Implementation of Decisive Plausible Logic (version 1.1)

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

Abstract

This is a Haskell implementation of Decisive Plausible Logic.[1] 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] DPL – an implementation of Decisive Plausible Logic.

1 Introduction

Plausible Logic [1, 2] is an improvement on Defeasible Logic.[3, 4] 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. This document describes an implementation of Decisive Plausible Logic similar to previous implementations of Defeasible Logic (*Deimos* [5, 6, 7]) and Plausible logic (*Phobos* [1, 2, 8], BPL [9], and CPL [10]).

Section 2 is a reformatting 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 is denoted by $X \subseteq Y$; the notation $X \subset Y$ means $X \subseteq Y$ and $X \neq Y$, and denotes that X is a *proper subset* of Y. The empty set is denoted by $\{\}$, 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, *Atm*, of (propositional) atoms; the set $\{\neg, \wedge, \vee,, \rightarrow, \Rightarrow, \rightharpoonup\}$ of connectives; the set $\{\mu, \alpha, \pi, \beta, \delta\}$ 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, a, or its negation, $\neg a$. A *clause*, $\vee L$, is the disjunction of a finite set, L, of literals. $\vee\{\}$ is the *empty clause* or *falsum* and is thought of as always being false. If l is a literal then we regard $\vee\{l\}$ as another notation for l and so each literal is a clause. A clause $\vee L$ 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 *dual-clause*, $\wedge L$, is the conjunction of a finite set, L, of literals. $\wedge\{\}$ is the *empty dual-clause* or *verum* and is thought of as always being true. If l is a literal then we regard $\wedge\{l\}$ as another notation for l and so each literal is a dual-clause. Thus $\wedge\{l\} = l = \vee\{l\}$. 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*, $\wedge C$, is the conjunction of a finite set, C, of clauses. A *dnf-formula*, $\vee D$, is the disjunction of a finite set, D, of dual-clauses. If c is a clause then we regard $\wedge\{c\}$ as another notation for c. If d is a dual-clause then we regard $\vee\{d\}$ as another notation for d. Thus both clauses and dual-clauses are both cnf-formulas and dnf-formulas. By a *formula* we mean any cnf-formula or any dnf-formula. A *tagged formula* is a formula preceded by a tag; so all tagged formulas have the form λf where $\lambda \in \{\mu, \alpha, \pi, \beta, \delta\}$, and f is a formula. The set of all literals is denoted by *Lit*; the set of all clauses is denoted by *Cls*; the set of all dual-clauses is denoted by *DCls*; the set of all cnf-formulas is denoted by *CnfFrm*; the set of all dnf-formulas is denoted by *DnfFrm*; and the set of all formulas is denoted by *Frm*.

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 $\sim f$ is $\neg f$; and $\sim\neg f$ is f. If L is a set of literals then $\sim L = \{\sim l : l \in L\}$. If $\vee L$ is a clause then $\sim\vee L = \wedge\sim L$. If $\wedge L$ is a dual-clause then $\sim\wedge L = \vee\sim L$. If E is a set of clauses or a set of dual-clauses then $\sim E = \{\sim e : e \in E\}$. If $\wedge C$ is a cnf-formula then $\sim\wedge C = \vee\sim C$. If $\vee D$ is a dnf-formula then $\sim\vee D = \wedge\sim D$. If F is a set of formulas then $\sim F = \{\sim f : f \in F\}$.

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

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_s, the set of plausible rules in R by R_p, and the set of defeaters in R by R_d. Also we define $R_{pd} = R_p \cup R_d$ and $R_{sp} = R_s \cup R_p$.

Let l be any literal. If C is any set of clauses define $C[l] = \{\vee L \in C : l \in L\}$ 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 $R[l] = \{r \in R : l = c(r)\}$ 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 $r_1 > r_2 > \ldots > r_n > r_1$. A relation is *acyclic* iff it is not cyclic.

If R is a set of rules then $>$ is a priority relation on R iff $>$ is an acyclic binary relation on R such that $>$ 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 $R[l;s] = \{t \in R[l] : t > s\}$ be the set of all rules in R with consequent l that beat s.

A *plausible description* of a situation is a 4-tuple $PD = (Ax, R_p, R_d, >)$ 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_d 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 $[1..n]$, 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 $\mathit{Res}(S)$. So $S \subseteq \mathit{Res}(S)$. Define $\mathit{Rsn}(S) = \mathit{Res}(S) - \{\vee\{\}\}$ to be the set of all non-empty clauses in $\mathit{Res}(S)$. Define $\mathit{Min}(S) = \{\vee L \in S : \text{if } K \subset L \text{ then } \vee K \notin S\}$. Then $\mathit{Min}(S)$ is the set of minimal clauses in S.

Let $PD = (Ax, R_p, R_d, >)$ be a plausible description. Define the set of *minimal contingent facts*, $\mathit{Fct}(Ax)$, generated from Ax by $\mathit{Fct}(Ax) = \mathit{Min}(\mathit{Rsn}(Ax)) - \{\vee L : \vee L \text{ is a tautology}\}$. The set of strict rules, R_s, generated from $\mathit{Fct}(Ax)$ is defined by $R_s = \{\sim(L - \{l\}) \rightarrow l : l \in L \text{ and } \vee L \in \mathit{Fct}(Ax)\}$. Define $R = R_s \cup R_p \cup R_d$ to be the set of rules *generated* from PD. The ordered pair $(R, >)$ is called a *plausible theory*.

Let $T = (R, >)$ be a plausible theory. The set of axioms in the plausible description from which T was generated is denoted by $\mathit{Ax}(T)$. Define $\mathit{Fct}(T) = \{\vee(\{c(r)\} \cup \sim A(r)) : r \in R_s\}$. Then $\mathit{Fct}(T)$ is the set from which R_s was generated and so $\mathit{Fct}(T) = \mathit{Fct}(\mathit{Ax}(T))$. If L is a set of literals then define $\mathit{Fct}(T;L) = \{\vee K \in \mathit{Fct}(T) : |K \cap L| \geq 2\}$. Define $\mathit{Inc}(T) = \{\sim L : \vee L \in \mathit{Min}(\mathit{Fct}(T) \cup \vee\{k, \sim k : k \in \mathit{Atm}\})\}$ to be the set of non-empty sets of literals which are inconsistent with the axioms of T. Hence $\{\} \notin \mathit{Inc}(T)$ and $\mathit{Inc}(T) \neq \{\}$. Define $\mathit{Inc}(T, l) = \{I - \{l\} : I \in \mathit{Inc}(T) \text{ and } l \in I\}$. Each member of $\mathit{Inc}(T, l)$ is a minimal set of literals which is *inconsistent with* l.

2.2 Decisive Plausible Logic

Given a plausible theory $T = (R, >)$ we define the following ten functions P, *Strict*, *Plaus*, *For*, *Nullified*, *Disabled*, *Discredited*, *Defeated*, *Beaten*, and *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 λf and any set B of literals and returns a result in {-1, 0, +1}. *Strict* takes any tagged literal λl and any set B of literals such that $l \notin B$ and returns a result in $\{-1, 0, +1\}$. *Plaus* takes any tagged literal λl 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 *Nullified* have the same domain and range as *Plaus*. *Disabled* takes any tagged literal λl such that $\lambda \neq \mu$ and any set B of literals such that $l \notin B$ and any set J of literals and returns a result in $\{-1, 0, +1\}$. *Discredited* takes any tagged literal λl such that $\lambda \neq \mu$ and any set B of literals such that $l \notin B$ and any literal j and returns a result in $\{-1, 0, +1\}$. *Defeated* takes any tagged literal λl such that $\lambda \neq \mu$ and any set B of literals such that $l \notin B$ and any rule s and returns a result in $\{-1, 0, +1\}$. *Beaten* and *Inappl* have the same domain and range as *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 $\max\{\} = -1$.

$\wedge$) $P(\lambda \wedge C, B) = \min\{P(\lambda c, B) : c \in C\}$.

$\vee$.1) If $\vee L$ is a tautology then $P(\lambda \vee L, B) = +1$.

$\vee$.2) If $\vee L$ is not a tautology then $P(\lambda \vee L, B) =$
$\max(\{P(\lambda l, B) : l \in L\}$
$\cup\{P(\lambda \wedge \sim(K - L), B) : \vee K \in \mathit{Fct}(T; L)\})$.

$\mathcal{L}$.1.1) If $l \in B$ then $P(\lambda l, B) = 0$
$\mathcal{L}$.1.2) If $l \notin B$ then $P(\lambda l, B) = \max\{\mathit{Strict}(\lambda l, B), \mathit{Plaus}(\lambda l, B)\}$.
$\mathcal{L}$.2) $\mathit{Strict}(\lambda l, B) = \max\{P(\lambda \wedge A(r), B \cup \{l\}) : r \in R_s[l]\}$.
$\mathcal{L}$.3) $\mathit{Plaus}(\lambda l, B) = \min\{\mathit{For}(\lambda l, B), \mathit{Nullified}(\lambda l, B)\}$.
$\mathcal{L}$.4) $\mathit{For}(\lambda l, B) = \max\{P(\lambda \wedge A(r), B \cup \{l\}) : r \in R_p[l]\}$.
$\mathcal{L}$.5) $\mathit{Nullified}(\lambda l, B) = \min\{\mathit{Disabled}(\lambda l, B, J) : J \in \mathit{Inc}(T, l)\}$.
$\mathcal{L}$.6) $\mathit{Disabled}(\lambda l, B, J) = \max\{\mathit{Discredited}(\lambda l, B, j) : j \in J\}$.
$\mathcal{L}$.7) $\mathit{Discredited}(\lambda l, B, j) = \min\{\mathit{Defeated}(\lambda l, B, s) : s \in R[j]\}$.
$\mathcal{L}$.8) $\mathit{Defeated}(\lambda l, B, s) =$
$\max\{\mathit{Beaten}(\lambda l, B, s), \mathit{Inappl}(\lambda l, B, s)\}$.
$\mathcal{L}$.9) $\mathit{Beaten}(\lambda l, B, s) = \max\{P(\lambda \wedge A(t), B \cup \{l\}) : t \in R_p[l; s]\}$.
$\mathcal{L}$.10.α) $\mathit{Inappl}(\alpha l, B, s) =$
$\min\{P(\alpha \sim \wedge A(s), B \cup \{l\}), -P(\alpha \wedge A(s), B \cup \{l\})\}$.
$\mathcal{L}$.10.π) $\mathit{Inappl}(\pi l, B, s) = P(\pi \sim \wedge A(s), B \cup \{l\})$.
$\mathcal{L}$.10.β) $\mathit{Inappl}(\beta l, B, s) = -P(\beta \wedge A(s), B \cup \{l\})$.
$\mathcal{L}$.10.δ) $\mathit{Inappl}(\delta l, B, s) =$
$\max\{P(\delta \sim \wedge A(s), B \cup \{l\}), -P(\delta \wedge A(s), B \cup \{l\})\}$.

In $\mathcal{L}$.1.2, if $\lambda = \mu$ then $P(\mu l, B) = \max\{\mathit{Strict}(\mu l, B)\} = \mathit{Strict}(\mu l, B)$.

Suppose T is a plausible theory, P is the proof function of T, f is a cnf-formula, and $\lambda \in \{\mu, \alpha, \pi, \beta, \delta\}$. The deducibility relation $\vdash$ is defined by $T \vdash +\lambda f$ iff $P(\lambda f, \{\}) = +1$, $T \vdash 0\lambda f$ iff $P(\lambda f, \{\}) = 0$, $T \vdash -\lambda f$ iff $P(\lambda f, \{\}) = -1$. We define $T(+\lambda) = \{f : T \vdash +\lambda f\}$, $T(0\lambda) = \{f : T \vdash 0\lambda f\}$, and $T(-\lambda) = \{f : T \vdash -\lambda f\}$. A *decisive plausible logic* consists of a plausible theory and its proof function.

2.3 Summary

A plausible description contains:

- definite facts, expressed as clauses;
- plausible rules ($\Rightarrow$);
- defeater rules ($\rightharpoonup$); and
- priorities to resolve conflicts between rules.

A plausible theory is derived from a plausible description by the computation of the strict rules ($\rightarrow$) that ensue from the definite facts.

Evaluation of $P(\lambda f, \{\})$, where $\lambda \in \{\mu, \alpha, \pi, \beta, \delta\}$ is a tag indicating the proof algorithm required, and f is a cnf-formula, returns either:

- +1, which means λf was proved;
- -1, which means there is no proof of λf; or
- 0, which means λf could not be decided due to looping.

The meaning of the proof algorithm tags is, in a roughly descending order of strictness/certainty:

- μ – monotonic, strict, like classical logic;
- α – the conjunction of π and β;
- π – plausible, probable, and propogates ambiguity;
- β – plausible, blocking ambiguity (as per the original Defeasible Logic of Nute[3]); and
- δ – the disjunction of π and β.

3 Installation

3.1 Obtaining this system

This system is available for download at `http://www.cit.gu.edu.au/~arock/`.

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 data 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 vaiables and obviating facts and the rules that use them;
- prints the instantiated, obviated description;
- prints the theory that is derived from that description;
- prints the combinations of input axioms with which the base theory will be augmented;
- 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.

4.2 Description file syntax

A DPL input file contains:

- a plain text representation of a plausible description, consisting of
  - axioms (strictly asserted facts expressed as clauses);
  - plausible rules;
  - defeater rules;
  - priority declarations;
- 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.

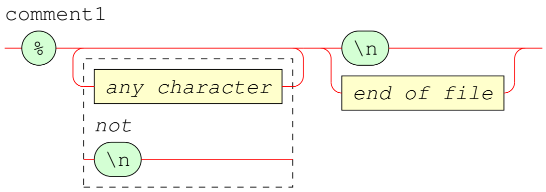
The rest of this section defines the syntax of a description file, working from the lowest levels, up. It end with some example description files.

Whitespace and comments

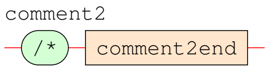
Any amount of whitespace is permitted before and after any symbol. Comments are treated as whitespace. There are two types:

- 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.

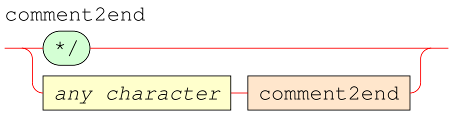
```
comment1 ::= "%" {<$any character$ ! "\\n">}
   ("\\n" | $end of file$);
level="lexical".
```



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



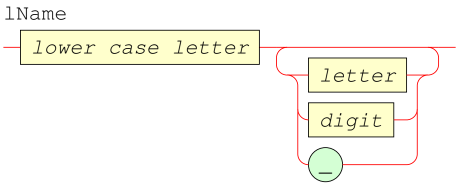
```
comment2end ::= "*/" | $any character$ comment2end;
level="lexical".
```



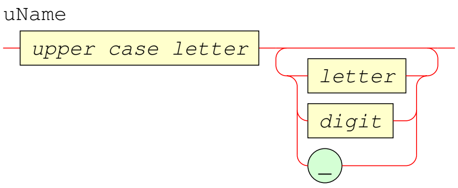
Names

Atoms and rule labels in a description 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".
```



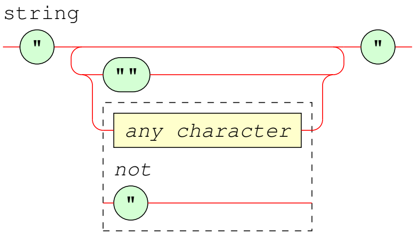
```
uName ::= $upper case letter$ {$letter$ | $digit$ | "_"};
level="lexical".
```



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.

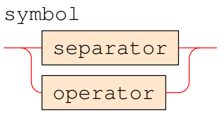
```
string ::= "\"" {"\"\"" | <$any character$ ! "\"">} "\"";
level="lexical".
```



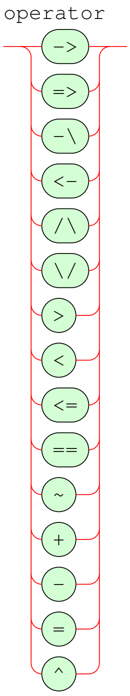
Symbols

These are the special sybmols on the language.

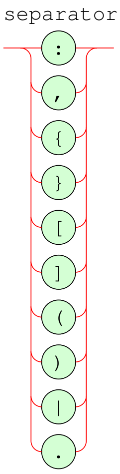
```
symbol ::= separator | operator;
level="lexical".
```



```
operator ::=   "->" | "=>" | "-\\" | "<-" | "/\\" | "\\/"
             | ">"  | "<"  | "<="  | "==" | "~"  | "+"
             | "-"  | "="  |  "^";
level="lexical".
```



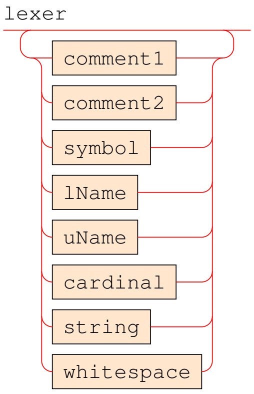
```
separator ::=   ":" | "," | "{" | "}" | "[" | "]"
              | "(" | ")" | "|" | ".";
level="lexical".
```



The complete lexical syntax

A DPL source consists of zero or more of the following kinds of tokens.

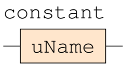
```
lexer ::= {comment1 | comment2 | symbol | lName | uName
  | cardinal | string | whitespace};
level="lexical".
```



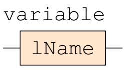
Constants and variables as arguments

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.

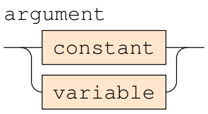
```
constant ::= uName;
level="grammar".
```



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



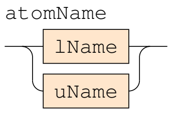
```
argument ::= constant | variable;
level="grammar".
```



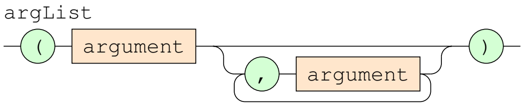
Atoms

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 `==`.

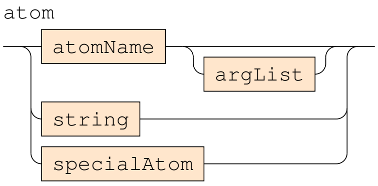
```
atomName ::= lName | uName;
level="grammar".
```



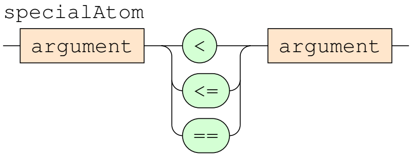
```
argList ::= "(" argument {"," argument} ")";
level="grammar".
```



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



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



Example atoms:

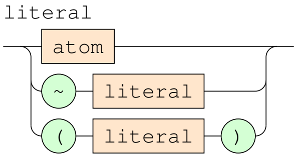
```
p                   p(A,B,c)
proposition_13      Proposition14(Const1,Const2,var_1)
"x > 0"             "command.equals(""sit"")"
```

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:

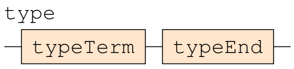
`p` `~p` `~(~p)` `p(A,B,c)` `~p(A,B,c)` `~"x > 3"`

Types

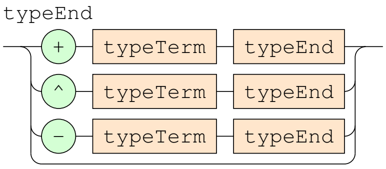
A type is a set of constants. Types are used to constrain the instantiation of declarations containing variables.

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.

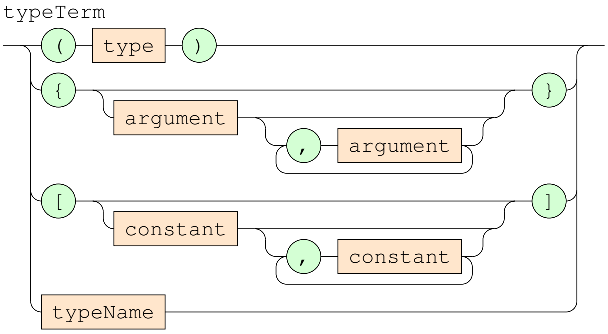
```
type ::= typeTerm typeEnd;
level="grammar".
```



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



```
typeTerm ::=   "(" type ")"
             | "{" [argument {"," argument}] "}"
             | "[" [constant {"," constant}] "]"
             | typeName;
level="grammar".
```

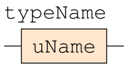


The use of delimiters `[` and `]`, rather than `{` and `}`, implies that the constants in that sequence form a total order. This causes the automatic generation of a lot of axioms of the forms x `<` y, x `<=` y and x `==` y.

Declaring named types

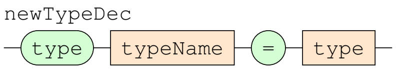
A type's name always starts with an upper case letter.

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



A named type is defined with a declaration of the following form, with the restriction that the arguments appearing in the enumerations are all constants and not variables.

```
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}.
```

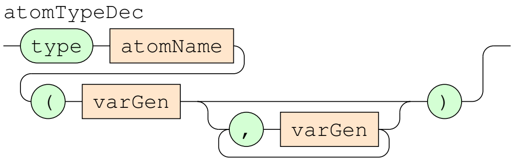
The Universe

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

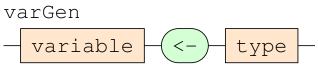
Asserting atom argument types

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:

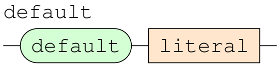
```
type eats(a <- Animal, o <- Organism).
type enemy(a1 <- Animal, a2 <- Animal - {a1}).
type boss(a1 <- Animal, a2 <- Animal, boss <- {a1, a2}).
type likes(a <- Animal, x <- Universe).
```

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".
```



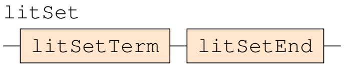
The following default facts are automatically asserted:

```
default ~(x < y).
default ~(x <= y).
default ~(x == y).
```

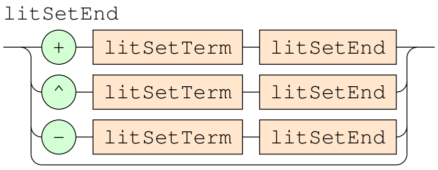
Literal sets

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.

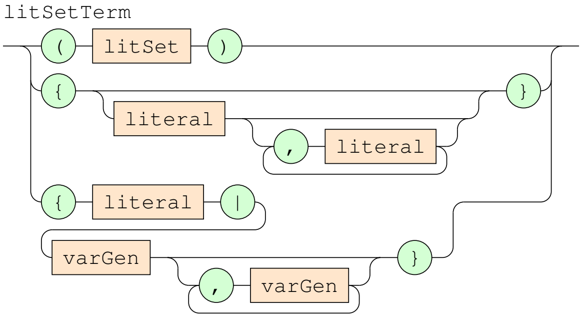
```
litSet ::= litSetTerm litSetEnd;
level="grammar".
```



```
litSetEnd ::=   "+" litSetTerm litSetEnd
              | "^" litSetTerm litSetEnd
              | "-" litSetTerm litSetEnd
              | $epsilon$;
level="grammar".
```



```
litSetTerm ::=   "(" litSet ")"
               | "{" [literal {"," literal }] "}"
               | "{" literal "|" \ varGen {"," varGen}"}";
level="grammar".
```



Example literal sets:

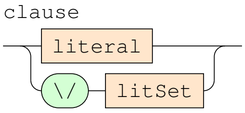
```
{}
{p, q, r}
{p} + {q, r}
{p, q, r} - {q}
({p} + {q, r}) - {q, r}
{see(x)} + {~see(y) | y <- Thing - {x}}
{see(x)} + {~see(y) | y <- Universe - {x}}
```

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

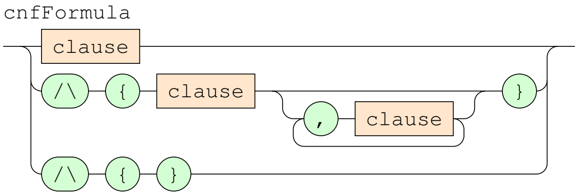
A clause is the disjunction (`\/` for $\bigvee$) of a set of literals. A literal is also a clause, as $\bigvee\{l\} = l$. `\/{}` (for $\bigvee\{\}$) is the empty clause, which is false.

```
clause ::= literal | "\\/" litSet;
level="grammar".
```



A cnf-formula is the conjunction (`/\` for $\bigwedge$) of a set of clauses. A clause is a cnf-formula, as $\bigwedge\{c\} = c$.

```
cnfFormula ::=   clause
               | "/\\" "{" clause {"," clause} "}"
               | "/\\" "{" "}";
level="grammar".
```



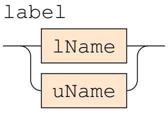
Example clauses and cnf-formulas:

clauses	*cnf-formulas*
`\/{}`	`/\{}`
`p`	`p`
`\/{p}`	`/\{p}`
`\/{a, b, ~c}`	`/\{\/{a, b}, ~c}`

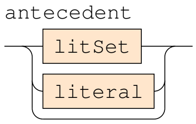
Rules

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.

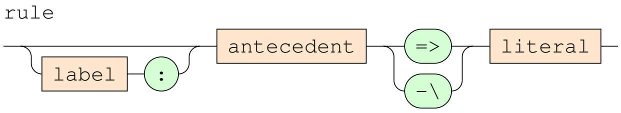
```
label ::= lName | uName;
level="grammar".
```



```
antecedent ::= litSet | literal | $epsilon$;
level="grammar".
```



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



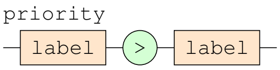
Example rules:

formal	*DPL*
$\{\} \Rightarrow p$	`{} => p`
$\{\} \Rightarrow p$	`=> p`
$\{a\} \rightharpoonup \neg b$	`{a} -\ ~b`
$\{a\} \rightharpoonup \neg b$	`a -\ ~b`
$\{a, b, c\} \rightharpoonup \neg d$	`{a, b, c} -\ ~d`

Priorities

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

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



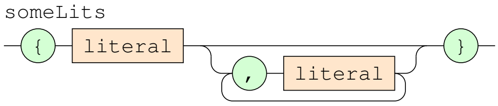
Example:

`R1 > R2`

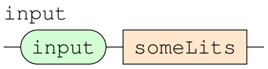
Input and ignore specifications

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.

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

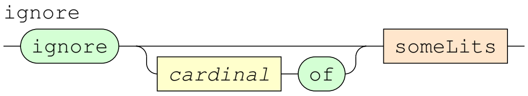


```
input ::= "input" someLits;
level="grammar".
```



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

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



Examples:

inputs and ignores	*axioms generated*

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

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

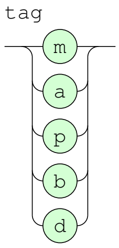
`input{a}.  input{b}.`
`ignore{a, ~b}.`
	`a.  b.`
	`~a.  b.`
	`~a.  ~b.`

`input{a}.  input{b}.`
`input{c}.  input{d}.`
`ignore 3 of {a, b, c, d}.`
	`a.  b.  ~c.  ~d.`
	`a.  ~b.  c.  ~d.`
	`a.  ~b.  ~c.  d.`
	`a.  ~b.  ~c.  ~d.`
	`~a.  b.  c.  ~d.`
	`~a.  b.  ~c.  d.`
	`~a.  b.  ~c.  ~d.`
	`~a.  ~b.  c.  d.`
	`~a.  ~b.  c.  ~d.`
	`~a.  ~b.  ~c.  d.`
	`~a.  ~b.  ~c.  ~d.`

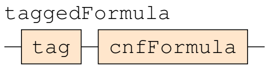
Tagged cnf-formulas

A tagged cnf-formula consists of: a proof level (`m` for μ, `a` for α, `p` for π, `b` for β, `d` for δ); and a cnf-formula.

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



```
taggedFormula ::= tag cnfFormula;
level="grammar".
```



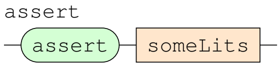
Examples:

`d /\{}`   `p p`   `p /\{p}`   `a /\{\/{a,b},~c}`

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 nagatively. There may be multiple assert declarations, each specifying a combination with which to perform proofs.

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

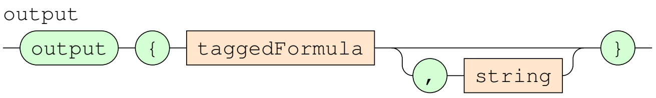


Assert declarations will be useful for testing descriptions with very many inputs, leading to too many combinations to performs proofs for. Their use will likely be temporary, it is recommended that they be included using the `-a` *`filename`* option.

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.

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



Examples:

```
output{p p}            output{a /\{\/{a,b},~c}}
output{p p, "IS_P"}    output{a /\{\/{a,b},~c}, "A_OR_B(X)"}
```

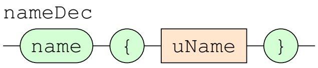
Name declaration

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 me used as the generated module name or as a part of all generated global definitions.

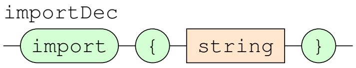
```
nameDec ::= "name" "{" uName "}";
level="grammar".
```



Example: `name{Nanook}`

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

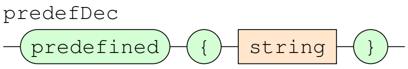
```
importDec ::= "import" "{" string "}";
level="grammar".
```



Example: `import{"package.Class"}`

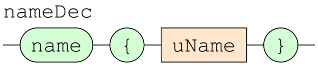
- The following declares that some token is assumed to be predefined in the target language.

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



Example `predefined{"elem"}`

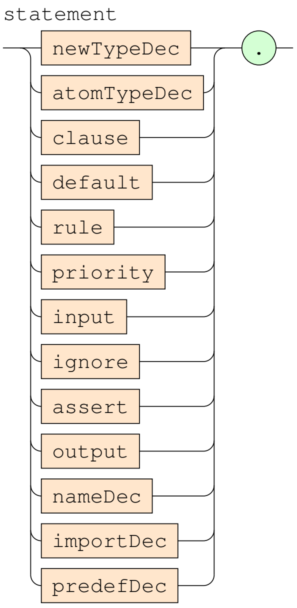
```
nameDec ::= "name" "{" uName "}";
level="grammar".
```



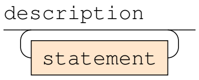
Description file

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 decarations for testing, and/or a name declaration and import and predefined token declarations for code generation. Each must be terminated with a period.

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



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



A description file typically has a name ending in ".d". Examples:

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

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

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

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

% If T > 25C then cool.
R2: tempGT25 => cool.
R2 > R1.

% 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}.

output{a cool}.
output{a ~cool}.
```

```
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)).
\/{~m(x), d(x)}.

% Siberian huskies are dogs (sh(x) -> d(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)}.

output{d b(N)}.
output{d ~b(N)}.

output{m f(R,N)}.
output{m ~f(R,N)}.

output{a f(R,N)}.
output{a ~f(R,N)}.

output{p f(R,N)}.
output{p ~f(R,N)}.

output{b f(R,N)}.
output{b ~f(R,N)}.

output{d f(R,N)}.
output{d ~f(R,N)}.

output{m f(N,R)}.
output{m ~f(N,R)}.

output{a f(N,R)}.
output{a ~f(N,R)}.

output{p f(N,R)}.
output{p ~f(N,R)}.

output{b f(N,R)}.
output{b ~f(N,R)}.

output{d f(N,R)}.
output{d ~f(N,R)}.
```

```
/*
** 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}.
```

```
% win(x) means x has won a game.
type win(x <- Player).

% shows (x, y) means player x is showing sign y.
type shows(x <- Player, y <- Sign).

% Equality.
default ~eq(x,y).
eq(x,x).

% What beats what.
default ~beats(x, y).
beats(Paper, Rock).
beats(Rock, Scissors).
beats(Scissors, Paper).

% By default, a player has not won.
DefaultWin: {} => ~win(x).

% By default, there is no tie.
DefaultTie: {} => ~tie.

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

% If noone wins, it is a tie.
Tie: {~win(x) | x <- Player} => tie.
Tie > DefaultTie.

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

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

```
/*
** file: JankenRules.d
**
** purpose: The rules for the Janken (paper, scissors,
**          rock) game.
**          This version can be used with DPL to output a
**          Haskell glue module.
*/

name{JankenRules}.

import{"JankenConstants"}.

 % The 3 hand signs.
type Sign = {Paper, Scissors, Rock}.

% The two players.
type Player = {A, B}.

% win(x) means x has won a game.
type win(x <- Player).

% shows (x, y) means player x is showing sign y.
type shows(x <- Player, y <- Sign).

% Equality.
default ~eq(x,y).
eq(x,x).

% What beats what.
default ~beats(x, y).
beats(Paper, Rock).
beats(Rock, Scissors).
beats(Scissors, Paper).

% By default, a player has not won.
DefaultWin: {} => ~win(x).

% By default, there is no tie.
DefaultTie: {} => ~tie.

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

% If noone wins, it is a tie.
Tie: {~win(x) | x <- Player} => tie.
Tie > DefaultTie.

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

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

The last example description is indented to be usd 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 `JankenRules.d` description file.

```
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)
```

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

```
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]
         putStr $ makeTableL ’ ’ $ transpose $
            (headings :) $ results
```

The program prints:

```
A        B        A wins? B wins? Tie?
Paper    Paper    -1      -1      +1
Paper    Scissors -1      +1      -1
Paper    Rock     +1      -1      -1
Scissors Paper    +1      -1      -1
Scissors Scissors -1      -1      +1
Scissors Rock     -1      +1      -1
Rock     Paper    -1      +1      -1
Rock     Scissors +1      -1      -1
Rock     Rock     -1      -1      +1
```

4.3 Invoking the DPL tool

Type the command:

```
DPL <options> <fileNames>
```

where the command line arguments include: *`fileNames`* – the names of the description files to process; and *`options`* – as follows:

- `+v` – be verbose, show progress information, and proof traces (the default);
- `-v` – don't be verbose;
- `-a` *`filename`* – append the contents of the named file to the descriptions before all processing (May be specified multiple times to append multiple files.);
- `+i` – show the combinations of input axioms with which proofs will be attempted (default if verbose, overrides `-v`);
- `-i` – don't show the combinations of input axioms with which proofs will be attempted (default if not verbose, overrides `+v`);
- `+p` – do proofs (the default);
- `-p` – don't do proofs (overrides `+t` and `+c`);
- `+q` – do proofs quickly using optimised theory data structures (overrides `+v`);
- `-q` – do proofs slowly using simple theory data structures (the default);
- `+t` – show the proof result summary table (the default);
- `-t` – don't show the proof result summary table;
- `+c` – convert results to C expressions;
- `-c` – don't convert results to C expressions (the default);
- `+s` – simplify the C expressions (the default);
- `-s` – don't simplify the C expressions;
- `+C` – export the theory as C data structures;
- `-C` – don't export the theory as C data structures (the default).
- `+h` – create a Haskell glue module; and
- `-h` – don't create a Haskell glue module (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 `typewriter` font, as are syntax specifying productions. Productions use the `::=` symbol and are commentary material, not formal Haskell code. The source code for the Haskell modules have been written in the literate style, and the following subsections have been produced directly from the Haskell+LaTeX source code.

5.1 DPLLexer

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

```
module DPL.DPLLexer (lexerL) where
```

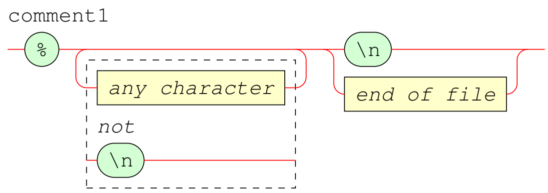
```
import Data.Char
```

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

5.1.1 Comments

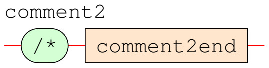
Comments in plausible sources follow the Prolog conventions. Comments that start with a percent sign (%) extend to the end of the line. Comments that start with the sequence `/*` extend to the next sequence `*/` and may span more than one line. Formally, the syntax for each type of comment is:

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

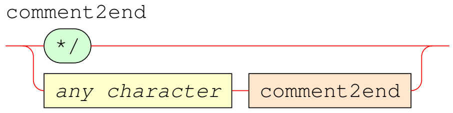


```
comment1L :: Lexer
comment1L =
   tokenL "%"
   <**> (many (satisfyL (/= ’\n’) "") *%> "")
   <**> (optional (literalL ’\n’) *%> "")
   %> " "
```

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



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

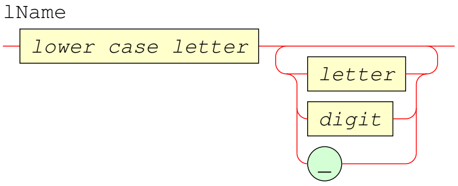


```
comment2L :: Lexer
comment2L =
   tokenL "/*" <**> comment2Lend %> " "
   where
   comment2Lend :: Lexer
   comment2Lend =
          tokenL "*/"
      <|> (satisfyL (\c -> True) "") <**> comment2Lend
```

5.1.2 Names

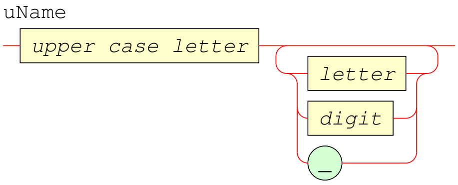
Literals, rule labels, constants and variables are all instances of names that occur in plausible sources. Two types are distinguished: those starting with lower case letters; and those starting with upper-case letters. Formally, the syntax for each type of name is:

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



```
lNameL :: Lexer
lNameL =
   (satisfyL isLower "lower-case letter" <**>
    ((many (satisfyL isNameChar "letter, digit, _"))
      *%> ""))
   %> "lName"
   where
   isNameChar c = isAlpha c || isDigit c || c == ’_’
```

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

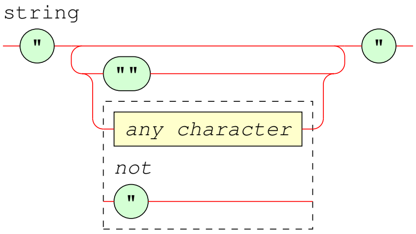


```
uNameL :: Lexer
uNameL =
   (satisfyL isUpper "upper-case letter" <**>
    ((many (satisfyL isNameChar "letter, digit, _"))
     *%> ""))
   %> "uName"
   where
   isNameChar c = isAlpha c || isDigit c || c == '_'
```

5.1.3 Strings

Atoms may be defined as arbitrary strings.

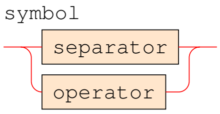
```
string ::= "\"" {"\"\"" | <$any character$ ! "\"">} "\"";
level="lexical".
```



`stringL` is implemented in `ABR.Configs`.

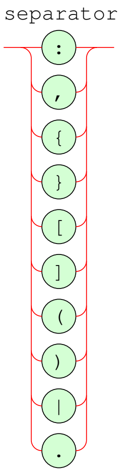
5.1.4 Symbols and everything else

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



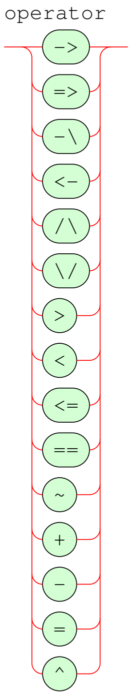
```
symbolL :: Lexer
symbolL = operatorL <|> separatorL
```

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



```
separatorL :: Lexer
separatorL =
       tokenL ":"   %> "symbol"
   <|> tokenL ","   %> "symbol"
   <|> tokenL "{"   %> "symbol"
   <|> tokenL "}"   %> "symbol"
   <|> tokenL "["   %> "symbol"
   <|> tokenL "]"   %> "symbol"
   <|> tokenL "("   %> "symbol"
   <|> tokenL ")"   %> "symbol"
   <|> tokenL "|"   %> "symbol"
   <|> tokenL "."   %> "symbol"
```

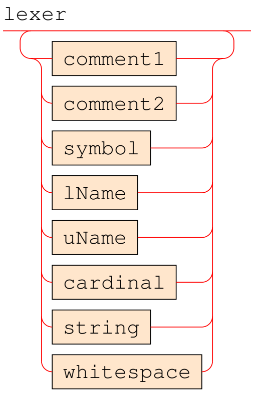
```
operator ::=   "->" | "=>" | "-\\" | "<-" | "/\\" | "\\/"
             | ">"  | "<"  | "<="  | "==" | "~"   | "+"
             | "-"  | "="  |  "^";
level="lexical".
```



```
operatorL :: Lexer
operatorL =
       tokenL "->"  %> "symbol"
   <|> tokenL "=>"  %> "symbol"
   <|> tokenL "-\\" %> "symbol"
   <|> tokenL "<-"  %> "symbol"
   <|> tokenL "/\\" %> "symbol"
   <|> tokenL "\\/" %> "symbol"
   <|> tokenL "<="  %> "symbol"
   <|> tokenL "<"   %> "symbol"
   <|> tokenL "=="  %> "symbol"
   <|> tokenL ">"   %> "symbol"
   <|> tokenL "~"   %> "symbol"
   <|> tokenL "+"   %> "symbol"
   <|> tokenL "-"   %> "symbol"
   <|> tokenL "="   %> "symbol"
   <|> tokenL "^"   %> "symbol"
```

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

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



```
lexerL :: Lexer
lexerL = listL [comment1L, comment2L, symbolL, lNameL,
      uNameL, cardinalL, stringL, whitespaceL
   ]
```

5.2 Constants

Module `Constants` implements constants for Decisive Plausible Logic.

```
module DPL.Constants (
      Constant(..), constantP, HasConstants(..)
   ) where
```

```
import Control.DeepSeq
import qualified Data.Set as S
```

```
import ABR.Parser
```

5.2.1 Data type

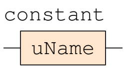
An `Constant` is a fixed token which may appear as an argument to an atom.

```
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` x C adds any constants in x to C.

```
   getConstants ::
      a -> S.Set Constant -> S.Set Constant
```

`hasConstants` x returns `True` iff x contains constants.

```
   hasConstants :: a -> Bool
   hasConstants 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.

```
module DPL.Variables (
      Variable(..), variableP, HasVariables(..),
      Substitution(..), Groundable(..)
   ) where
```

```
import Control.DeepSeq
import qualified Data.Set as S
```

```
import ABR.Parser
```

```
import DPL.Constants
```

5.3.1 Data type

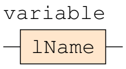
An `Variable` is a token which may appear as an argument to an atom, to be instatiated with constants.

```
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` x V adds any variables in x to V.

```
   getVariables ::
      a -> S.Set Variable -> S.Set Variable
```

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

```
   hasVariables :: a -> Bool
   hasVariables 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 `:->-` c replaces a variable v with a constant c. Substitutions may be composed. s_1 `:>->` s_2 first performs s_1 and then s_2. `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` v c x returns x with all occurrences of variable v replaced by constant c.

```
   ground1 :: Variable -> Constant -> a -> a
```

`ground` s x applies substitution s to x.

```
   ground :: Substitution -> a -> a
   ground s x = case s of
      NullSub    -> x
      v :->- c   -> ground1 v c x
      s1 :>-> s2 -> ground s2 $ ground s1 x
```

`rename` v v' x returns x with all occurrences of variable v replaced by another variable v'.

```
   rename :: Variable -> Variable -> a -> a
```

5.3.5 Instance declarations

Showing

```
instance Show Variable where

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

Collecting Variables

```
instance HasVariables Variable where

   getVariables = S.insert
```

Grounding

```
instance Groundable a => Groundable [a] where

   ground1 v c = map (ground1 v c)

   rename v v' = map (rename v v')
```

NFData (for deepseq)

```
instance NFData Variable where

   rnf (Variable v) = v `deepseq` ()
```

5.4 Arguments

Module `Arguments` implements arguments for Decisive Plausible Logic.

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

import Control.DeepSeq

import ABR.Parser

import DPL.Constants
import DPL.Variables
```

5.4.1 Data type

An `Argument` of an atom may be either:

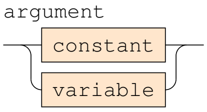
- a constant (`Const`); or
- a variable (`Var`).

```
data Argument =   Const Constant
                | Var   Variable
                deriving (Eq, Ord)
```

5.4.2 Parsers

The syntax for an argument is:

```
argument ::= constant | variable;
level="grammar".
```



which is implemented by `argumentP`.

```
argumentP :: Parser Argument
argumentP = (
          constantP @> Const
      <|> variableP @> Var
   )
```

5.4.3 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 v c a = case a of
      Const c'              -> Const c'
      Var   v' | v == v'    -> Const c
               | otherwise -> Var   v'

   rename v v' a = case a of
      Const c                -> Const c
      Var   v'' | v == v''   -> Var v'
                | otherwise -> Var v''
```

NFData (for deepseq)

```
instance NFData Argument where

   rnf a = case a of
      Const c -> c `deepseq` ()
      Var   v -> v `deepseq` ()
```

5.5 Atoms

Module `Atoms` implements atoms for Decisive Plausible Logic.

```
module DPL.Atoms (
      Atom(..), atomNameP, atomP, HasAtoms(..)
   ) where

import Control.DeepSeq
import qualified Data.Set as S

import ABR.Parser
import ABR.Text.Showing
import ABR.Data.List

import DPL.Constants
import DPL.Variables
import DPL.Arguments
```

5.5.1 Data type

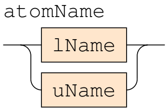
An `Atom` is a proposition symbol, `Prop`. An atom may have a list of arguments.

```
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`.

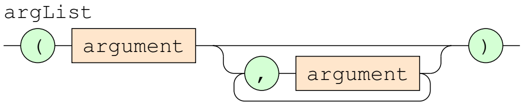
```
atomName ::= lName | uName;
level="grammar".
```



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

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

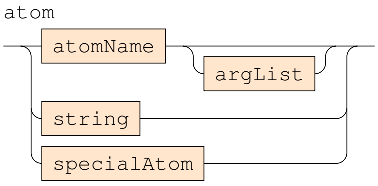
```
argList ::= "(" argument {"," argument} ")";
level="grammar".
```



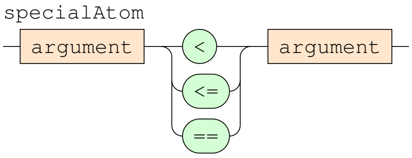
```
argListP :: Parser [Argument]
argListP =
   literalP "symbol" "("
    *> argumentP
   <*> many (literalP "symbol" "," *> argumentP)
   <*  nofail (literalP "symbol" ")")
   @> cons
```

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

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



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



which is implemented by `atomP`.

```
atomP :: Parser Atom
atomP =
           argumentP
       <*> (     literalP "symbol" "<"
             <|> literalP "symbol" "<="
             <|> literalP "symbol" "=="
           )
       <*> nofail argumentP
       @> (\(a1,((_,op,_),a2)) -> Prop op [a1,a2])
   <|>     atomNameP
       <*> optional argListP
       @> (\(n,ass) -> case ass of
              []   -> Prop n []
              [as] -> Prop n as
           )
   <|> tagP "string"
       @> (\(_,s,_) -> Prop s [])
```

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`.

```
class HasAtoms a where
```

`getAtoms` *x A* adds any atoms in *x* to *A*.

```
   getAtoms :: a -> S.Set Atom -> S.Set Atom
```

5.5.4 Instance declarations

Showing

```
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 ')'
```

Collecting Atoms

```
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
```

NFData (for deepseq)

```
instance NFData Atom where

   rnf (Prop n as) = n `deepseq` (as `deepseq` ())
```

5.6 OAtoms

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

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

import Data.Array.IArray
import qualified Data.Map as M
import qualified Data.Set as S

import DPL.Atoms
```

5.6.1 Data types

An `OAtom` is a *positive* integer that uniquely identifies a correspoding `Atom`.

```
type OAtom = Int
```

An `AtomMap` maps `Atom`s to their corresponding `OAtom`s.

```
type AtomMap = M.Map Atom OAtom
```

An `OAtomMap` maps `OAtom`s to their corresponding `Atom`s.

```
type OAtomMap = Array Int Atom
```

5.6.2 Building atom maps

`mkAtomMaps` x returns (N, am, oam), where x is something that contains N distinct `Atom`s, and for each of them, (am maps them to a corresponding `OAtom`, and oam maps them back.

```
mkAtomMaps :: HasAtoms a => a -> (Int, AtomMap, OAtomMap)
mkAtomMaps x =
   let as = S.toList $ getAtoms x S.empty
       n = length as
       am = M.fromList $ zip as [1..n]
       oam = array (1,n) $ zip [1..n] as
   in (n, am, oam)
```

5.6.3 Looking up atoms

`toOAtom` am a returns `Atom` a's corresponding `OAtom`.

```
toOAtom :: AtomMap -> Atom -> OAtom
toOAtom am a = case M.lookup a am of
   Just oa -> oa
   Nothing -> error $ "DPL ERROR: toOAtom " ++ show a
```

`toAtom` oam oa returns `OAtom` oa's corresponding `Atom`.

```
toAtom :: OAtomMap -> OAtom -> Atom
toAtom = (!)
```

5.7 Literals

Module `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

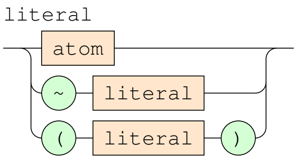
A `Literal` is any atom a (`Pos`) or its negation $\neg a$ (`Neg`).

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

5.7.2 Parser

The syntax for a literal is:

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



which shows that a literal may be multiply negated and embedded in parentheses, and is implemented by `pLiteralP`.

```
pLiteralP :: Parser Literal
pLiteralP =
       atomP @> Pos
   <|> literalP "symbol" "~" *> pLiteralP @> neg
   <|> literalP "symbol" "(" *> pLiteralP
       <* literalP "symbol" ")"
```

5.7.3 Negation

Class `Negatable` includes types that may be logically negated.

```
class Negatable a where
```

`neg` x negates x.

```
   neg :: a -> a
```

`pos` x returns the "positive" x. For example if a is an atom, `pos` $a =$ `pos` $\sim a = a$.

```
   pos :: a -> a
```

5.7.4 Instance declarations

Ordering

```
instance Ord Literal where
```

```
   compare l l’ = case l of
      Pos a -> case l’ of
         Pos a’ -> compare a a’
         Neg _  -> GT
      Neg a -> case l’ of
         Pos _  -> LT
         Neg a’ -> compare a’ a
```

Showing

```
instance Show Literal where
```

```
   showsPrec p l = case l of
      Pos a -> shows a
      Neg a -> showChar ’~’ . shows a
```

Negation

```
instance Negatable Literal where
```

```
   neg l = case l of
      Pos a -> Neg a
      Neg a -> Pos a
```

```
   pos l = case l of
      Pos a -> Pos a
      Neg a -> Pos a
```

```
instance Negatable [Literal] where
```

```
   neg = map neg
```

```
   pos = map pos
```

Collecting Atoms

```
instance HasAtoms Literal where
```

```
    getAtoms l = case l of
       Pos a -> getAtoms a
       Neg a -> getAtoms a
```

Collecting constants

```
instance HasConstants Literal where
```

```
   getConstants l cs = case l of
      Pos a -> getConstants a cs
      Neg a -> getConstants a cs
```

Collecting variables

```
instance HasVariables Literal where
```

```
   getVariables l vs = case l of
      Pos a -> getVariables a vs
      Neg a -> getVariables a vs
```

Grounding

```
instance Groundable Literal where

   ground1 v c l = case l of
      Pos a -> Pos $ ground1 v c a
      Neg a -> Neg $ ground1 v c a

   rename v v’ l = case l of
      Pos a -> Pos $ rename v v’ a
      Neg a -> Neg $ rename v v’ a
```

NFData (for deepseq)

```
instance NFData Literal where

   rnf l = case l of
      Pos a -> a ‘deepseq‘ ()
      Neg a -> a ‘deepseq‘ ()
```

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 correspoding `Literal`.

```
type OLiteral = Int
```

5.8.2 Looking up literals

`toOLiteral` *am l* returns `Literal` *l*’s corresponding `OLiteral`.

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

`toLiteral` *oam ol* returns `OLiteral` *ol*’s corresponding `Literal`.

```
toLiteral :: OAtomMap -> OLiteral -> Literal
toLiteral oam ol | ol < 0     = Neg $ toAtom oam $ abs ol
                 | otherwise = Pos $ 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)
```

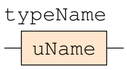
A `TypeTable` is a mapping from named types to their values.

```
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`.

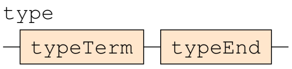
```
typeName ::= uName;
level="grammar".
```



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

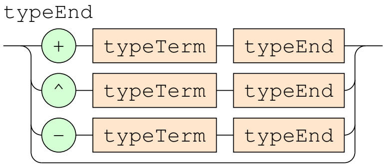
Types are parsed by `typeP`.

```
type ::= typeTerm typeEnd;
level="grammar".
```



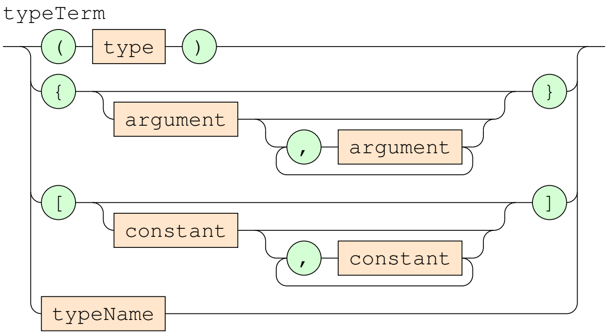
```
typeP :: Parser Type
typeP = typeTermP <*> typeEndP
        @> (\(t,f) -> f t)
```

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



```
typeEndP :: Parser (Type -> Type)
typeEndP =
           literalP "symbol" "+"
        *> nofail’ "type expected" typeTermP
       <*> typeEndP
        @> (\(t, f) -> f . (:+ t))
   <|>     literalP "symbol" "^"
        *> nofail’ "type expected" typeTermP
       <*> typeEndP
        @> (\(t, f) -> f . (:^ t))
   <|>     literalP "symbol" "-"
        *> nofail’ "type expected" typeTermP
       <*> typeEndP
        @> (\(t, f) -> f . (:- t))
   <|>     epsilonA
        #> id
```

```
typeTerm ::=   "(" type ")"
             | "{" [argument {"," argument}] "}"
             | "[" [constant {"," constant}] "]"
             | typeName;
level="grammar".
```



```
typeTermP :: Parser Type
typeTermP =
           literalP "symbol" "("
        *> nofail’ "type expected" typeP
       <*  nofail (literalP "symbol" ")")
   <|>     literalP "symbol" "{"
        *> optional (
                  argumentP
             <*> many (
                        literalP "symbol" ","
                     *> nofail’ "argument expected"
                           argumentP
                  )
           )
       <*  nofail (literalP "symbol" "}")
       @> (\es -> case es of
             []  -> TEnum S.empty
             [p] -> TEnum $ S.fromList $ cons p
          )
   <|>     literalP "symbol" "["
        *> optional (
                  constantP
             <*> many (
                        literalP "symbol" ","
                     *> nofail’ "constant expected"
                           constantP
                  )
           )
       <*  nofail (literalP "symbol" "]")
       @> (\es -> case es of
             []  -> TOrd []
             [p] -> TOrd $ cons p
          )
   <|> typeNameP
       @> TName
```

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. `evalTypes` *nts*, where *nts* is a list of pairs (n, t) associating a name with a type, returns either `Left` *tt* or `Right` *msg*, where *tt* is a `TypeTable` mapping names to fully evaluated types and *msg* is an error message.

```
evalTypes :: [(TypeName, Type)] -> Either TypeTable String
evalTypes nts =
   let ns = map fst nts
       dups = nub [n | (n:ns’) <- tails ns, n ‘elem‘ ns’]
       tt = M.fromList nts
   in if not (null dups) then
      Right $ unlines $ map (\n -> "Type " ++ show n ++
         " is muliply defined.") dups
   else
      eval ns tt
   where
   eval :: [TypeName] -> TypeTable
           -> Either TypeTable String
   eval ns tt = case ns of
      []     -> Left tt
      (n:ns’) -> case evaln n tt [] of
         Left tt’  -> eval ns’ tt’
         Right msg -> Right msg
   evaln :: TypeName -> TypeTable -> [TypeName]
            -> 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’,tt’) ->
                  Left $ M.insert n t’ tt’
               Right msg     -> Right msg
   evalt :: Type -> TypeTable -> [TypeName]
            -> 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 ++ " in the \
               \declaration of a named type contains \
               \variables."
         else
            Left (t, tt)
      TOrd cs -> Left (t, tt)
      TName n -> case evaln n tt ns of
         Left tt’  -> case M.lookup n tt’ of
            Nothing -> Right $ "Type " ++ show n ++
               " is used but not declared."
            Just t  -> Left (t, tt’)
         Right msg -> Right msg
      t1 :+ t2 -> case evalt t1 tt ns of
         Left (TEnum as1,tt’) -> case evalt t2 tt’ ns of
            Left (TEnum as2,tt’’) ->
               Left (TEnum (S.union as1 as2), tt’’)
            Right msg -> Right msg
         Right msg -> Right msg
      t1 :^ t2 -> case evalt t1 tt ns of
         Left (TEnum as1,tt’) -> case evalt t2 tt’ ns of
            Left (TEnum as2,tt’’) ->
               Left (TEnum (S.intersection as1 as2), tt’’)
            Right msg -> Right msg
         Right msg -> Right msg
      t1 :- t2 -> case evalt t1 tt ns of
         Left (TEnum as1,tt’) -> case evalt t2 tt’ ns of
            Left (TEnum as2,tt’’) ->
               Left (TEnum (S.difference as1 as2), tt’’)
            Right msg -> Right msg
         Right msg -> Right msg
```

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.

```
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'` *tt t* returns *t* reduced to alist of constants, where *tt* is the table of named types.

```
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.

```
class HasTypes a where
```

`getOrderings` *os x* adds any orderings in *x* to *os*.

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

5.9.6 Instance declarations

Showing

```
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 '(' . shows t1 .
         showString " + " . shows t2 . showChar ')'
      t1 :^ t2 -> showChar '(' . shows t1 .
         showString " ^ " . shows t2 . showChar ')'
      t1 :- t2 -> showChar '(' . shows t1 .
         showString " - " . shows t2 . showChar ')'
```

Collecting constants

```
instance HasConstants Type where

   getConstants t cs = case t of
      TEnum as -> foldr getConstants cs $ S.toList as
      TOrd as  -> 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)
```

Collecting variables

```
instance HasVariables Type where

   getVariables t vs = case t of
      TEnum as -> foldr getVariables vs $ S.toList as
      TOrd _   -> vs
      TName n  -> vs
      t1 :+ t2 -> getVariables t2 $ getVariables t1 vs
      t1 :^ t2 -> getVariables t2 $ getVariables t1 vs
      t1 :- t2 -> getVariables t2 $ getVariables t1 vs
```

Grounding

```
instance Groundable Type where

   ground1 v c t = case t of
      TEnum as -> TEnum $ S.fromList $ map (ground1 v c) $
         S.toList as
      TOrd cs  -> TOrd cs
      TName n  -> TName n
      t1 :+ t2 -> ground1 v c t1 :+ ground1 v c t2
      t1 :^ t2 -> ground1 v c t1 :^ ground1 v c t2
      t1 :- t2 -> ground1 v c t1 :- ground1 v c t2

   rename v v' t = case t of
      TEnum as -> TEnum $ S.fromList $ map (rename v v') $
         S.toList as
      TOrd cs  -> TOrd cs
      TName n  -> TName n
      t1 :+ t2 -> rename v v' t1 :+ rename v v' t2
      t1 :^ t2 -> rename v v' t1 :^ rename v v' t2
      t1 :- t2 -> rename v v' t1 :- rename v v' t2
```

Collecting Orderings

```
instance HasTypes Type where

   getOrderings os t = case t of
      TEnum _  -> os
      TOrd cs  -> cs : os
      TName _  -> 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.

```
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.Atoms
import DPL.Types
import DPL.Literals
```

5.10.1 Data types

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

```
data NewTypeDec = TypeName := Type
                  deriving (Eq, Ord)
```

An `AtomTypeDec` asserts the type of every argument to an atom

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

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

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

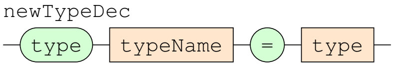
A `Default` fact is not really a type declaration, but its purpose is purely to control instantiation, so this is as good a place to define it as anywhere. A default fact declaration is a literal to be instantiated to obviate rules.

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

5.10.2 Parsers

New type declarations are parsed by `newTypeDecP`.

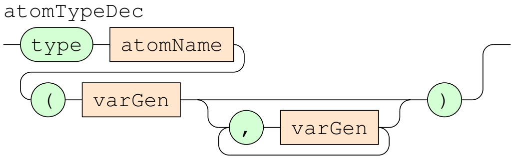
```
newTypeDec ::= "type" typeName "=" type;
level="grammar".
```



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

Atom type declarations are parsed by `atomTypeDecP`.

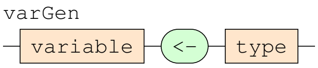
```
atomTypeDec ::= "type" atomName \
                "(" varGen {"," varGen} ")";
level="grammar".
```



```
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`.

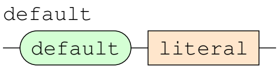
```
varGen ::= variable "<-" type;
level="grammar".
```



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

Default facts are parsed by `defaultP`.

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



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

5.10.3 Instance declarations

Showing

```
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 " .
      shows l
```

Collecting constants

```
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

```
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

```
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)
```

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) =
      foldl 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

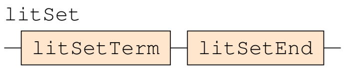
- an enumerated set of literals (`LEnum`);
- the union of two sets of literals (`::+`);
- the intersection between two types (`::^`);
- the difference between two types (`::-`); or
- a comprehensive specification (`LComp`).

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

5.11.2 Parsers

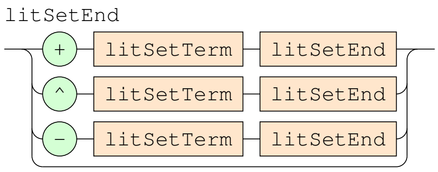
Types are parsed by `litSetP`.

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



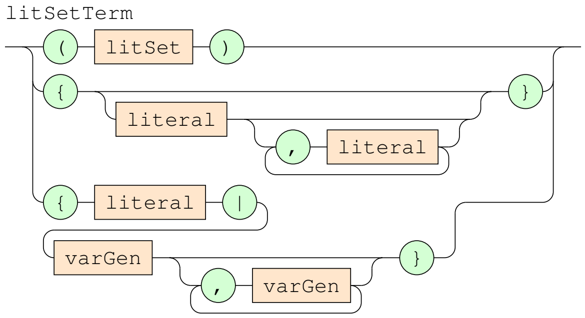
```
litSetP :: Parser LitSet
litSetP = litSetTermP <*> litSetEndP
          @> (\(t,f) -> f t)
```

```
litSetEnd ::=   "+" litSetTerm litSetEnd
              | "^" litSetTerm litSetEnd
              | "-" litSetTerm litSetEnd
              | $epsilon$;
level="grammar".
```



```
litSetEndP :: Parser (LitSet -> LitSet)
litSetEndP =
              literalP "symbol" "+"
           *> nofail' "literal set expected" litSetTermP
          <*> litSetEndP
           @> (\(t, f) -> f . (::+ t))
      <|>     literalP "symbol" "^"
           *> nofail' "literal set expected" litSetTermP
          <*> litSetEndP
           @> (\(t, f) -> f . (::^ t))
      <|>     literalP "symbol" "-"
           *> nofail' "literal set expected" litSetTermP
          <*> litSetEndP
           @> (\(t, f) -> f . (::- t))
      <|>     epsilonA
           #> id
```

```
litSetTerm ::=    "(" litSet ")"
                | "{" [literal {"," literal }] "}"
                | "{" literal "|" \ varGen {"," varGen}"}";
level="grammar".
```



```
litSetTermP :: Parser LitSet
litSetTermP =
              literalP "symbol" "("
           *> nofail' "literal set expected" litSetP
          <*  nofail (literalP "symbol" ")")
      <|>     literalP "symbol" "{"
           *> pLiteralP
          <*> literalP "symbol" "|"
           *> nofail' "variable generator expected" varGenP
          <*> many (
                    literalP "symbol" ","
                 *> nofail' "variable generator expected"
                       varGenP
              )
          <*  nofail (literalP "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')
              )
      <|>     literalP "symbol" "{"
           *> optional (
                     pLiteralP
                 <*> many (
                          literalP "symbol" ","
                       *> nofail' "literal expected" pLiteralP
                     )
              )
          <*  nofail (literalP "symbol" "}")
           @> (\lss -> LEnum (case lss of
                 []       -> S.empty
                 [(l,ls)] -> S.fromList (l : ls)
              ))
```

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 flattend. The class is parameterized over both the unflattened and flattened types.

```
class HasLitSets a b where
```

`flatten` tt x is a `Check` that attempts to flatten x, but may fail with a message. tt is the table of named types.

```
  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 $
        foldr getConstants 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

Precondtion: The variable being grounded must be realatively 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)

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

Flattening

```
instance HasLitSets LitSet [Literal] where
```

Precondition: ls must contain no variables other than that appearing in comprehensions.

```
  flatten tt ls = case ls of
     LEnum ls'   -> CheckPass $ S.toList ls'
     ls1 ::+ ls2 -> case flatten tt ls1 of
        CheckPass ls1' -> case flatten tt ls2 of
           CheckPass ls2' -> flatten tt $ LEnum
              $ S.union (S.fromList ls1') (S.fromList ls2')
           CheckFail msg  -> CheckFail msg
        CheckFail msg  -> CheckFail msg
     ls1 ::^ ls2 -> case flatten tt ls1 of
        CheckPass ls1' -> case flatten tt ls2 of
           CheckPass ls2' -> flatten tt $ LEnum
              $ S.intersection (S.fromList ls1') (S.fromList ls2')
           CheckFail msg  -> CheckFail msg
        CheckFail msg  -> CheckFail msg
     ls1 ::- ls2 -> case flatten tt ls1 of
        CheckPass ls1' -> case flatten tt ls2 of
           CheckPass ls2' -> flatten tt $ LEnum
              $ S.difference (S.fromList ls1') (S.fromList ls2')
           CheckFail msg  -> CheckFail msg
        CheckFail msg  -> CheckFail msg
     LComp l vgs       ->
        let unfold :: [VarGen] -> [Substitution]
            unfold vgs = case vgs of
               []             -> [NullSub]
               (v :<- t) : vgs ->
                  [s :>-> s'
                  | s <- [v :->- c
                         | c <- evalType' tt t],
                    s' <- unfold (ground s vgs)
                  ]
        in CheckPass [ground s l | s <- unfold vgs]
```

Collecting Orderings

```
instance HasTypes LitSet where

  getOrderings os ls = case ls of
     LEnum ls'   -> os
     ls1 ::+ ls2 -> getOrderings (getOrderings os ls1) ls2
     ls1 ::^ ls2 -> getOrderings (getOrderings os ls1) ls2
     ls1 ::- ls2 -> getOrderings (getOrderings os ls1) ls2
     LComp l vgs -> foldl getOrderings os 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
```

```
import Data.List
import qualified Data.Map as M
import Data.Maybe

import ABR.Parser
import ABR.Data.List
import ABR.Text.Showing
import ABR.Control.Check

import DPL.Constants
import DPL.Variables
import DPL.Atoms
import DPL.Literals
import DPL.LitSets
import DPL.Types
```

5.12.1 Data types

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

- comprehensively (`Clause`); or
- as a list (`Or`).

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

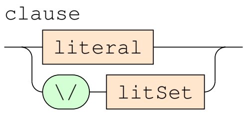
A `CnfFormula` is the conjunction (`CNF`) of a set of clauses.

```
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.

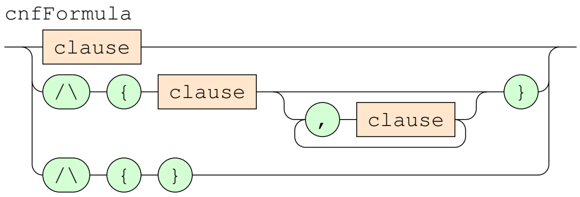
```
clause ::= literal | "\\/" litSet;
level="grammar".
```



```
clauseP :: Parser Clause
clauseP =
       pLiteralP
       @> Clause . mkLitSet
   <|> literalP "symbol" "\\/"
       *> nofail' "literal set expected" litSetP
       @> Clause
```

`cnfFormulaP` parses cnf-formulas.

```
cnfFormula ::=   clause
               | "/\\" "{" clause {"," clause} "}"
               | "/\\" "{" "}";
level="grammar".
```



```
clauseSetP :: Parser [Clause]
clauseSetP =
       literalP "symbol" "{"
       *> clauseP
       <*> many (literalP "symbol" ","
                 *> nofail' "clause expected" clauseP)
       <* literalP "symbol" "}"
       @> snub . cons
   <|> literalP "symbol" "{"
       <*> nofail (literalP "symbol" "}")
       #> []
```

```
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

```
class MaybeTautology a where
```

`isTautology` *C* returns `True` iff *C* is a tautology.

```
   isTautology :: a -> Bool
```

5.12.4 Resolution

`resolve` *c d* returns `Just` 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. $Or[\sim e, \sim b, a, c]$. Postcondition: If there is a resolvent, it is returned with the same ordering.

```
resolve :: Clause -> Clause -> Maybe Clause
resolve (Or cs) (Or ds) = resolve’ cs (reverse ds) [] []
   where
   resolve’, resolve’’ :: [Literal] -> [Literal] ->
      [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:rs) (q:qs’) rs’
                   else
                      resolve’ (q:qs) rs qs’ (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:rs) (q:qs’) rs’
                   else
                      resolve’ (q:qs) rs qs’ (r:rs’)
         Neg _  ->
            resolve’ qs (r:rs) (q:qs’) rs’
   resolve’’ qs [] qs’ rs’ =
      Just $ Or $ mnub (reverse qs’ ++ qs) rs’

   resolve’’ [] rs qs’ rs’ =
      Just $ Or $ mnub (reverse qs’) (reverse rs ++ rs’)

   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
                      Nothing
                   else if qa < ra then
                      resolve’’ qs (r:rs) (q:qs’) rs’
                   else
                      resolve’’ (q:qs) rs qs’ (r:rs’)
      Neg qa -> case r of
         Pos ra -> if qa == ra then
                      Nothing
                   else if qa < ra then
                      resolve’’ qs (r:rs) (q:qs’) rs’
                   else
                      resolve’’ (q:qs) rs qs’ (r:rs’)
         Neg _  -> resolve’’ qs (r:rs) (q:qs’) rs’
   {- build :: [Literal] -> [Literal] -> [Literal] ->
      Maybe Clause
   build qs qs’ rs’ =
      Just $ Or $ mnub (reverse qs’ ++ qs) rs’ -}
```

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

```
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] -> shows 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 -> Clause $ ground1 v c ls
      Or ls     -> Or $ map (ground1 v c) ls

   rename v v' cl = case cl of
      Clause ls -> Clause $ rename v v' ls
      Or ls     -> Or $ map (rename v v') ls

instance Groundable CnfFormula where

   ground1 v c (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 :>
```

5.13.1 Data type definition

A `Label` is used to identify a rule.

```
type Label = String
```

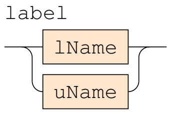
A `Priority` asserts with r_1 `:>` r_2 that r_1 beats r_2.

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

5.13.2 Parser

`labelP` parses labels.

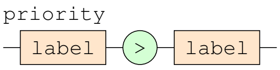
```
label ::= lName | uName;
level="grammar".
```



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

`priorityP` parses priorities.

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



```
priorityP :: Parser Priority
priorityP =
   labelP <*> literalP "symbol" ">"
   *> nofail’ "label expected" labelP
   @> (\(n1,n2) -> n1 :> n2)
```

5.13.3 Instance declarations

Showing

```
instance Show Priority where

   showsPrec p (l :> l’) =
      showString l . showString " > " . showString l’
```

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.Atoms
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 `Strict`, `Plaus`ible or a `Defeat`er. 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 represeation 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)
```

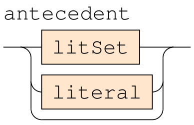
`Rule` is the preferred shorthand.

```
type Rule = PRule Literal
```

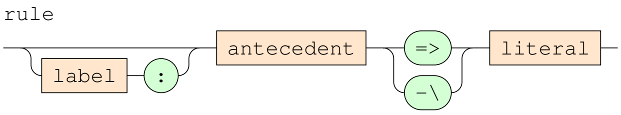
5.14.2 Parsers

The syntax for a rule is:

```
antecedent ::= litSet | literal | $epsilon$;
level="grammar".
```



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



which is implemented by `ruleP`.

```
antecedentP :: Parser LitSet
antecedentP =
       litSetP
   <|> pLiteralP @> mkLitSet
   <|> epsilonA #> LEnum S.empty

ruleP :: Parser Rule
ruleP = (optional (labelP
               <* literalP "symbol" ":")
            @> concat)
        <*> (antecedentP @> Left)
        <*> (    literalP "symbol" "=>"
             <|> literalP "symbol" "-\\")
        <*> pLiteralP
        @> (\(l,(as,((_,arrow,_),c))) -> (case arrow of
                "=>"  -> Plaus
                "-\\" -> Defeat
             ) l c as)
```

5.14.3 Interrogating rules

Class `IsRule` provides the properties of rules required for proofs.

```
class IsRule r l where
```

`ant` r returns $\bigwedge A(r)$. `negant` r returns $\bigwedge {\sim}A(r)$. Precondition: The rule’s antecdent 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 ls -> 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 ls  -> getConstants ls cs1
              Right ls -> foldr getConstants cs1 ls
       in cs2
```

Collecting variables

```
instance HasVariables Rule where

    getVariables r vs =
       let vs1 = getVariables (rcon r) vs
           vs2 = case rant r of
              Left ls  -> getVariables ls vs1
              Right ls -> foldr getVariables vs1 ls
       in vs2
```

Collecting Atoms

```
instance HasAtoms Rule where

    getAtoms r as =
       let as1 = getAtoms (rcon r) as
           as2 = case rant r of
              Left ls  -> getAtoms ls as1
              Right ls -> foldr getAtoms 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 -> ls
         _        -> error $ "ant: rule not flattened\
            \: " ++ show r
      )

   negant r = [neg (case rant r of
         Right ls -> ls
         _        -> error $ "negant: rule not flatte\
            \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)
```

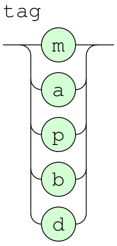
A `TaggedCnfFormula` is the target for proofs. Constructor `Tag` associates a tag and a cnf-formula.

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

5.15.2 Parsers

`pTagP` parses a tag.

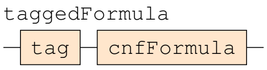
```
tag ::= "m" | "a" | "p" | "b" | "d";
level="grammar".
```



```
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` parses a tagged cnf-formula.

```
taggedFormula ::= tag cnfFormula;
level="grammar".
```



```
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) =
      shows t . shows (CNF lss)

instance Show (TaggedCnfFormula OLiteral) where

   showsPrec p (Tag t lss) =
      shows t . shows lss
```

Collecting Constants

```
instance HasConstants l =>
   HasConstants (TaggedCnfFormula l) where
```

```
    getConstants (Tag _ lss) cs =
       foldr getConstants cs (concat lss)
```

Collecting variables

```
instance HasVariables l =>
   HasVariables (TaggedCnfFormula l) where

    getVariables (Tag _ lss) vs =
       foldr getVariables vs (concat lss)
```

Collecting Atoms

```
instance (HasAtoms l) => HasAtoms (TaggedCnfFormula l) where

    getAtoms (Tag _ lss) as =
       foldr getAtoms as (concat lss)
```

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.Atoms
import DPL.Literals
import DPL.LitSets
import DPL.Formulas
import DPL.Rules
import DPL.Priorities
import DPL.Tags
import DPL.Types
import DPL.TypeDecs
import DPL.DPLLexer
```

5.16.1 Data type definitions

A plausible `Description` consists of:

- a list of new type declarations (`dnt`);
- a set of axioms (`dax`);
- a set of default facts (`ddef`);
- a set of plausible rules (`drp`);
- a set of defeater rules (`drd`);
- and a set of priorities (`dpri`).

Additionally we associate with a description:

- a mapping from names to fully evaluated types (`dtt`);
- the sets of input literals to use as inputs (`din`);
- the sets of input literals to ignore in combination (`dig`);
- the sets of input literals specifically asserted (`das`);
- 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]
   }
```

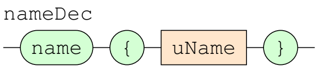
A `Statement` is an intermediate data structure used while parsing.

```
data Statement =   NT  NewTypeDec
                 | AT  AtomTypeDec
                 | Ax  Clause
                 | Def Default
                 | Rul Rule
                 | Pri Priority
                 | In  [Literal]
                 | Ig  [[Literal]]
                 | As  [Literal]
                 | Out (TaggedCnfFormula Literal,
                        Maybe String)
                 | Nam String
                 | Imp String
                 | Pre String
```

5.16.2 Parser

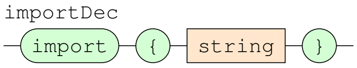
Gode generation hints

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



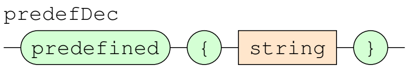
```
nameDecP :: Parser String
nameDecP =
        literalP "lName" "name"
     *> literalP "symbol" "{"
     *> tagP "uName"
    <*  nofail (literalP "symbol" "}")
     @> (\(_,n,_) -> n)
```

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



```
importDecP :: Parser String
importDecP =
        literalP "lName" "import"
     *> literalP "symbol" "{"
     *> tagP "string"
    <*  nofail (literalP "symbol" "}")
     @> (\(_,cs,_) -> cs)
```

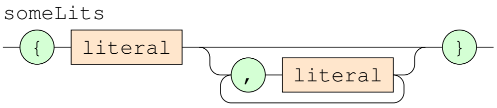
```
predefDec ::= "predefined" "{" string "}";
level="grammar".
```



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

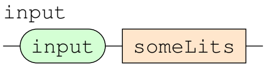
Inputs

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



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

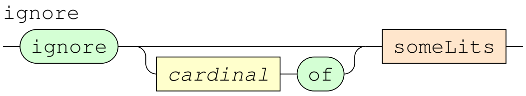
```
input ::= "input" someLits;
level="grammar".
```



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

Ignores

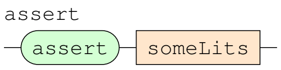
```
ignore ::= "ignore" [$cardinal$ "of"] someLits;
level="grammar".
```



```
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 combina\
                   \tions at line " ++ show (l + 1)
             else
                combinations n' ls
       )
```

Asserts

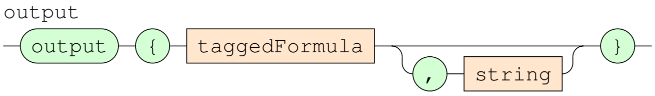
```
assert ::= "assert" someLits;
level="grammar".
```



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

Outputs

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

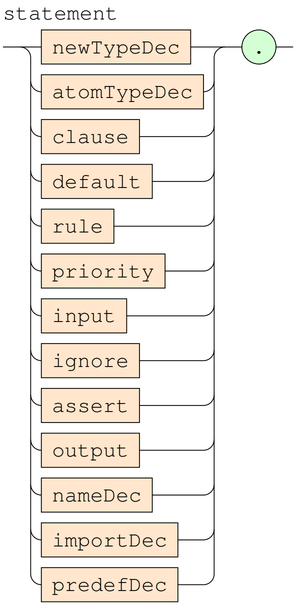


```
outputP :: Parser (TaggedCnfFormula Literal, Maybe String)
outputP =
   (   literalP "lName" "output"
    *> (    literalP "symbol" "{"
         *> nofail taggedFormulaP
        <*> optional (
                  literalP "symbol" ","
               *> tagP "string"
            )
        <* nofail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

Descriptions

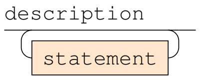
`descriptionP` parses a description and the input, ignore and output specifications.

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



```
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  lss -> d {dig = lss ++ dig d}
      As  ls   -> d {das = ls : das d}
      Out ts   -> d {dout = ts : dout d}
      Def df   -> d {ddef = df : ddef d}
      Nam n    -> d {dnam = n}
      Imp cs   -> d {dimp = cs : dimp d}
      Pre cs   -> d {dpre = cs : dpre 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)))
          u = TypeName "Universe" := TEnum (S.map Const cs)
          os = concat [
                  foldl getOrderings [] (dnt d),
                  foldl getOrderings [] (dat d),
                  foldl getOrderings [] (dax d),
                  foldl getOrderings [] (drp d),
                  foldl getOrderings [] (drd d)
               ]
          d’ :: Description
          d’ = d {
                dnt  = snub $ u : dnt 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))
               (dnt d’)) of
            Left tt   -> tt
            Right msg -> error msg
         }
```

5.16.3 Semantic checks

`labelCheck` is a Check that all of the labels occuring 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 (\(l :> l’) -> [l, l’]) $
          dpri d
       undefs = filter (‘notElem‘ rls) pls
   in case undefs of
      [] -> CheckPass d
      _   -> CheckFail $ "Undefined labels: " ++
         unwords undefs
```

5.16.4 Asserting Orderings

```
assertOrderings :: [[Constant]] -> [Clause]
assertOrderings = concatMap ao
   where
   ao :: [Constant] -> [Clause]
   ao cs = 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]
   lt c c’ = mc $ Pos $ Prop "<"  [Const c, Const c’]
   le 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\
      \ription. There are variables, but there are no cons\
      \tants."
   | otherwise = CheckPass d {
           dax = nub $
              concatMap (instances (dtt d) (dat d))
              (dax d),
           drp = nub $
              concatMap (instances (dtt d) (dat d))
              (drp d),
           drd = nub $
              concatMap (instances (dtt d) (dat d))
              (drd d),
           ddef = nub $
              concatMap (instances (dtt d) (dat d))
              (ddef 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 .
```

```
              concatMap (instances (dtt d) (dat d)))
              (das d),
           dout = nub $
              concatMap (instances (dtt d) (dat d))
              (dout d)
        }
```

Class `Instantiable` has instances that can be instantiated by the grounding of variables.

```
class Instantiable a where
```

`instantiate` *tt ats x* creates as many instances of *x* as grounding of all the free variables demands and the substitution that created each instance. *tt* is the table of named types. *ats* is the list of atom type assertions. For each instance the substitution that defines that instance is also returned

```
   instantiate :: TypeTable -> [AtomTypeDec] -> a ->
      [(a, Substitution)]
```

`instances` *tt ats x* creates as many instances of *x* as grounding of all the free variables demands. *tt* is the table of named types. *ats* is the list of atom type assertions.

```
   instances :: TypeTable -> [AtomTypeDec] -> a -> [a]
   instances tt ats = map fst . instantiate tt ats
```

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.

```
assertCheck :: Check Description Description String
assertCheck d
   | null (das d) = CheckPass d
   | otherwise    = case gen (din d) (dig d) of
      []     -> CheckPass d
      r1 : _ ->
         if and [null (ls \\ ls’) && null (ls’ \\ ls)
                | let ls = map pos r1,
                  ls’ <- map (map pos) (das d)]
         then CheckPass d
         else CheckFail "Assert does not match the\
            \ declared inputs. Hint: Inspect the\
            \ grounded 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 asserted as a fact, then that rule is useless and can be omitted. The `obviateCheck` passes a description after removing such obviously true literals from antecedents and such obviously useless rules.

```
obviateCheck :: Check Description Description String
obviateCheck d =
   let facts = [l | Or [l] <- dax d]
       defaults = [l | Default l <- ddef d]
       test :: Literal -> Maybe Bool
       test l
          | l ‘elem‘ facts     = Just True
          | neg l ‘elem‘ facts = Just False
          | l ‘elem‘ defaults  = Just True
          | neg l ‘elem‘ defaults  = Just False
          | otherwise          = Nothing
       check :: [Literal] -> Maybe [Literal]
       check as = case as of
          []     -> Just []
          a : as’ -> case test a of
             Just True  -> check as’
             Just False -> Nothing
             Nothing    -> case check as’ of
                Nothing   -> Nothing
                Just as’’ -> Just (a : as’’)
       obviateP, obviateD :: Rule -> Maybe Rule
       obviateP (Plaus l c (Right as)) = case check as of
          Nothing  -> Nothing
          Just as’ -> Just (Plaus l c (Right as’))
       obviateD (Defeat l c (Right as)) = case check as of
          Nothing  -> Nothing
          Just as’ -> Just (Defeat l c (Right as’))
   in CheckPass d {
          drp = nub $ catMaybes $ map obviateP $ drp d,
          drd = nub $ catMaybes $ map obviateD $ drd d
      }
```

5.16.8 Generating runs

`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.

```
generateRuns :: Description -> [[Literal]]
generateRuns d
   | null (das d) = gen (din d) (dig d)
   | otherwise = das 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.

```
gen :: [[Literal]] -> [[Literal]] -> [[Literal]]
gen inss igss = case inss of
   []          -> [[]]
   ins : inss’ ->
      let qss = case ins of
             [q] -> [[q], [neg q]]
             _   -> (map (\(b,e,a) ->
                reverse (map neg b) ++ [e] ++ map neg a)
                . fragments) ins
          genSeq igss qs = case qs of
             []      -> gen inss’ igss
             q : qs’ -> case applyIgnores q igss of
                Nothing    -> []
                Just igss’ ->
                   map (q :) $ genSeq igss’ qs’
      in concatMap (genSeq igss) qss
```

`applyIgnores` *l igss* returns either: `Nothing`, when the literal *l* is knocked out by an element of *igss* that equals [*l*]; or `Just` *igss′*, where *igss′* contains any elements of *igss* not knocked out because they contain ¬*l*.

```
applyIgnores :: Literal -> [[Literal]]
   -> Maybe [[Literal]]
applyIgnores l igss = case igss of
   []                    -> Just []
   igs : igss’
      | igs == [l]       -> Nothing
      | neg l ‘elem‘ igs -> applyIgnores l igss’
      | otherwise        -> case applyIgnores l igss’ of
         Nothing     -> Nothing
         Just igss’’ -> Just $ (delete l igs) : igss’’
```

`generateAxioms` *d ls* returns the description *d* augmented by the axioms *ls*.

```
generateAxioms :: Description -> [Literal] -> Description
generateAxioms d ls = d {
      dax = snub $ dax d ++ map (Or . (:[])) ls,
      din = [],
      dig = []
   }
```

5.16.9 Loading a description

`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.

```
loadDescription :: FilePath ->
   IO (CheckResult Description String)
loadDescription path = do
   source <- readFile path
   case (checkParse ((dropWhite . nofail .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 "\n% 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)
         | (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)
         ]

instance Instantiable VarGen where

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

instance Instantiable Default where

   instantiate tt ats (Default l) =
      [(Default l’, s)
      | (l’, 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']
            ]
         Var v'@(Variable n)
            | "_VG_" `isPrefixOf` n ->
               [((a, vg) : avgs, s)
               | (avgs, s) <- instantiate tt ats  avgs'
               ]
            | otherwise             ->
               [((ground s a, vg) : avgs, s :>-> s'')
               | (s,s') <- [(v' :->- c, v :->- c)
                           | c <- evalType' tt t],
                 (avgs, s'') <- instantiate tt ats
                    [(ground s a, ground (s :>-> 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] -> ts
             _    -> error $ "Too many type assertions for\
                \ " ++ n ++ "/" ++ show n
      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 l = case l 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' : xs', s :>-> 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)
         ]
      ls1 ::+ ls2 ->
         [(ls1' ::+ ls2', s1 :>-> s2)
         | (ls1', s1) <- instantiate tt ats ls1,
           (ls2', s2) <- instantiate tt ats (ground s1 ls2)
         ]
      ls1 ::^ ls2 ->
         [(ls1' ::^ ls2', s1 :>-> s2)
         | (ls1', s1) <- instantiate tt ats ls1,
           (ls2', s2) <- instantiate tt ats (ground s1 ls2)
         ]
      ls1 ::- ls2 ->
         [(ls1' ::- ls2', s1 :>-> s2)
         | (ls1', s1) <- instantiate tt ats ls1,
           (ls2', s2) <- instantiate tt ats (ground s1 ls2)
         ]
      LComp l vgs ->
         [ (LComp l' vgs', s :>-> s')
         | (l', s) <- instantiate tt ats l,
           (vgs', s') <- instantiate tt ats (ground s vgs)
         ]
```

```
instance Instantiable Clause where

   instantiate tt ats c = case c of
      Clause ls ->
         [(Clause ls', s)
         | (ls', s) <- instantiate tt ats ls
         ]
      Or ls     ->
         [(Or ls', s)
         | (ls', s) <- instantiate tt ats ls
         ]
```

```
instance Instantiable Rule where

   instantiate tt ats r = case r of
      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 outout with \
         \string"
      Nothing -> [((Tag tag lss', Nothing), s)
                 | (lss',s) <- instantiate tt ats lss]
```

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
```

```
import DPL.Priorities
import DPL.Formulas
import DPL.Descriptions
import DPL.Tags
```

5.17.1 Data type definitions

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

```
data PTheory r l = Theory {
      tax   :: [Clause],
      trsn  :: [Clause],
      tfct  :: [Clause],
      trs   :: [r l],
      trp   :: [r l],
      trd   :: [r l],
      tpri  :: [Priority],
      tinc  :: [[l]],
      tincl :: M.Map l [[l]]
   }
```

`Theory` is the preferred shorthand.

```
type Theory = PTheory PRule Literal
```

5.17.2 Making theories from descriptions

`mmin` S returns $\mathit{Min}(S) = \{\vee L \in S : \text{if } K \subset L \text{ then } \vee K \notin S\}$.

```
mmin :: [Clause] -> [Clause]
mmin s = map Or $ noSuperSets [l | Or l <- s]
```

`fct` Ax returns $\mathit{Fct}(Ax) = \mathit{Min}(\mathit{Rsn}(Ax)) - \{\vee L : \vee L \text{ is a tautology}\}$.

```
fct :: [Clause] -> [Clause]
fct ax = filter (not . isTautology) $ mmin $ rsn ax
```

`makeTheory` d transforms description d into a theory.

```
makeTheory :: Description -> Theory
makeTheory d = let
```

$$R_s = \{{\sim}(L - \{l\}) \rightarrow l : l \in L \text{ and } \vee L \in \mathit{Fct}(Ax)\}.$$

```
      tax' = dax d
      trsn' = rsn tax'
      tfct' = fct(tax')
      trs' = [Strict l (Right (map neg (ls \\ [l])))
             | Or ls <- tfct', l <- ls]
```

$$R = R_s \cup R_p \cup R_d$$

```
      trp' = drp d
      trd' = drd d
      r = trs' ++ trp' ++ trd'
```

$$\mathit{Inc}(T) = \{{\sim}L : \vee L \in \mathit{Min}(\mathit{Fct}(T) \cup \vee\{k, {\sim}k : k \in \mathit{Atm}\})\}$$

```
      allAtoms = S.toList $ getAtoms d S.empty
      tinc' = [map neg ls | Or ls <- mmin
               (tfct' ++ [Or [Neg a, Pos a]
                        | a <- allAtoms])]
```

$$\mathit{Inc}(T, l) = \{I - \{l\} : I \in \mathit{Inc}(T) \text{ and } l \in I\}.$$

```
      allLits = snub $ concat tinc'
      tincl' :: Literal -> [[Literal]]
      tincl' l = noSuperSets $ snub [ls \\ [l] |
                    ls <- tinc', l `elem` ls]
      addIncl :: Literal -> M.Map Literal [[Literal]]
         -> M.Map Literal [[Literal]]
      addIncl l = M.insert l (tincl' l)
   in Theory {
      tax = tax',
      trsn = trsn',
      tfct = tfct',
      trs = trs',
      trp = trp',
      trd = trd',
      tpri = dpri d,
      tinc = tinc',
      tincl = foldr addIncl M.empty allLits
   }
```

5.17.3 Interrogating theories

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

```
class IsTheory t r l where
```

`tfctl` $T\ L$ returns $\mathit{Fct}(T; L) = \{\vee K \in \mathit{Fct}(T) : |K \cap L| \geq 2\}$.

```
   tfctl :: t r l -> [l] -> [[l]]
```

`inc` $T\ l$ returns $\mathit{Inc}(T, l)$.

```
   inc :: t r l -> l -> [[l]]
```

`rsl` $T\ l$ returns $R_s[l]$.

```
   rsl :: t r l -> l -> [r l]
```

`rpl` $T\ l$ returns $R_p[l]$.

```
   rpl :: t r l -> l -> [r l]
```

`rl` $T\ l$ returns $R[l]$.

```
   rl :: t r l -> l -> [r l]
```

`rpls` $T\ l\ r$ returns $R_p[l; s]$.

```
   rpls :: t r l -> l -> r l -> [r l]
```

5.17.4 Instance declarations

Showing

```
instance Show Theory where

   showsPrec p t =
        showString "% Ax:\n"
      . showWithTerm ".\n" (tax t)
      . showString "% Rsn(Ax):\n"
      . showWithTerm ".\n" (trsn t)
      . showString "% Fct(Ax):\n"
      . showWithTerm ".\n" (tfct t)
      . showString "% strict rules:\n"
      . showWithTerm ".\n" (trs t)
      . showString "% plausible rules:\n"
      . showWithTerm ".\n" (trp t)
      . showString "% defeaters:\n"
      . showWithTerm ".\n" (trd t)
      . showString "% priorities:\n"
      . showWithTerm ".\n" (tpri t)
      . showString "% Inc(T):\n"
      . showWithTerm ".\n" (tinc t)
      . showString "% Inc(T,l):\n"
      . showWithTerm ".\n" (M.toList (tincl t))
```

Collecting atoms

```
instance HasAtoms Theory where

   getAtoms t as =
      let as1 = foldr getAtoms as  $ trs t
          as2 = foldr getAtoms as1 $ trp t
          as3 = foldr getAtoms as2 $ trd t
      in as3

instance HasAtoms (Description,Theory) where

   getAtoms (d,t) as =
      let as1 = getAtoms d as
          as2 = getAtoms t as1
      in as2
```

Theory interrogation

`filterByC` $l\ rs$ selects the rules in rs with consequent l.

```
filterByC :: Literal -> [Rule] -> [Rule]
filterByC l = filter ((== l) . rcon)
```

`filterByBeats` *r rs* selects the rules in *rs* that beat rule *r*.

```
filterByBeats :: Theory -> Rule -> [Rule] -> [Rule]
filterByBeats t s = case s of
   Strict _ _ -> const []
   _          ->
      let l = rlbl s
      in filter (\r -> (rlbl r :> l) ‘elem‘ tpri t)
```

```
instance IsTheory PTheory PRule Literal where

   tfctl t ls =
      [ks | Or ks <- tfct t,
           length [k | k <- ks, k ‘elem‘ ls] >= 2]

   inc t l = case M.lookup l (tincl t) of
      Just js -> js
      Nothing -> []

   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,
             TypeSynonymInstances, OverlappingInstances #-}

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

import Data.Array
import Data.List
import qualified Data.Map.Strict as M

import ABR.Data.List
import ABR.Text.Showing

import DPL.OAtoms
import DPL.Literals
import DPL.OLiterals
import DPL.Rules
import DPL.Theories
import DPL.Priorities
import DPL.Formulas
import DPL.Descriptions
```

5.18.1 Data type definitions

Optimised rules are assigned optimised labels (`OLabel`s) 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 antecents, 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 OLabel OAIndex (Array OAIndex [l])
```

`ORule` is the preferred shorthand.

```
type ORule = OPRule OLiteral
```

An optimised theory `OPTheory`, constructor `OTheory`, contains:

- `otam` – the `AtomMap` that maps from `Atom`s to `OAtom`s (which range from 1 to N);
- `otoam` – the `OAtomMap` that maps from `OAtom`s to `Atom`s;
- `otants` – 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.);
- `otrpl` – the indices of the plausible rules presorted by consequent (This array is indexed from $-N$ to N. Element 0 should be empty.);
- `otrl` – the indices of the 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 involed in priorities.);
- `otfct` – optimised `ftct`; and
- `otincl` – optimised `tincl`.

```
data OPTheory r l = OTheory {
      otam    :: AtomMap,
      otoam   :: OAtomMap,
      otants  :: Array OAIndex [l],
      otrs    :: Array ORIndex (r l),
      otrsl   :: Array l [ORIndex],
      otrpl   :: Array l [ORIndex],
      otrl    :: Array l [ORIndex],
      otrpls  :: Array (l,OLabel) [ORIndex],
      otfct   :: [[l]],
      otincl  :: Array l [[l]]
   }
```

`OTheory` is the preferred shorthand.

```
type OTheory = OPTheory OPRule OLiteral
```

5.18.2 Conversions

`toOTheory` *D* and *T* makes an optimised theory from *T*. Make sure *D* is the grounded version.

```
toOTheory :: Description -> Theory -> OTheory
toOTheory d t =
   let (n, am, oam) = mkAtomMaps (d,t)
       ml = toOLiteral am
       mml = map ml
       labels = snub $ map rlbl (trp t) ++ map rlbl (trd t)
       nl = length labels
       labelMap = M.fromList $ zip labels [1..]
       mapLbl lbl = case M.lookup lbl labelMap of
          Just i  -> i
          Nothing -> 0
       rules = trp t ++ trd t ++ trs t
       np = length $ trp t
       nd = length $ trd t
       ns = length $ trs t
       nr = np + nd + ns
       antecedents =
          snub [mml as | Right as <- map rant rules]
       nants = length antecedents
       ants = array (0, nants - 1) $ zip [0..] antecedents
       findAnt :: [Literal] -> OAIndex
       findAnt ls =
          let a = mml ls
              f :: OAIndex -> OAIndex -> OAIndex
              f i j =
                 let m = (i + j) ‘div‘ 2
                 in if a == ants ! m then m
                    else if a < ants ! m then f i (m-1)
                    else f (m+1) j
          in f 0 (nants - 1)
       rs = array (0, nr - 1)
          [(i, ORule i (mapLbl l) (findAnt as) ants)
```

```
          | (i, r) <- zip [0..] rules
          , let l = case r of Strict _ _ -> ""
                              _ -> rlbl r
          , let Right as = rant r]
       priMap = foldl (\t (l,l’) ->
          (M.insertWith (++) l [l’] t)) M.empty $
          map (\(l :> l’) -> (mapLbl l, mapLbl l’)) $
          tpri t
       beats l l’ = case M.lookup l priMap of
          Just ls -> l’ ‘elem‘ ls
          Nothing -> False
       otrsl’ = accumArray (flip (:)) [] (-n,n) $
          map (\(Strict c (Right as), i) ->
          (ml c, i)) $ zip (trs t) [np + nd .. nr - 1]
       otrpl’ = accumArray (flip (:)) [] (-n,n) $
          map (\(Plaus l c (Right as), i) ->
          (ml c, i)) $ zip (trp t) [0 .. np - 1]
       otrdl’ = accumArray (flip (:)) [] (-n,n) $
          map (\(Defeat l c (Right as), i) ->
          (ml c, i)) $ zip (trd t) [np .. np + nd - 1]
       otrl’ = array (-n,n) [(l, otrsl’ ! l ++ otrpl’ ! l
         ++ otrdl’ ! l) | l <- [-n..n]]
       otrpls’ = accumArray (flip (:)) [] ((-n,1), (n,nl))
          [((c,l), r) | c <- [-n..n], l <- [1..nl],
           r <- otrpl’ ! c, let ORule _ l’ _ _ = rs ! r,
           l’ ‘beats‘ l]
   in OTheory {
      otam    = am,
      otoam   = oam,
      otants  = ants,
      otrs    = rs,
      otrsl   = otrsl’,
      otrpl   = otrpl’,
      otrl    = otrl’,
      otrpls  = otrpls’,
      otfct   = [mml ls | Or ls <- tfct t],
      otincl  = accumArray (\_ x -> x) [] (-n,n) $
         map (\(l,lss) -> (ml l, map mml lss)) $
         M.toList $ tincl t
   }
```

5.18.3 Instance declarations

Showing

```
instance Show ORule where

   showsPrec p (ORule i l a ants) = shows i . showChar ’ ’
      . shows l . showString ": " . shows a

instance Show OTheory where

   showsPrec p t =
        showString "% Atom map:\n   "
      . showWithSep "\n   " (M.toList (otam t))
      . showString "\n% Antecedents:\n   "
      . showWithSep "\n   " (assocs (otants t))
      . showString "\n% Rules:\n   "
      . showWithSep "\n   " (assocs (otrs t))
      . showString "\n% Strict rules by consequent:\n   "
      . showWithSep "\n   " (assocs (otrsl t))
      . showString "\n% plausible rules by consequent:\
        \\n   "
      . showWithSep "\n   " (assocs (otrpl t))
      . showString "\n% the rules by consequent:\n   "
      . showWithSep "\n   " (assocs (otrl t))
      . showString "\n% rules that beat other rules by\
        \ consequent:\n   "
      . showWithSep "\n   " (assocs (otrpls t))
      . showString "\n% fct:\n   "
      . showWithSep "\n   " (otfct t)
      . showString "\n% incl:\n   "
      . showWithSep "\n   " (assocs (otincl t))
```

Interrogating rules

```
instance IsRule OPRule OLiteral where

   ant (ORule _ _ a ants) = map (:[]) $ ants ! a

   negant (ORule _ _ a ants) = [neg (ants ! a)]
```

Interrogating theories

```
instance IsTheory OPTheory OPRule OLiteral where

   tfctl t ls =
      [ks | ks <- otfct t,
           length [k | k <- ks, k ‘elem‘ ls] >= 2]

   inc t l = otincl t ! l

   rsl t l = map (otrs t !) $ otrsl t ! l

   rpl t l = map (otrs t !) $ otrpl t ! l

   rl t l = map (otrs t !) $ otrl t ! l

   rpls t c (ORule _ l _ _ )
      | l == 0     = []
      | otherwise = map (otrs t !) $ otrpls t ! (c,l)
```

5.19 Proof Results

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

```
module DPL.ProofResults (ProofResult(..)) where

import Control.DeepSeq

import DPL.Literals
```

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).

```
data ProofResult = 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  -> Zero
      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. Different instances of this class allow variations of the proof algorithm, such as printing a trace.

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

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

```
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.Literals
import DPL.Theories
import DPL.Tags
import DPL.ProofResults
import DPL.Formulas
import DPL.Rules
import DPL.Descriptions
import DPL.OTheories
import DPL.OLiterals
```

5.20.1 PlausibleX classes: overloaded inference

Classes `Plausible` X 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 a auxiliary functions that are implemented by each instance.

Auxiliary functions

These methods of classes `Plausible`X need to implemented for each instance, unless the default is adequate.

```
class (Monad monad) => Plausible monad state where
```

`return_` returns a `ProofResult` wrapped up in the monad.

```
   return_ :: ProofResult -> state ->
      monad (ProofResult, state)
   return_ result state = return (result, state)
```

`neg_` does a proof wrapped up in the monad and negates its result.

```
   neg_ ::
      (state -> monad (ProofResult, state))
      -> state -> monad (ProofResult, state)
   neg_ result state = do
      (r,state’) <- result state
      return (neg r, state’)
```

`just_` does a proof wrapped up in the monad. You only need to call this where `max_`, `min_`, and `return_` have not appeared in the definition.

```
   just_ ::
      (state -> monad (ProofResult, state))
      -> state -> monad (ProofResult, state)
   just_ result state = result state
```

`min_` returns the min of a sequence of proof computations. `max_` returns the max.

```
   min_, max_ ::
      [state -> monad (ProofResult, state)]
      -> state -> monad (ProofResult, state)
   min_ ps state = case ps of
      []       -> return (Plus, state)
      [p]      -> p state
      p : ps’ -> do
         (r’,state’) <- p state
         case r’ of
            Minus -> return (Minus, state’)
            _      -> do
               (r’’, state’’) <- min_ ps’ state’
               return (min r’ r’’, state’’)
   max_ ps state = case ps of
      []       -> return (Minus, state)
      [p]      -> p state
      p : ps’ -> do
         (r’,state’) <- p state
         case r’ of
            Plus -> return (Plus, state’)
            _     -> do
               (r’’, state’’) <- max_ ps’ state’
               return (max r’ r’’, state’’)
```

```
class (Monad monad) => PlausibleL monad state l where
```

`label_` *label* F λ f B *result* may be used to generate a trace for *result* using the *label* that names the rule being used, F as the name of the current proof function, λ as the current proof tag, f as the current cnf formula, and the current B.

```
   label_ ::
      String -> String -> Tag -> [[l]]
      -> [l] -> state
      -> (state -> monad (ProofResult, state))
      -> monad (ProofResult, state)
   label_ _ _ _ _ _ state result = result state
```

`label_ls` is the same as `label_` for use in proof functions with an extra `[l]` parameter.

```
   label_ls :: String -> String -> Tag -> [[l]] -> [l] ->
      [l] -> state -> (state -> monad (ProofResult, state))
      -> monad (ProofResult, state)
   label_ls _ _ _ _ _ _ state result = result state
```

`label_l` is the same as `label_` for use in proof functions with an extra `l` parameter.

```
   label_l :: String -> String -> Tag -> [[l]] -> [l] ->
      l -> state -> (state -> monad (ProofResult, state))
      -> monad (ProofResult, state)
   label_l _ _ _ _ _ _ state result = result state
```

```
class (Monad monad, IsRule r l) =>
   PlausibleRL monad state r l where
```

`label_r` is the same as `label_` for use in proof functions with an extra `Rule` parameter.

```
   label_r :: String -> String -> Tag -> [[l]]
      -> [l] -> r l -> state
      -> (state -> monad (ProofResult, state))
      -> monad (ProofResult, state)
   label_r _ _ _ _ _ _ state result = result 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, Plausible monad state,
   PlausibleL monad state l, PlausibleRL monad state r l,
   IsTheory t r l, IsRule r l, Negatable l, Ord l, Eq l) =>
   PlausibleTRL monad state t r l where
```

`p` is the main proof function P. `p` T (`Tag` λ f) B returns $P(\lambda f, B)$, were f is a cnf formula.

$\wedge$) $P(\lambda \wedge C, B) = \min\{P(\lambda c, B) : c \in C\}$.
$\vee$.1) If $\vee L$ is a tautology then $P(\lambda \vee L, B) = +1$.
$\vee$.2) If $\vee L$ is not a tautology then $P(\lambda \vee L, B) =$
$\max(\{P(\lambda l, B) : l \in L\}$
$\cup\{P(\lambda \wedge {\sim}(K - L), B) : \vee K \in \mathit{Fct}(T; L)\})$.
$\mathcal{L}$.1.1) If $l \in B$ then $P(\lambda l, B) = 0$
$\mathcal{L}$.1.2) If $l \notin B$ then $P(\lambda l, B) = \max\{\mathit{Strict}(\lambda l, B), \mathit{Plaus}(\lambda l, B)\}$.

In $\mathcal{L}$.1.2, if $\lambda = \mu$ then
$P(\mu l, B) = \max\{\mathit{Strict}(\mu l, B)\} = \mathit{Strict}(\mu l, B)$.

```
   p :: t r l -> TaggedCnfFormula l -> [l] -> state ->
      monad (ProofResult, state)
   p = p’
```

We define `p` in terms of another function `p’` of exactly the same type so that `p` can be overridden for state maintenance without needing to reimplement much.

```
   p’ :: t r l -> TaggedCnfFormula l -> [l] -> state ->
      monad (ProofResult, state)
   p’ t (Tag lambda [[l]]) b state
      | l ‘elem‘ b   =
         label_ "L.1.1" "P" lambda [[l]] b state $
            return_ Zero
      | lambda == Mu =
         label_ "L.1.2" "P" Mu [[l]] b state $
            just_ (strict t (Tag Mu [[l]]) b)
      | otherwise    =
```

```
      label_ "L.1.2" "P" lambda [[l]] b state $
         max_ [strict t (Tag lambda [[l]]) b,
               plaus  t (Tag lambda [[l]]) b]
p’ t (Tag lambda [ls]) b state
   | isTautology ls =
      label_ "\\/.1" "P" lambda [ls] b state $
         return_ Plus
   | otherwise  =
      label_ "\\/.2" "P" lambda [ls] b state $
         max_ ([p t (Tag lambda [[l]]) b | l <- ls] ++
               [p t (Tag lambda (map ((: []) . neg)
                  (snub ks \\ ls))) b
               | ks <- tfctl t ls])
p’ t (Tag lambda lss) b state =
   label_ "/\\" "P" lambda lss b state $
      min_ [p t (Tag lambda [ls]) b | ls <- lss]
```

$\mathcal{L}$.2) $\mathit{Strict}(\lambda l, B) = \max\{P(\lambda\wedge A(r), B \cup \{l\}) : r \in R_s[l]\}$.

```
strict :: t r l -> TaggedCnfFormula l -> [l] -> state ->
   monad (ProofResult, state)
strict t (Tag lambda [[l]]) b state =
   label_ "L.2" "Strict" lambda [[l]] b state $
      max_ [p t (Tag lambda (ant r)) (b +: l)
           | r <- rsl t l]
```

$\mathcal{L}$.3) $\mathit{Plaus}(\lambda l, B) = \min\{\mathit{For}(\lambda l, B), \mathit{Nullified}(\lambda l, B)\}$.

```
plaus :: t r l -> TaggedCnfFormula l
   -> [l] -> state
   -> monad (ProofResult, state)
plaus t (Tag lambda [[l]]) b state =
   label_ "L.3" "Plaus" lambda [[l]] b state $
      min_ [for t (Tag lambda [[l]]) b,
            nullified t (Tag lambda [[l]]) b]
```

$\mathcal{L}$.4) $\mathit{For}(\lambda l, B) = \max\{P(\lambda\wedge A(r), B \cup \{l\}) : r \in R_p[l]\}$.

```
for :: t r l -> TaggedCnfFormula l
   -> [l] -> state
   -> monad (ProofResult, state)
for t (Tag lambda [[l]]) b state =
   label_ "L.4" "For" lambda [[l]] b state $
      max_ [p t (Tag lambda (ant r)) (b +: l)
           | r <- rpl t l]
```

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

```
nullified :: t r l ->  TaggedCnfFormula l
   -> [l] -> state
   -> monad (ProofResult, state)
nullified t (Tag lambda [[l]]) b state =
   label_ "L.5" "Nullified" lambda [[l]] b state $
      min_ [disabled t (Tag lambda [[l]]) b js
           | js <- inc t l]
```

$\mathcal{L}$.6) $\mathit{Disabled}(\lambda l, B, J) = \max\{\mathit{Discredited}(\lambda l, B, j) : j \in J\}$.

```
disabled :: t r l ->  TaggedCnfFormula l
   -> [l] -> [l] -> state
   -> monad (ProofResult, state)
disabled t (Tag lambda [[l]]) b js state =
   label_ls  "L.6" "Disabled" lambda [[l]] b js state $
      max_ [discredited t (Tag lambda [[l]]) b j
           | j <- js]
```

$\mathcal{L}$.7) $\mathit{Discredited}(\lambda l, B, j) = \min\{\mathit{Defeated}(\lambda l, B, s) : s \in R[j]\}$.

```
discredited :: t r l ->  TaggedCnfFormula l
   -> [l] -> l -> state
   -> monad (ProofResult, state)
discredited t (Tag lambda [[l]]) b j state =
   label_l "L.7" "Discredited" lambda [[l]] b j state $
      min_ [defeated t (Tag lambda [[l]]) b s
           | s <- rl t j]
```

$\mathcal{L}$.8) $\mathit{Defeated}(\lambda l, B, s) =$
$\max\{\mathit{Beaten}(\lambda l, B, s), \mathit{Inappl}(\lambda l, B, s)\}$.

```
defeated :: t r l ->  TaggedCnfFormula l
   -> [l] -> r l -> state
   -> monad (ProofResult, state)
defeated t (Tag lambda [[l]]) b s state =
   label_r "L.8" "Defeated" lambda [[l]] b s state $
      max_ [beaten t (Tag lambda [[l]]) b s,
            inappl t (Tag lambda [[l]]) b s]
```

$\mathcal{L}$.9) $\mathit{Beaten}(\lambda l, B, s) = \max\{P(\lambda\wedge A(t), B \cup \{l\}) : t \in R_p[l; s]\}$.

```
beaten :: t r l ->  TaggedCnfFormula l
   -> [l] -> r l -> state
   -> monad (ProofResult, state)
beaten t (Tag lambda [[l]]) b s state =
   label_r "L.9" "Beaten" lambda [[l]] b s state $
      max_ [p t (Tag lambda (ant u)) (b +: l)
           | u <- rpls t l s]
```

$\mathcal{L}$.10.α) $\mathit{Inappl}(\alpha l, B, s) =$
$\min\{P(\alpha{\sim}\wedge A(s), B \cup \{l\}), -P(\alpha\wedge A(s), B \cup \{l\})\}$.
$\mathcal{L}$.10.π) $\mathit{Inappl}(\pi l, B, s) = P(\pi{\sim}\wedge A(s), B \cup \{l\})$.
$\mathcal{L}$.10.β) $\mathit{Inappl}(\beta l, B, s) = -P(\beta\wedge A(s), B \cup \{l\})$.
$\mathcal{L}$.10.δ) $\mathit{Inappl}(\delta l, B, s) =$
$\max\{P(\delta{\sim}\wedge A(s), B \cup \{l\}), -P(\delta\wedge A(s), B \cup \{l\})\}$.

```
inappl :: t r l ->  TaggedCnfFormula l -> [l] -> r l ->
   state -> monad (ProofResult, state)
inappl t (Tag Alpha [[l]]) b s state =
   label_r "L.10.a" "Inappl" Alpha [[l]] b s state $
      min_ [p t (Tag Alpha (negant s)) (b +: l),
            neg_ (p t (Tag Alpha (ant s)) (b +: l))]
inappl t (Tag Pi [[l]]) b s state =
   label_r "L.10.p" "Inappl" Pi [[l]] b s state $
      just_ (p t (Tag Pi (negant s)) (b +: l))
inappl t (Tag Beta [[l]]) b s state =
   label_r "L.10.b" "Inappl" Beta [[l]] b s state $
      just_ $ neg_ $ p t (Tag Beta (ant s)) (b +: l)
inappl t (Tag Delta [[l]]) b s state =
   label_r "L.10.d" "Inappl" Delta [[l]] b s state $
      max_ [p t (Tag Delta (negant s)) (b +: l),
            neg_ (p t (Tag Delta (ant s)) (b +: l))]
```

5.20.2 Maybe () instances of PlausibleX

These instances will not print traces. Call `p` like this to suppress the trace:

```
   let Just result = p theory (Tag lambda f) [] ()
   in ...

instance Plausible Maybe () where { }

instance PlausibleL Maybe () Literal where { }

instance PlausibleRL Maybe () PRule Literal where { }

instance PlausibleTRL Maybe () PTheory PRule Literal where
   { }
```

5.20.3 IO instance of PlausibleX

These instance will print traces. See `doProofs` below for an example call to `p` that prints a trace.

```
instance Plausible IO String where

   return_ result indent = do
      putStr " "
      return (result, indent)

   neg_ result indent = do
      putStrLn $ indent ++ "- ("
      (r,_) <- result (".  " ++ indent)
      putStrLn $ indent ++ ") = " ++ show (neg r)
      return (neg r,indent)

   just_ result indent = do
      putStrLn " = ("
      (r,_) <- result (".  " ++ indent)
      putStr $ indent ++ ") "
      return (r,indent)
```

```
   min_ ps indent = do
         putStrLn " = min {"
         (r,_) <- min' ps (".  " ++ indent)
         putStr $ indent ++ "} "
         return (r,indent)
      where
      min' ps state = case ps of
         []      -> return (Plus, state)
         [p]     -> p state
         p : ps' -> do
            (r',state') <- p state
            case r' of
               Minus -> return (Minus, state')
               _     -> do
                  (r'', state'') <- min' ps' state'
                  return (min r' r'', state'')
   max_ ps indent = do
         putStrLn " = max {"
         (r,_) <- max' ps (".  " ++ indent)
         putStr $ indent ++ "} "
         return (r,indent)
      where
      max' ps state = case ps of
         []      -> return (Minus, state)
         [p]     -> p state
         p : ps' -> do
            (r',state') <- p state
            case r' of
               Plus -> return (Plus, state')
               _    -> do
                  (r'', state'') <- max' ps' state'
                  return (max r' r'', state'')
```

```
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
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, indent)

   label_ls label func lambda f b ls indent p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show ls ++ ")"
      putStr $ indent ++ goal
      (r,_) <- p indent
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, indent)

   label_l label func lambda f b l indent p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show l ++ ")"
      putStr $ indent ++ goal
      (r,_) <- p indent
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, indent)
```

```
instance PlausibleRL IO String PRule Literal where

   label_r label func lambda f b r indent p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show r ++ ")"
      putStr $ indent ++ goal
      (r,_) <- p indent
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, indent)
```

```
instance PlausibleTRL IO String PTheory PRule Literal where
   {}
```

5.20.4 Optimised instances of PlausubleX

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

```
type History = M.Map
   (TaggedCnfFormula OLiteral, [OLiteral]) ProofResult
```

```
instance Plausible Maybe History where { }

instance PlausibleL Maybe History OLiteral where { }

instance PlausibleRL
   Maybe History OPRule OLiteral where { }

instance PlausibleTRL
   Maybe History OPTheory OPRule OLiteral where

   p t tf b h = case M.lookup (tf,b) h of
      Nothing -> do
         (r,h') <- p' t tf b h
         return (r, M.insert (tf,b) r h')
      Just r ->
         return (r, h)
```

5.20.5 Noisy optimised instances of PlausubleX

These instances will use optimised data structures, and memoing using a `History`, but will provide some diagnotic 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 ++ ") = " ++ show (neg r)
      return (neg r, (indent,h))

   just_ result (indent,h) = do
      putStrLn " = ("
      (r,_) <- result (".  " ++ indent, h)
      putStr $ indent ++ ") "
      return (r,(indent,h))

   min_ ps (indent,h) = do
         putStrLn " = min {"
         (r,(_,h')) <- min' ps (".  " ++ indent, h)
         putStr $ indent ++ "} "
         return (r,(indent,h'))
      where
      min' ps state = case ps of
         []      -> return (Plus, state)
         [p]     -> p state
         p : ps' -> do
            (r',state') <- p state
            case r' of
               Minus -> return (Minus, state')
               _     -> do
                  (r'', state'') <- min' ps' state'
                  return (min r' r'', state'')
   max_ ps (indent,h) = do
         putStrLn " = max {"
         (r,(_,h')) <- max' ps (".  " ++ indent, h)
         putStr $ indent ++ "} "
         return (r,(indent,h'))
      where
      max' ps state = case ps of
         []      -> return (Minus, state)
         [p]     -> p state
         p : ps' -> do
            (r',state') <- p state
            case r' of
               Plus -> return (Plus, state')
               _    -> do
                  (r'', state'') <- max' ps' state'
                  return (max r' r'', state'')
```

```
instance PlausibleL IO (String,History) OLiteral where

   label_ label func lambda f b (indent,h) p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ")"
      putStr $ indent ++ goal
      (r,(_,h')) <- p (indent,h)
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, (indent, h'))
```

```
   label_ls label func lambda f b ls (indent,h) p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show ls ++ ")"
      putStr $ indent ++ goal
      (r,(_,h’)) <- p (indent,h)
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, (indent,h’))

   label_l label func lambda f b l (indent,h) p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show l ++ ")"
      putStr $ indent ++ goal
      (r,(_,h’)) <- p (indent,h)
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, (indent,h’))

instance PlausibleRL
   IO (String,History) OPRule OLiteral where

   label_r label func lambda f b r (indent,h) p = do
      let goal = func ++ "(" ++ show (Tag lambda f)
             ++ ", " ++ show b ++ ", " ++ show r ++ ")"
      putStr $ indent ++ goal
      (r,(_,h’)) <- p (indent, h)
      putStrLn $ "= " ++ show r
         ++ ", by " ++ label
      return (r, (indent,h’))

instance PlausibleTRL
   IO (String,History) OPTheory OPRule OLiteral where

   p t tf b (indent,h) = case M.lookup (tf,b) h of
      Nothing -> do
         (r,(_,h’)) <- p’ t tf b (indent,h)
         putStrLn $ indent ++ "new result: P" ++
            show (tf,b) ++ " = " ++ show r
         return (r, (indent, M.insert (tf,b) r h’))
      Just r -> do
         putStrLn $ indent ++ "used old result: P" ++
            show (tf,b) ++ " = " ++ show r
         return (r, (indent,h))
```

5.20.6 Do one proof with a description

`doProof` *D ls tf* does one proof for inputs *ls* and output *tf* with description *D*, printing traces, and returning the result.

```
doProof :: Description -> [Literal] ->
   TaggedCnfFormula Literal -> IO ProofResult
doProof d ls tf = do
   let t = makeTheory $ generateAxioms d ls
   (r,_) <- p t tf [] ""
   return r
```

`doProofSilently` *D ls tf* does one proof for inputs *ls* and output *tf* with description *D*, returning the result.

```
doProofSilently :: Description -> [Literal] ->
   TaggedCnfFormula Literal -> ProofResult
doProofSilently d ls tf =
   let t = makeTheory $ generateAxioms d ls
       Just (r,_) = p t tf [] ()
   in r
```

5.20.7 Do all proofs for a description

`doProofs` *D* does all the proofs for all the inputs and outputs specified in description *D*, printing traces, and returning the added axioms and results for each set of axioms.

```
doProofs :: Description -> IO [([Literal],[ProofResult])]
doProofs d = do
   let proveD :: [Literal] -> IO ([Literal],[ProofResult])
       proveD axs = do
          putStrLn $ "\nAxioms: " ++ show axs
          let t = makeTheory $ generateAxioms d axs
              prove1 :: TaggedCnfFormula Literal
                 -> IO (ProofResult, String)
              prove1 tf = p t tf [] ""
          print t
          ris <- mapM (prove1 . fst) $ dout d
          return (axs, map fst ris)
   mapM proveD (generateRuns d)
```

`doProofsSilently` *D* does all the proofs for all the inputs and outputs specified in description *D*, returning the added axioms and results for each set of axioms.

```
doProofsSilently ::
   Description -> [([Literal],[ProofResult])]
doProofsSilently d =
   let proveD :: [Literal] -> ([Literal],[ProofResult])
       proveD axs =
          let t = makeTheory $ generateAxioms d axs
              prove1 :: TaggedCnfFormula Literal ->
                 (ProofResult, ())
              prove1 tf =
                 let Just result = p t tf [] ()
                 in result
              ris = map (prove1 . fst) $ dout d
          in (axs, map fst ris)
   in map proveD (generateRuns d)
```

`doProofsQuicklySilently` *D* does all the proofs for all the inputs and outputs specified in description *D*, returning the added axioms and results for each set of axioms. It does it using optimised theories for speed.

```
doProofsQuicklySilently ::
   Description -> IO [([Literal],[ProofResult])]
doProofsQuicklySilently d = do
   let proveD ::
         ([Literal], Int) -> IO ([Literal],[ProofResult])
       proveD (axs,i) = do
          let ot = toOTheory d $ makeTheory $
                   generateAxioms d axs
              proveAll :: History ->
                 [TaggedCnfFormula Literal] ->
                 [ProofResult]
              proveAll h tfs = case tfs of
                 []                     -> []
                 Tag lambda lss : tfs’ ->
                    let otf = Tag lambda $ map
                           (map (toOLiteral (otam ot))) lss
                        Just (r,h’) = p ot otf [] h
                    in r : proveAll h’ tfs’
              rs = proveAll M.empty $ map fst $ dout d
          rs ‘deepseq‘ hPutStrLn stderr $ show i
          return (axs, rs)
       runs = generateRuns d
   hPutStrLn stderr $ show $ length $!! runs
   mapM proveD $ zip (runs) (reverse [0..length runs - 1])
```

`doProofsQuickly` *D* does all the proofs for all the inputs and outputs specified in description *D*, returning the added axioms and results for each set of axioms. It does it using optimised theories for speed.

```
doProofsQuickly ::
   Description -> IO [([Literal],[ProofResult])]
doProofsQuickly d =
   let proveD :: [Literal] -> IO ([Literal],[ProofResult])
       proveD axs = do
          putStrLn $ "\nAxioms: " ++ show axs
          let ot = toOTheory d $ makeTheory $
                   generateAxioms d axs
              proveAll :: History ->
                 [TaggedCnfFormula Literal] ->
                 IO [ProofResult]
              proveAll h tfs = case tfs of
                 []                     ->
                    return []
                 Tag lambda lss : tfs’ -> do
                    let otf = Tag lambda $ map
                           (map (toOLiteral (otam ot))) lss
                    (r,(_,h’)) <- p ot otf [] ("",h)
                    rs <- proveAll h’ tfs’
                    return $ r : rs
          print ot
          rs <- proveAll M.empty $ map fst $ dout d
```

```
          return (axs, rs)
   in mapM proveD (generateRuns d)
```

`proofsTable` *D runs* 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.Descriptions
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 ()
makeCExprs d t simpC = do
   let cases :: [((TaggedCnfFormula Literal, Maybe String),
          [(ProofResult,[Literal])])]
       cases = [(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') = do
          let lss = [ls | (Plus,ls) <- t']
          lss' <- if simpC then simplify lss
                           else return lss
          let cex = makeCExpr lss'
              cex' = unlines $
                 map (\cs -> "   " ++ cs ++ " \\") $
                 lines cex
          case ms of
             Nothing  -> putStrLn $ show tf ++ " =\n\n   "
                ++ cex
             Just s   -> do
                putStr $ "#define " ++ unString s ++
                   " ( \\\n   " ++ cex'
                putStrLn ")\n"
   mapM_ displayCase cases
```

`makeCExpr` *lss* renders the formula *lss* in something close to C.

```
makeCExpr :: [[Literal]] -> String
makeCExpr lss =
   let ccc lss = case lss of
              []  -> "false"
              lss ->
                 concat $ intersperse "|| " $ map cc lss
       cc ls = case ls of
             [] -> "true"
             ls -> drop 3 $ unlines $ map ("   " ++ ) $
                   wordWrap 71 $ concat $
                   intersperse " && " $ map c ls
       c l = case l of
          Pos a -> unString $ show a
          Neg a -> ('!' :) $ unString $ show a
   in ccc 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 lss
      bss' <- qmSimplify bss
      return $ fromBitss ps $ bss'
```

Conversion to bits

`toBit` *l* converts literal *l* to a bit.

```
toBit :: Literal -> QMBit
toBit l = case l of
   Pos _ -> One
   Neg _ -> Zer
```

`toBits` *ls* converts a list of literals *ls* to a list of bits.

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

`toBitss` *lss* converts lists of literals *lss* to lists of bits.

```
toBitss :: [[Literal]] -> [[QMBit]]
toBitss = map toBits
```

Conversion back to literals

`fromBit` *p b* converts bit *b* back to a literal, using the *positive* literal *p* as a reference.

```
fromBit :: Literal -> QMBit -> Maybe Literal
fromBit p b = case b of
   Zer -> Just $ neg p
   Dsh -> Nothing
   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 ps bs = catMaybes $ zipWith fromBit ps bs
```

`fromBitss` *ps bss* converts lists of bits to lists of literals, using the positive references *ps*.

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

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
```

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

infixr 5 +-+, +^+
```

5.22.1 Constructing C

`displayComment` *h css* writes lines *css* to file with handle *h*, encapusulated in a C multi-line comment.

```
displayComment :: Handle -> [String] -> IO ()
displayComment hdl css = do
   hPutStrLn hdl "/*"
   hPutStr hdl $ unlines $ 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"
   ]
```

Class `AsC` overloads `asC`. `asC` *x* renders *x* as a C identifer.

```
class AsC a where

   asC :: a -> String
```

`asCchars` *cs* replaces characters that occur in *cs* that may not occur in a C identifier with underscores.

```
asCchars :: String -> String
asCchars cs =
   let leading = case cs of
          '\"' : _ -> "q_"
          _        -> ""
       cs' = filter (`notElem` " \"") cs
       replace [] = []
       replace cs'@(c:cs)
          | "<=" `isPrefixOf` cs'
             = "_LE_" ++ replace (tail cs)
          | "==" `isPrefixOf` cs'
             = "_EQ_" ++ replace (tail cs)
          | c == '<'
             = "_LT_" ++ replace cs
          | c == '_' || isAlphaNum c
             = c : replace cs
          | otherwise
             = '_' : replace cs
   in leading ++ replace cs'
```

5.22.2 The transformation

`makeCTheory` *t* outputs *t* as C definitions to files `theory.h` and `theory.c`.

```
makeCTheory :: FilePath -> Description -> Theory -> IO ()
makeCTheory path d t = do
```

`ot` is the optimised theory. The atoms in this theory will have optimised values 1 . . . `maxAtom`. `name` is the identifier for this theory. It defaults to the capitalised description file name, but may be overridden by a name declaration. `tag` *cs* qualifies a symbol with this name. `numStrict`, `numPlaus`, `numDef` are the numbers of strict, plausible, and defeater rules, respectively. `numRules` is the sum of those. The inputs. `inputs` is the list of all literals that are inputs. `numLabels` is the number of optimised labels (> 0).

```
   let ot :: OTheory
       ot = toOTheory d t
       (1,maxAtom) = bounds (otoam ot)
       name :: String
       name = if dnam d /= "/DEFAULT/" then dnam d else
          map toUpper $ reverse $ takeWhile isAlpha $
          dropWhile (not . isAlpha) $ dropWhile isAlpha $
          reverse path
       tag :: String -> String
       tag cs = name +-+ cs
       numPlaus = length (trp t)
       numDef = length (trd t)
       numStrict = length (trs t)
       numRules = numPlaus + numDef + numStrict
       inputs :: [Literal]
       inputs = snub $ map pos $ concat $ din d
       ((_,1),(_,numLabels)) = bounds $ otrpls ot
       outputs :: [Literal]
       outputs = snub $ map pos $ concat [concat lss |
          (Tag _ lss,_) <- dout d]
```

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. `putHln` and `putCln` println 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 $ "\n#ifndef " +-+ tag "THEORY_H_\n\
      \#define " +-+ tag "THEORY_H_\n"
```

The theory prefix.

```
   putHln $ "#ifndef THEORY_PREFIX\n\
      \#define THEORY_PREFIX" +^+ name ++" \n\
      \#endif\n"
```

Define the stack size.

```
   putHln $ "#define" +^+ tag "STATICSTACKSIZE 32768\n\
      \#ifndef STATICSTACKSIZE\n\
      \#define STATICSTACKSIZE" +^+ tag "STATICSTACKSIZE\n\
      \#endif\n"
```

Import header files.

```
   putHln "#include \"plaus.h\"\n"
   putCln "\n#include \"plaus.h\""
   putCln $ "#include \"" ++ tag "theory.h\"\n"
```

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.

```
   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 ::
          (ORIndex -> 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 :: (OAIndex -> IO ()) -> IO ()
       forAnts f =
          let (lo,hi) = bounds $ otants ot
          in mapM_ f [lo .. hi]
       forInputs, rofInputs:: (Literal -> IO ()) -> IO ()
       forInputs f = mapM_ f inputs
       rofInputs f = mapM_ f rinputs
       rinputs = 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 (\i -> putHln $ "#define" +^+
      lJustify 40 (tag "LN" +-+ asC (otoam ot ! i)) ++ " "
      ++ rJustify 10 (asC (negate i)))
   putHln $ "#define" +^+ lJustify 40 (tag "L_NONE") ++ " "
      ++ rJustify 10 "0"
   forAtoms (\i -> putHln $ "#define" +^+
      lJustify 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 (\i -> putHln $ "#define" +^+
      lJustify 40 (tag "AN" +-+ asC (otoam ot ! i)) ++ " "
      ++ rJustify 10 (show (maxAtom - i)))
   putHln $ "#define" +^+ lJustify 40 (tag "A_NONE") ++ " "
      ++ rJustify 10 (show maxAtom)
   forAtoms (\i -> putHln $ "#define" +^+
      lJustify 40 (tag "AP" +-+ asC (otoam ot ! i)) ++ " "
      ++ rJustify 10 (show (maxAtom + i)))
```

Counts.

```
   putHln $ "\n#define" +^+ tag "A_OFFS\t" ++ tag "A_NONE\
      \\t /* Add this to literals to get array index. \
      \*/\n\n#define" +^+ tag "NUM_LITERALS\t" ++
      show (2 * maxAtom + 1) ++ "\t /* This includes " ++
      tag "L_NONE */"
   putHln $ "\n#define" +^+ tag "THEORY_LITFACTS\t0\t\
      \/* facts stated in the theory that are in lit */\n"

   displayComment dotH ["array indices for the facts gen\
      \erated from inputs"]
   let factIndex :: (Literal,Int) -> IO ()
       factIndex (l,i) = putHln  $ "#define" +^+
          lJustify 40 (tag "INP" +-+ asC l) ++ " (" ++
          tag "THEORY_LITFACTS + " ++ show i ++ ")"
   mapM_ factIndex $ zip inputs [0..]
   putHln $ "\n#define" +^+ tag "NUM_INPUTS\t" ++
      show (length inputs) ++ "\t/* determines the number\
      \ of facts */"
   putHln $ "\n#define" +^+ tag "THEORY_FACTS\t0\t/* total\
      \ number of facts stated in the theory */\n"
   putHln $ "#define" +^+ tag "NUM_FACTS\t(" ++
      tag "THEORY_FACTS + " ++ tag "NUM_INPUTS)\
      \\t/* total number of facts */\n"
   putHln $ "#define" +^+ tag "THEORY_NAME " ++
      tag "Theory"
   putHln $ "#define" +^+ tag "THEORY_CHK" +^+ tag "chk\n"
   putHln $ "#define" +^+ tag "NUM_NONSTRICT" +^+
      show (numPlaus + numDef) ++ "\n"
```

Arrays and methods used for generality

```
displayComment dotH [tag "inputs is an array of \
   \strings denoting the inputs"]
putHln $ "extern char *" ++ tag "inputs[" ++
   tag "NUM_INPUTS+1];\n"
displayComment dotH [tag "outputs is an array of \
   \strings denoting the outputs"]
putHln $ "#define" +^+ tag "NUM_OUTPUTS\t" ++
   show (length outputs) ++ "\t/* total number of decla\
   \red outputs */\n"
putHln $ "extern char *" ++ tag "outputs[" ++
   tag "NUM_OUTPUTS+1];\n"
displayComment dotH [tag "atoms is an array of \
   \strings denoting the axioms"]
putHln $ "extern char *" ++ tag "atoms[" ++
   tag "A_NONE+1];\n"
displayComment dotH [tag "proofTags is an array of \
   \strings denoting the proofs"]
putHln $ "extern char *" ++ tag "proofTags[5];\n"
displayComment dotH ["these are the inputs, used in \
   \generic checking method '" ++ tag "chk'"]
putHln "enum\n{"
forInputs (\x -> putHln $ "\t" ++ tag (asC x) ++ ",")
putHln $ "\n\t" ++ tag "numInputs\n};\n"
displayComment dotH [tag "chk(i,v) is used to check an\
   \ input with index i to the value v"]
putHln $ "int " ++ tag "chk(int input, int value);\n"
```

Defining the rule numbers.

```
displayComment dotH [
   "these are the rule numbers",
   "",
   tag "Rxx: rule",
   "",
   "the scrict rules also contains pre-computed rule \
    \numbers for inputs",
   "that turn into facts and strict rules once \
    \asserted:",
   "",
   tag "RS_N_xx: strict rules for negative inputs",
   tag "RS_P_xx: strict rules for positive inputs"]
putHln $ "enum " ++ tag "Ruleset\n{\n"
forRules (\i -> putHln $ "\t" ++ tag "R" ++ show i
   ++ ",")
putHln "\n/* The strict rules below only become active\
   \ for the corresponding inputs. */\n"
forInputs (\x -> putHln $ "\t" ++ tag "RS_N" +-+ (asC x)
   ++ ",")
rofInputs (\x -> putHln $ "\t" ++ tag "RS_P" +-+ (asC x)
   ++ ",")
putHln $ "\n\t" ++ tag "NUM_RULES"
putHln "};\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 "FACT(inp, p, n) do {\t\\\n\t\t" ++
        tag "RS1[" ++ tag "A##p##_##inp]\t= " ++
        tag "RS1##p##_Fact_##inp;\t\\\n\t\t" ++
        tag "RS1[" ++ tag "A##n##_##inp]\t= " ++
        tag "RS1_NONE;\t\\\n\t\t" ++
        tag "R1[" ++ tag "A##p##_##inp]\t= " ++
        tag "R1##p##_Fact_##inp;\t\\\n\t\t" ++
        tag "R1[" ++ tag "A##n##_##inp]\t= " ++
        tag "R1_NONE;\t\\\n\t\t" ++
        tag "Inc[" ++ tag "A##p##_##inp]\t= " ++
```

```
      tag "Inc_NONE;\t\\\n\t\t" ++
      tag "Inc[" ++ tag "A##n##_##inp]\t= &" ++
      tag "NRAnts[" ++
      tag "RS_##n##_##inp];\t\\\n\t\t" ++
      tag "Facts[" ++ tag "INP_##inp]\t= " ++
      tag "Fact##p##_##inp;\t\\\n\t" ++
      "} while (0)\n"

putHln $ "#define " ++ tag "CHK(inp,v) if (v) " ++
   tag "FACT( inp , P, N); else " ++ tag "FACT( inp ,\
   \ N, P);\n"
```

Defining macros to reset theory status

```
displayComment dotH ["\"unset\" facts: reset the theory\
      \ back to before a fact of an input was known",
   "this does not properly support partial resetting\
      \ at this stage" ]
putHln ("#define" +^+
   tag "TFRESET(inp) do {\t\\\n\t\t" ++
   tag "RSl[AP_##inp]\t= " ++
   tag "RSlP_##inp;\t\\\n\t\t" ++
   tag "RSl[AN_##inp]\t= " ++
   tag "RSlN_##inp;\t\\\n\t\t" ++
   tag "Rl[AP_##inp]\t= " ++
   tag "RlP_##inp;\t\\\n\t\t" ++
   tag "Rl[AN_##inp]\t= " ++
   tag "RlN_##inp;\t\\\n\t\t" ++
   tag "Inc[AP_##inp]\t= " ++
   tag "IncP_##inp;\t\\\n\t\t" ++
   tag "Inc[AN_##inp]\t= " ++
   tag "IncN_##inp;\t\\\n\t\t" ++
   tag "Facts[INP_##inp]\t= NULL;\t\\\n\t" ++
   "} while (0)\n")
putHln $ "#define" +^+ tag "THEORESET() do {\t\\"
forInputs (\x -> putHln $ "\t\t" ++ tag "TFRESET(" ++
   asC x ++ ");\t\\")
putHln "\t} while (0)\n"
```

Declare all theory table references

```
displayComment dotH ["theory table references needed \
   \by the above macros"]
putHln $ "\nextern sstack_t " ++ tag "Stack;\n"
```

Facts

```
forInputs (\x -> putHln $ "extern osetelement_t " ++
   tag "FactN" +-+ asC x ++ "[];")
rofInputs (\x -> putHln $ "extern osetelement_t " ++
   tag "FactP" +-+ asC x ++ "[];")
putHln $ "\nextern oset_t *" ++
   tag "Facts[" ++ tag "NUM_FACTS + 1];\n"
```

Declare positive and negative antecedents

```
forAnts (\i -> putHln $ "extern literal_t" +^+ tag "Ant"
   ++ show i ++ "[];")
putHln $ "\nextern literal_t " ++ tag "Ant_Empty[];\n"
putHln $ "extern cnf_t " ++ tag "RAnts[" ++
   tag "NUM_RULES+1];\n"
forAnts (\i -> putHln $ "extern literal_t" +^+
   tag "NAnt" ++ show i ++ "[];")
putHln $ "\nextern cnf_t " ++
   tag "NRAnts[" ++ tag "NUM_RULES+1];\n"
```

Declare variables to be used for storing all rules by consequent.

```
rofAtoms (\i -> putHln $ "extern ruleno_t " ++ tag "RlN"
   +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "extern ruleno_t " ++ tag "Rl_NONE[];"
forAtoms (\i -> putHln $ "extern ruleno_t " ++ tag "RlP"
   +-+ asC (otoam ot ! i) ++ "[];")
putHln ""
forInputs (\x -> putHln $ "extern ruleno_t " ++
   tag "RlN_Fact" +-+ asC x ++ "[];")
rofInputs (\x -> putHln $ "extern ruleno_t " ++
   tag "RlP_Fact" +-+ asC x ++ "[];")
putHln $ "\nextern oset_t *" ++ tag "Rl[];\n"
```

Declare variables to be used for storing all strict rules by consequent.

```
rofAtoms (\i -> putHln $ "extern ruleno_t " ++
   tag "RSlN" +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "extern ruleno_t " ++ tag "RSl_NONE[];"
forAtoms (\i -> putHln $ "extern ruleno_t " ++
   tag "RSlP" +-+ asC (otoam ot ! i) ++ "[];")
putHln ""
forInputs (\x -> putHln $ "extern ruleno_t " ++
   tag "RSlN_Fact" +-+ asC x ++ "[];")
rofInputs (\x -> putHln $ "extern ruleno_t " ++
   tag "RSlP_Fact" +-+ asC x ++ "[];")
putHln $ "\nextern oset_t *" ++ tag "RSl[];\n"
```

Declare variables to be used for storing all plausable rules by consequent.

```
rofAtoms (\i -> putHln $ "extern ruleno_t " ++
   tag "RPlN" +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "extern ruleno_t " ++ tag "RPl_NONE[];"
forAtoms (\i -> putHln $ "extern ruleno_t " ++
   tag "RPlP" +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "\nextern oset_t *" ++ tag "RPl[];\n"
```

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

```
rofAtoms (\i -> do
   forRules (\x -> putHln $ "extern ruleno_t " ++
      tag "RPlsN" +-+ asC (otoam ot ! i) +-+ "R" ++
      asC x ++ "[];")
   putHln ("\nextern oset_t *" ++ tag "RPlsN" +-+
      asC (otoam ot ! i) ++ "[];\n")
  )
forRules (\x -> putHln $ "extern ruleno_t " ++
   tag "RPls_NONE_R" ++ asC x ++ "[];")
putHln ("\nextern oset_t *" ++ tag "RPls_NONE_[];\n")
forAtoms (\i -> do
   forRules (\x -> putHln $ "extern ruleno_t " ++
      tag "RPlsP" +-+ asC (otoam ot ! i) +-+ "R"
      ++ asC x ++ "[];")
   putHln ("\nextern oset_t *" ++ tag "RPlsP" +-+
      asC (otoam ot ! i) ++ "[];\n")
  )
putHln $ "extern oset_t **" ++ tag "RPls[];\n"
```

Declare variables for storing inpompatibilites

```
rofAtoms (\i -> putHln $ "extern literal_t " ++
   tag "IncNP" +-+ asC (otoam ot ! i) ++ "[];")
forAtoms (\i -> putHln $ "extern literal_t " ++
   tag "IncPN" +-+ asC (otoam ot ! i) ++ "[];")
putHln ""

rofAtoms (\i -> putHln $ "extern cnf_t " ++
   tag "IncN" +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "extern cnf_t " ++ tag "Inc_NONE[];"
forAtoms (\i -> putHln $ "extern cnf_t " ++
   tag "IncP" +-+ asC (otoam ot ! i) ++ "[];")
putHln $ "\nextern cnf_t *" ++ tag "Inc[];\n"

putHln $ "extern int " ++ tag "nInc[];\n"
putHln $ "extern theory_t " ++ tag "Theory;\n"
```

Start *theory.c* file. Define string arrays holding input/atom names. This code is currently only used by *testct.c*.

```
  displayComment dotC [
     "inputs is an array of strings denoting the inputs" ]
  putCln $ "char *" ++ tag "inputs[] = { " ++
     concatMap (\x -> "\"" ++ asC x ++
     "\", ") inputs ++ "NULL };\n"
  displayComment dotC ["outputs is an array of strings \
     \denoting the outputs"]
  putCln $ "char *" ++ tag "outputs[] = { " ++
     concatMap (\x -> "\"" ++ asC x ++
     "\", ") outputs ++ "NULL };\n"
  displayComment dotC [
     "atoms is an array of strings denoting the axioms" ]
  putCln $ "char *" ++ tag "atoms[] = { \"NONE\"" ++
     concatMap (\i -> ", \"" ++ asC (otoam ot ! i)
     ++ "\"") [1 .. maxAtom] ++ " };\n"
--   displayComment dotC [
--      "proofTags is an array of strings denoting the \
--      \proofs" ]
```

```
--   putCln $ "char *" ++ tag "proofTags[] = {\"mu\", \
--      \\"alpha\", \"pi\", \"beta\", \"delta\" };\n"
   displayComment dotC [
      "cooler_chk(input, value) is used to check an input \
      \with index input to the value" ]
   putCln $ "int " ++ tag "chk(int input, int value)\n\
      \{\n\
      \\tswitch (input)\n\
      \\t{"
   forInputs (\x -> putC $ "\t\tcase" +^+ tag (asC x) ++
      ":\n\t\t\t" ++ tag "CHK(" ++ asC x ++ ", value);\n\
      \\t\t\tbreak;\n\n")
   putCln "\t\tdefault:\n\
      \\t\t\treturn -1;\n\
      \\t}\n\n\
      \\treturn 0;\n\
      \}\n"
```

Define stack size

```
   displayComment dotC [
      "just create a static stack of " ++
      tag "STATICSTACKSIZE elements" ]
   putCln $ "sstack_t " ++ tag "Stack = { " ++
      tag "STATICSTACKSIZE, " ++ tag "Stack.element };\n"
```

Define facts ***NEED TO TEST***

```
   let litToC n | n > 0     =
          tag "LP" +-+  asC (otoam ot ! n)
                | otherwise =
          tag "LN" +-+ asC (otoam ot ! abs n)
       antsToC f []  = "0"
       antsToC f [x] = "1, " ++ f x
       antsToC f xs  = show (2*length xs) ++
          concatMap (\y -> ", " ++ f y ++ " , 0 ") xs
   mapM_ (\(n,x) -> putCln $ "osetelement_t " ++
      tag "Fact" ++ show n ++ "[] = { " ++
         case x of
            []  -> "0"
            [y] -> "1, " ++ litToC y
            _   -> show (2*length x) ++ concatMap
               (\y -> ", " ++ litToC y ++ ", 0") x
         ++ " };"
      ) (zip [1..] (otfct ot))
```

Define facts for inputs

```
   displayComment dotC [
      "pre-computed facts for inputs used \
          \for actual proofs:" ]
   forInputs (\x -> putCln $ "osetelement_t " ++
      tag "FactN" +-+ asC x ++ "[] = { 2, " ++ tag "LN"
      +-+ asC x ++ ", 0 };")
   rofInputs (\x -> putCln $ "osetelement_t " ++
      tag "FactP" +-+ asC x ++ "[] = { 2, " ++ tag "LP"
      +-+ asC x ++ ", 0 };")
   putCln ""

   putCln $ "oset_t *" ++
      tag "Facts[" ++ tag "NUM_FACTS + 1] =\n{"
   putCln "\tNULL\n};\n"
```

Define antecedents for all rules

```
   displayComment dotC [
      "these are the antecedents of all the rules",
          "",
          "the first parameter is the size of the \
          \following formula",
      "sets of clauses must always be 0-terminated!" ]
   forAnts (\i -> putCln $ "literal_t" +^+ tag "Ant" ++
      show i ++ "[] = {" +^+ antsToC litToC (otants ot ! i)
      +^+ "};")
   putCln ""
   displayComment dotC ["pre-computed empty list anteced\
      \ent for inputs that become facts:" ]
   putCln $ "literal_t " ++ tag "Ant_Empty[] = { 0 };\n"
```

Bind rules to their antecedents.

```
   let showAntIndex :: ORIndex -> String
       showAntIndex r =
          let ORule _ _ a _ = otrs ot ! r
          in show a
   putCln $ "cnf_t" +^+ tag "RAnts["
      ++ tag "NUM_RULES+1] =\n{"
   forRules (\i -> putCln $ "\t(cnf_t)" +^+ tag "Ant" ++
      showAntIndex i ++ ",")
   putCln "\n\t/* pre-computed strict rules for \
      \inputs that will become facts: */"
   mapM_ (\_ -> putCln $ "\t(cnf_t) " ++
      tag "Ant_Empty,") [1..2*length inputs]
   putCln "\n\tNULL\n};\n"
```

Define negative antecendents for all rules and bind rules to them

```
   displayComment dotC [
      "negative antecedents: same as above, \
          \but literals are reversed" ]
   let antsToC' f [] = "0"
       antsToC' f xs = show (length xs) ++ concatMap
          (\y -> ", " ++ f y) xs
   forAnts (\i -> putCln $ "literal_t" +^+ tag "NAnt" ++
      show i ++ "[] = { " ++ antsToC' litToC (map (neg)
      (otants ot ! i)) ++ " };")
   putCln ""
   putCln $ "cnf_t" +^+ tag "NRAnts[" ++
      tag "NUM_RULES+1] =\n{"
   forRules (\i -> putCln $ "\t(cnf_t)" +^+ tag "NAnt" ++
      showAntIndex i ++ ",")
   putCln "\n\t/* pre-computed strict rules for \
      \inputs (used once they become facts): */"
   mapM_ (\_ -> putCln $ "\t(cnf_t) " ++
      tag "Ant_Empty,") [1..2*length inputs]
   putCln "\n\tNULL\n};\n"
```

Define rules, indexed by consequent

```
   displayComment dotC [
      "All the rules, indexed by consequent" ]
   let listRules :: [ORIndex] -> String
       listRules rs = "[] = {" +^+ show (length rs) ++
          concatMap (\r -> "," +^+ tag "R" ++ show r) rs
          +^+ "};"
       listRules' :: [ORule] -> String
       listRules' rs = "[] = {" +^+ show (length rs) ++
          concatMap (\(ORule i _ _ _) -> "," +^+ tag "R"
          ++ show i) rs +^+ "};"
   rofAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RlN" +-+
      asC (otoam ot ! i) ++ listRules (otrl ot ! (neg i)))
   putCln $ "ruleno_t " ++ tag "Rl_NONE" ++
      listRules (otrl ot ! 0)
   forAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RlP" +-+
      asC (otoam ot ! i) ++ listRules (otrl ot ! i))
```

Define pre-computed rules that need to be used for inputs that have become facts.

```
   displayComment dotC [
      "pre-computed rules that need to be used for \
      \inputs that have become facts:" ]
   forInputs (\x -> putCln $ "ruleno_t " ++ tag "RlN_Fact"
      +-+ asC x ++ "[] = { 1, " ++ tag "RS_N" +-+
      asC x ++ " };")
   rofInputs (\x -> putCln $ "ruleno_t " ++ tag "RlP_Fact"
      +-+ asC x ++ "[] = { 1, " ++ tag "RS_P" +-+
      asC x ++ " };")
   putCln $ "\noset_t *" ++ tag "Rl[] =\n{"
   rofAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RlN"
      +-+ asC (otoam ot ! i) ++ ",")
   putCln $ "\t(oset_t *) " ++ tag "Rl_NONE,"
   forAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RlP"
      +-+ asC (otoam ot ! i) ++ ",")
   putCln "\tNULL\n};\n"
```

Define strict rules, indexed by consequent

```
   displayComment dotC [
      "All strict rules, indexed by consequent" ]
   rofAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RSlN" +-+
      asC (otoam ot ! i) ++ listRules (otrsl ot ! (neg i)))
   putCln $ "ruleno_t " ++ tag "RSl_NONE" ++
      listRules (otrsl ot ! 0)
   forAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RSlP" +-+
      asC (otoam ot ! i) ++ listRules (otrsl ot ! i))
   putCln ""
```

```
displayComment dotC [
   "pre-computed strict rules for inputs.",
   "these are not active by default, but are used",
   "for those that inputs turn into facts" ]
forInputs (\x -> putCln $ "ruleno_t " ++ tag "RS1N_Fact"
   +-+ asC x ++ "[] = { 1, " ++ tag "RS_N" +-+ asC x
   ++ " };")
rofInputs (\x -> putCln $ "ruleno_t " ++ tag "RS1P_Fact"
   +-+ asC x ++ "[] = { 1, " ++ tag "RS_P" +-+ asC x ++
   " };")
putCln $ "\noset_t *" ++ tag "RSl[] =\n{"
rofAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RS1N"
   +-+ asC (otoam ot ! i) ++ ",")
putCln $ "\t(oset_t *) " ++ tag "RSl_NONE,"
forAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RS1P"
   +-+ asC (otoam ot ! i) ++ ",")
putCln "\tNULL\n};\n"
```

Define plausible rules, indexed by consequent

```
displayComment dotC [
   "All plausible rules, indexed by consequent" ]
rofAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RP1N" +-+
   asC (otoam ot ! i) ++ listRules (otrpl ot ! neg i))
putCln $ "ruleno_t " ++ tag "RPl_NONE" ++
   listRules (otrpl ot ! 0)
forAtoms (\i -> putCln $ "ruleno_t" +^+ tag "RP1P" +-+
   asC (otoam ot ! i) ++ listRules (otrpl ot ! i))
putCln $ "\noset_t *" ++ tag "RPl[] =\n{"
rofAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RP1N"
   +-+ asC (otoam ot ! i) ++ ",")
putCln $ "\t(oset_t *) " ++ tag "RPl_NONE,"
forAtoms (\i -> putCln $ "\t(oset_t *) " ++ tag "RP1P"
   +-+ asC (otoam ot ! i) ++ ",")
putCln "\tNULL\n};\n"
```

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

```
putCln ""
displayComment dotC [
   "Rules that beat another rule, indexed by \
       \consequent and RULES that get beaten" ]
let listRpls :: OLiteral -> ORIndex -> String
    listRpls c r =
       let ORule _ l _ _ = otrs ot ! r
       in listRules (otrpls ot ! (c, l))
rofAtoms (\i -> do
   let inp = asC (otoam ot ! i)
   forPD (\r -> putCln $ "ruleno_t " ++ tag "RPlsN" +-+
      inp +-+ "R" ++ show r ++ listRpls (neg i) r)
   putCln $ "\noset_t *" ++ tag "RPlsN" +-+
      inp ++ "[] =\n{"
   forPD (\r -> putCln $ "\t(oset_t *) " ++
       tag "RPlsN" +-+ inp +-+ "R" ++ show r ++ ",")
   putCln "\tNULL\n};\n"
  )
forPD (\r -> putCln $ "ruleno_t " ++ tag "RPls_NONE_R"
   ++ show r ++ listRpls 0 r)
putCln $ "\noset_t *" ++ tag "RPls_NONE[] =\n{"
forPD (\r -> putCln $ "\t(oset_t *) " ++
   tag "RPls_NONE_R" ++ show r ++ ",")
putCln "\tNULL\n};\n"
forAtoms (\i -> do
   let inp = asC (otoam ot ! (abs i))
   forPD (\r -> putCln $ "ruleno_t " ++ tag "RPlsP" +-+
      inp +-+ "R" ++ show r ++ listRpls i r)
   putCln $ "\noset_t *" ++ tag "RPlsP" +-+ inp
      ++ "[] =\n{"
   forPD (\r -> putCln $ "\t(oset_t *) " ++ tag "RPlsP"
      +-+ inp +-+ "R" ++ show r ++ ",")
   putCln "\tNULL\n};\n"
  )

putCln $ "oset_t **" ++ tag "RPls[] =\n{"
rofAtoms (\i -> putCln $ "\t" ++ tag "RPlsN" +-+
   asC (otoam ot ! i) ++ ",")
putCln $ "\t" ++ tag "RPls_NONE,"
forAtoms (\i -> putCln $ "\t" ++ tag "RPlsP" +-+
   asC (otoam ot ! i) ++ ",")
putCln "\tNULL\n};\n"
```

Define the incompatibilities.

```
displayComment dotC ["incompatibilities"]
let getIncs i j k = putC $ ", " ++
       litToC (((otincl ot ! i) !! j) !! k)
    getIncs’ cs f i j = do
       putC $ "literal_t " ++ tag cs +-+
          asC (otoam ot ! i) +-+ show j ++ "[] = { " ++
          show (length ((otincl ot ! (f i)) !! j))
       mapM_ (getIncs (f i) j)
          [0..length ((otincl ot ! (f i)) !! j) - 1]
       putCln " };"
    getIncs’’ cs f i = mapM_ (getIncs’ cs f i)
       [0 .. length (otincl ot ! (f i)) - 1]
rofAtoms (getIncs’’ "IncNP" neg)
forAtoms (getIncs’’ "IncPN" id)
putCln ""

let printIncs i n np = do
       let inp = asC (otoam ot ! abs i)
           printIncs’ j =
              putC $ "(cnf_t)" +^+ tag np +-+ inp +-+
                 show j ++ ", "
       putC $ "cnf_t" +^+ tag n +-+
          asC (otoam ot ! abs i) ++ "[] = { "
       mapM_ printIncs’ [0 .. length (otincl ot ! i)-1]
       putCln "NULL };"
rofAtoms (\i -> printIncs (neg i) "IncN" "IncNP")
putCln $ "cnf_t " ++ tag "Inc_NONE[] = { NULL };"
forAtoms (\i -> printIncs i "IncP" "IncPN")

putCln $ "\ncnf_t *" ++ tag "Inc[] =\n{"
rofAtoms (\i -> putCln $ "\t" ++ tag "IncN" +-+
   asC (otoam ot ! i) ++ ",")
putCln $ "\t" ++ tag "Inc_NONE,"
forAtoms (\i -> putCln $ "\t" ++ tag "IncP" +-+
   asC (otoam ot ! i) ++ ",")
putCln "\tNULL\n};\n"

putCln $ "int " ++ tag "nInc[] = { " ++
   show (length (otincl ot ! (-maxAtom))) ++
       concatMap (\i -> ", " ++
       show (length (otincl ot ! i)))
       [-maxAtom + 1 .. maxAtom] ++
       " };\n"
```

Define the theory

```
putCln $ "theory_t " ++ tag "Theory =\n{\n" ++
   "\t&" ++ tag "Stack,\n\t" ++ tag "Facts,\n\n\t" ++
   tag "RAnts,\n\t" ++ tag "NRAnts,\n\n\t" ++
   tag "Rl,\n\t" ++ tag "RSl,\n\t" ++ tag "RPl,\n\t" ++
   tag "RPls,\n\n\t" ++ tag "Inc,\n\t" ++
   tag "nInc,\n\t" ++ tag "A_OFFS,\n\t" ++
   tag "NUM_NONSTRICT\n};\n"
```

Write the header file closing stuff.

```
putHln $ "\n#endif /* " +-+ tag "THEORY_H_ */"
```

Close the files.

```
hClose dotH
hClose dotC
```

5.22.3 Instance declarations

```
instance AsC Atom where

   asC = asCchars . show

instance AsC OAtom where

   asC a = show a

instance AsC Literal where

   asC = asCchars . show
```

5.23 MakeHGlue

Module `MakeHGlue` generates a Haskell module with a function for each of the outputs specified in 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` is the name of this description, either declared in it or derived from its file name. `scriptName` is the output file name. `script` is the actual file handle.

```
   let name, scriptName :: String
       name = (\(c:cs) -> toUpper c : cs)
          (if dnam d /= "/DEFAULT/"
           then dnam d
           else reverse $ takeWhile isAlpha $
                dropWhile (not . isAlpha) $
                dropWhile 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.Descriptions", "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 = dout d
       args = case ((nofail . total) programL)
                   (preLex (unlines inputs)) of
          OK (tlps,_) -> snub [a | (("varid",a),_) <- tlps]
             \\ map unString (dpre d)
          Error _ _   -> error "DPL can not lex \
             \the inputs as Haskell."
   comment $ "The inputs are:" : inputs
   blank
   comment $ "The outputs are:" : map show outputs
   blank
   comment $ "The arguments will be:" : args ++
      ["the_description"]
```

Now generate the functions:

```
   let makeFun :: (TaggedCnfFormula Literal, Maybe String)
          -> IO ()
       makeFun (tf,mcs) = do
          let fname = case mcs of
                 Just n  -> unString n
                 Nothing -> concatMap (\c -> case c of
                       ’ ’ -> "_"
                       ’~’ -> "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
              cnf’ (ls:ls’:lss) = "[" ++ cnf’’ ls ++ ","
                 ++ cnf’ (ls’ : lss)
              cnf’ [ls] = "[" ++ cnf’’ ls ++ "]"
              cnf’ [] = "]"
              cnf’’ (l:l’:ls) = lit l ++ ","
                 ++ cnf’’ (l’:ls)
              cnf’’ [l] = lit l ++ "]"
              cnf’’ [] = "]"
              lit (Pos a) = "Pos (" ++ atm a ++ ")"
              lit (Neg a) = "Neg (" ++ atm a ++ ")"
              atm (Prop cs as) = "Prop " ++ show cs ++ " ["
                 ++ arg as
              arg (a:a’:as) = arg’ a ++ "," ++ arg (a’:as)
              arg [a] = arg’ a ++ "]"
              arg [] = "]"
              arg’ (Const (Constant cs)) = "Const \
                 \(Constant " ++ show cs ++ ")"
              arg’ (Var _) = error "DPL can’t generate a \
                 \ glue module -- variable in an input\
                 \ or an output."
              makeInp :: (Bool, Literal) -> IO ()
              makeInp (first,l) = do
                 put $ "      " ++
                    (if first then " " else ",") ++ "if "
                    ++ unString (show l)
                 put $ "       then " ++ lit l
                 put $ "       else " ++ lit (neg l)
          blank
          comment ["Required proof: "++ show tf,
                   "Optionally supplied name: " ++ show mcs,
                   "Selected name: " ++ fname]
          put $ unwords (fname : args ++
             ["the_description"]) ++ " ="
          put "   doProofSilently the_description ["
          mapM_ makeInp $ zip (True : repeat False) $
             concat $ din d
          put $ "      ] (" ++ "Tag " ++ tag ++ ")"
   mapM_ makeFun outputs
```

All done. Close.

```
   hClose script
```

5.24 DPL

Module `DPL` implements the proof tool for Decisive Plausible Logic.

```
module Main (main, run, runc) where

import System.Environment
import Control.Monad

import ABR.Control.Check
import ABR.Parser
import ABR.Util.Args
import ABR.Parser.Lexers
import ABR.Parser.Checks

import DPL.DPLLexer
import DPL.LitSets
import DPL.Descriptions
import DPL.Literals
import DPL.Inference
import DPL.Theories
import DPL.MakeCExprs
import DPL.OTheories
import DPL.MakeCTheory
import DPL.MakeHGlue
```

5.24.1 Compiler launch

`main` gets the names of the files to process and calls `run` to process each in turn.

```
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 "@LONELY" options
          else options
       appendNames = lookupQueue "a" options'
   options'' <- (do
      css <- mapM readFile appendNames
      let css' = zipWith (\n cs -> "\n/* *** appended: "
             ++ n ++ " *** */\n\n" ++ cs) appendNames css
      return $ insertParam "@APPEND" (concat css') options'
     )
   case files of
      []    -> putStrLn "DPL: no files."
      paths -> mapM_ (run options'') paths
```

5.24.2 Interpreter launch

`run` *options path* processes the file at *path* in the context of the user's *options*.

```
run :: Options -> FilePath -> IO ()
run options path = do
   let noisy = lookupFlag "v" options True
       quick = lookupFlag "q" options False
       proving = lookupFlag "p" options True
       makeCexs = lookupFlag "c" options False
       makeCth = lookupFlag "C" options False
       makeHglu = lookupFlag "h" options False
       simplifyC = lookupFlag "s" options True
       showTable = lookupFlag "t" options True
       showInputs = lookupFlag "i" options noisy
       lonely = lookupFlag "@LONELY" options False
   unless lonely $ putStrLn $
      "\n=========== BEGIN: " ++ path ++ " ===========\n"
   source' <- readFile path
   let source = source' ++ lookupParam "@APPEND" options ""
   when noisy (do
      putStrLn "--- input source ---\n"
      putStrLn source
     )
   case (checkParse ((dropWhite . nofail . total) lexerL)
      (total descriptionP) &? labelCheck) source of
      CheckPass d -> do
         when noisy (do
            putStrLn "--- description ---\n"
            print d
           )
         case (groundCheck &? flatten (dtt d) &?
               obviateCheck) d of
            CheckPass d -> do
               when noisy (do
                  putStrLn "--- grounded description ---\n"
                  print d
                  putStrLn "--- base theory ---\n"
                 )
               case assertCheck d of
                  CheckPass d -> do
                     let t = makeTheory d
                     when (noisy && proving) $ print t
                     when showInputs (do
                        let combs = generateRuns d
                        putStrLn $ "--- "
                           ++ show (length combs)
                           ++ " input combinations ---\n"
                        putStrLn $ unlines $ map show combs
                       )
                     when (noisy && quick) (do
                          putStrLn "--- optimised base \
                              \theory ---\n"
                          print $ toOTheory d t
                       )
                     when proving (do
                        when noisy $ putStrLn "--- proofs\
                           \ ---"
                        runs <- if noisy && quick then
                              doProofsQuickly d
                           else if quick then
                              doProofsQuicklySilently d
                           else if noisy then
                              doProofs d
                           else
                              return $ doProofsSilently d
                        when showTable (do
                           when (noisy || makeCexs) $
                              putStrLn "\n--- summary \
                                 \---\n"
                           putStr $ proofsTable d runs
                          )
                        when makeCexs (do
                           when (noisy || showTable) $
                              putStrLn "\n--- C \
                                 \expressions ---\n"
                           makeCExprs d runs simplifyC
                          )
                       )
                     when makeCth $ makeCTheory path d t
                     when makeHglu $ makeHGlue path d
                  CheckFail msg ->
                     putStrLn $ "--- error ---\n\n" ++ msg
            CheckFail msg ->
               putStrLn $ "--- error ---\n\n" ++ msg
      CheckFail msg ->
         putStrLn $ "--- error ---\n\n" ++ msg
   unless lonely $ putStrLn $
      "\n=========== END: " ++ path ++ " ===========\n"
```

`runc` *path* processes the file at *path* in the context of the options that select for only the summary table and the C expression.

```
runc :: FilePath -> IO ()
runc path = do
   let options :: Options
       options = assertFlagMinus "p" $
                 assertFlagPlus "c" $
                 assertFlagPlus "t" $
                 emptyOptions
   run options path
```

References

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

[2] David Billington and Andrew Rock. Propositional plausible logic: Introduction and implementation. *Studia Logica*, 67:243–269, 2001. 1

[3] D. Nute. Defeasible logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 353–395. Oxford University Press, 1994. 1, 2.3

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

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

[6] Michael J. Maher, Andrew Rock, Grigoris Antoniou, David Billington, and Tristan Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001. 1

[7] Andrew Rock. *Deimos*: A query answering Defeasible logic system. Technical report, (continually) in preparation. 1

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

[9] Andrew Rock. Implementation of Basic Plausible Logic. Technical report, (continually) in preparation. 1

[10] Andrew Rock. Implementation of Constructive Plausible Logic. Technical report, (continually) in preparation. 1

[11] Andrew Rock. ABR Haskell libraries. Technical report, (continually) in preparation. 5

Index