \text{rsl} T L \) returns \( \text{Inc}(T,l) \).
\( \text{rsl} :: t r l \rightarrow \{ [1] \} \rightarrow \{ [1] \} \)
\( \text{rpm} T L \) returns \( R_p[l,l] \).
\( \text{rpm} :: t r l \rightarrow \{ [1] \} \rightarrow \{ [1] \} \)

5.17.4 Instance declarations

Showing

\( \text{instance Show Theory where} \)
\( \text{showsPrec} \ p \ t = \)
\( \text{showsString} \ "\% \text{Ax}: \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{Ax}: \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{Rsn}(Ax): \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{Rsn}(Ax): \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{Rsn}(Ax): \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{plausible rules}: \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{plausible rules}: \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{defeaters}: \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{defeaters}: \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{strict rules}: \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{strict rules}: \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{priorities}: \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{priorities}: \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{Inc}(T): \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{Inc}(T): \text{\textbackslash}n" \ (\text{tax} t) \)
\( . \ \text{showString} \ "\% \text{Inc}(T): \text{\textbackslash}n" \)
\( . \ \text{showWithTerm} \ "\% \text{Inc}(T): \text{\textbackslash}n" \ (\text{tax} t) \)

Collecting atoms

\( \text{instance HasAtoms Theory where} \)
\( \text{getAtoms t as} = \)
\( \text{let as1 = foldr \text{getAtoms} as \$ \ \text{trs} t \}
\( \text{as2} = \text{foldr} \text{getAtoms as1} \$ \ \text{trp} t \}
\( \text{as3} = \text{foldr} \text{getAtoms as2} \$ \ \text{trd} t \}
\( \text{in as3} \)
\( \text{instance HasAtoms (Description,Theory) where} \)
\( \text{getAtoms (d,t) as} = \)
\( \text{let as1 = getAtoms d as \}
\( \text{as2} = \text{getAtoms t as1} \}
\( \text{as3} = \text{getAtoms as2} \}
\( \text{in as2} \)

Theory interrogation

\( \text{filterByC} I rs \) selects the rules in \( rs \) with consequent \( l \).
\( \text{filterByC :: Literal} \rightarrow \{ \text{Rule} \} \rightarrow \{ \text{Rule} \} \)
\( \text{filterByC} I = \text{filter} ((== l) \ . \ \text{rcon}) \)

November 25, 2013 34
filterByBeats \( r \) \( s \) selects the rules in \( r \) that beat rule \( r \).

filterByBeats :: Theory \rightarrow Rule \rightarrow [Rule] \rightarrow [Rule]

filterByBeats \( t \) \( s \) = case \( s \) of
  Strict _ _ \rightarrow const []
  _ \rightarrow
    let l = rlbl s
    in filter (\( r \)\rightarrow (rlbl \( r \) \geq 1) \ 'elem' \( t \) \( p \) \( t \))

instance IsTheory OTheory FRule Literal where
  tfctl \( t \) \( ls \) = [ks | Or ks \< 'tfct t, length \( \{ k \mid k \< 'ks, k \ 'elem' \( ls \} \) \geq 2]
  inc t l = case M.lookup 1 (tincl t) of
    Just js \rightarrow js
    Nothing \rightarrow []
  rsl t l = filterByC l (trs t)
  rpl t l = filterByC l (trp t)
  rl t l = filterByC l (trs t ++ trp t ++ trd t)
  rpls t l s = filterByBeats t s (rpl t l)

5.18 OTheories

Module OTheories implements optimised theories for Decisive Plausible Logic.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, TypeSynopsisInstances, OverlappingInstances #-}

module DPL.OTheories (OLabel, ORIndex, OAIndex, OPRule(..), ORule, OTheory, toOTheory ) where

import Data.Array
import qualified Data.Map.Strict as M
import Data.List
import ABR.Text.Showing
import ABR.Data.List
import qualified Data.Map.Strict as M
import qualified Data.List as List
import Data.Array

5.18.1 Data type definitions

Optimised rules are assigned optimised labels (OLabel) which are 0 for a rule that was originally unlabelled and therefore not involved in the priority relation, or 1 to \( M \) for labelled rules, where \( M \) is the number of distinct rule labels.

- type OLabel = Int
- Each optimised rule will be uniquely identified by an index number (ORIndex), unrelated to its label.

- type ORIndex = Int
- Many optimised rules will have the same antecedents. We will store them just once. Each unique antecedent will be identified by a unique index number (OAIndex).

- type OAIndex = Int
- Optimised rules (OPRule), constructor (ORule) only need store their antecedents, as they are stored presorted by kind and consequent. But attached to each rule, we also need a link to the lookup array of actual antecedents. (I think this will be OK.)

- data OPRule l = ORule ORIndex ORLabel OAIndex (Array OAIndex [l])
  the preferred shorthand.

- type ORule = OPRule GLiteral
- An optimised theory (OTheory), constructor (OTry) contains:
  - otam – the AtomMap that maps from \( \text{Atoms} \) to \( \text{GAatoms} \) (which range from 1 to \( N \)).
  - otom – the \( \text{GAtomMap} \) that maps from \( \text{GAatoms} \) to \( \text{Atoms} \).
  - ot ants – the array of all unique antecedents.
  - otrs – an array mapping every rule's index to the actual optimised rule.
  - otrsl – the indices of the strict rules presorted by consequent (This array is indexed from \(-N\) to \(N\). Element 0 should be empty).
  - otrl – the indices of the plausible rules presorted by consequent (This array is indexed from \(-N\) to \(N\). Element 0 should be empty).
  - otrpls – the indices of the rules that beat another rule, presorted by consequent (This array is indexed from \(-N,1\) to \(N,M\). Elements \((0,i)\) should be empty. There are no elements \((i,0)\) as rules with 0 labels are not involved in priorities;)
  - otfct – optimised ftct; and
  - otincl – optimised tincl.

data OTheory r l = OTheory {
  otam :: AtomMap,
  otom :: GAtomMap,
  otrants :: Array OAIndex [1],
  otrs :: Array ORIndex [r l],
  otrsi :: Array l [ORIndex],
  otrp :: Array l [ORIndex],
  otrpl :: Array (l,OLabel) [ORIndex],
  otfct :: [[1]],
  otrct :: Array l [[1]]
} OTheory is the preferred shorthand.

toOTheory \( D \) and \( T \) makes an optimised theory from \( T \). Make sure \( D \) is the grounded version.

toOTheory :: Description -> Theory -> OTheory

5.18.2 Conversions

let \( n, an, oan \) = mkAtomMaps \( (d,t) \)
  \( ml = toGLiteral an \)
  \( mml = map \( m \) \)
  \( labels = snub \$ map rlbl (trp t) ++ map rlbl (trd t) \)
  \( ml = length labels \)
  \( labelMap = M.fromList \$ zip labels [1..] \)
  \( mapLbl lbl = case M.lookup 1 lbla labelMap of \)
    Just i \rightarrow i
    Nothing \rightarrow 0
  \( rules = trp t ++ trd t ++ trs t \)
  \( np = length \( \text{rules} \) \)
  \( nd = length \( \text{trd} \) \)
  \( ns = length \( \text{trs} \) \)
  \( nr = np + nd + ns \)
  \( antecedents = snub mml as \mid Right as <- map rant rules \)
  \( nants = length \text{antecedents} \)
  \( ants = array \( (0, \text{nants} - 1) \$ \) zip \( [0..] \) \text{antecedents} \)

findAnt :: [Literal] -> OAIndex

findAnt \( d \) \( t \)

let \( a = mml \) \( l \)
  _ = f (oaIndex -> OAIndex -> OAIndex)
  _ = f (i j)
  _ = let m = \((i + j) \ 'div' 2 \)
      in if m == ants \( t \) then m
         else if a < ants \( t \) then f i (m-1)
         else f (m+1) j
  in f 0 (nants - 1)
  \( rs = array (0, nrs - 1) \)
  \( [(i, ORule i (mapLbl l) (\( \text{findAnt} \) as) ants) \)
negant (ORule _ _ a ants) = [neg (ants ! a)]

5.19 Proof Results
Module ProofResults implements a data type that represents
all the possible results on attempting a proof.

5.19.1 Data type
Proof functions have three-valued results. A ProofResult
is one of: Plus (for +1), Zero (for 0); and Minus (for −1).
dataProofResult = Minus | Zero | Plus
deriving (Eq, Ord)

5.19.2 Instance declarations
Showing
instance Show ProofResult where
  showsPrec p r = case r of
    Minus -> showString "-1"
    Zero -> showChar '0'
    Plus -> showString "+1"

DeepSeq
instance NFData ProofResult where {}

Negation
instance Negatable ProofResult where
  neg r = case r of
    Minus -> Plus
    Zero -> Negatable
    Plus -> Minus

pos _ = Plus

5.20 Inference
Module Inference defines the proof functions for Decisive
Plausible Logic. They are defined as methods of a type class.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances,
    TypeSynonymInstances, OverlappingInstances #-)

module DPL.Inference (Plausible(..), doProof, doProofSilently,
    doProofs, doProofsSilently, doProofsQuicklySilently,
    doProofsQuickly, proofsTable) where

November 25, 2013 36
import Control.DeepSeq
import Data.List
import qualified Data.Map.Strict as M
import System.IO
import ABR.Data.List
import ABR.Text.String
import DPL.OTheories
import DPL.Rules
import DPL.ProofResults
import DPL.Tags
import DPL.Theories
import DPL.Literals
import System.IO
import qualified Data.Map.Strict as M

5.20.1 PlausibleX classes: overloaded inference

Classes PlausibleX allows overloading of the inference conditions over the monad through which a proof is threaded, and the state information passed along with results. The aim is to implement the proof functions once in terms of auxiliary functions that are implemented by each instance.

Auxiliary functions

These methods of classes PlausibleX need to be implemented for each instance, unless the default is adequate.

class (Monad monad) => Plausible monad state where

return_ :: ProofResult -> state -> monad (ProofResult, state)

neg_ :: (state -> monad (ProofResult, state)) -> state -> monad (ProofResult, state)

just_ :: (state -> monad (ProofResult, state)) -> state -> monad (ProofResult, state)

max_ ps state = case ps of
  [] -> return (Min, state)
  [p] -> p state
  _ -> return (Max, state)

min_ ps state = case ps of
  [] -> return (Max, state)
  [p] -> p state
  _ -> return (Min, state)


Proof functions

These are the proof functions that define Decisive Plausible Logic. They have the following default implementations, in terms of the auxiliary functions above.

class (Monad monad) => PlausibleRL monad state t r l where

label_r :: String -> String -> Tag -> [l] -> [l] -> state -> monad (ProofResult, state)

label_ls :: String -> String -> Tag -> [l] -> [l] -> state -> monad (ProofResult, state)


November 25, 2013 37
{ $\{\lambda l, B \}$.}

\textbf{L.2) Strict $L(B) = \max\{P(\lambda \lambda \lambda (r), B \cup \{l\}) : r \in R_0\{l\}\}$.}

\textbf{L.3) Plaus $L(B) = \min\{\text{For}(L, B), \text{Nullified}(L, B)\}$.}

\textbf{L.4) For $L(B) = \max\{P(\lambda \lambda \lambda \lambda (r), B \cup \{l\}) : r \in R_0\{l\}\}$.}

\textbf{L.5) Nullified $L(B) = \min\{\text{Disabled}(L, B, J) : J \in \text{Inc}(T, l)\}$.}

\textbf{L.6) Disabled $L(B, J) = \max\{\text{Discredited}(L, B, j) : j \in J\}$.}

\textbf{L.7) Discredited $L(B, j) = \min\{\text{Defeated}(L, B, s) : s \in R[j]\}$.}

\textbf{L.8) Defeated $L(B, s) = \max\{\text{Beaten}(L, B, s), \text{Inappl}(L, B, s)\}$.}

\begin{itemize}
  \item[] \textbf{L.9) Beaten $L(B, s) = \max\{P(\lambda \lambda \lambda \lambda (t), B \cup \{l\}) : t \in R_0\{l\}\}$.}
  \item[] \textbf{L.10.a) Inappl $L(\alpha, B, s) = \min\{\text{Defeated}(\alpha, B, s) : B \in \{l\}\}$.}
  \item[] \textbf{L.10.b) Inappl $L(\beta, B, s) = \min\{P(\beta \lambda \lambda \lambda \lambda (\alpha), B \cup \{l\}) : B \in \{l\}\}$.}
  \item[] \textbf{L.10.d) Inappl $L(\delta, B, s) = \max\{P(\delta \lambda \lambda \lambda \lambda (\alpha), B \cup \{l\}) : B \in \{l\}\}$.}
\end{itemize}

\section{5.20.2 Maybe \{\} instances of PlausibleX}

These instances will not print traces. Call \texttt{p} like this to suppress the trace:

\begin{itemize}
  \item[] \texttt{let Just result = \texttt{p} theory (\texttt{Tag lambda f}) \{\} () in ...}
  \item[] \texttt{instance Plausible Maybe () \{\} in ...}
  \item[] \texttt{instance PlausibleRL Maybe () \texttt{PRule} Literal where \{\}}
  \item[] \texttt{instance PlausibleTRL Maybe () \texttt{PTheory} \texttt{PRule} Literal where \{\}}
\end{itemize}

\section{5.20.3 IO instance of PlausibleX}

These instance will print traces. See \texttt{doProofs} below for an example call to \texttt{p} that prints a trace.
5.20.4 Optimised instances of PlausibleX

These instances will not print traces and use optimised data structures, and memoing using a History, but will provide some diagnostic progress information.

instance Plausible Maybe History where {}
instance PlausibleL Maybe History OLiteral where {}
instance PlausibleRL Maybe History ORule OLiteral where {}
instance PlausibleTRL Maybe History OTheory ORule OLiteral where {}

5.20.5 Noisy optimised instances of PlausibleX

These instances will use optimised data structures, and memoing using a History, but will provide some diagnostic progress information.

instance Plausible IO (String,History) where
return_ result (indent,h) = do
putStr " 
return (result, (indent,h))
neg_ result (indent,h) = do
putStrLn $ indent ++ "- (" 
(r,_) <- result (". " ++ indent, h)
putStrLn $ indent ++ 
return (neg r, (indent,h))
just_ result (indent,h) = do
putStrLn " = (" 
(r,_) <- result (". " ++ indent, h)
putStr $ indent ++ 
return (r,(indent,h))

5.20.6 Optimised instances of PlausibleX

These instances will not print traces and use optimised data structures, and memoing using a History.

instance Plausible Maybe History where {}
instance PlausibleL Maybe History OLiteral where {}
instance PlausibleRL Maybe History ORule OLiteral where {}
instance PlausibleTRL Maybe History OTheory ORule OLiteral where {}

instance Plausible IO (String,History) where
return_ result (indent,h) = do
putStr " 
return (result, (indent,h))
neg_ result (indent,h) = do
putStrLn $ indent ++ "- (" 
(r,_) <- result (". " ++ indent, h)
putStrLn $ indent ++ 
return (neg r, (indent,h))
just_ result (indent,h) = do
putStrLn " = (" 
(r,_) <- result (". " ++ indent, h)
putStr $ indent ++ 
return (r,(indent,h))

instance PlausibleL IO String Literal where
label_ label func lambda f b indent p = do
let goal = func ++ "(" ++ show (Tag lambda f) ++ ", " ++ show b ++ ")"
putStr $ indent ++ goal 
(r,_) <- p indent
putStr $ indent ++ "++ show r 
++ ", by ++ label 
return (r, indent)

instance PlausibleRL IO String PRule Literal where
label_r label func lambda f b indent p = do
let goal = func ++ "(" ++ show (Tag lambda f) ++ ", " ++ show b ++ ")"
putStr $ indent ++ goal 
(r,_) <- p indent
putStr $ indent ++ "++ show r 
++ ", by ++ label 
return (r, indent)

instance PlausibleTRL IO String PTheory PRule Literal where
label_l label func lambda f b indent p = do
let goal = func ++ "(" ++ show (Tag lambda f) ++ ", " ++ show b ++ ")"
putStr $ indent ++ goal 
(r,_) <- p indent
putStr $ indent ++ "++ show r 
++ ", by ++ label 
return (r, indent)
null
return (axs, rs)
in mapM proveD (generateRuns d)

proofsTable takes the description D and the results from doProofs runs and builds a table to display the results.

proofsTable :: Description -> [(Literal,ProofResult)] -> String
proofsTable d runs =
  let headings =
      replicate ((length . fst . head) runs) "" ++
      map (show . fst) (dout d)
  row (axs,rs) = map show axs ++ map show rs
  in makeTableL ' ' $ headings : map row runs

5.21 MakeCExprs
Module MakeCExprs packages the transformation of a collection of proof results depending on some input parameters as a C Boolean expression.

module DPL.MakeCExprs (makeCExprs) where
import Data.List
import Data.Maybe
import Data.Array.IArray
import Data.Array.MArray
import Data.Array.IO
import System.IO
import ABR.Data.List
import ABR.Text.String
import ABR.Logic.QuineMcClusky
import DPL.Literals
import DPL.Description
import DPL.ProofResults
import DPL.Tags
import DPL.Formulas

makeCExprs D runs takes the description D and the results from doProofs runs and prints a listing of the equivalent C Boolean expressions.

makeCExprs :: Description -> [(Literal,ProofResult)] -> Bool -> IO ()
machine CExprs d t simpC = do
  let cases :: [(TaggedCnfFormula Literal, Maybe String),(ProofResult,[Literal])] =
      [(ts, zip rs lss) |
        let (lss,rss) = unzip t,
        (ts,rs) <- zip (dout d) (transpose rss)]
displayCase :: ((TaggedCnfFormula Literal, Maybe String), (ProofResult,[Literal])) -> IO ()
displayCase ((tf,ms),t') =
  let lss = [ls | (Plus,ls) <- t']
  lss' <- if simpC then simplify lss
  else return lss
  cex = makeCExpr lss'
  cex' = unlines $ map (\cs -> " " ++ cs ++ " \"") $ lines cex
  case ms of
    Nothing -> putStrLn $ show tf ++ " =
    Just s -> do
      putStrLn $ "define "+ unString s ++ " " ++ cex'
      putStrLn "\n"
  mapN displayCase cases
makeCExprs lss renders the formula lss in something close to C.

makeCExpr :: [Literal] -> String
makeCExpr lss =
  let cc lss = case lss of
    [] -> "false"
    lss ->
      concat $ intersperse "|| " $ map cc lss
    cc lss = case lss of
      [] -> "true"
      lss ->
    lss = map (\l -> show l) lss

5.21.1 Quine-McCluskey glue
simplify lss simplifies formula lss using the Quine-McCluskey method.

simplify :: [Literal] -> IO [Literal]
simplify lss = case lss of
  [] -> return []
  lss -> do
    let bss = toBitss lss
    ps = map pos $ head bss
    bss' <- qmSimplify bss
    return $ fromBitss ps $ bss'

Conversion to bits

toBit :: Literal -> QMBit
toBit l = case l of
  Pos _ -> One
  Neg _ -> Zer
toBits is converts a list of literals lss to a list of bits.

toBits :: [Literal] -> [QMBit]
toBits lss = map toBit lss

Conversion back to literals

fromBit :: QMBit -> Maybe Literal
fromBit One = Just l
fromBit Zer = Nothing
fromBit Dsh = Nothing
fromBit Zero = Nothing
fromBit One = Just p

fromBits ps bs converts a list of bits bs back to a list of literals, using the positive references ps.

fromBits :: [Literal] -> [QMBit] -> [Literal]
fromBits bs ps = catMaybes $ zipWith fromBit bs ps

fromBits ps bs converts lists of bits to lists of literals, using the positive references ps.

fromBits :: [Literal] -> [QMBit] -> [Literal]
fromBits lss ps = map (fromBits ps) lss

5.22 MakeCTheory
Module MakeCTheory packages the transformation of an optimised theory into a C data structure.

{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, TypeSynonymInstances, OverlappingInstances #-}
module DPL.MakeCTheory (makeCTheory) where
import System.IO
import Data.Array
import Data.Char
import Data.List
import ABR.Data.List
import ABR.Text.String

November 25, 2013 41
import DPL.Atoms
import DPL.Descriptions
import DPL.Theories
import DPL.OTheories
import DPL.Literals
import DPL.OLiterals
import DPL.Literals
import DPL.OTheories
import DPL.Theories
import DPL.Descriptions

infixr 5 +-+, +^+

import DPL.Tags
import DPL.OAtoms
import DPL.Rules
import DPL.OLiterals
import DPL.Literals
import DPL.OTheories
import DPL.Theories
import DPL.Descriptions

5.22.1 Constructing C

displayComment h cs writes lines css to file with handle h, encapsulated in a C multi-line comment.
displayComment :: Handle -> [String] -> IO ()
displayComment hdl css = do
  hPutStrLn hdl "$"
  hPutStr hdl "$ lines $ map (" + " ++ ) css
  hPutStrLn hdl "$"

(cs ++ cs') catenates cs and cs' ensuring that there is exactly one underscore at their connection point.

(++): String -> String -> String
++ = catenateWith '_'

(cs ++ cs') catenates cs and cs' ensuring that there is exactly one space at their connection point.

(++): String -> String -> String
++ = catenateWith ' '

headerComment h c name writes a file header comment to the file with handle h. The comment labels the file as having name theory.c, and containing the theory named name.

headerComment :: Handle -> Char -> String -> IO ()
headerComment hdl c name = let kind = case c of
  'c' -> "definition"
  'h' -> "header"
  _ -> "???"
in  displayComment hdl ["file: " ++ name ++ "theory." ++ [c],
  "purpose: " ++ name ++ "theory" ++ kind ++ " file",
  "created by: DPL"]

headerComment h c name = do
  hPutStrLn hdl " */"
  hPutStr hdl $ unlines $ map (" * ") ++ css
  hPutStrLn hdl "/*"

headerComment :: Handle -> Char -> String -> IO ()
headerComment dotC 'c' name = headerComment dotH 'h' name

let putH, putHln, putC, putCln :: String -> IO ()
putCln cs = do hPutStrLn dotC cs
putC cs = do hPutStr dotC cs
putHln = hPutStrLn dotH
putH = hPutStr dotH

let tag cs = name ++ cs
  numPlaus = length (otrplot ot)
  numDef = length (otd t)
  numStrict = length (otrplot ot)
  numRules = numPlaus + numDef + numStrict
  inputs :: [Literal]
  inputs = snub $ map pos $ concat lss
  outputs :: [Literal]
  outputs = snub $ map pos $ concat lss

let putCln $ "#include \\
  "theory.h"
  \\
  #define " +^+ tag "STATICSTACKSIZE"
  \\
  #ifndef THEORY_PREFIX

let putHln $ "#define" +^+ tag "THEORY_H_

open the files and write header comments. dotH and dotC are the handles to the .h and .c files respectively. putH and putC print to each of these files.

dotH <- openFile (tag "theory.h") WriteMode
dotC <- openFile (tag "theory.c") WriteMode
headerComment dotH 'h' name
headerComment dotC 'c' name

let putH, putHln, putC, putCln :: String -> IO ()
putH = hPutStr dotH
putHln = hPutStrLn dotH
putC cs = do hPutStr dotC cs
hFlush dotC
putCln cs = do hPutStrLn dotC cs
hFlush dotC

write the preprocessor stuff to the top of the header file.

putHln $ "*/
#define THEORY_PREFIX"

the theory prefix.

putHln $ "#ifdef THEORY_PREFIX"

define the stack size.

putHln $ "#define STATICSTACKSIZE 32768"

import header files.

putHln $ "#include \plaus.h"
putC ln $ "#include \plaus.h"

Define some common loops. forAtoms loops from 1 to maxAtom. rofAtoms loops the other way: forStrict loops over the unique rule numbers assigned to the strict rules in the base theory. Similarly forPlaus and forDef loop over the plausible and defeater rule numbers. forRules loops over all of those. forAnts loops over all of the unique antecedent indices. forInputs loops over the atoms that are inputs. rofInputs loops the other way.

November 25, 2013
let forAtoms, rofAtoms :: (OAtom -> IO ()) -> IO ()
forAtoms f = mapM_ f \[1..maxAtom\]
rofAtoms f = mapM_ f \[maxAtom, maxAtom-1 .. 1\]
forPlaus, forDef, forStrict, forPD, forRules ::
\(\text{OIndex} \to \text{IO} ()\) -> IO ()
forPlaus f = mapM_ f \[0 .. numPlaus - 1\]
forDef f = mapM_ f \[numPlaus .. numPlaus + numDef - 1\]
forStrict f = mapM_ f \[numPlaus + numDef .. numRules - 1\]
forPD f = do
forPlaus f
forDef f
forRules f = mapM_ f \[0 .. numStrict + numPlaus + numDef - 1\]
forAnts :: \(\text{OIndex} \to \text{IO} ()\) -> IO ()
forAnts f = let
(lo,hi) = bounds \(\text{ontants ot}\)
in mapM_ f \[lo .. hi\]

forInputs, rofInputs:: (Literal -> IO ()) -> IO ()
forInputs f = mapM_ f inputs
rofInputs f = mapM_ f inputs
inputs = reverse inputs
--
forLabels :: (OLabel -> IO ()) -> IO ()
forLabels f = mapM_ f \[1 .. numLabels\]

Define the literals.

displayComment doTH[
"These are the literals in the order used in the\" theory.,
","
"As literals, they can be negated, so we can\" t \"use an enum here."
]
rofAtoms \(\lambda i \to \text{putHln} \$ \"\#define\" +^+\)
1Justify 40 (tag \"LH\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (negate (i)))
putHln \$ \"\#define\" ++ 1Justify 40 (tag \"L_NONE\") ++ " +
++ rJustify 10 0"
forAtoms \(\lambda i \to \text{putHln} \$ \"\#define\" +
1Justify 40 (tag \"LP\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (i)))
hPutChar dotH \"\n"

Define the array indices.

displayComment doTH[
"array indices for the literals"
]
rofAtoms \(\lambda i \to \text{putHln} \$ \"\#define\" ++
1Justify 40 (tag \"AN\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (negate (i)))
putHln \$ \"\#define\" ++ 1Justify 40 (tag \"A_NONE\") ++ " +
++ rJustify 10 maxAtom"
forAtoms \(\lambda i \to \text{putHln} \$ \"\#define\" +
1Justify 40 (tag \"AP\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (i)))
hPutChar dotH \"\n"

Counts.

putHln \$ \"\#define\" ++ tag \"A\"OFFS\.viewModel ++ tag \"A\"NONE\"[
\("/\) Add this to literals to get array index. \"/
\"/\) #define ++ tag \"NUM_LITERAL\" +^+\"
show (2 * maxAtom + 1) ++ "\t" ++ tag \"NUM_INPUTS\" +^+\"
putHln \$ \"\#define\" ++ tag \"THEORY_LITFACTS\" +^+\"
\/*/ facts stated in the theory that are in lit ++\"
displayComment doTH[\"array indices for the facts gen\"
\erated from inputs\"]
let factIndex :: (Literal,Int) -> IO ()
factIndex (l,i) = putHln \$ \"\#define\" ++
1Justify 40 (tag \"LM\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (i)))
putHln \$ \"\#define\" ++ 
1Justify 40 (tag \"LM_NONE\") ++ " +
++ rJustify 10 0"
forAnts \(\lambda i \to \text{putHln} \$ \"\#define\" +
1Justify 40 (tag \"LP\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (i)))
hPutChar dotH \"\n"

displayComment doTH[
\"these are the rule numbers","
"," the strict rules also contains pre-computed rule 
\"numbers for inputs","
\"that turn into facts and strict rules once \"asserted:\","
"," \"RS_N_x: strict rules for negative inputs\"
"," \"RS_P_x: strict rules for positive inputs\"
\"]
forInputs \(\lambda x \to \text{putHln} \$ \"\#define\" +^+\"
show i ++ tag \"R\" ++ show i ++ \"
forRules \(\lambda i \to \text{putHln} \$ \"\#define\" +
1Justify 40 (tag \"A\"NONE +^+\"
++ rJustify 10 maxAtom)
forAnts \(\lambda i \to \text{putHln} \$ \"\#define\" +
1Justify 40 (tag \"AP\" +^+ asC (otoam ot ! i)) ++ " +
++ rJustify 10 (asC (i)))
hPutChar dotH \"\n"

Defining macros for checking facts
displayComment doTH[
\"the following code should be auto-generated to \"
\"assert facts (inputs being","
"," \"true or false\"]
putHln \$ \"\#define\" ++
tag \"NUM_FACTS\" +^+\"
tag \"THEORY_FACTS\" +^+ tag \"NUM_INPUTS\"
\"/\) total number of facts ++\"
putHln \$ \"\#define\" ++ tag \"THEORY_NAME\" ++
tag \"Theory\"
putHln \$ \"\#define\" ++ tag \"THEORY_CHK\" +^+ tag \"chk\"n"
putHln \$ \"\#define\" ++ tag \"NUM_NONSRICT\" ++
show (numPlaus + numDef) +^+ \n"

Arrays and methods used for generality
displayComment doTH[\tag inputs is an array of \"
strings denoting the inputs\"]
putHln \$ \"\#define\" ++ tag \"inputs\" +^+ tag \"NUM_INPUTS\" +^+\"
displayComment doTH[\tag outputs is an array of \"
strings denoting the outputs\"]
putHln \$ \"\#define\" ++ tag \"NUM_OUTPUTS\" +^+\"

putHln \$ \"\#define\" ++ tag \"NUM_LITERALS\" +^+\"

putHln \$ \"\#define\" ++ tag \"THEORY_CHK\" +^+\"

Arrays and methods used for generality
Define macros to reset theory status

displayComment doth ("\"unset\"") facts: reset the theory\"
"back to before a fact of an input was known\
tag "RAnts 2 \n \n P; N; else \"tag "FACT( inp, P, N);\n"

declare variables to be used for storing all rules that beat each other, by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n	tag "RPl\n	tag "RPlN\n	tag "RPlN_Fact\n	tag "RPlP\n	tag "RPlP_Fact\n	tag "RPI\n	tag "RPI\n	tag "RPI\n	tag "RPI\n	tag "RPI\n	
declare variables to be used for storing all plausible rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n	tag "RPl\n	tag "RPlN\n	tag "RPlN_Fact\n	tag "RPlP\n	tag "RPlP_Fact\n	tag "RPI\n	tag "RPI\n	tag "RPI\n	tag "RPI\n	
declare variables to be used for storing all rules that beat each other, by consequent.

rofAtoms (i -> do

forRules (\x -> putHln $ "extern rule_t \n	tag "RPl\n	tag "RPlN\n	tag "RPlN_Fact\n	tag "RPlP\n	tag "RPlP_Fact\n	tag "RPI\n	tag "RPI\n	
declare variables to be used for storing incompatibilities

rofAtoms (i -> putHln $ "extern literal_t \n
tag "IncNP\n	tag "IncNP\n	tag "IncNP\n	tag "IncNP\n	tag "IncNP\n	tag "IncNP\n	tag "IncNP\n
declare positive and negative antecedents

forAtoms (i -> putHln $ "extern literal_t \n	tag "Ant\n	tag "Ant\n	tag "Ant\n	tag "Ant\n	tag "Ant\n	tag "Ant\n
declare variables to be used for storing all rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n	tag "RIN\n	tag "RIN\n	tag "RIN\n	tag "RIN\n	tag "RIN\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "external rule_t \n
tag "RM\n	tag "RM\n	tag "RM\n	tag "RM\n	tag "RM\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "RIP\n	tag "RIP\n	tag "RIP\n	tag "RIP\n	tag "RIP\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "R\n	tag "R\n	tag "R\n	tag "R\n	tag "R\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "R\n	tag "R\n	tag "R\n	tag "R\n	tag "R\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "R\n	tag "R\n	tag "R\n	tag "R\n	tag "R\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "R\n	tag "R\n	tag "R\n	tag "R\n	tag "R\n
declare variables to be used for storing all strict rules by consequent.

rofAtoms (i -> putHln $ "extern rule_t \n
tag "R\n	tag "R\n	tag "R\n	tag "R\n	tag "R\n
declare variables to be used for storing all strict rules by consequent.
Define pre-computed facts for inputs used \n\(\forall\) for actual proofs: \n\)

\[
\text{listRules :: [ORule] -> String}
\]

\[
\text{listRules rs} = "[\] = "++ show (length rs) ++ concatMap (\r -> ", " ++ tag (asC r)) rs +^+ ";"
\]

\[
\text{defrules indexed by consequent}
\]

\[
\text{all the rules, indexed by consequent}
\]

\[
\text{let listRules :: [ORule] -> String}
\]

\[
\text{listRules rs} = "[\] = "++ show (length rs) ++ concatMap (\r -> ", " ++ tag (asC r)) rs +^+ ";"
\]

\[
\text{defrules need to be used for inputs that have become facts.}
\]

\[
\text{all pre-computed rules that need to be used for \n\(\forall\) inputs that have become facts:}
\]

\[
\text{let listRules :: [ORule] -> String}
\]

\[
\text{listRules rs} = "[\] = "++ show (length rs) ++ concatMap (\r -> ", " ++ tag (asC r)) rs +^+ ";"
\]

\[
\text{def rules, indexed by consequent}
\]

\[
\text{all strict rules, indexed by consequent}
\]

\[
\text{let rofAtoms :: [ORule] -> String}
\]

\[
\text{rofAtoms ols = "(\i -> putCln \$ "+ tag (asC ols)) ++ show (length ols) ++ concatMap (\r -> ", " ++ tag (asC r)) rs +^+ ";"
\]

\[
\text{bind rules to their antecedents.}
\]

\[
\text{let showAntIndex :: ORule -> String}
\]

\[
\text{showAntIndex ols = "+ tag (asC ols) ++ show (length ols) ++ concatMap (\r -> ", " ++ tag (asC r)) rs +^+ ";"
\]
Define rules that beat other rules, indexed by consequent

```haskell
putCln $ "\n";
forAtoms (\i -> putCln $ \(t\) ++ \(tag\) \("RPlsP" ++ asC (otom ot ! i) ++ ","
    \);)
putCln $ "\n";
forPD (\r -> putCln $ \(tag\) \("RPls\[
\] = \{ \n";
forAtoms (\i -> putCln $ \(t\) ++ \(tag\) \("RPls_NONE\," ++ asC (otom ot ! i) ++ ","
    \);)
```

Define rules that beat another rule, indexed by consequent

```haskell
forAtoms (\i -> putCln $ \(t\) ++ \(tag\) \("RPlsP" ++ asC (otom ot ! i) ++ ","
    \);)
```

Define plausible rules, indexed by consequent

```haskell
forAtoms (\i -> putCln $ \(t\) ++ \(tag\) \("RPls\[
\] = \{ \n";
forAtoms (\i -> putCln $ \(t\) ++ \(tag\) \("RPls_NONE\," ++ asC (otom ot ! i) ++ ","
    \);)
```

Define rules that beat other rules, indexed by consequent and the rule that gets beaten

```haskell
let inp = asC (otoam ot ! abs i)
forPD (\r -> putCln $ \("ruleno_t" ++ tag \("RPls" ++ asC (otom ot ! i) ++ ",")
    \);)
```

Define the incompatibilities.

```haskell
let getsInc i j k = putCln $ "\n"
litToC ((\(otincl ot ! i) !! j) !! k)
getsInc' cs f i j = do
putCln $ \(tag\) \("RSlP" ++ asC (otom ot ! i) ++ show j ++ "[\] = \{ \n"
    \);
mapM_ (getsInc' cs f i)
forAtoms (\i -> putCln $ \("RSlP") ++ asC (otom ot ! i) ++ ","
    \);
```

Define the theory

```haskell
let hClose dotC
hClose dotC
```

5.22.3 Instance declarations

```haskell
instance AsC Atom where
    asC = asCChars . show
    instance AsC OAterm where
```

5.23 MakeHGlue

Module MakeHGlue generates a Haskell module with a function for each of the outputs specified in a description. These functions will return a ProofResult. Their parameters will be generated from the inputs specified in the theory. They won’t necessarily all be simple booleans.
module DPL.MakeHGlue (makeHGlue) where

import System.IO
import Data.Char
import Data.List
import ABR.Parser
import ABR.Text.String
import ABR.HaskellLexer
import ABR.Data.List
import DPL.Constants
import DPL.Arguments
import DPL.Atoms
import DPL.Literals
import DPL.Tags
import DPL.Descriptions

makeHGlue path D writes out a Haskell glue module for description D which was read from file path.

makeHGlue :: FilePath -> Description -> IO ()
makeHGlue path d = do

name, scriptName :: String
name = (
    if dnam d /= "/DEFAULT/"
    then dnam d
    else reverse $ takeWhile isAlpha $ dropWhile (not . isAlpha) $ reverse path)

scriptName = name ++ ".hs"

script <- openFile scriptName WriteMode

Some Haskell generating conveniences.

let put :: String -> IO ()
    put = hPutStrLn script

blank :: IO ()
    blank = put ""

comment, imports :: [String] -> IO ()
    comment = mapM_ (
        cs -> put$ "-- " ++ cs)

    imports = mapM_ (
        cs -> put$ "import " ++ cs)

The file headings.

comment [scriptName, 
    "This file was generated from " ++ path]

blank

put "$ module " ++ name ++ " where"

blank

imports $ ["DPL.Constants", "DPL.Arguments", 
    "DPL.Atoms", "DPL.Literals", "DPL.Tags", 
    "DPL.Inference"]
++ map unString (dimp d)

blank

Now collect up all the inputs and extract from them all the things that look like candidates for parameters.

let inputs = map (unString . show) $ concat $ din d

outputs = doit d

args = case (nofail . total) programL

let makeFun :: (TaggedCnfFormula Literal, Maybe String) -> IO ()

    makeFun (tf, mcs) = do

        let fname = case mcs of

            Just n -> unString n

            _ -> concatMap (\c -> case c of

                _ -> "",

                "not_" -> "not",

                c -> if isAlphaNum c || c == ",",

                then [c] else ",") $ show tf

        tag = case tf of

            Tag Mu cnf -> "Mu [" ++ cnf' cnf

            Tag Alpha cnf -> "Alpha [" ++ cnf' cnf

            Tag Pi cnf -> "Pi [" ++ cnf' cnf

            Tag Beta cnf -> "Beta [" ++ cnf' cnf

            Tag Delta cnf -> "Delta [" ++ cnf' cnf

            Tag Man [ls] -> ["++ cnf' ls ++ "]"

            Tag Var [ls] -> ["++ cnf' ls ++ "]"

            Tag Const [ls] -> ["++ cnf' ls ++ "]"

            Tag Var (l:l') -> l ++ l' ++ "]"

            Tag Const (c:cs) -> "Const \[Constant ++ show cs ++ "]"

            Tag Const (c:cs) -> "Const " ++ cs ++ "]"

            Tag Var _ -> error "DPL can’t generate a glue module -- variable in an input or an output."

        makeInp :: (Bool, Literal) -> IO ()
        makeInp (first, l) = do

            if first then 
                put$ " if " ++ unString (show l)
            else 
                put$ " then " ++ lit l

        blank

        comment ["Required proof: " ++ show tf, 
            "Optionally supplied name: " ++ show mcs, 
            "Selected name: " ++ fname]

        put $ unwords (fname ++ name ++ "the_description") ++ " ="

        put $ unwords (fname ++ name ++ "the_description") ++ " ="

        put $ unwords (fname ++ name ++ "the_description") ++ " ="

        put $ unwords (fname ++ name ++ "the_description") ++ " ="

        mapM_ makeFun outputs

        All done. Close.

        hClose script

5.24 DPL

Module DPL implements the proof tool for Decisive Plausible Logic.
5.24.1 Compiler launch

```
main :: IO ()
main = do
  args <- getArgs
  let options, files = findOpts [FlagS "p", FlagS "c", FlagS "t", FlagS "q", FlagS "v", FlagS "i", FlagS "s", FlagS "C", FlagS "h", QueueS "a"] args
  options' = if length files == 1 then assertFlagPlus "GLOONLY" options else options
  appendNames = lookupQueue "a" options' options'' <- (do
    let (options, files) = findOpts [FlagS "p", FlagS "c", FlagS "s", FlagS "C", FlagS "h", QueueS "a"] args
    unless lonely $ putStrLn 
    "+++ n ++ " *** */\n
    |

    "+++ appended: ":

    ++ n ++ " *** */\n
    cs) appendNames css
    putStrLn $ unlines $ map show combs
    putStrLn $ "--- input combinations ---
    " ++ show (length combs)
    putStrLn $ "--- summary ---"

    let t = makeTheory d
    putStrLn $ toOTheory d t
    putStrLn $ "---

    " ++ path ++ " ---"

    putStrLn $ "--- proofs\n    \ ---"

    putStrLn $ "--- proofs\n    \ ---"

    putStrLn $ "--- proofs\n    \ ---"

    putStrLn $ "--- proofs\n    \ ---"
```

5.24.2 Interpreter launch

```
runc options path processes the file at path in the context of the user’s options.
```

References


Index

\( \rightarrow \), 2
\([m..n]\), 2
\(|S|\), 2
\(\Rightarrow\), 2
\(\lor\), 2
\(\neg\), 2
\(\rightarrow\), 2
\(\Rightarrow\), 2
\(\subseteq\), 2
\(\setminus\), 3
\{\}, 2
\(:*\), 18
\(:\neg\), 18
\(:\Rightarrow\), 15
\(:\neg\), 22
\(:\neg\), 22
\(<\), 21
\(\neg\), 20
\(\Rightarrow\), 25
\(\Rightarrow\), 15
\(\neg\), 18

\(\alpha\), 2, 3, 8, 27
acyclic, 2
\(\text{Alpha}\), 27
alphabet, 2
ant, 26
antecedent, 2, 7, 26, 31, 35, 42
syntax, 7, 8, 26
\(\text{A}(R)\), 2
\(\text{A}(r)\), 2, 26
argList
syntax, 6, 10
\(\text{Argument}\), 15
argument
syntax, 5, 15
\(\text{argumentP}\), 15
arguments
command line, 4, 12
in atoms, 4–6, 14–16, 18, 21
\(\text{Arguments}\)
module, 15
\(\text{arrow}(r)\), 2
assert
syntax, 8, 29
assertCheck, 31
\(\text{Atm}\), 2
\(\text{Atom}\), 16
atom, 15
syntax, 6, 16
\(\text{AtomMap}\), 17
atomName
syntax, 5, 6, 16
\(\text{atomNameP}\), 16
\(\text{atomP}\), 16
\(\text{Atoms}\)
module, 15
\(\text{AtomTypeDec}\), 21
\(\text{atomTypeDec}\)
syntax, 6, 7, 21
\(\text{atomTypeDecP}\), 21

Ax, 3, 34
axioms, 3, 4, 12, 28, 30, 31, 40
\(\text{Ax}(T)\), 3
\(\beta\), 2
\(\text{Beaten}\), 3
\(\text{Beta}\), 27
\(C[l]\), 2
\(\text{Clause}\), 24
clause, 2
syntax, 7, 24
\(\text{clauseP}\), 24
\(\text{Cls}\), 2
\(\text{CNF}\), 24
cnf-formula, 2
\(\text{CnfFormula}\), 24
cnfFormula
syntax, 7, 24
cnfFormulaP, 24
\(\text{CnfFrm}\), 2
\(\text{collecting}\)
constants, 15
variables, 15
comment1
syntax, 4, 12
comment2
syntax, 4, 12
comment2end
syntax, 4, 12
complement, 2
consequent, 2
\(\text{Const}\), 15
\(\text{Constant}\), 14
constant, 15
syntax, 5, 14
\(\text{constantP}\), 14
constants
collecting, 15
\(\text{Constants}\)
module, 14
contingent clause, 2
\(\epsilon(R)\), 2
\(\epsilon(r)\), 2
cyclic, 2
\(\delta\), 2
das, 28
dax, 28
\(\text{DCls}\), 2
ddef, 28
decisive plausible logic, 3
\(\text{Default}\), 21
default
syntax, 7, 21
defaultP, 21
\(\text{Defeat}\), 26
\(\text{Defeated}\), 3
defeater arrow, 2
defeater rule, 2
\(\text{Delta}\), 27
\(\text{Description}\)
description
syntax, 9, 30
descriptionP, 29
\(\text{Descriptions}\)
module, 28
dig, 28
dimp, 28
din, 28
\(\text{Disabled}\), 3
dnan, 28
dnf-formula, 2
\(\text{DoFrm}\), 2
dnt, 28
doProof, 40
doProofs, 40

November 25, 2013 49