The *pathlp* 2.x System User Manual

Igal Khitron[1] Mira Balaban[1] Michael Kifer[2]

Last edited on 30.06.2017

[1]Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva 84105, Israel, khitron,mira@cs.bgu.ac.il

[2]Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, U.S.A., kifer@cs.stonybrook.edu

Contents

Part 1: Language

1.1 Description

1. The semantics of *PathLP* is based on Well Founded Models for logic programs, so query answers can have three truth values: *true*, *undefined*, or *false*.
2. You can find the Background Theories in Appendices.
3. You can use a cut (*!*) in a usual way.
4. Facts, rules, and constraints can appear only in files (including the file entered from stdin).
5. Queries can be entered anywhere.
6. In files, *?-* query start symbol avoids printing any language output. *??-* prints regular query output. *:-* omits the answer but prints a warning on fail.
7. In command line *?-* can be used, but *:-* can not. *??-* can be omitted.

1.2 Fundamentals

1. *PathLP* is a logic programming language being developed by Mira Balaban and Michael Kifer, basing on **F-Logic**. It is planned to be a basic level for an **F-OML** Modeling Language built on it. As a logic programming language, *PathLP* may be regarded as knowledge base that includes data (facts), inference rules and integrity constraints.
2. The language is built on **path** term. *PathLP* program describes a directed graph with name labels on nodes and edges and information about the types of the nodes. Therefore there are two kinds of edges: Value edges and type one; and two kinds of relationships: Type membership and types subsetting.
3. **A variable**: Noted by a name preceded by a question mark: *?x* means a variable called *x*; *?_x* means a variable called *_x*. Variable name starting with an underscore is a **don't care** variable. *?_* and *?* are **disposable** (anonymous) variables (any appearance has a new value).
4. **Object path expression** (or path expression): Main grammar structure. The expression looks like $x.y_1[z_1]$. ... $.y_{n-1}[z_{n-1}].y_n[z_n]$, or $x.y_1[z_1]$. ... $.y_{n-1}[z_{n-1}].y_n$. x is a node (**object**) label with an outgoing edge (**property**) labeled by y_1 to the node whose value (label) is z_1, with outgoing edges path during y_2, y_3, ..., y_n edges until a node possibly labeled by z_n. A path expression value is its last node.

5. A bracketed expression (a **guard**) may or may not appear after any node in path expression. It cannot be used after the first outgoing node in expression. A path expression without followed guard on last edge is called an **unguarded path expression** and defines a group of all possible expression values in current database. Generally, the guard is a non-empty list of unguarded path expressions. For example, $a.b.c[d.e[?f].g, ?f.h, ?x]$ describes a path $a.b.c$ to the node whose label is also a value of paths $d.e.g$, $d.e.h$, and of the variable $?x$.
6. **Membership** relationship: Noted by a colon. *x:y* means *x* value belongs to the *y* type.
7. **Subsetting** relationship: Noted by a double colon. *x::y* means type *x* is included in type *y*. This relationship is transitive.
8. **Type path expression**: As nodes values graph, there also exists a nodes types graph. An expression $x!y_1[z_1]!y_2[z_2]!$ ... $!y_n[z_n]$ describes a type path, meaning: Any object of the type *x* can have an edge labeled y_1 to an object of the type z_1, which can have an edge labeled y_2, and so on (example 2). Type expression can limit a number of outgoing edges using a cardinality constraint. *x!y[z]{a..b}* describes that any object of type *x* should have no less than *a* and no more than *b* *y*-labeled edges whose destination node belongs to a type *z*. Any border can be a number or a variable. UML asterisk can be used as a legal language number. Cardinality constraint absence implies *{0..*}*.
9. Long path expressions can be used, as in $a.b.c[?x].d.e[?y]$, in other words you can build a long path omitting a guard anywhere. A long type path expression can have a cardinality constraint only after (and describing) the last edge.
10. **Facts and rules**: Semantic structures look like those in **Prolog**. For example, loading a file:
    a.b[c];
    a.b[d];
    ?s:f :- a.b[?s], a.m(?s)[?x], ?x>10;
    adds two facts and a rule to the current database. They are reasoned as usual.
11. **Queries**: Reasoning triggers. For example, a query
    ?- (?p = prop4 or ?p = prop11), q!?p[?m], ?m:class1;
    means: Find any values for *?p* and *?m* such as *p=prop4* or *p=prop11*, and also a *q* type object can have *?p* labeled edges with destination *?m* that belongs to *class1* (meta-) class. Usually queries used in command line and the *?-* can be omitted there.
12. **Constraints**: Another semantic structure (example 3). It is noted by *!-* and describes forbidden states. This means a constraint that was added to a database as a special query must fail. Such queries are checked during a so-called **system stability check process**. Failed query means succeeded constraint, and succeeded query means system stability downfall. For example, a constraint
    !- ?a.b[?x], ?x>10;
    prohibits *b* labeled edges from any *?a* to any *?x*, such as *?x*'s value is a number great than 10. The system stability check process is usually called after every file(s) load. Cardinality constraints are considered regular ones.
13. **Logical equality**: Noted by *:=:* (example 4). <*x*> *:=:* <*y*> means that two terms <*x*> and <*y*> (simple or compound, grounded or not) can be unified.

14. Logic operators that can be used in queries are: *and* (or *,*), *or*, *not*.
15. \+ can be used too, even it is not a logic operator, but a regular *fail_if*.

Part 2: System

The system manual is organized as follows: ....

2.1 Installation

Linux or Unix. Fill in the fields at the top of the file `pathlp.sh` according to your installation. Also ensure this file is executable. This can be done, for instance, as follows:
chmod +x pathlp.sh
Once the correct settings are provided, you can start the system using the `pathlp` command.

System requirements:

- **XSB**: included in download.
- **readlink**: highly recommended (it is almost certain that it is already installed).
- **rlwrap**: also highly recommended. This provides command line history, tab autocompletion (preserved language words, history, and file system items), and `Ctrl-R` history search.

Windows. Fill in the fields at the top of the file `pathlp.bat` according to your installation. Once the correct settings are provided, you can start the system using the `pathlp` shortcut.

System requirements:

- **XSB**: included in download.

Extensions. You can also use these optional files:

1. `pathlp.png` and `pathlp.ico`: PathLP icons for `Linux/Unix` and `Windows`, respectively.
2. `pathlp-install.xml`: new Mime type creation script in `Linux`
3. `pathlp.js`: code indentation in a `KDE` system (like in `Kate`).
4. `pathlp.xml`: code highlighting in a `KDE` system (like in `Kate`).
5. `katetest.ppl` and `katetest.pdf`: illustration of the use of `pathlp.xml` in KDE.

Follow Mime Extensions instructions to create a new Mime type in `Linux` with right icon, and, if you have *`Kate`*, to install highlighting and indentation for this Mime type (for `Linux` and `Windows`). You can find indentation rules in the same appendix. `Windows` *`Kate`* is available on windows.kde.org, Finland 4.7.4 mirror.

2.1.2 Linux Executables

In `Linux/Unix`, executable files containing *PathLP* programs can be created. To do so, put
#! /usr/bin/env pathlp
as the first line in the `*.ppl` files and make them executable:
chmod +x <filename>

Note that for this to work the command *pathlp* must be found by the shell, so the directory where *pathlp* is located must be in the `PATH` system variable. In C-shell based shells, such as **csh**, **tcsh**, this can be accomplished via the following command
setenv PATH $PATH:<directory where the pathlp shell script is found>
In sh-based shells such as **bash**, this can be done with the command:
export PATH=$PATH:<directory where the pathlp shell script is found>
You can place this command in the startup file of your system's shell.

2.2 The *pathlp* System

1. **Interactive system.** When a *pathlp* session begins, a prompt is shown. If the application cannot identify the time zone, the timestamp part of the prompt is omitted.
2. When a line finishes, it is checked for beeing completed input. Ignoring some special cases, the line should stop with ;. Otherwise, next line read and the process is redone. Beware of ambiguity: If you need something as *call((true;false))@_prolog;* in two lines, do not write
   call((true;
   false))@_prolog;
   because the first line is wrong parsed as all input. The semicolon could be moved to the second line start, for example.
3. The semicolon as whole input is treated as an empty line.
4. Empty lines are also allowed as input.
5. The *EOF* character (*Ctrl-D* in `Unix`-based systems, *Ctrl-Z*+Enter in `Windows`) terminates the session if it is entered at an empty line, and is ignored in the middle of a line.
6. When *pathlp* prints out an answer, it waits for further user instructions similarly to most **Prolog** systems. To get the next answer, type the semicolon (;) or space followed by a carriage return. If no further answers are needed, type the carriage return.
7. **File loading.** Use *load/2, 1*, *loadnew/2, 1*, or list syntax Builtins and in Dynamic Modules.
8. A line
   cd <directory name>

changes the current directory to the given one. A line *cd* returns to the directory where the application was invoked.

9. Any expression from the previous paragraph as command line tail is ignored, current working directory is changed, and the input continues to be read as the same line. This feature helps in working with a **KDE** system (like ***Kate***).
10. **Comments.** There are two types of comments: *% <comment>* marks everything as a comment till the end of line. */* <comment> */* marks everything in-between /* and */ as a comment. It can be used in the middle of the code.
11. **Variables check.** The compiler checks every rule and fact for singleton variables. Rules are checked also for unsafe variables (those that appear only in the head). You can suppress the warnings by converting the variable to be a silent (the name starts with the underscore symbol). The checks can be turned off using the *warnings/1* builtin.

2.2.2 Prolog Delegation

1. The full backquoted line is treated as full input. Any delimiters, comments, and changing directory commands are allowed outside.
2. Every term, query single expression or whole line delimited using backquotes are passed down to **Prolog** without parsing. For instance
   `` `:- import append/3 from basics.` ``
   However, variables inside such **Prolog** that you want to use in regular code must begin with the question mark, as in *PathLP* and unlike **Prolog**. E.g., *?x=ac, ?y=ad,* `` `?x @< ?y` ``. Question marks that do not designate a variable and backslash symbols must be escaped with another backslash.
3. @ after predicate activates the module system. F.i., *select([1, 2, 3, 4], ?x, ?y)@basics* calls accordant predicate in **Prolog** *basics* module. See Dynamic Modules for different use.
4. *_prolog* as module name means **Prolog** *usermod*, as in *repeat@_prolog*.
5. A variable as a module name can be used, but it must be bound at the time the call was made.
6. All infix **Prolog** operators can be used.
7. For math, use infix *is*. See details in Math.
8. The *pathlp* supports **DCG** (Definite clause grammar), using infix operator – –>, but redundant parentheses are advisable.
9. Language special symbols can be converted to infix by two followed spaces. The unification *?x = a < b* fails. For variables coupling use *?x = a < b*. On the other hand, extra space after equality symbol fails both options. The convertible symbols are [**<, >, =<, >=, =, :, .**]. A comma is also like this, but it has special roles, so be extra careful. This feature does not work at the end of the line, for normal comments usage.

2.3 Miscellanea

1. No property starting with underscore can be defined. The fact
   a._b[c];
   does not compile. The fact
   a!?x[b];
   is not reasoned by *a!_p[b]* query.
2. The idiom &<*character*> can be used to get the ASCII numeric value of <*character*>. For instance, &*A* is 65, &_ is 32.
3. Inside quoted atoms, the symbol & is used in the reverse sense: &<*xyz*>, where <*xyz*> is a three digit decimal number between 032 and 126, represents the ASCII character with the given ASCII code. To include the symbol & and the backslash inside quoted atoms, they must be escaped by backslash.
4. Double-quoted text has the same meaning as in **Prolog**: it's a list of ASCII characters represented as their numeric values. Usual escape sequences can be used in it.
5. Cardinality constraint infinity asterisk is a regular number, as in *3<**.
6. A compound term with zero arguments like *f()* is a synonym for a simple term *f*.
7. You can add a name to constraint as one word constant before the code. It is ignored by parser, but printed during system stability check process. For example:
   This_constraint _checks_for_negative _positives !- ?x:posnumber, ?x =< 0;
8. Keep in your mind the two primary differences between the negation kinds: 'not' and \+ (example 6). The first one is grounding. If you want to get the right answer, do not use unbound variables in \+ negation. The second one is recursion. The usage of \+ with recursive relation can cause problems.
9. **Rules and facts order**: If same structure has rules and facts, the facts are reasoned first, and the rules after them. For example, in the code:
   a.b[c];
   a.b[?x] :- !, ?x = d;
   a.b[e];
   a.b[?y] :- ?y = f;
   the query *?- a.b[?p]* returns *c*, *d*, and surpisingly *e*. It is better to have a practice writing all the facts before the rules.
10. More than that, there is no guaranty to reasoning order in facts. In rules, the only guaranty is proper cut usage. Do not expect to some reasoning particular order. You shouldn't also to expect to answers orders from facts before then rules, only for side-effects.
11. If you want to put a fact between rules, just convert it to a rule:
    a.b[?x] :- !, ?x = d;
    a.b[e] :- true;
    a.b[?y] :- ?y = f;
12. If you need to use numbers in path expressions, separate them by space: *3. 4[x]*, because *3.4[x]* will be misparsed as real number.

13. One more problem: multiple guard. The code
    a.b[c,d] :- <Body>;
    is converted to
    a.b[?x] :- [c,d]._member[?x], <Body>;
    So, if *Body* has a cut, *d* can be never reasoned.
14. Also, the fact
    a.b[c,d];
    is converted to a rule
    a.b[?x] :- [c,d]._member[?x];
    and as one, is a part of rules order, as explained above.

2.4 Tabling

1. The system uses tabling mechanism. The **tabling** (memoization) is an ability of a program to record each query call in a table. So each call to query checks first if there is an answer in the table. If this query was already called, the saved answer is returned, and there is no need to calculate the query once again.
2. The tabling has some adventages. The first one is runtime. If there is no need to rerun queries, the time of their run is saved.
3. The next adventage is infinite loop avoidance. The code
    ?x.path[?y] :- ?x.path[?z], ?z.edge[?y];
    ?x.path[?y] :- ?x.edge[?y];
    a.edge[b];
    b.edge[c];
    c.edge[a];
    ?- ?x.path[?y];
    could enter to infinite loop if written in **Prolog** (so called *left recursion*), but works in *PathLP* because of tabling. The reasoning mechanism checks on call *?x.path[?z]* if there was such a call, finds that *?x.path[?y]* was started and not finished yet, understands there could be infinited loop here and cuts the decision tree. So we'll get the right answer.
4. On the other hand, the code
    ?x.path([?x, ?y])[?y] :- ?x.edge[?y];
    ?x.path([?x|?p])[?y] :- ?x.edge[?z], ?z.path(?p)[?y];
    a.edge[b];
    b.edge[c];
    c.edge[a];
    ?- ?x.path(?p)[?y];
    which not just finds nodes pairs, which number is final, even the number of possibilities to reach them is infinite, but also collects the path, enters to infinite loop. The cause is tabling: the table should be filled with all the answers before the first is reasoned.

5. To help on this, there is a new path expression type added: **untabled property**. It works as regular type path expression, but is not tabled.
6. In queries only, the difference is like that: if you think some call to path expression has infinite number of answers, make this call untabled by doubling the path expression dot symbol. For example:
   ?x.path([?x, ?y])[?y] :- ?x.edge[?y];
   ?x.path([?x|?p])[?y] :- ?x.edge[?z], ?z..path(?p)[?y];
   a.edge[b];
   b.edge[c];
   c.edge[a];
   ?- ?x..path(?p)[?y];
   works properly giving all final part of answers you wanted by backtracking.
7. There is no difference in facts and rules.
8. Any property can be called in tabled and untabled way in the code and executes the same code, except tabling.
9. You are recommended to think if there could be infininite number of answers on property call and to double the dot in such cases. For example:
   ?- [1,2,3]._pappend([4,5,6])[?list];
   ?- ?x.._pappend([4,5,6])[?y];
10. One else tabling difference is recording the full call as table entry. It causes infinite loop on usage of infinite parameters. For example:
    ?- ?_x=[3|?_x], ?_x._plength[?length];
    is a bad choice. You should double dot here to get the right answer.
11. Same way, you can make **untabled membership** call, as in
    ?- ..., ?x::Type5, ... ;.
12. Aware from left recursion in untable calls. Rearrange you program to avoid infinite loops.

2.4.2 Bounded rationality

1. There is a powerfull mechanism (example 11) that helps to avoid infinite loops, called *bounded rationality*. It means, outlined, that when there are too much answers in the table, we can prone the decision tree.
2. The default MAXDEPTH (see below) value is 5,000.
3. There are three possible cases of bounded rationality usage.
4. The first one called **subgoal abstraction**. It means, if we have a wrong code code for Church numerals
   ?a:nochurch :- s(?a):nochurch;
   the query
   ?- ?x:nochurch;
   should fail. It doesn't, entering in infinite loop. To avoid this, the subgoal abstraction prones

the decision tree when receiving MAXDEPTH tree depth and marks the brunch failed, getting the right answer.

5. The subgoal abstraction works well also with lists. It means, if we have a wrong code
?a:nolist :- [?|?a]:nolist;
the query
?- ?x:nolist;
should fail. And indeed, subgoal abstraction prones the tree, getting the right answer.
6. The second case called **radial restraint**. It means. if we have a code for Church numerals
zero:church;
s(?x):church :- ?x:church;
the query
?- ?x:church;
will not give even one answer, because there is infinite answers number and the table will never be filled. To avoid this, the radial restraint stops the computation process receiving MAXDEPTH functor depth. The longest answer returned as undefined not fully ground:
Undefined answer:
?x = s(s(s(...(s(s(s(<temp-var>))))...)))
to allow unification with longer answers.
7. The third case needed because the platform still does not allow radial restraint on lists. So, for now, the lists are just truncated. It means, if we have a code
[]:list;
[?|?x]:list :- ?x:list;
the query
?- ?x:list;
will return by backtracking all the lists with length less than MAXDEPTH.

2.4.3 Depth control

1. There is a mechanism that allows to control the computation depth. There is a builtin *depth(<flag>, <value>)*, which marks relevant flags. Use it very carefully.
2. The flag *answer* has an positive integer value of maximal radial restraint functor depth.
3. The flag *list* has an positive integer value of maximal radial restraint list depth.
4. The flag *subgoal* has an positive integer value of maximum subgoal abstraction depth.
5. The flag *a(answer)* has a value of action maded on maximal radial restraint functor depth. The values are bounded_rationality (default), failure, error, warning.
6. The flag *a(list)* has a value of action maded on maximal radial restraint list depth. The values are failure (default), error, warning.
7. The flag *a(subgoal)* has a value of action maded on maximal subgoal abstraction depth. The values are abstract (default), failure, error.
8. Bounded rationality means functor radial restraint usage.
9. Abstract means subgoal abstraction usage.

10. Failure means fail on this depth.
11. Error means throw an error on this depth.
12. Warning means warn and continue computation on this depth.
13. *depth(<flag>, <value>)* influences on query computations. For system stability check process use *sdepth(<flag>, <value>)*.
14. You can use, of course, the regular builtins features: <*value*> as variable returns the current one, and <*value*> as *d* puts the default value. You can see the defaults in *table depth default* flag using *state/0*.

2.5 Builtins Properties

1. A query *a._size[?x]* is a getter for a quantity of outgoing edges from a node *a*.
2. A query *a._size(b)[?x]* is a getter for a quantity of outgoing edges labeled *b* from a node *a*.
3. A query *a._count[?x]* is a getter for a quantity of members of the type *a*. All three properties run until timeout finished (10 seconds and can be changed by *timeout/1*), and return undefined * after this.
4. A query *a._var[?boolean]* checks if the object is a variable.
5. A query *a._ground[?boolean]* checks if the object are fully ground.
6. A query *a._equal[b]* checks if two objects are identical.

2.5.2 Lists Library

1. There is a builtin lists library in the system.
2. *?maybelist._pislist[?boolean]* returns *true* iff the argument is a list.
3. *?list._pmember[?element]* succeeds if the *?element* is included in the *?list*. *undefined* answer is used in controversial situations. Warns on non-set list or attempt to find a variable in the list that already includes it. The warnings can be suppressed by the builtin *warnings/1*. Both relations can treat difference lists.
4. *?list._plength[?length]* succeeds if *?list* length is *?length*. *undefined* answer is used for open lists. All three properties can treat infinite lists, but you are encouraged to use untabled call.
5. *?list1._pappend(?list2)[?list]* succeeds if *?list1* and *?list2* appended to the *list*. Every argument should be a proper list or a variable.
6. *?listslist._pappend[?list]* succeeds if the *?listlist* is a proper list of proper lists (maybe variables) that can be appended to *list*. It should be a proper list or a variable too.

2.6 Builtins

1. *help/0* prints help screen.
2. *halt/0* finishes the session.

3. *halt/1* finishes the session with parameter errorlevel exit status.
4. *load/1* loads the argument atom file name or file names list. The list can have regular syntax and it is treated from left to right. Underscore as file name reads stdin and stops on *EOF*. Every language file should have the ***.ppl*** extension. It is compiled (except if it was not edited from last load) and loaded to existing database. ***.P*** extension consults **Prolog** file. All other names are checked first for accordant language name (a call *load(file1)* searches for ***file1.ppl)***, then for **Prolog** name, and then tried as is for **Prolog** consulting.
5. [< *names list*>] can be used instead.
6. *loadnew/1* - as *load/1*, but the *resetsystem/0* is called first. If there is only one file to load, the system reset done only before success loading.
7. [[<*names list*>]] can be used instead. See more in Dynamic Modules.
8. *state/0* prints system configuration values.
9. *resetsystem/0* destroys the database and restores the default equality and typing flags.
10. *stable/1* used as *stable(?)* runs system stability check process. See in Dynamic Modules the explanation.
11. *pwrite/1*+ prints all the arguments.
12. *pwriteln/0*+ prints all the arguments, if any, and breaks the line.
13. <*conf*>*/1* where <*conf*> is configuration built in changes the system state. The argument can be one of two setting options, *d* for default (prints the new value), or variable to instantiate the current option. The built ins are:

Name	Default	Description	Other
warnings	on	Print compilation and _pmember warnings	off
answers	wait	Wait after each query answer	all
equality	empty	Don't use language logical equality *	normal
typing	inference	Use typing inference *	checking
stability	automatic	Check system stability after each load call	initiated
tracing	no	Omit stack backtrace on errors	trace

14. The timeout can be changed using *timeout/1*. The parameter is in seconds, at least 0.001, or * for infinite.
15. See more in Dynamic Modules (also for * in the table) and Tabling.

2.6.2 Program Control

1. *if <query formula> then <query formula> else <query formula>* is a regular form of language condition statement. For example:
   ?- if a.b[?q], ?q:c then pwriteln(?q) else pwriteln(none);
2. The *else* part can be omitted, it means fail on else.
3. You can see the nesting order from example:
   if 3<4 then if 5>6 then pwriteln(yes) else pwriteln(no);

is treated as
if 3<4 then (if 5>6 then pwriteln(yes)) else pwriteln(no);

4. *Else* evaluates only the nearest query expression. For grouping use parentheses:
   if 3>4 then pwriteln(no) else (pwriteln(yes), pwriteln(absolutely));
5. *true*, *otherwise*, *false*, *fail* and *undefined* can be used as is.

2.6.3 Aggregation

1. The system supports data aggregation from the version **2.0** (example 10). There are six aggregation operators: *bagof*, *setof*, *sum*, *average*, *max* and *min*. All of them have the same syntax.
2. To find all the answers of some query, use
   bagof(<aggregation variable>, (<grouping variable>,) (<query formula>,)+ <result>)*
   For instance, to find all answers for the third element in the list that are smaller than 5:
   ?- bagof(?x, _pmember([?a, ?b, ?x], [[1, 5, 3], [4, 5, 6], [7, 8, 9]]), ?x < 5, ?res);
   The needed list returns in *?res*. Pay attention you can insert any positive number of query formulas without extra parentheses.
3. The grouping part allows to get partial answers according to the grouping variables.
   ?- bagof(?x, ?b, _pmember([?a, ?b, ?x], [[1, 5, 3], [4, 5, 6], [7, 8, 9]]), ?x < 5, ?res);
   returns the backtracking answers of *?res* for each different value of *?b*.
4. You should know that the aggregation variable (i.e. the first one) is locally quantified. This allows you usage of non silent variables without getting unbounded answer. You can use the same variable name outside of the aggregation operator scope, and they are not unificated.
5. The *setof* operator removes duplications from the answers.
6. The *sum* operator sums the answers, if they are numbers list. * can be used.
7. Same way *average* calculates the average, *max* the maximum and *min* the minimum. They fail for empty lists.

2.7 Testing

1. Regression tests engine is now added to *pathlp*.
2. The default system configuration is turned off testing. You can change this using *testing/1*. The options are *work* (default), *debug*.
3. The accordand command line arguments are **-work**, **-debug**.
4. The simple way to add tests to you program is like this: For each test add two object path expressions.
5. The first one is *<testname>.answer(<varsnum>)[<answerslist>];*. For example, if you want to run a test *test1*, which checks two variables, *?x* and *?y*, and the answers should be *?x = 3, ?y = 5* and *?x = 8, ?y = 9*, write:
   test1.answer(2)[[[3, 5], [8, 9]]];

6. If you have no variables, use regval (*true*, *undefined*, *false*) in the guard. For example, if the test call *test3* should fail, write:
   test3.answer(0)[false];
7. If you want to check undefined answers separately, use <*true list*> / <*undefined list*> guard. For example, if the answers should be *?x = 3, ?y = 5*; *Undefined: ?x = 8, ?y = 9*; *?x = 33, ?y = 55*; *Undefined: ?x = 88, ?y = 99*, write:
   test5.answer(2)[[[3, 5], [33, 55]] / [[8, 9], [88, 99]]];
8. Usage *in/1* fanctor as the answer tests if the real answers list is a sublist of the test answer. For example, *test9.answer(1)[in([[1], [3], [5]])]* means you want the answers be nothing more than these numbers (but at least one answer).
9. Usage *one/1* in place of *in/1* works the same, and also checks that there is exactly one answer from the list.
10. The second path expression is the test call. Use a rule
    <testname>.call[<variables list>] :- <test code>; For example, for above examples you could write:
    test1.call[[?x, ?y]] :- ?x.someedge[?y], ?y:someclass;
    test3.call[[]] :- 3 > 4;
11. To test on each program compilation, add as the last line the directive *:- test(MODNAME);*
12. On running with *stability* flag is *initiated*, the system stability check process fail is counted as test error.
13. You can do more than just simple testing on compilation. For example, the testing could be done anytime, using two builtins below.
14. The builtin *test/0* runs all the tests that are loaded to the application database.
15. The builtin *test/1* runs all the test that are in the current module database.
16. So, as mentioned above, use *MODNAME* as the parameter in programs, if you want to run the tests in the module this program is loaded to.
17. If the parameter is a variable, it is unified to a boolean test result.
18. You can write the tests in separate file, of course, and include them using *#include* directive.
19. You can run the test in the same program on incremental database. For example, the file:
    <tests path expressions or include lines>
    <some facts and rules>
    :- test(MODNAME);
    <some more facts and rules>
    :- test(MODNAME);
    runs the tests twice: on the first program part, and on all of it.
20. If the test runs more than 10 seconds, you get an error *Time limit exceeded*. It comes for infinite loops avoidance. You can change the timeout using *timeout/1*.

2.8 Command Line

1. All command line parameters are executed from left to right (example 12). Any not recognized parameter is treated as file name and loaded by *load/1*. See in Dynamic Modules for this and more.
2. **-stdin** or _ executes *load(_)*.
3. **-help** or **-h** executes *help/0*. Call with **-h** only exits after printing.
4. **-halt** stops the session and ignores the rest.
5. **-hlt <STATUS>** does the same, returning *STATUS* error level.
6. **-reset** executes *resetsystem/0*.
7. **-stab** executes *stable/1* for default module (See more in Dynamic Modules).
8. **-won**, **-woff** executes *warnings/1*.
9. **-await**, **-aall** executes *answers/1*.
10. **-eempty**, **-enormal** executes *equality/1*.
11. **-tinf**, **-tcheck** executes *typing/1*.
12. **-saut**, **-sinit** executes *stability/1*.
13. **-bno**, **-btrace** executes *tracing/1*.
14. **-state** executes *state/0*.
15. Parameters can be passed to **XSB**, using environment variable by calling (once per session and until a different call is used):

	tcsh, csh	**sh, bash, dash, ksh**	**Windows**
Single parameter	setenv xsbparam -p	export xsbparam=-p	set xsbparam=-p
Multy parameters	setenv xsbparam ′-p -S′	export xsbparam=′-p -S′	set xsbparam=-p -S
Remove all	unsetenv xsbparam	unset xsbparam	set xsbparam=

16. If you attempt to use any parameters permanently, add them as **XSB** parameters in *pathlp.sh* or *pathlp.bat*.
17. If you want to reset the system, use before the file name **-reset** explicitly.
18. If you want to run the system stability check process, run it explicitly. It will run automatically just once, after all the arguments treat.

2.9 Dynamic Modules

1. The *pathlp* system supports Dynamic modules (example 7). The database can be splitted into different modules and file can be loaded to particular module.
2. The prompt prints module name, if not default.
3. Without Dynamic modules usage the system uses always *def* default module.
4. There is always current module. If not changed, it is the default one, *def*.

5. The built in *newmodule/1* creates new module with parameter name. The name should be bounded and not used for any existing module.
6. The built in *module/1* changes the current module to parameter. The module should exist. Usage with a variable unifies the current module.
7. The built in *load/2* loads the first parameter file or files list to second parameter module. The module should exist. The synonym is *[file1, file2, ...|module]*. If the **Prolog** file is loaded, the module name is ignored.
8. The built in *load/1* loads to the current module.
9. The built in *loadnew/2* loads the first parameter file or files list to second parameter module, erasing this module. The synonym is *[[file1, file2, ...]|module]*.
10. The built in *loadnew/1* does this with the current module.
11. The built in *stable/1* runs the system stability check process on the parameter module. Unbound variable parameter checks all existing modules, and the variable is unified to a boolean answer - iff the check succeeded.
12. The built in *resetsystem/0* still erases all the database. If you want to erase the database in particular module, use *[[]|modulename]*. If you want to do this in current module, use *[[]]*.
13. The built ins *equality/2* and *typing/2* work for the second parameter module.
14. The built ins *equality/1* and *typing/1* work for the current module.
15. The command line arguments for the modules are **-new <modulename>** for *newmodule/1*, **-module <modulename>** for *module/1*, **-tom <modulename> <filename>** for *load/2*, **-stb <modulename>** for *stable/1*, and **-stab** for *stable(def)*.
16. To check specific module during the reasoning, use @ symbol. For example, the query:
    ?- ?x.?y[?z]@module1, ?z:?y, ?y!?x[?t]@module2.
    finds all the possibilities, checking in *module1*, current module, and *module2* respectively. The variable module name should be bound. The reasoning with non existing module fails.
17. For consistency, the current module in code in a file is the module the file is loaded to, otherwise use explicit module name.
18. In files only *MODNAME* string is replaced with the name of the module this file is loaded to.

2.10 Libraries

1. Any program can load library files, using extension ***.lpp**. To do so, insert to the code a line
   ### <libname>.lpp
   where <libname> is a library file. The format is strict. Don't use extra spaces or comments in this line. A name with directory path is used as is. Otherwise, the file is searched in **<PathLP_dir>/lpp**, and then in current directory.
2. You can create your own libraries. Use the predicate *load(<filename>.ppl, lpp)*. The library **<filename>.lpp** is created in the same directory as <filename>.
3. Therefore, using *newmodule(lpp)* is forbidden.

4. Of course, you can include *PathLP* files.
   #include "filename"
   makes the job. Think about inclusion at the end of file, to avoid line numbering problems.

2.11 Preprocessor

1. All non command line input is passed through the preprocessor before compilation. *#define* defines constants or parameterized macros.
   #define X 3
   replaces all standalone *X* appearances with *3*,
   #define same(x) x..x
   replaces an expression *{same(3)}* with *{3..3}*.
2. *#defeval* works almost the same, except evaluating the inner macros values statically and not during substitution.
3. *#include "<filename>"* includes a given file in the code. You can use the relative path, the current directory is **<PathLP_dir>**.
4. *#ifeq <A> <B>* compiles next code lines if two expressions are literally the same, after macros substitution. For example,
   #ifeq X 3
5. *#ifdef <constant>* compiles the next code lines if the constant was defined.
6. *#ifneq* and *#ifndef <constant>* work in the opposite way.
7. *#if <expression>* evaluates the given expression arithmetically and compile the next code lines if the value is zero. Conditions can be combined using &&, ||, and *!* as negation. *defined(<constant>)* succeeds if the constant was defined.
8. Paragraph 4-8 statements finishes with one of the paragraph 10-12 statements.
9. *#else* compiles the next lines if the condition fails.
10. *#elif <condition>* is an "else" with inner "if". It can be used more than once and maybe *#else* as last one.
11. *#endif* finishes the condition statement. Nested "if" statements can be used.
12. *#exec <command>* replaces the line with the result of OS call. For example,
    #exec echo ":- pwriteln('" `date` "');"
    prints compilation timestamp on each run.
13. *#eval <expression>* replaces the line with an arithmetical expression result. The code
    :- writeln(
    #eval 3 + 4
)@_prolog;
    prints 7.
14. You can use *#mode* statement in the usual preprocessors way.
    #mode standard pathlp
    returns to the normal mode.

Part 3: Examples

All example files are in **<PathLP_dir>/examples**.

3.1 Simple Queries

Code (**example1**.ppl):

```
a.b[c];
a.b[d];
a.e[f];
g.b[c];
t.?x[?y] :- a.?x[?y];
```

• Logger:

```
khitron [9:18]~/Desktop/PathLP>pathlp

PathLP [09:18] > ?- cd examples
PathLP [09:18] (~/Desktop/PathLP/examples) > ?- [example1];
[PathLP: compiling example1.ppl]
[PathLP: file example1.ppl has 5 lines]
true
PathLP [09:18] (~/Desktop/PathLP/examples) > ?- cd
PathLP [09:18] > ?- ?R.?T[?V];

Answer:
?R = t
?T = e
?V = f;

Answer:
?R = t
```

```
?T = b
?V = d;

Answer:
?R = t
?T = b
?V = c;

Answer:
?R = a
?T = b
?V = c;

Answer:
?R = a
?T = b
?V = d;

Answer:
?R = a
?T = e
?V = f;

Answer:
?R = g
?T = b
?V = c;
false
PathLP [09:19] > ?- ?.?R[?V];

Answer:
?R = e
?V = f;

Answer:
?R = b
?V = d;

Answer:
?R = b
?V = c;
```

```
Answer:
?R = b
?V = c;

Answer:
?R = b
?V = d;

Answer:
?R = e
?V = f;

Answer:
?R = b
?V = c;
false
PathLP [09:19] > ?- answers(all);
true
PathLP [09:19] > ?- ?.?R[?V];

Answer:
?R = b
?V = c

Answer:
?R = b
?V = d

Answer:
?R = e
?V = f
PathLP [09:19] > ?- answers(d);
answers = wait
true
PathLP [09:19] > ?- ?.?R[?V];

Answer:
?R = e
?V = f;
```

```
Answer:
?R = b
?V = d;

Answer:
?R = b
?V = c;

Answer:
?R = b
?V = c;

Answer:
?R = b
?V = d;

Answer:
?R = e
?V = f;

Answer:
?R = b
?V = c;
false
PathLP [09:19] > ?- a.g[?x];
false
PathLP [09:19] > ?- a.?g[?x];

Answer:
?g = b
?x = c;

Answer:
?g = b
?x = d;

Answer:
?g = e
?x = f;
false
PathLP [09:20] > ?- a._size[?x];
```

```
Answer:
?x = 3;
false
PathLP [09:20] > ?- a._size(?s)[?x];
false
PathLP [09:20] > ?- a._size(b)[?x];

Answer:
?x = 2;
false
PathLP [09:20] > ?- a._size(s)[?x];
false
```

3.2 Type Path Expressions and System Stability Check Process

Code (**example2**.ppl):

```
a.b[c];
a.b[d];
a.e[f];
g.b[c];
t.?x[?y] :- a.?x[?y];
class1!b[class2]{3..4};
```

• Logger:

```
khitron [12:20]~/Desktop/PathLP>pathlp

PathLP [12:20] > ?- cd examples
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- [example2];
[PathLP: compiling example2.ppl]
[PathLP: file example2.ppl has 6 lines]
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- [_];
> a:class1;
>
[PathLP: compiling from standard input]
[PathLP: file from standard input has 1 line]

The system is unstable:
```

```
In module def the type cardinality constraint
class1!b[class2]{3..4};
is not satisfied for:
Object a
The system stability check finished
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- [_];
>
[PathLP: compiling from standard input]
[PathLP: file from standard input has 0 lines]

The system is unstable:

In module def the type cardinality constraint
class1!b[class2]{3..4};
is not satisfied for:
Object a
The system stability check finished
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- [_];
> a.b[m];
>
[PathLP: compiling from standard input]
[PathLP: file from standard input has 1 line]
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- ?!?[?]{2..5};
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- ?!?[?]{4..5};
false
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- ?!?[?]{4..4};
false
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- ?!?[?]{0..*};
true
PathLP [12:20] (~/Desktop/PathLP/examples) > ?- ?!?[?];
true
```

3.3 Constraints

Code (**example3**.ppl):

```
a.b[c];
a.b[d];
a.e[f];
g.b[c];
t.?x[?y] :- a.?x[?y];
a.e[5];
a.e[10];
a.e[14];
!- a.?[?x],
   ?x>8;
```

• Logger:

```
khitron [12:25]~/Desktop/PathLP>pathlp

PathLP [12:25] > ?- cd examples
PathLP [12:25] (~/Desktop/PathLP/examples) > ?- [example3];
[PathLP: compiling example3.ppl]
[PathLP: file example3.ppl has 10 lines]

The system is unstable:

In module def the constraint:
!- a.?[?x], ?x>8;
is not satisfied for:

Answer:
?x = 10

Answer:
?x = 14
The system stability check finished
true
```

3.4 Logical Equality

Code (**example4**.ppl):

a:=:b;
b:=:c;
q.w[c];

• Logger:

```
khitron [12:39]~/Desktop/PathLP>pathlp

PathLP [12:39] > ?- cd examples
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- [example4];
[PathLP: compiling example4.ppl]
[PathLP: file example4.ppl has 3 lines]
true
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- ?x.?y[?z];

Answer:
?x = q
?y = w
?z = c;
false
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- equality(normal);
true
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- ?x.?y[?z];

Answer:
?x = q
?y = w
?z = b;

Answer:
?x = q
?y = w
?z = a;

Answer:
?x = q
?y = w
?z = c;
false
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- equality(d);
equality = empty
true
```

```
PathLP [12:40] (~/Desktop/PathLP/examples) > ?- ?x.?y[?z];

Answer:
?x = q
?y = w
?z = c;
false
```

3.5 Typing Inference

In typing inference mode the type of edge destination is implied from type path expressions.

Code (**example5**.ppl):
A!b[C];
a:A;

• Logger:

```
khitron [12:43]~/Desktop/PathLP>pathlp

PathLP [12:45] > ?- cd examples
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- [example5];
[PathLP: compiling example5.ppl]
[PathLP: file example5.ppl has 2 lines]
true
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- ?x:?y;

Answer:
?x = a
?y = A;
false
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- [_];
> a.b[p];
>
[PathLP: compiling from standard input]
[PathLP: file from standard input has 1 line]
true
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- ?x:?y;
```

```
Answer:
?x = p
?y = C;

Answer:
?x = a
?y = A;
false
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- typing(checking);
true
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- ?x:?y;

Answer:
?x = a
?y = A;
false
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- typing(d);
typing = inference
true
PathLP [12:45] (~/Desktop/PathLP/examples) > ?- ?x:?y;

Answer:
?x = p
?y = C;

Answer:
?x = a
?y = A;
false
```

3.6 Two Types of Negation

PathLP supports two types of negation: `not` and `\+`.

Generally, one should use `not` for negation, as this is the type of negation that has the correct well-founded semantics in all cases. For instance, in the following program, the edge *p* works while the edge *q* doesl not:

Code (**example6**.ppl):

```
% not adventage
example.p[?] :- not example.p[?];
example.q[?] :- \+ example.q[?];

% \+ adventage
‘:- table w/1.‘
‘w(0).‘

example.checkbp[?] :- abolish_table_pred(w/1)@_prolog,
                      !,
                      time(checkbpproc)@_prolog;

`checkbpproc :- basics:between(1, 1000000, X),
                \\\\+ w(X),
                fail.`

example.checktnot[?] :- abolish_table_pred(w/1)@_prolog,
                        !,
                        time(checktnotproc[?])@_prolog;

`checktnotproc :- basics:between(1, 1000000, X),
                  not w(X),
                  fail.`
```

• Logger:

```
PathLP [17:27] > ?- [example6];
[PathLP: compiling example6.ppl]
[PathLP: file example6.ppl has 23 lines]
true
PathLP [17:27] > ?- example.p[?];
undefined
PathLP [17:27] > ?- example.q[?];
Removing incomplete tables...
TABLING ERROR (Illegal cut over incomplete tabled predicate)
```

However, $\backslash+$ has its uses because it is much faster. Of course, in general it can give a wrong answer or worse (not terminate), but if applied to a non-recursive goal then the effect is the same as that of `not`.

• Logger:

```
PathLP [17:28] > ?- example.checkbp[?];
```

```
% 0.39 CPU in 0.391 seconds (99% CPU)
false
PathLP [17:28] > ?- example.checktnot[?];
% 0.297 CPU in 0.295 seconds (100% CPU)
false
```

3.7 Dynamic Modules

Code (**example7**.ppl):

```
??- pwrite(Before, ': '),
      ?x:?y,
      ?y:?z@prt;
 ?- newmodule(prt);
 ?- pwrite('Please enter line a:b; and end of file for module def'),
    [_];
 ?- pwrite('Please enter line b:c; and end of file for module prt'),
    [_|prt];
??- pwrite(After, ': '),
      ?x:?y,
      ?y:?z@prt;
 ?- module(prt);
??- pwrite('Another place: '),
      ?x:?y,
      ?y:?z@prt;
```

• Logger:

```
khitron [19:28]~/Desktop/PathLP/examples>pathlp

PathLP [19:28] > ?- [example7];
[PathLP: compiling example7.ppl]
[PathLP: file example7.ppl has 15 lines]
Before: false
Please enter line a:b; and end of file for module def> a:b;
>
[PathLP: compiling from standard input]
[PathLP: file from standard input has 1 line]
Please enter line b:c; and end of file for module prt> b:c;
>
```

```
[PathLP: compiling from standard input]
[PathLP: file from standard input has 1 line]
After:
Answer:
?x = a
?y = b
?z = c;
false
Another place:
Answer:
?x = a
?y = b
?z = c;
false
true
PathLP [19:29] {prt} > ?- ?x:?y, ?y:?z@prt;
false
PathLP [19:29] {prt} > ?-
```

3.8 Reasoning

Code (**example8**.ppl):

```
Dir!child[Node]{2..2};
Node!size[Int]{1..1};
File!content[string]{1..1};

?F.size[?S] :- ?F:File,
               ?F.content[?C],
               ?C._plength[?S];
?D.size[?S] :- ?D:Dir,
                ?D.child[?C1],
                ?D.child[?C2],
                ?C1 != ?C2,
                ?C1.size[?S1],
                ?C2.size[?S2],
                ?S is ?S1+?S2;

Dir :: Node;
```

```
File :: Node;

d1:Dir;
d2:Dir;
f1:File;
f2:File;
f3:File;

d1.child[f1];
d1.child[f2];
d2.child[d1];
d2.child[f3];

f1.content["aa"];
f2.content["aaa"];
f3.content["aaaaa"];

??- d2.size[?S];
```

• Logger:

```
khitron [17:38]~/Desktop/PathLP/examples>pathlp
PathLP [17:38] > ?- [example8];
[PathLP: compiling example8.ppl]
[PathLP: file example8.ppl has 34 lines]

Answer:
?S = 10;
false
true
PathLP [17:38] > ?-
```

3.9 Imaginary Aggregation

Code (**example9**.ppl):

```
Dir!child[Node]{1..*};
Node!size[Int]{1..1};
File!content[string]{1..1};

?F.size[?S] :- ?F:File,
              ?F.content[?C],
              ?C._plength[?S];
?D.size[?S] :- ?D:Dir,
               ?D.allchild[?Childs],
               ?Childs.nodeSizes[?Sizes],
               ?Sizes.sumlist[?S];

?D.allchild[?Childs] :- ?D._size(child)[?N],
                        ?Childs._plength[?N],
                        ?Childs.children[?D],
                        !;
?D.allchild[[]] :- not ?D._size(child)[?];

[?X|?Ys].children[?D] :- ?D.child[?X],
                         ?Ys.children[?D],
                         not ?Ys._pmember[?X];
[].children[?];

[?Node|?Nodes].nodeSizes[[?S|?Sizes]] :- ?Node.size[?S],
                                         ?Nodes.nodeSizes[?Sizes];
[].nodeSizes[[]];

[].sumlist[0];
[?X|?Ys].sumlist[?S] :- ?Ys.sumlist[?YsSum],
                        ?S is ?X + ?YsSum;

Dir::Node;
File::Node;

d0:Dir;
d1:Dir;
f0:File;
f1:File;
f2:File;
f3:File;
f4:File;

d0.child[f0];
d0.child[d1];
d1.child[f1];
d1.child[f2];
```

d1.child[f3];
d1.child[f4];

f0.content["a"];
f1.content["aa"];
f2.content["aaa"];
f3.content["aaaa"];
f4.content["aaaaa"];

"a":string;
"aa":string;
"aaa":string;
"aaaa":string;
"aaaaa":string;

?x:Int :- integer(?x)@_prolog;

• Logger:

```
khitron [18:52]~/Desktop/PathLP/examples>pathlp
PathLP [18:52] > ?- [example9];
[PathLP: compiling example9.ppl]
[PathLP: file example9.ppl has 62 lines]
true
PathLP [18:52] > ?- d0.size[?S0];

Answer:
?S0 = 15;
false
PathLP [18:52] > ?- d1.size[?S1];

Answer:
?S1 = 14;
false
PathLP [18:52] > ?- f3.size[?S3];

Answer:
?S3 = 4;
false
```

3.10 Aggregation

Code (**example10**.ppl):

```
Dir!child[Node]{1..*};
Node!size[Int]{1..1};
File!content[string]{1..1};

?F.size[?S] :- ?F:File,
               ?F.content[?C],
               ?C._plength[?S];
?D.size[?S] :- ?D:Dir,
                sum(?size, ?D.child[?Child], ?Child.size[?size], ?S);

Dir::Node;
File::Node;

d0:Dir;
d1:Dir;
f0:File;
f1:File;
f2:File;
f3:File;
f4:File;

d0.child[f0];
d0.child[d1];
d1.child[f1];
d1.child[f2];
d1.child[f3];
d1.child[f4];

f0.content["a"];
f1.content["aa"];
f2.content["aaa"];
f3.content["aaaa"];
f4.content["aaaaa"];

"a":string;
"aa":string;
"aaa":string;
"aaaa":string;
```

"aaaaa":string;

?x:Int :- integer(?x)@_prolog;

• Logger:

```
khitron [17:10]~/Desktop/PathLP/examples>pathlp
PathLP [17:10] > ?- [example10];
[PathLP: compiling example10.ppl]
[PathLP: file example10.ppl has 41 lines]
true
PathLP [17:10] > ?- d0.size[?S0];

Answer:
?S0 = 15;
false
PathLP [17:10] > ?- d1.size[?S1];

Answer:
?S1 = 14;
false
PathLP [17:10] > ?- f3.size[?S3];

Answer:
?S3 = 4;
false
PathLP [17:10] > ?- ?size = hello, sum(?size, ?D,
                  > ?D.child[?_Child], ?_Child.size[?size], ?S);

Answer:
?size = hello
?D = d0
?S = 15;

Answer:
?size = hello
?D = d1
?S = 14;
false
PathLP [17:10] > ?- sum(?size, ?D,
                  > (?D.child[?_Child], ?_Child.size[?size]) or
                  > (?D:File,?D.size[?size]), ?S);
```

```
Answer:
?D = d0
?S = 15;

Answer:
?D = d1
?S = 14;

Answer:
?D = f0
?S = 1;

Answer:
?D = f1
?S = 2;

Answer:
?D = f2
?S = 3;

Answer:
?D = f3
?S = 4;

Answer:
?D = f4
?S = 5;
false
```

3.11 Bounded rationality

Code (**example11**.ppl):

```
:- depth(subgoal, 30),
   depth(answer, 30),
   depth(list, 30);

:- answers(all);
```

?a:nochurch :- s(?a):nochurch;

?a:nolist :- [?|?a]:nolist;

zero:church;
s(?x):church :- ?x:church;

[]:list;
[?|?x]:list :- ?x:list;

• Logger:

```
khitron-tapuz2 [17:40]~/Desktop/PathLP>pathlp

PathLP [17:40] > ?- % Wrapping lines for readability.

PathLP [17:40] > ?- [example11];
[PathLP: compiling example11.ppl]
[PathLP: file example11.ppl has 15 lines]
true

PathLP [17:40] > ?- ?x:nochurch;
false

PathLP [17:40] > ?- ?x:nolist;
false

PathLP [17:40] > ?- ?x:church;

Answer:
?x = zero

Answer:
?x = s(zero)

Answer:
?x = s(s(zero))

Answer:
?x = s(s(s(zero)))
```

```
Answer:
?x = s(s(s(s(zero))))

Answer:
?x = s(s(s(s(s(zero)))))

Answer:
?x = s(s(s(s(s(s(zero))))))

Answer:
?x = s(s(s(s(s(s(s(zero)))))))

Answer:
?x = s(s(s(s(s(s(s(s(zero))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(zero)))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(zero))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(zero))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(zero))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(zero))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(zero)))))))))))))))))
```

```
Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))))))))))
```

```
Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero)))))))))))))))))))))))))))))

Answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(zero))))))))))))))))))))))))))))))

Undefined answer:
?x = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
(s(_h8491)))))))))))))))))))))))))))))))

PathLP [17:40] > ?- ?x:list;

Answer:
?x = []

Answer:
?x = [_h8408,_h8410,_h8412,_h8414,_h8416,_h8418,_h8420,_h8422,
_h8424,_h8426,_h8428,_h8430,_h8432,_h8434,_h8436,_h8438,_h8440,
_h8442,_h8444,_h8446,_h8448,_h8450,_h8452,_h8454,_h8456,_h8458,
_h8460,_h8462,_h8464]

Answer:
?x = [_h8478,_h8480,_h8482,_h8484,_h8486,_h8488,_h8490,_h8492,
_h8494,_h8496,_h8498,_h8500,_h8502,_h8504,_h8506,_h8508,_h8510,
_h8512,_h8514,_h8516,_h8518,_h8520,_h8522,_h8524,_h8526,_h8528,
_h8530,_h8532]

Answer:
?x = [_h8546,_h8548,_h8550,_h8552,_h8554,_h8556,_h8558,_h8560,
_h8562,_h8564,_h8566,_h8568,_h8570,_h8572,_h8574,_h8576,_h8578,
_h8580,_h8582,_h8584,_h8586,_h8588,_h8590,_h8592,_h8594,_h8596,
_h8598]

Answer:
?x = [_h8612,_h8614,_h8616,_h8618,_h8620,_h8622,_h8624,_h8626,
_h8628,_h8630,_h8632,_h8634,_h8636,_h8638,_h8640,_h8642,_h8644,
_h8646,_h8648,_h8650,_h8652,_h8654,_h8656,_h8658,_h8660,_h8662]

Answer:
?x = [_h8676,_h8678,_h8680,_h8682,_h8684,_h8686,_h8688,_h8690,
_h8692,_h8694,_h8696,_h8698,_h8700,_h8702,_h8704,_h8706,_h8708,
_h8710,_h8712,_h8714,_h8716,_h8718,_h8720,_h8722,_h8724]

Answer:
?x = [_h8738,_h8740,_h8742,_h8744,_h8746,_h8748,_h8750,_h8752,
_h8754,_h8756,_h8758,_h8760,_h8762,_h8764,_h8766,_h8768,_h8770,
_h8772,_h8774,_h8776,_h8778,_h8780,_h8782,_h8784]

Answer:
?x = [_h8798,_h8800,_h8802,_h8804,_h8806,_h8808,_h8810,_h8812,
_h8814,_h8816,_h8818,_h8820,_h8822,_h8824,_h8826,_h8828,_h8830,
_h8832,_h8834,_h8836,_h8838,_h8840,_h8842]

Answer:
?x = [_h8856,_h8858,_h8860,_h8862,_h8864,_h8866,_h8868,_h8870,
_h8872,_h8874,_h8876,_h8878,_h8880,_h8882,_h8884,_h8886,_h8888,
_h8890,_h8892,_h8894,_h8896,_h8898]

Answer:
?x = [_h8912,_h8914,_h8916,_h8918,_h8920,_h8922,_h8924,_h8926,
_h8928,_h8930,_h8932,_h8934,_h8936,_h8938,_h8940,_h8942,_h8944,
_h8946,_h8948,_h8950,_h8952]

Answer:
?x = [_h8966,_h8968,_h8970,_h8972,_h8974,_h8976,_h8978,_h8980,
_h8982,_h8984,_h8986,_h8988,_h8990,_h8992,_h8994,_h8996,_h8998,
_h9000,_h9002,_h9004]

Answer:
?x = [_h9018,_h9020,_h9022,_h9024,_h9026,_h9028,_h9030,_h9032,
_h9034,_h9036,_h9038,_h9040,_h9042,_h9044,_h9046,_h9048,_h9050,
_h9052,_h9054]

Answer:
?x = [_h9068,_h9070,_h9072,_h9074,_h9076,_h9078,_h9080,_h9082,
_h9084,_h9086,_h9088,_h9090,_h9092,_h9094,_h9096,_h9098,_h9100,
_h9102]

Answer:
?x = [_h9116,_h9118,_h9120,_h9122,_h9124,_h9126,_h9128,_h9130,
_h9132,_h9134,_h9136,_h9138,_h9140,_h9142,_h9144,_h9146,_h9148]

Answer:
?x = [_h9162,_h9164,_h9166,_h9168,_h9170,_h9172,_h9174,_h9176,
_h9178,_h9180,_h9182,_h9184,_h9186,_h9188,_h9190,_h9192]

Answer:
?x = [_h9206,_h9208,_h9210,_h9212,_h9214,_h9216,_h9218,_h9220,
_h9222,_h9224,_h9226,_h9228,_h9230,_h9232,_h9234]

Answer:
?x = [_h9248,_h9250,_h9252,_h9254,_h9256,_h9258,_h9260,_h9262,
_h9264,_h9266,_h9268,_h9270,_h9272,_h9274]

Answer:
?x = [_h9288,_h9290,_h9292,_h9294,_h9296,_h9298,_h9300,_h9302,
_h9304,_h9306,_h9308,_h9310,_h9312]

Answer:
?x = [_h9326,_h9328,_h9330,_h9332,_h9334,_h9336,_h9338,_h9340,
_h9342,_h9344,_h9346,_h9348]

Answer:
?x = [_h9362,_h9364,_h9366,_h9368,_h9370,_h9372,_h9374,_h9376,
_h9378,_h9380,_h9382]


?x = [_h9396,_h9398,_h9400,_h9402,_h9404,_h9406,_h9408,_h9410,
_h9412,_h9414]

Answer:
?x = [_h9428,_h9430,_h9432,_h9434,_h9436,_h9438,_h9440,_h9442,
_h9444]

Answer:
?x = [_h9458,_h9460,_h9462,_h9464,_h9466,_h9468,_h9470,_h9472]
```

```
Answer:
?x = [_h9486,_h9488,_h9490,_h9492,_h9494,_h9496,_h9498]

Answer:
?x = [_h9512,_h9514,_h9516,_h9518,_h9520,_h9522]

Answer:
?x = [_h9536,_h9538,_h9540,_h9542,_h9544]

Answer:
?x = [_h9558,_h9560,_h9562,_h9564]

Answer:
?x = [_h9578,_h9580,_h9582]

Answer:
?x = [_h9596,_h9598]

Answer:
?x = [_h9612]

PathLP [17:40] > ?-
```

3.12 Command Line

khitron [14:34] **~>pathlp -new module11 -module module11 -sinit -aall wer/aap wer/mkf -saut -stb module11 -module def wer/mdk -halt > logger.txt**
creates a new dynamic module *module11*, make it current, turn off the system stability check process, turn on batch answers printing, load two files, *~/wer/aap.ppl* and *~/wer/mkf.ppl*, turn on the stability check again, run it for previous files, make the default module be the current one, load there the file *~/wer/mdk.ppl*, stop the application and redirect all the output not including errors to a file *~/logger.txt*.

Appendix A: Background Theories

Typing theories

```
% Subtype implication.
?x!?y[?z] :- nonvar(?z)@_prolog,
             ?x!?y[?t],
             ?t::?z;
% Typing polymorphism.
?x:?y :- ?z::?y,
         ?x:.?z;
% Typing inference.
?x:?y :- typing(?kind, MODNAME),
         ?kind = inference,
         ?z!?t[?y],
         ?u:.?z,
         ?u.?t[?x];
% Subtyping transitivity.
?x::?y :- ?x::?z,
          ?z::?y,
          ?x != ?y;
```

Equality theories

```
?x:=:?x;                 % Reflexivity
?x:=:?y :- ?y:=:?x;      % Symmetry
?x:=:?y :- ?x:=:?z,      % Transitivity
           ?y:=:?z;
```

Appendix B: Math

1. The *pathlp* uses the math mechanism, imported from *XSB Prolog*.
2. You can use as query element the code <*ans*> *is* <*expression*>, where if <*ans*> is a variable to write the answer to, and <*expression*> is (fully) ground math expression.
3. Usage a number as <*ans*> evaluates the expression and checks if the answer fits it.
4. The possible math functions are listed below.
5. *X + Y* is replaced by the sum of *X* and *Y*.
6. *X - Y* is replaced by the differnce of *X* and *Y*.
7. *X * Y* is replaced by the multiplication of *X* and *Y*.
8. *X / Y* is replaced by the division of *X* and *Y*.
9. *X div Y* or *X // Y* are replaced by the integer division of *X* and *Y*.
10. *X* $\vee$ *Y* is replaced by the bitwise and of *X* and *Y*.
11. *X* $\wedge$ *Y* is replaced by the bitwise or of *X* and *Y*. For sure, keep the space after the backslash.
12. *X xor Y* or *X >< Y* is replaced by the bitwise xor of *X* and *Y*.
13. *X >> Y* is replaced by the logical shift right of *X*, *Y* positions.
14. *X << Y* is replaced by the logical shift left of *X*, *Y* positions.
15. *min(X, Y)* is replaced by the minimum of *X* and *Y*.
16. *max(X, Y)* is replaced by the maximum of *X* and *Y*.
17. *ceiling(X)* is replaced by the ceiling of *X*.
18. *floor(X)* is replaced by the floor of *X*.
19. *truncate(X)* is replaced by the truncation of *X*.
20. *round(X)* is replaced by the integer number closest to *X*.
21. *float(X)* is replaced by the float number equal to *X*.
22. *X mod Y* is replaced by the *X* modulo *Y*: $X\ mod\, Y = X - \lfloor X\ /\ Y \rfloor * Y$.
23. *X rem Y* is replaced by the remainder of the division *X* on *Y*: $X\ rem\ Y = X - X\ div\ Y * Y$.
24. *X ^ Y* is replaced by the *X* power *Y*.
25. *X ** Y* is replaced by the *float(X ^ Y)*.
26. *sqrt(X)* is replaced by the square root of *X*.
27. *sign(X)* is replaced by the sign of *X*: -1, 0 or 1.
28. *pi*, *e*, *epsilon* are the numbers: π, e and the difference between 1.0 and the next possible float.
29. Use 0 - X for the opposite number to X.

Appendix C: Mime Extensions

New mime type creation instructions.

1. Run *cd <PathLP_dir>/extensions*.
2. Run *xdg-icon-resource install --size 16 ./pathlp.png pathlp-icon*.
3. Locate the system mime write permitted directory. It is possibly ~**/.local/share/mime** or **/usr/share/mime**. Otherwise try to find a directory **.../share/mime**. Anyway it should conclude subdirectory **packages**.
4. Run *cp ./pathlp-install.xml <mime_dir>/packages*.
5. Run *update-mime-database <mime_dir>*.

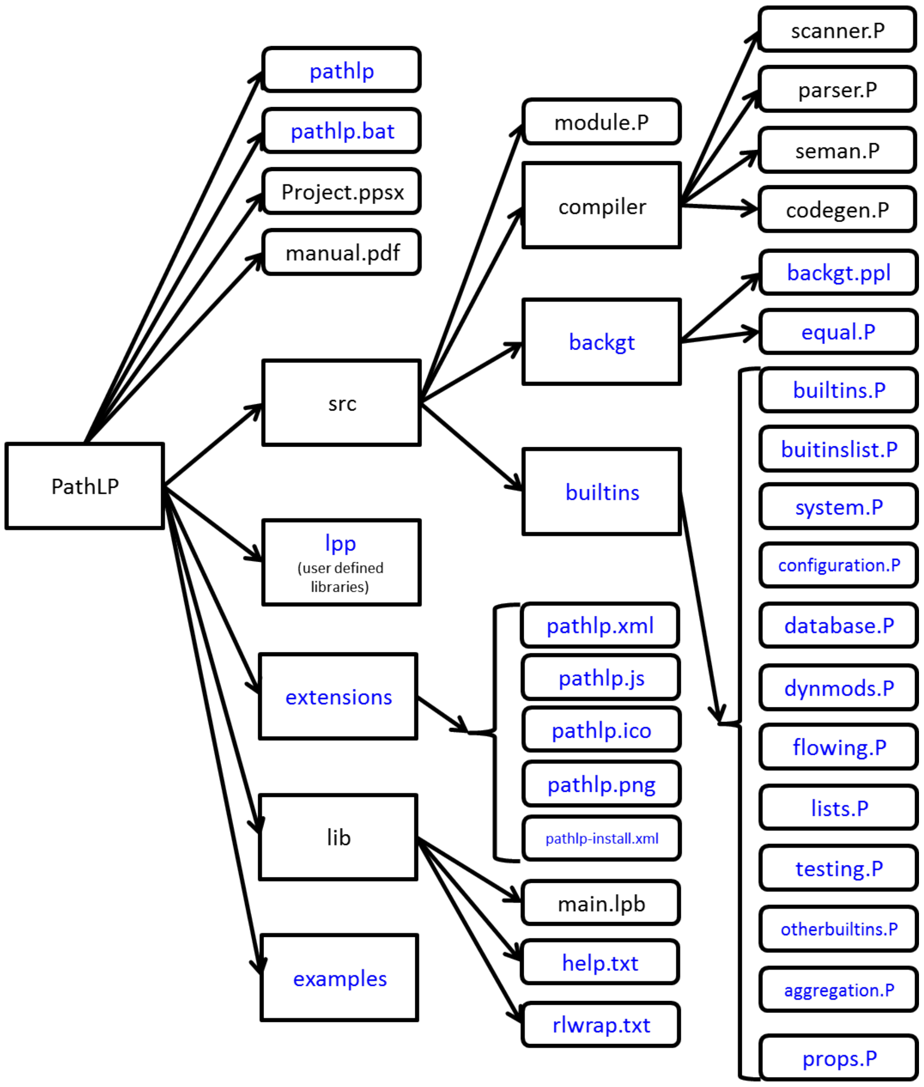
Kate configuration instructions.

1. Locate KDE apps write permitted directory. Try one of directories ~**/.kde*/share/apps** or **/usr/share/kde*/apps**. Otherwise try to find a possible directory **.../share/kde*/apps** or maybe **.../.kde*/share/apps**. ***** means nothing or version number. In `Windows` it is **KDE\share\apps**. Also use there backslash in place of slash in next paragraphs.
2. If **<apps_dir>/katepart** not exists, run *mkdir <apps_dir>/katepart*.
3. If **<apps_dir>/katepart/syntax** not exists, run *mkdir <apps_dir>/katepart/syntax*.
4. If **<apps_dir>/katepart/script** not exists, run *mkdir <apps_dir>/katepart/script*.
5. If **<apps_dir>/katepart/script/indentation** not exists, run *mkdir <apps_dir>/katepart/script/indentation*.
6. Run *cd <PathLP_dir>/extensions*.
7. Run *cp ./pathlp.xml <apps_dir>/katepart/syntax*.
8. Run *cp ./pathlp.js <apps_dir>/katepart/script*.
9. Run *cp ./pathlp.js <apps_dir>/katepart/script/indentation*.
10. (Re)open **Kate** or another **KDE** editor.
11. Choose menu **Settings**->**Configure Kate**.
12. Click **Editor component**->**Open/Save**.
13. Open **Modes & Filetypes** tab.
14. Choose in **Filetype** option **Source/PathLP** (create it in `Windows`).
15. Ensure in **Indentation mode** option **PathLP** and approve.

Indentation rules.

1. In the middle of the code each next line is indented. Usually, for the second character after the main symbol. For example:
   a.b[c] :- d.e[f],
   is indented below the *d*.
   ??- pwriteln(5), is indented below the *p*.
2. If there is no main symbol, default indentation is 8 characters.
3. If you'll change indentation, the result is the maximum between your and system's choice.
4. If you use multiline comments (/* ... */), after pressing /* a space automatically inserted. If you press enter, the next line is started with * (below the one in the previous line) and space. If you'll press / after the space in the new line, as "* /", the space is removed.
5. If new line happens not in the middle of the code, the line is not indented.

Appendix D: Source Structure



Appendix E: Bibliography

[1] Khitron, I., Kifer, M., Balaban, M.: PathLP: A Path-oriented Logic Programming Language. http://pathlp.sf.net (2011)

[2] Balaban, M., Kifer, M.: An Overview of F-OML: An F-Logic Based Object Modeling Language. (In: Workshop on OCL and Textual Modeling, MoDELS 2010)

[3] Balaban, M., Kifer, M.: Logic Based Model-Level Software Development with F-OML. (In: MoDELS 2011)

[4] Kifer, M., Lausen, G.: F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema. (In: SIGMOD International Conference on Management of Data, ACM, Portland, Ore., May 31 - June 2, 1989) pp. 134–146

[5] Warren, D.: XSB: A Logic Programming and Deductive Database System. http://xsb.sf.net (2000)

Index