The DPL (Decisive Plausible Logic) Tool

Andrew Rock

November 25, 2013

Abstract

This documents the DPL tool, which implements the Decisive Plausible Logic of David Billington.

1 Plausible Logic in short

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 – *m*onotonic, strict, like classical logic;
- α – the conjunction of π *a*nd β;
- π – *p*lausible, *p*robable, and *p*ropogates ambiguity;
- β – plausible, *b*locking ambiguity (as per the original Defeasible Logic of Nute; and
- δ – the *d*isjunction of π and β.

2 Using DPL

2.1 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.

2.2 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:

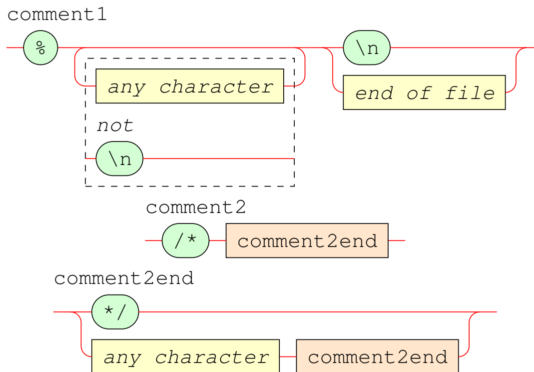
– types (sets of constants that may be bound to variable arguments to atoms);
– the types of the arguments of specific atoms;
– default facts that only operate in the absence of an explicit alternative;
– inputs (additional axioms to asserted, alternately positively and negatively, when performing proofs);
– 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

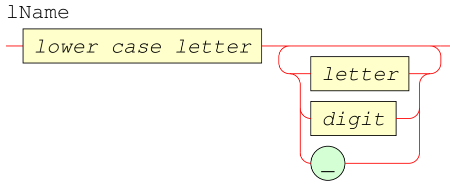
Any amount of whitespace is permitted before and after any symbol. Comments are treated as whitespace. There are two types:

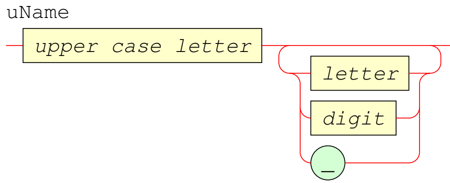
- Comments that begin with a `%` extend to the end of the line.
- Comments that begin with `/*` extend to the next `*/` and may extend across many lines.



Names

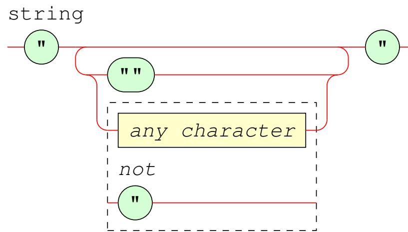
Atoms and rule labels in a description are names that start with letters and may include digits and underscores. Names that start with upper and lower case letters are distinguished for different purposes.





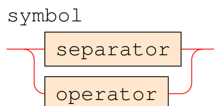
Strings

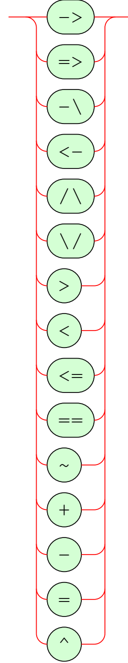
Atoms may also be represented as double-quote-delimited strings. To embed a double-quote in a string without prematurely terminating the string, repeat it.

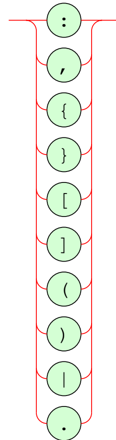


Symbols

These are the special sybmols on the language.

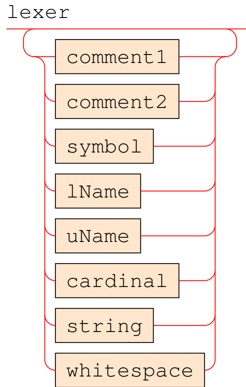






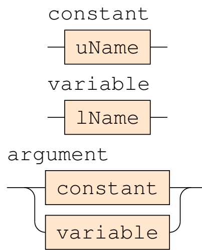
The complete lexical syntax

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



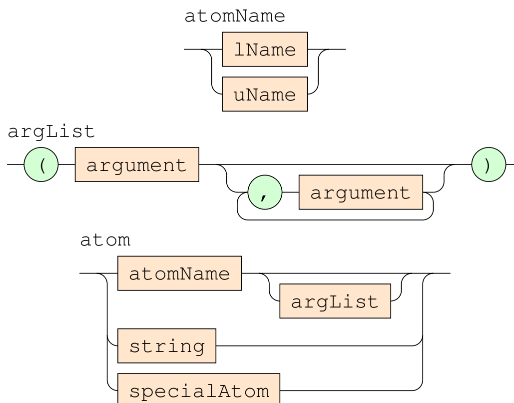
Constants and variables as arguments

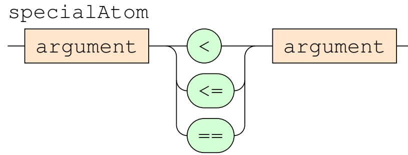
Constants and variables may optionally appear as arguments to atoms. Constants always start with upper case letters and variables always start with lower case letters.



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 **==**.





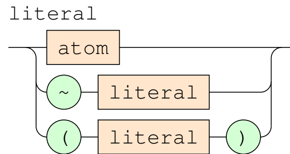
Example atoms:

```
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 (¬).



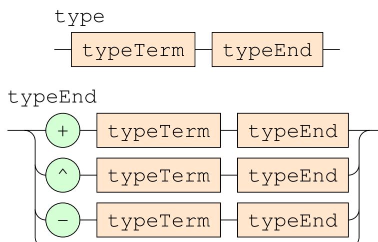
Example literals:

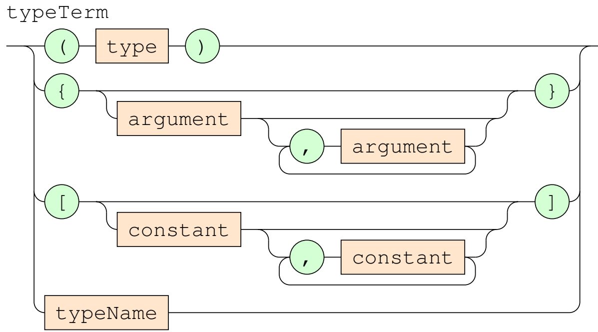
```
p   ~p   ~(~p)   p(A,B,c)   ~p(A,B,c)   ~"x > 3"
```

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.

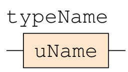




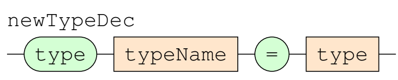
The use of delimiters [and], rather than { and }, implies that the constants in that sequence form a total order. This causes the automatic generation of a lot of axioms of the forms x < y, x <= y and x == y.

Declaring named types

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



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.



Examples:

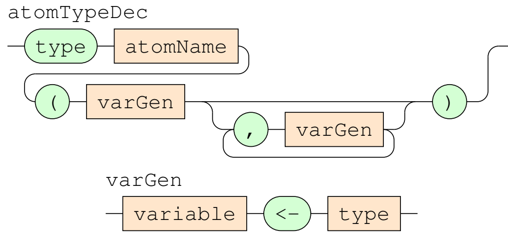
```
type Animal = {Cat, Dog, Human}.
type Vegetable = {Grass, Carrot}.
type Organism = Animal + Vegetable + {Mushroom}.
type Beast = Animal - {Human}.
```

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.



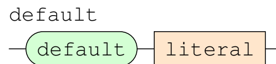
Examples:

```
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.



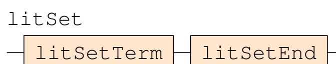
The following default facts are automatically asserted:

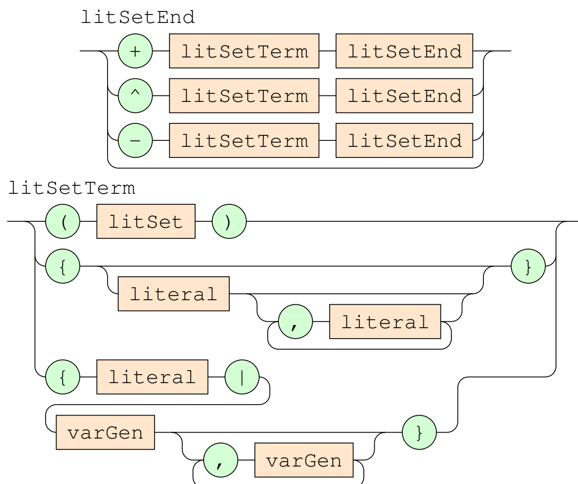
```
default ~(x < y).
default ~(x <= y).
default ~(x == y).
```

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.





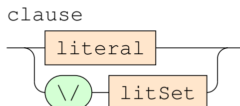
Example literal sets:

```
{}
{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}}
```

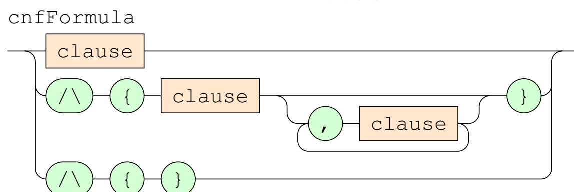
Note that `x` is a free variable that will be instantiated to all of the constants in the `Universe` unless constrained by an atom type assertion.

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.



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

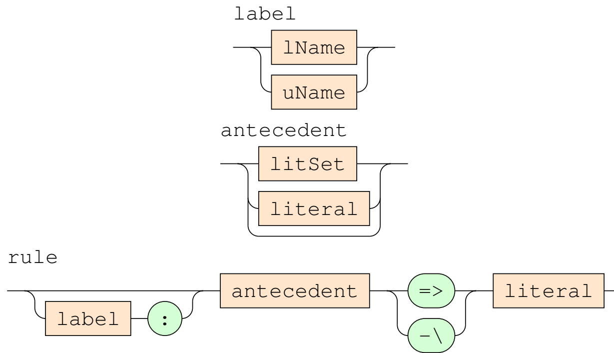


Example clauses and cnf-formulas:

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

Rules

Strict rules (`->` for $\rightarrow$) do not appear in plausible descriptions. Only plausible rules (`=>` for $\Rightarrow$) and defeater rules (`-\` for $\rightharpoonup$) 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.

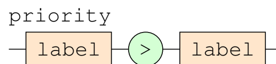


Example rules:

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.



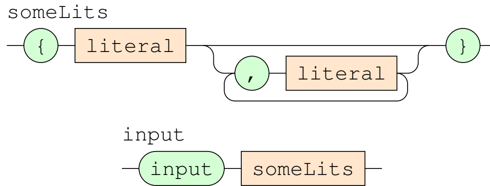
Example:

`R1 > R2`

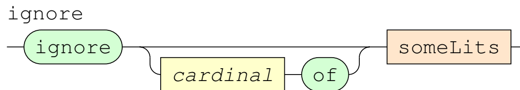
Input and ignore specifications

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.



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



Examples:

inputs and ignores	*axioms generated*

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

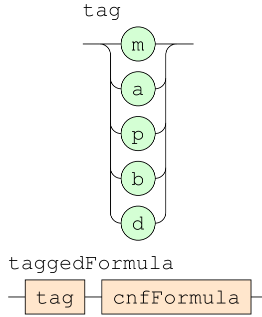
`input{a, b}.`	`a.  ~b.` `~a.  b.`
`input{a, ~b}.`	`a.  b.` `~a.  ~b.`

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

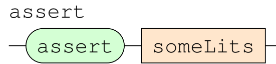
`input{a}.   input{b}.`
`input{c}.   input{d}.`
`ignore 3 of {a, b, c, d}.`
	`a.  b.  ~c.  ~d.`
	`a.  ~b.  c.  ~d.`
	`a.  ~b.  ~c.  d.`
	`a.  ~b.  ~c.  ~d.`
	`~a.  b.  c.  ~d.`
	`~a.  b.  ~c.  d.`
	`~a.  b.  ~c.  ~d.`
	`~a.  ~b.  c.  d.`
	`~a.  ~b.  c.  ~d.`
	`~a.  ~b.  ~c.  d.`
	`~a.  ~b.  ~c.  ~d.`

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.



taggedFormula



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

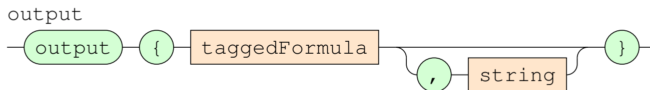
assert someLits

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



Examples:

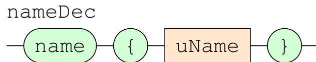
```
output{p p}           output{a /\{\/{a,b},~c}}
output{p p, "IS_P"}   output{a /\{\/{a,b},~c}, "A_OR_B(X)"}
```

Name declaration

Code generation hints

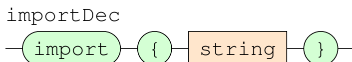
The DPL tool is sometimes used to generate code in other languages. These declarations are hints to help those processes:

- 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.



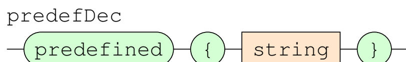
Example: `name{Nanook}`

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

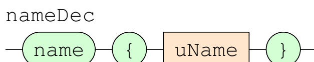


Example: `import{"package.Class"}`

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

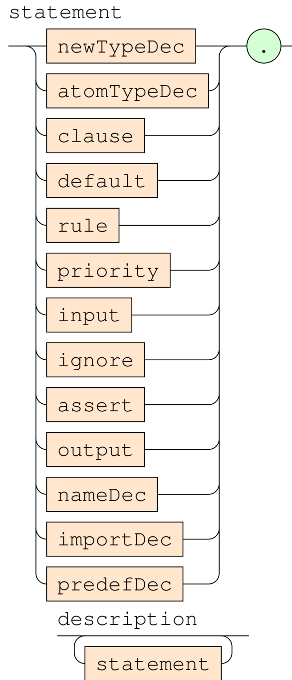


Example `predefined{"elem"}`



Description file

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



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

2.3 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).