L specification (draft)

Evgenii Balai

March 12, 2016

Contents

1 Syntax **2**
- 1.1 Symbols . . . 2
- 1.2 Basic Terms . . . 2
- 1.3 Constant Declarations . . . 3
- 1.4 Set Expressions . . . 3
- 1.5 Type Declarations . . . 3
- 1.6 Quantified Terms . . . 4
- 1.7 Terms . . . 4
- 1.8 Atoms . . . 4
- 1.9 Literals . . . 4
- 1.10 Sentences . . . 4
- 1.11 Maybe Constructs . . . 4
- 1.12 Cardinality Constraints . . . 5
- 1.13 Rules . . . 5
- 1.14 Program . . . 5
- 1.15 Language Grammar . . . 6
- 1.16 Comments . . . 8

2 Semantics **8**
- 2.1 Constant Declarations . . . 8
- 2.2 Arithmetic Terms . . . 9
- 2.3 Type Declarations . . . 9
- 2.4 Programs with Variables and Ground Programs . . . 10
- 2.5 Program Models . . . 11
- 2.6 Program models . . . 13

3 Examples **13**
- 3.1 Simple Examples . . . 13
- 3.2 Safety Obligations . . . 14
- 3.3 K-vertex Connectivity of Graphs . . . 15

A L program for checking safety obligations **18**

B L program for checking K-connectivity of a graph **19**

1 Syntax

1.1 Symbols

L symbols are non-empty strings of ascii characters divided into the following disjoint categories:

- integer numerals
- identifiers
- special characters

An ***integer numeral*** is a string of one or more decimal digits (0-9).

An ***identifier*** is a string of one letter followed by zero or more letters, digits, and underscores.

A ***special character*** is one of the following characters:

`> = < + - * / % { } ( ) , . | \`

A ***symbol*** is an integer numeral, an identifier or a special character.

1.2 Basic Terms

Basic terms are divided into the following categories:

- constants
- variables
- typed variables
- arithmetic terms
- functional terms

A ***constant*** is either an integer numeral or an identifier starting with a lowercase letter.

A ***variable*** is an identifier starting with an uppercase letter.

A ***typed variable*** is a string of the form *id var*, where *id* is an identifier also referred to as *type name* and *var* is a variable.

An ***arithmetic term*** is a string of the form $t \diamond u$, where: each of t and u is either an integer numeral, a variable, or an arithmetic term; and $\diamond$ is a special symbol in the set {`'+', '-', '*', '/', '%'`} (with `'%'` standing for modulo operator). Parentheses can optionally be omitted in which case standard operator precedences apply.

A ***functional term*** is a string of the form $f(t_1, \ldots t_n)$ where $t_1, \ldots, t_n$ are basic terms, f is an identifier also referred to as a ***functional symbol***, and $n > 0$.

Let t and t' be basic terms. We will say that t' is a ***subterm*** of t iff at least one of the following conditions holds:

- $t = t'$
- t is of the form $t' \diamond t_1$, where t_1 is some basic term

- t is of the form $t_1 \diamond t'$, where t_1 is some basic term
- t is of the form $f(t_1, \ldots t_n)$ and $t' \in \{t_1, \ldots, t_n\}$
- there exists a basic term t_1 such that t' is a subterm of t_1 and t_1 is a subterm of t

We say that t' is a *proper subterm* of t if t' is a subterm of t and $t' \neq t$.

A term t is called ***ground*** iff at least one of the following holds:

- t is an identifier or an integer numeral; or
- all proper subterms of t are ground

1.3 Constant Declarations

A ***constant declaration*** is of the form

$$\texttt{const}\ c = v. \tag{1}$$

where c is an identifier also referred to as ***constant name*** and v is a ground arithmetic term, an integer numeral, or an identifier. We will say that the constant c is *defined* by a program if the program contains a declaration of the form (1).

1.4 Set Expressions

A ***set expression*** is of one of the following forms:

- $\{t_1, t_2, \ldots, t_n\}$

  where $t_1, \ldots, t_n$ are ground terms and $n > 0$.

  A shorthand $\{l..r\}$ (where each of l and r is either a constant name, an integer numeral, or a ground arithmetic term) may be used to represent the set of all integers in the closed interval $[l, r]$. The numeric value represented by l must be strictly less than that by r.

- An identifier.

- t `where` V_1 *in* $type_1, \ldots, V_n$ *in* $type_n$

  where t is a term, $\{V_1, \ldots, V_n\}$ is the set of all variables occurring in t and $type_1, \ldots type_n$ are identifiers.

- $(S_1 \diamond S_2)$, where S_1 and S_2 are set expressions and $\diamond$ is a set-theoretic operator, denoted by one of the special characters in $\{+, *, \backslash\}$. Parentheses can be omitted in which case $*$ and $\backslash$ have higher precedence than $+$ and all operations are left-associative.

1.5 Type Declarations

A ***type declaration*** is of the form

$$\texttt{type}\ t = set\_expr. \tag{2}$$

where t is an identifier and *set_expr* is a set expression defined in section 1.4. We will say that the type t is *defined* by a program if it contains a declaration of the form (2).

1.6 Quantified Terms

A ***quantified term*** is of the form:

$$quantifier \; p$$

Where $quantifier$ is an identifier in the set {`every`, `some`}, p is an identifier also referred to as the *type* of the quantified term.

We will refer to a quantified term starting with the quantifier `some`(`every`) as an ***existentially quantified*** (***universally quantified***) term.

1.7 Terms

A ***term*** is either a basic term or a quantified term.

1.8 Atoms

Atoms are divided into two categories:

- predicate atoms
- built-in atoms

A ***predicate atom*** is a string of the form $p(t_1, \ldots t_n)$ where p is an identifier also referred to as a ***predicate name*** and $t_1, \ldots, t_n$ are terms with $n \geq 0$. A predicate atom is called ***basic*** if $t_1, \ldots, t_n$ are basic terms.

A ***built-in atom*** is a string of the form $t_1 \preceq t_2$ where t_1 and t_2 are basic terms and $\preceq \in$ {`'<','>', '>=','<=','=','!='`} is a string of special characters.

An atom is called ***ground*** if all the terms occurring in the atom are ground.

1.9 Literals

A ***literal*** is of one of the forms:

- a, where a is an atom
- `not` b, where b is a basic atom

1.10 Sentences

A ***sentence*** is either a literal or an expression of the one of the forms (A *or* B), (A *and* B), where A and B are sentences. A sentence of the form A *and* B can also be written as A, B. Parentheses can be omitted in which case *and* has higher precedence than *or*. A sentence is called a ***predicate sentence***, if all the atoms occurring in the sentence are predicate atoms.

1.11 Maybe Constructs

A ***maybe construct*** is of the form

$$\texttt{maybe}\; p(t_1, \ldots t_n)$$

where $p(t_1, \ldots, t_n)$ is a basic predicate atom.

1.12 Cardinality Constraints

A ***cardinality constraint*** is of the form

$$v_1 <= |\{p(t_1, \dots t_n)\}| <= v_2.$$

where

1. each of v_1 and v_2 is a ground arithmetic term, an integer numeral, or a constant name,
2. $p(t_1, \dots, t_n)$ is a basic predicate atom.

1.13 Rules

A ***rule*** can be of two forms:

$$head. \tag{3}$$

or

$$head \;\texttt{if}\; sentence. \tag{4}$$

where *head* is one of the following:

1. a predicate sentence;
2. a maybe construct;
3. a cardinality constraint.

and *sentence* is a sentence defined in section (1.10). We will also refer to *head* and *sentence* as the ***head*** and the ***body*** of the corresponding rule respectively.

1.14 Program

A ***program*** is a collection of statements, each of which is either a constant declarations, a type declaration, or a rule.

Constant declarations of a program must satisfy the following conditions:

1. Each constant name must occur in the left-hand side of exactly one constant declaration.
2. Each identifier which is a subterm of the right-hand side of a constant declaration must occur in the left-hand side of a preceding constant declaration.

Type declarations of a program must satisfy the following conditions:

1. Each type name must occur in the left-hand side of exactly one type declaration.
2. Each identifier occurring as a subexpression of the right-hand side of a type declaration must be a type name defined by a preceding type declaration.

Rules of a program must satisfy the following conditions:

1. The leftmost occurrence of any variable V in a rule must be in the head of the rule and must be preceeded by a type name (also referred to as *the type* of the variable in this rule).
2. For every typed variable $t\ Var$, the identifier t is a type defined by the program.
3. Every identifier occurring in the rule as a subterm of an arithmetic term is a constant name defined by the program.

1.15 Language Grammar

1.15.1 Terms

$term ::= basic\_term \mid quantified\_term$
$basic\_term ::= numeric\_constant \mid variable \mid identifier \mid identifier\ variable$
$\qquad \mid arithmetic\_term \mid functional\_term$
$ground\_term ::= numeric\_constant \mid identifier \mid$
$\qquad \mid ground\_arithmetic\_term \mid ground\_functional\_term$
$arithmetic\_term ::= \texttt{-(}T0\texttt{)} \mid \texttt{-}T0 \mid \texttt{(}T0\ infix_1\ T1\texttt{)} \mid \texttt{(}T1\ infix_2\ T2\texttt{)} \mid$
$\qquad \mid T0\ infix_1\ T1 \mid T1\ infix_2\ T2$
$ground\_arithmetic\_term ::= \texttt{-(}T0\_g\texttt{)} \mid \texttt{-}T0\_g \mid \texttt{(}T0\_g\ infix_1\ T1\_g\texttt{)} \mid \texttt{(}T1\_g\ infix_2\ T2\_g\texttt{)} \mid$
$\qquad \mid T0\_g\ infix_1\ T1\_g \mid T1\_g\ infix_2\ T2\_g$

$infix_1 ::= \texttt{+} \mid \texttt{-}$
$infix_2 ::= \texttt{*} \mid \texttt{/} \mid \texttt{\%}$
$infix ::= infix_1 \mid infix_2$
$T0 ::= T1 \mid T0\ infix_1\ T1$
$T1 ::= T2 \mid T1\ infix_2\ T2$
$T2 ::= (T0) \mid variable \mid numeric\_constant \mid identifier \mid identifier\ variable$

$T0\_g ::= T1\_g \mid T0\_g\ infix_1\ T1\_g$
$T1\_g ::= T2\_g \mid T1\_g\ infix_2\ T2\_g$
$T2\_g ::= (T0\_g) \mid numeric\_constant \mid identifier$

$functional\_term ::= identifier\ \texttt{(}\ terms\ \texttt{)}$
$ground\_functional\_term ::= identifier\ \texttt{(}\ ground\_terms\ \texttt{)}$

$quantified\_term ::= quantifier\ identifier$
$quantifier ::= \texttt{every} \mid \texttt{some}$
$basic\_terms ::= basic\_term \mid basic\_term\texttt{,}\ basic\_terms$
$ground\_terms ::= ground\_term \mid ground\_term\texttt{,}\ ground\_terms$
$terms ::= term \mid term\texttt{,}\ terms$

1.15.2 Constant Declarations

$const\_decl ::= \texttt{const}\ identifier\texttt{=}ground\_arithmetic\_term. \mid identifier\texttt{=}identifier.$
$\qquad \mid identifier\texttt{=}numeric\_constant.$

1.15.3 Type Declarations

$type\_decl$ ::= `type` $identifier$ `=` set_expr.
$limit$ ::= $identifier$ | $numeric\_constant$ | $ground\_arithmetic\_term$
set ::= $\{[ground\_terms]\}$ [1]
$range$ ::= $\{limit..limit\}$
set_expr ::= $ST0$
set_constr ::= $basic\_term$ `where` $tvars$
$tvars$ ::= $tvar$ | $tvar$`,`$tvars$
$tvar$ ::= $variable$ `in` $identifier$
$ST0$ ::= $ST1$ | $ST0$ `+` $ST1$
$ST1$ ::= $ST2$ | $ST1$ `*` $ST2$ | $ST1 \setminus ST2$
$ST2$ ::= `(`$ST0$`)` | set | $range$ | set_constr | $identifier$

1.15.4 Atoms

$atom$::=$predicate\_atom$ | $built\_in$
$predicate\_atom$ ::= $identifier$[`(`$terms$`)`]
$basic\_predicate\_atom$ ::= $identifier$[`(`$basic\_terms$`)`]
$built\_in$ ::= $basic\_term$ op $basic\_term$
op ::= `>` | `<` | `>=` | `<=` | `=` | `!=`

1.15.5 Literals

$predicate\_literal$ ::= $predicate\_atom$ | `not` $predicate\_atom$
$literal$ ::= $atom$ | `not` $predicate\_atom$

1.15.6 Sentences

$sentence$ ::= $s3$
$s2$ ::= $s2$ | $s3$ `or` $s2$
$s1$ ::= $s1$ | $s2$ `and` $s1$ | $s2$ `,` $s1$
$s0$ ::= $literal$ | `(`$s3$`)`
$predicate\_sentence$ ::= $predicate\_s3$
$predicate\_s2$ ::= $predicate\_s2$ | $predicate\_s3$ `or` $predicate\_s2$
$predicate\_s1$ ::= $predicate\_s1$ | $predicate\_s2$ `and` $predicate\_s1$ | $predicate\_s2$ `,` $pedicate\_s1$
$predicate\_s0$ ::= $predicate\_literal$ | `(`$predicate\_s3$`)`

1.15.7 Maybe Statements

$maybe\_st$ ::= `maybe` $basic\_predicate\_atom$

1.15.8 Cardinality Constraints

$bound$ ::= $arithmetic\_term$ | $numeric\_constant$ | $identifier$
$card\_constr$ ::= $bound$ `<=` `|`$\{basic\_predicate\_atom\}$`|` `<=` $bound$

[1] Square brackets around $basic\_terms$ mean that $basic\_terms$ are optional, that is, {} is a valid expression for non-terminal set

1.15.9 Rules

rule ::= *head*`.`|*head* `if` *sentence*`.`
head ::= *predicate_sentence* | *maybe_st* | *card_constr*

1.15.10 Program

program ::= *statements*
statements ::= *statement* | *statement*, *statements*
statement ::= *const_decl* | *type_decl* | *rule*

1.16 Comments

L programs can have comments starting with `/*` and ending with `*/` . For example:

```
/*
 This is a
 multiline comment
*/
type t = {1,2,3}. /* this is a type declaration */
p(t X) if /* this is a rule */ q(X).
```

2 Semantics

We define the semantics of an L program Π in terms of *models* of P. A model is, intuitively, a minimal set of atoms which, satisfies the rules of the program. The notion of satisfiability is defined in section 2.5 with all necessary background provided in sections 2.1 - 2.4.

The semantics of a program P containing comments coincide with the semantics of the program obtained from P by removing the comments. In the rest of this section we will consider programs not containing comments.

2.1 Constant Declarations

Let Π be an L program starting with a constant declaration of the form

$$\texttt{const}\ cn = gar\_term.$$

Where cn is referred to as a constant name and *gar_term* is a ground arithmetic term. By condition 2 for constant declarations defined in section 1.14, *gar_term* cannot contain identifiers as subterms, therefore its value v can be obtained as defined in section 2.2.

The models Π coincide with the models of program Π' obtained from Π by:

1. Removing the declaration `const` $cn = gar\_term$.
2. Replacing every subterm of a term in Π equal to cn with the numeric constant v.

By condition 2 for constant declarations defined in section 1.14, if the program Π' starts another constant declaration, its right hand side cannot contain numeric constants, therefore its semantics can be defined in the same manner as for Π.

Therefore, it is sufficient to define the semantics for programs not containing constant declarations (and, therefore, not containing constant names in arithmetic terms either).

2.2 Arithmetic Terms

A program may contain ground arithmetic terms constructed from integer numerals and operations '+' (plus), '-' (minus), '*' (multiplication), '/' (integer division), '%' (modulo)[2] Each arithmetic term has *a value*. The meaning and precedence of operations '+', '-', '*' is as usual. The operation '/' has the same precedence as '*' and is defined as

$$a/b := sgn(a*b)*(|a|\ div\ |b|)$$

where

1. $sgn(a*b) = \begin{cases} 1 & \text{if } a*b \geq 0 \\ -1 & \text{otherwise .} \end{cases}$
2. for $a \geq 0$ and $b \geq 0$ $a\ div\ b$ is a floor division, i.e, $a\ div\ b$ is the largest integer such that $b*(a\ div\ b) \leq a$.

The operation '/' has the same precedence as '*' and is defined as

$$a\%b := a - n*(a/n)$$

All operations are associated from left to right.

The semantics of programs containing undefined arithmetic operations (division by zero or modulo with its second operand equal to zero) is undefined.

2.3 Type Declarations

Type declarations of a program Π define a mapping $\mathscr{D}_\Pi$ from identifiers (also called type names) and set expressions of Π to sets of ground terms. If q is a type name or a sort expression, $\mathscr{D}_\Pi(q)$ denotes the set of ground terms it is mapped to.

The mapping is defined as follows:

1. For every sort expression of the form

$$\{t_1, \ldots, t_n\}$$

$\mathscr{D}_\Pi(\{t_1, \ldots, t_n\})$ is $\{t_1, \ldots, t_n\}$

[2] Current implementation allows only numerals in the range from 0 to 2,147,483,647. Moreover, correct evaluation for an arithmetic term is guaranteed only if all its subterms have values within the range.

2. For every sort expression of the form

$$t \texttt{ where } V_1 \ in \ type_1, \ldots, V_n \ in \ type_n$$

$\mathscr{D}_\Pi(t \texttt{ where } V_1 \ in \ type_1, \ldots, V_n \ in \ type_n)$ is $\{t | V_1 \in \mathscr{D}_\Pi(type_1), \ldots, V_n \in \mathscr{D}_\Pi(type_n)\}$

3. For every set expression of the form

$$S_1 \diamond S_2$$

$\mathscr{D}_\Pi(S_1 \diamond S_2)$ is $\mathscr{D}_\Pi(S_1) \odot \mathscr{D}_\Pi(S_2)$, where $\odot$ is a set operation: union, intersection, or difference when $\diamond$ is $+$, $*$ or $\setminus$ correspondingly.

4. For every type declaration of the form

$$\texttt{type } tn = set\_expr$$

$\mathscr{D}_\Pi(tn)$ is $\mathscr{D}_\Pi(set\_expr)$

In the remainder of the section we will mostly use $\mathscr{D}_\Pi$ to obtain the values which correspond to the type names of Π.

2.4 Programs with Variables and Ground Programs

For a program Π containing variables we obtain a corresponding *ground* program Π^g as follows:

1. each rule r containing variables is replaced with a maximal collection of rules, each of which corresponds to a unique substitution of variables (together with possibly preceeding type names) with ground terms. A variable v can be replaced with a term f if
   - there in an occurrence of a typed variable $t \ v$ in r, and
   - $f \in \mathscr{D}(t)$
2. each arithmetic term is evaluated as described in section 2.2.

For example, consider the program

```
type type1 = {1,2,5}.
type type2 = {1,2}.

p(type1 X, type2 Y) if X+Y = 7.
maybe q(type1 X).
1<=|{t(type1 X, type2 Y)}| <= 2 if q(Y).
```

The corresponding ground program is:

```
p(1, 1) if 2 = 7
p(1, 2) if 3 = 7
p(2, 1) if 3 = 7
p(2, 2) if 4 = 7
p(5, 1) if 6 = 7
p(5, 2) if 7 = 7
maybe q(1).
maybe q(2).
maybe q(5).
1<=|{t(type1 X, 1)}| <= 2 if q(1).
1<=|{t(type1 X, 2)}| <= 2 if q(2).
```

Therefore, any program Π can be viewed as a shorthand for the corresponding ground program Π^g. In the following sections we will define the semantics of ground programs.

2.5 Program Models

A model of Π is a set of ground atoms satisfying certain conditions.

To describe the conditions, we first introduce some definitions. We will call a predicate atom $p(t_1, \ldots, t_n)$ *basic* if all the terms $t_1, \ldots t_n$ are basic.

Similarly, a sentence is called basic if all the atoms occurring in the sentence are basic.

Definition 1. *(A set ground atoms satisfying a basic predicate atom)*
A set of ground atoms A satisfies a simple predicate atom $p(t_1, \ldots t_n)$ if and only if $p(t_1, \ldots t_n) \in A$.

□

Definition 2. *(A set of ground atoms satisfying a built-in atom)*
A set of ground atoms satisfies a ground built-in atom $t_1 \diamond t_2$ iff one of the following conditions is satisfied:

1. $\diamond$ is $=(\neq)$ and $t_1 = t_2 (t_1 \neq t_2)$;

2. $\diamond$ is $< (\leq, \geq, >)$, t_1 and t_2 are integer numerals representing numbers N_1 and N_2 respectively and $N_1 < N_2$ $(N_1 \leq N_2, N_1 \geq N_2, N_1 > N_2)$;

3. $\diamond$ is $< (\leq, \geq, >)$, t_1 and t_2 are identifiers starting with a lowercase letter, and t_1 lexicographically smaller than (smaller than or equal to, greater than or equal to, greater) t_2.

□

The semantics of a program containing a built-in atom $t_1 \diamond t_2$ where $\diamond \in \{<, \leq, \geq, >\}$ is *defined* if and only of both t_1, t_2 are both constants or integer numerals.

Definition 3. *(A set of ground atoms satisfying a literal)*
A set of ground atoms A satisfies a literal l if one of the following conditions is satisfied:

1. l is of the form a, where a is an atom satisfied by A

2. l is of the form `not` a, where a is an atom not satisfied by A

Definition 4. *(A set of ground atoms satisfying a basic sentence)*
A set of of ground atoms A satisfies a basic sentence S ($A \vdash S$) if one of the following conditions is satisfied:

1. S is a literal satisfied by A
2. S is of the form S_1 *or* S_2 (S_1 *and* S_2) and A satisfies one (both) of the sentences S_1 and S_2

□

Let S be a sentence. And let

- $\{U_1, \ldots U_n\}$ be the set of all universally quantified terms of S whose types are $Tu_1, \ldots, Tu_n$ respectively;
- $\{E_1, \ldots E_m\}$ be the set of all existentially quantified terms of S whose types are $Te_1, \ldots, Te_m$ respectively;

For a set $\{T_1, \ldots, T_k\}$ of quantified terms of S, we denote the sentence obtained from S by replacing every occurrence of a quantified term T_i with t_i by $S|_{\{T_k=t_1,\ldots,T_k=t_k\}}$.

Definition 5. *(A set of ground atoms satisfying a sentence)*
A set of ground terms A satisfies S iff

$$\exists(e_1, \ldots, e_n) \in \mathscr{D}(Te_1)\times\ldots\times\mathscr{D}(Te_n) : \forall(u_1, \ldots, u_m) \in \mathscr{D}(Tu_1)\times\ldots\times\mathscr{D}(Tu_m) :$$
$$(A \vdash S|_{\{E_1=e_1,\ldots,E_m=e_m,U_1=u_1,\ldots U_n=u_n\}})$$

□

Definition 6. *(A set of ground atoms satisfying a cardinality constraint)*
Let A be a set of ground atoms and

$$v_1 <= |\{p(t_1, \ldots t_n)\}| <= v_2.$$

be a cardinality constraint of a program Π. Let $X_1 \ldots, X_n$ be all variables occurring in the constraint and $Tx_1, \ldots Tx_n$ be the types of the variables $X_1 \ldots, X_n$ correspondingly. Let S be a set of ground atoms of Π of the form $p(t'_1, \ldots t'_n)$ each of which is obtained from $p(t_1, \ldots t_n)$ by replacing all occurrences of variables with elements of their corresponding types.
A satisfies the constraint $v_1 <= |\{p(t_1, \ldots t_n)\}| <= v_2$ iff $v_1 \leq |S| \leq v_2$.

□

As described in section 1.13, program rules may be in one of the two forms. We sometimes say that a rule of the form

$$head.$$

has a body which is satisfied by any set of ground atoms and use the canonical form

$$head \texttt{ if } body.$$

even if the body is not present.

Definition 7. *(A set of ground atoms satisfying a rule)*
A set of ground atoms A satisfies a rule, whose head is either a cardinality constraint or a sentence, if one of the following conditions holds:

1. A does not satisfy *body*;

2. A satisfies both *body* and *head*.

□

2.6 Program models

Let Π be an L program. For every atom a occurring, let a' be a fresh atom not occurring in Π, such that for any two distict atoms $a_1 \neq a_2$ of Π, $a'_1 \neq a'_2$. By Π' we denote a program obtained form Π by:

1. replacing all maybe contructs of the form `maybe` l with l *or* (`not` l).

2. replacing all literals of the form `not` a with a'.

Let A' denote the set of all new atoms introduced in P'.

Definition 8. *(A model of a program)*
Let A be a set of ground atoms of a program Π; A is a model of Π if and only if there exists a set of atoms B of Π' such that the following conditions are satisfied:

1. $A = B \setminus A'$.

2. B is the mimimal set satisfying the rules of Π' whose heads are predicate sentences.

3. B satisfies all the rules of Π' whose heads are cardinality constraints.

□

3 Examples

3.1 Simple Examples

The program Π_1:

```
a.
b if a.
```

has exactly one model $\{a, b\}$.

The program Π_2:

```
a if b.
```

has exactly one model $\{\}$, because it does not contain maybe literals or cardinality constraints, and $\{\}$ is the minimal set of atoms satisfying the only rule of the program.

The program Π_3:

```
type t1 = {5,6,7}.
type t2 = {0,1,2}.
p(t2 N) if q(N+5).
maybe q(t1 N).
1<=|{q(t1 N)}<=2.
```

has six models:

```
{p(1), p(2), q(6), q(7)},
{p(2), q(7)},
{p(1), q(6)},
{p(0), p(1), q(5), q(6)},
{p(0), p(2), q(5), q(7)},
{p(0), q(5)}
```

3.2 Safety Obligations

The safety obligations are met if

1. The system requirements have been certified;
2. The process for insuring validation has been followed, and
3. The system has passed all required inspections.

The 3 conditions for meeting safety obligations can be defined by the following L rule:

```
safetyObligationsMet if
   requirementsCertified and
   validationProcessFollowed and
   passed(every requiredInspection).
```

The system requirements are certified if they are sound and complete. This is expressed by the following L rule:

```
requirementsCertified if
   requirementsSound and
   requirementsComplete.
```

The validation process has been followed if sections $A - E$ of code $825/A/6$ have been satisfied. The code sections are represented by identifiers of the form 825_A_6_X, where X is a character in the range A-Z. The corresponding L rule is:

```
validationProcessFollowed if
   satisfied(code_825_A_6_A) and
   satisfied(code_825_A_6_B) and
   satisfied(code_825_A_6_C) and
   satisfied(code_825_A_6_D) and
   satisfied(code_825_A_6_E).
```

The set of required inspection is represented by a type consisting of two elements:

```
type requiredInspection = {epa_i_652_6B_714_A, epa_i_652_6B_714_B}.
```

An EPA safety hearing is passed if it is not pending:

```
passed(epaFine_j_652_6B_710_C H) if not pending(H).
```

The first inspection named EPA i/652/6B/714/A is passed if we have completed all required forms and every EPA safety hearing is passed:

```
passed(epa_i_652_6B_714_A) if
   completed(every requiredFromEPA714) and
   passed(every epaFine_j_652_6B_710_C).
```

The second inspection named EPA i/652/6B/714/B is passed if we have paid all fines required under previous infractions under EPA code j/652/6B/710/C:

```
passed(epa_i_652_6B_714_B) if
   paid(every epaFine_j_652_6B_710_C).
```

All the forms, hearings and infractions mentioned in the previous two definitions are defined by types:

```
type epaSafetyHearing = {es1,es2}.
type requiredFromEPA714 = {rfe1} .
type epaFine_j_652_6B_710_C = {efj1,efj2,efj3}.
```

A complete program with type declarations and rules put in the right order is given in appendix A

3.3 K-vertex Connectivity of Graphs

A graph is called *K-vertex-connected* (or simply *K-connected*) if it has more than K vertices and remains connected whenever fewer than K vertices are removed.

We consider the undirected graph shown in Figure 1.

The number of nodes in the graph is stored in a constant n:

```
const n = 5.
```

The number K is also a constant:

```
const k = 2.
```

The nodes of the graph are represented with a type *node*.

```
type node = {1..n}.
```

The edges of the graph are represented with the facts below. The atom $edge(i, j)$ for integers i and j is true if and only if there is an edge from node i to node j in the graph. The edges are defined as follows:

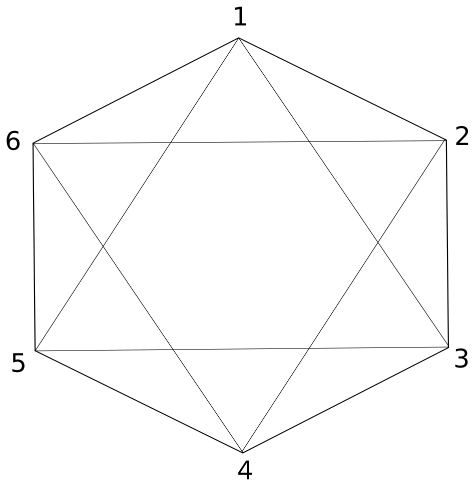

Figure 1: Complete undirected graph with 5 nodes

```
edge(node X, node Y) if X%n = (Y+1)%n.
edge(node X, node Y) if X%n = (Y+2)%n.
edge(node X, node Y) if edges(X,Y)
```

where the last rule is needed to represent the undirectedness of the graph. We will check $K - connectedness$ by trying to remove up to $K - 1$ nodes from the graph and checking whether the graphs remains connected. For a node N `removed(N)` is true if N is removed from the graph. Any node may be removed from the graph:

```
maybe removed(node N).
```

We are only interesting in the models where less than K nodes are removed:

```
0 <= |{removed(node N)}| <= k-1.
```

To defined the connectedness of a graph, we first define a `reachable(X,Y)` relation, which is true if and only if there exists a path from X to Y in the graph not containing removed nodes:

Any node which wasn't removed is reachable from itself:

```
reachable(node X, X) if not removed(X).
```

A node Y is reachable from node X if they are both not removed and there is an edge from X to Y:

```
reachable(node X,node Y) if edge(X,Y)
                            and not removed(X)
                            and not removed(Y).
```

To define reachability for nodes not connected by an edge, we need an auxiliary relation $reachable\_through(X, Z, Y)$ which says "a node Y is reachable from node X through node Z".

$reachable\_through(X, Z, Y)$ holds if Z is reachable from X, Y is reachable from Z and none of the nodes X,Y,Z was removed:

```
reachable_though(node X,node Z, node Y) if reachable(X,Z),
                            and reachable(Z,Y)
                            and not removed(X)
                            and not removed(Y)
                            and not removed(Z).
```

Finally, a node Y is reachable from node X if Y is reachable from X through some node:

```
reachable(node X, node Y)
                if reachable_through(node X, some node, node Y).
```

The graph is k-connected if any two nodes that were not removed are reachable from each other. We next define the disconnected relation: the graph is disconnected if there exists a pair of nodes which are not reachable from each other.

```
disconnected(node X, node Y) if not reachable(X, Y)
                   and not removed(X)
                   and not removed(Y).

disconnected_graph if disconnected(some node, some node).
```

If there exists at least one way to remove at most $k - 1$ such that the graph is disconnected, the graph is not k-connected. We can check this by, first, putting a constraint requiring the graph to be disconnected:

```
1<=|{disconnected_graph}|<=1.
```

The graph is not K-connected if and only there exists at least one model of the program.

The program has no models for $k <= 4$ but has a model for $k = 5$. That is, the graph on figure 3.3 is $4 - \textit{connected}$ but not $5 - \textit{connected}$ (for example, the nodes {2,3,4,5} can be removed from the graph to make it disconnected).

A complete program for this example is given in appendix B

A L program for checking safety obligations

```
type requiredInspection = {epa_i_652_6B_714_A, epa_i_652_6B_714_B}.
type epaSafetyHearing = {es1,es2}.
type requiredFromEPA714 = {rfe1} .
type epaFine_j_652_6B_710_C = {efj1,efj2,efj3}.


safetyObligationsMet if
   requirementsCertified and
   validationProcessFollowed and
   passed(every requiredInspection).

requirementsCertified if
   requirementsSound and
   requirementsComplete.

validationProcessFollowed if
   satisfied(code_825_A_6_A) and
   satisfied(code_825_A_6_B) and
   satisfied(code_825_A_6_C) and
   satisfied(code_825_A_6_D) and
   satisfied(code_825_A_6_E).


passed(epaFine_j_652_6B_710_C H) if not pending(H).

passed(epa_i_652_6B_714_A) if
   completed(every requiredFromEPA714) and
   passed(every epaFine_j_652_6B_710_C).


passed(epa_i_652_6B_714_B) if
   paid(every epaFine_j_652_6B_710_C).
```

B L program for checking K-connectivity of a graph

```
const n = 6.
const k = 5.

type node = {1..n}.
edge(node X, node Y) if X%n = (Y+1)%n.
edge(node X, node Y) if X%n = (Y+2)%n.
edge(node X, node Y) if edge(Y,X).

maybe removed(node N).
0 <= |{removed(node N)}| <= k-1.
reachable(node X, X) if not removed(X).

reachable(node X,node Y) if edge(X,Y)
                            and not removed(X)
                            and not removed(Y).

reachable_through(node X,node Z, node Y) if reachable(X,Z)
                            and reachable(Z,Y)
                            and not removed(X)
                            and not removed(Y)
                            and not removed(Z).


reachable(node X,node Y) if reachable_through(X,some node, Y).

disconnected(node X, node Y)  if    not reachable(X, Y)
                   and not removed(X)
                   and not removed(Y).


disconnected_graph  if   disconnected(some node, some node).


1<=|{disconnected_graph}|<=1.
```