Jekejeke Minlog Reference

Version 1.0.3, December 24th, 2018

XLOG Technologies GmbH

Jekejeke Prolog

Minimal Logic 1.0.3

Language Reference

Author: XLOG Technologies GmbH
Jan Burse
Freischützgasse 14
8004 Zürich
Switzerland

Date: December 24th, 2018
Version: 0.32

Participants: None

Warranty & Liability

To the extent permitted by applicable law and unless explicitly otherwise agreed upon, XLOG Technologies GmbH makes no warranties regarding the provided information. XLOG Technologies GmbH assumes no liability that any problems might be solved with the information provided by XLOG Technologies GmbH.

Rights & License

Trademarks

Table of Contents

Change History

Jan Burse, June 15th, 2011, 0.1:
- Initial version.

Jan Burse, June 5th, 2012, 0.2:
- Forward chaining examples, syntax and theory section completed.

Jan Burse, July 14th, 2012, 0.3:
- Finite domain examples, syntax and theory section completed.

Jan Burse, October 15th, 2012, 0.4:
- Some fixes and little solver example introduced.

Jan Burse, November 19th, 2012, 0.5:
- Index section, conversation section, tracing revisited and Bonner's examples introduced.

Jan Burse, February 26th, 2013, 0.6:
- Stability analysis removed.

Jan Burse, May 23th, 2013, 0.7:
- Clauses with variable sharing introduced.

Jan Burse, June 6th, 2013, 0.8:
- Disjunction in forward chaining and last string ops introduced.

Jan Burse, August 3rd, 2013, 0.9:
- Forward chaining advising and attribute variables introduced.

Jan Burse, October 2nd, 2013, 0.10:
- Subject to occurs check example introduced.

Jan Burse, April 18th, 2014, 0.11:
- Finite domain solver is not preloaded anymore.

Jan Burse, June 13th, 2014, 0.12:
- Forward closure and chart parsing is not preloaded anymore.

Jan Burse, August 19th, 2014, 0.13:
- Small restructuring of CLP(FD) and back and forth operators.

Jan Burse, February 11th, 2015, 0.14:
- The CLP(FD) examples adapted to new top-level constraint display.

Jan Burse, May 28th, 2015, 0.15:
- Title page introduced and message lists removed.

Jan Burse, August 10th, 2015, 0.16:
- Small extension of CLP(FD) to single infinite search.

Jan Burse, January 4th, 2016, 0.18:
- Small extension of CLP(FD) to use objects.

Jan Burse, February 25th, 2016, 0.19:
- New continuation queue API and when/2 predicate introduced.

Jan Burse, March 7th, 2016, 0.20:
- Better extensibility of the forward chaining and hook integration.

Jan Burse, April 13th, 2016, 0.21:
- New miscellaneous definitions section and new indexes section.

Jan Burse, May 16th, 2016, 0.22:
- New modules regex, rdf and sparql introduced.

Jan Burse, June 23th, 2016, 0.23:
- Delay and undefined error in CLP(FD) bridging.

Jan Burse, November 14th, 2016, 0.24:
- Documentation of {}/1 and CLP(FD) test cases updated.

Jan Burse, December 13th, 2016, 0.25:
- Primitive array support in CLP(FD).

Jan Burse, March 11th, 2017, 0.26:
- Some improvements of attribute variables and trailing.

Jan Burse, May 7th, 2017, 0.27:
- Some addition to elementary arithmetic.

Jan Burse, September 17th, 2017, 0.28:
- Type 1 and Type 2 attributed variables API.

Jan Burse, October 13th, 2017, 0.29:
- New trailed named variables module.

Jan Burse, July 05th, 2018, 0.30:
- Zimmerman Algorithm for integer square root.

Jan Burse, October 19th, 2018, 0.31:
- New module "chr" and module "asp".

Jan Burse, December 24th, 2018, 0.32:
- Randomized labelling introduced.

1 Introduction

This document gives a reference of the Jekejeke Minlog module.

- **Minlog Examples:** We show some examples uses of the Jekejeke Minlog module.
- **Minlog Conversations:** We show how to interact with the Jekejeke Minlog module.
- **Minlog Syntax:** We show what syntax the Jekejeke Minlog module accepts.
- **Minlog Theory:** The Jekejeke Minlog module comes with a set of new predicates.
- **Appendix Example Listing:** The full source code of the Jekejeke Minlog module examples is given.

2 Minlog Examples

We show some examples of the use of the Jekejeke Minlog module. Readers who might be interested in getting a quick grip Jekejeke Minlog module and who have already a basic knowledge of forward chaining might stick to this section only.

- **Bonner's Examples:** Two small examples from the Jekejeke Minlog tool box for hypothetical and counter factual reasoning.

- **Animals Revisited:** Normal Prolog rules can be turned into forward chaining rules by simply declaring the relevant predicates forward. The given example shows a small expert system.

- **Palindrome Revisited:** It is also possible to execute grammar rules in a forward manner. The given example allows detecting palindromes.

- **Money Revisited:** We show how a letter puzzle can be solved with the bundled finite domain constraint solver.

- **Little Solver:** We present a sketch on how a little custom constraint solver can be implemented via forward chaining rules.

- **Type Inference:** We show how a sound type inference algorithm can be implemented in using a subject to occurs check constraint.

2.1 Bonner's Examples

We demonstrate hypothetical and counter factual reasoning as provided by the module "hypo". We essentially replicate the examples from Bonner [3][4]. A student has to take mandatory German and optionally French or Italian courses before he can grade. We can express this as a set of Prolog rules as follows:

```
/* must take german and can choose between french and italian */
grade(S) :- take(S, german), take(S, french).
grade(S) :- take(S, german), take(S, italian).
```

We further assume a database with some facts. These facts express state of affairs for students. The facts need to be either dynamic or thread local, since static facts cannot be modified by the module "hypo". We need to mark them multi-file, so that we can modify them from the top-level as well:

```
/* hans has already taken french */
:- multifile take/2.
:- thread_local take/2.
take(hans, french).
```

Hypothetical reasoning now consists of assuming further facts or rules. The module "hypo" provides the predicates assumea/1 and assumez/1 for this purpose. They are analogous to asserta/1 and assertz/1, except that they only add the given fact or rule temporarily for the duration of the continuation:

```
/* hans would not grade if he takes also italian */
?- assumez(take(hans, italian)), grade(hans).
No

/* hans would grade if he takes also german */
?- assumez(take(hans, german)), grade(hans).
Yes ;
No
```

On the other hand, counter factual reasoning consists of retiring existing facts or rules. The module "hypo" provides the predicates retire/1 and retireall/1 for this purpose. They are analogous to retract/1 and retractall/1, except that they only remove a found fact or rule temporarily for the duration of the continuation: The facts we are working with are now:

```
/* anna has already taken german, french and italian*/
:- multifile take/2.
:- thread_local take/2.
take(anna, german).
take(anna, french).
take(anna, italian).
```

Whereas in hypothetical reasoning a previously failing query turned into a succeeding one, in counterfactual reasoning the converse can be observed. A previously succeeding query can turn into a succeeding one. This monotonicity holds if the program does not contain negation as failure or other non-monotonic constructs:

```
/* anna would grade if she would not have taken italian */
?- retire(take(anna, italian)), grade(anna).
Yes ;
No

/* anna would not grade if she would not have taken german */
?- retire(take2(anna, german)), grade(anna).
No
```

It is possible to use the toolbox efficiently for more elaborate applications such as adversarial planning [5]. Also note that in practical applications often the continuation variant [6] of hypothetical and counter factual reasoning is used. These variants are also provide by the module "hypo". We refer the interested reader to the module documentation.

2.2 Animals Revisited

We demonstrate forward chaining as provided by the module "delta". As an example, we take a small expert system for animals. This is the same example as from the runtime library where an ordinary Prolog backward chaining solution is given. As already explained in the previous section, the multi-file directive helps us modifying facts from the top-level:

```
:- multifile motion/1, skin/1, diet/1.
:- thread_local motion/1, skin/1, class/1, diet/1, animal/1.
```

Ordinary Prolog backward chaining uses rules of the form "Goal :- Sub-Goals". They can be entered with the help of the operator (:-)/2. Forward chaining uses rules of the form "Action <= Condition". The operator (<=)/2 is provided by the module "delta". The module "delta" also provides the post/1 action and the posted/1 condition:

```
post(class(mamal)) <= posted(motion(walk)), posted(skin(fur)).
post(class(fish)) <= posted(motion(swim)), posted(skin(scale)).
post(class(bird)) <= posted(motion(fly)), posted(skin(feather)).

post(animal(rodent)) <= posted(class(mamal)), posted(diet(plant)).
post(animal(cat)) <= posted(class(mamal)), posted(diet(meat)).
post(animal(salmon)) <= posted(class(fish)), posted(diet(meat)).
post(animal(eagle)) <= posted(class(bird)), posted(diet(meat)).
```

Ordinary Prolog backward chaining is invoked by sending a goal to the Prolog interpreter. The Prolog interpreter then tries to find answer substitutions in a top-down fashion. The module "delta" provides a forward chaining engine, which is invoked by sending a goal to it. The forward chaining engine then tries to find answer sets in a bottom-up fashion.

```
write('The animal is '), write(X), nl <= posted(animal(X)).
```

We are interested in an answer for the animal/1 predicate. We therefore added the above forward chaining rule as well to the Prolog text, so that when an animal fact arrives, a message is written. The module "delta" allows such a forward chaining rule, since an action can be an arbitrary Prolog goal. Here are some results:

```
?- post(motion(walk)), post(skin(fur)).
Yes

?- post(motion(walk)), post(skin(fur)), post(diet(meat)).
The animal is cat
Yes

?- post(motion(walk)), post(skin(fur)), post(diet(plant)).
The animal is rodent
Yes
```

The module delta not only delivers hypothetical forward chaining but also counter factual forward chaining, which we did not demonstrate here. The condition phaseout/1 and phaseout_posted/1 can be used to temporarily remove facts and rules during forward chaining. We refer the interested reader to the module documentation.

2.3 Palindrome Revisited

It is also possible to execute grammar rules in a forward manner by a chart parser. To be able to use the chart parser in a Prolog text one has first to load the chart library. Loading the chart library will install a term expansion that will convert chart rules first into forward chaining rules and then into ordinary Prolog delta rules. Loading the chart library requires the Jekejeke Minlog capability and is done as follows:

```
:- use_module(library(minimal/chart)).
```

The example is an adaption of the example already found in the language reference of the Jekejeke Prolog runtime library. It allows detecting palindromes. The way forward grammar rules are currently conceived it is not anymore possible to use them to generate text. It is only possible to detect text. In ordinary DCG rules text is passed around in the logical variables of the non-terminals. In chart DCG rules the text will be represented as facts for the predicate ‘D’/3. For the text “bert” the following facts will be temporarily added:

```
'D'(97, 3, 4).
'D'(110, 2, 3).
'D'(110, 1, 2).
'D'(97, 0, 1).
```

The facts will be supplied to the forward chaining component in the above order, from the last text element to the first text element, right-to-left. This is a little bit unusual but has certain advantages. First since the order is fixed it is possible to check for arrived facts only in the first literal of a forward grammar rule. This reduces the number of generated delta rules and speeds up the execution of the parser. Second it is possible to use auxiliary conditions as known from normal definite clause grammars since the goals will be executed left-to-right as soon as the left-most fact arrives.

But we have to see to it that the non-terminals generate ground attributes, since non-ground attributes would currently not yet be correctly preserved by the forward chaining component. This bottom-up condition is not a problem for the palindrome example. The terminal predicate 'D'/3 needs to be declared locally, so that the generated code finds locally some declaration for. Further we need a forward/1 declaration for those non-terminals that do not occur as some first non-terminal of a DCG chart rule. The DCG chart rules itself are denoted by the operator (==:)/2:

```
:- multifile 'D'/3.
:- thread_local 'D'/3, palin/4.
:- forward palin/5.

palin([], [Middle]) ==:
   [Middle].
palin([Middle], []) ==:
   [Middle, Middle].
palin([Border | List], Middle) ==:
   [Border], palin(List, Middle), [Border].
```

The words/3 predicate or alternatively the words/4 predicate from the module chart can be used to start the chart parser. These commands will add the ‘D’/3 facts and set the forward chaining engine in motion, so that these commands will already compute the forward chaining closure. The commands will do so in a temporary manner:

```
?- words("bert", 0, _), listing('D'/3).
:- thread_local 'D'/3.
'D'(116, 3, 4).
'D'(114, 2, 3).
'D'(101, 1, 2).
'D'(98, 0, 1).
```

The chart/3 construct can then be used to query a non-terminal. The goal expansion will already expand such a construct at compile time, so that as long as the first argument of chart/3 is not a variable the end-user doesn’t has to worry about performance. We can detect the same text as with the normal definite clause grammars. Both the positive and the negative example do their job also in forward chaining:

```
?- words("racecar", 0, N), chart(palin(X,Y), 0, N).
X = [114,97,99],
Y = [101]

?- words("bert", 0, N), chart(palin(X,Y), 0, N).
No
```

Forward grammar rules will loop when there are productions with a single goal in the body that form a cycle. On the other hand if this cycle condition is not met forward grammar rules can easily deal with left recursion and they can also easily identify parse chunks.

2.4 Money Revisited

We show how a letter puzzle can be solved with the bundled finite domain constraint solver. The letter puzzle is already found in the language reference of the Jekejeke Prolog runtime library. There the puzzle was solved via the method of generate and test. This method uses goals G_1, .., G_n to enumerate exemplars of a collection and goals T_1,..,T_m to verify a condition. The problem is then solved via backtracking over the following query:

```
G1, .., Gn, T1, .., Tm
```

For constraint solving the problem is posed the same way. But the constraint solver will not perform any backtracking during the setup phase. Only in the solving phase the constraint solver will then intelligently pick and rearrange the goals. The heuristic search component is responsible for picking the first generator G_i. The forward checking component will then cross out relevant tests T_j, .., T_k:

```
Gi, Tj, .., Tk
```

If a test fails backtracking is issued until the recent generator is exhausted. The process might then even backtrack to previous generators but later pick again totally different generators. If all tests succeed a further generator and tests are picked. The search succeeds when all generators and relevant tests have been processed.

Let's turn to the letter puzzle. To be able to use the finite domain constraint solver parser in a Prolog text one has first to load the clpfd library. Loading the clpfd library will define the predicates and syntax operators of the finite domain constraint solver. Loading the clpfd library requires the Jekejeke Minlog capability and is done as follows:

```
:- use_module(library(finite/clpfd)).
```

Since our finite domain constraint solver allows the use native Prolog variables we can use these to represent individual letters. The part that generates the permutations will therefore read as follows. The predicate ins/2 states that the letters will have values in the range of 0..9. The next predicate all_different/1 then states that the letters should be distinct:

```
puzzle(X) :-
   X = [S,E,N,D,M,O,R,Y],
   X ins 0..9,
   all_different(X),
```

The code then continues by phrasing the condition that needs to be verified. These conditions read as in the original example except that the Prolog relations (=:=)/2 and (=/=)/2 are replaced by the relations (#=)/2 and (#\=)/2 of the finite domain constraint solver.

```
   M #\= 0,
   S #\= 0,
             1000*S + 100*E + 10*N + D +
             1000*M + 100*O + 10*R + E #=
   10000*M + 1000*O + 100*N + 10*E + Y,
```

The predicate label/1 will then start the constraint There is no need to retrieve the variable values via the predicate indomains/2 since labelling will only succeed with all the given variables instantiated to a constant.

```
   label(X).
```

The source code of the puzzle/1 predicate is found in the appendix. It can be ensure loaded from the top-level. To then call the puzzle/1 predicate from the top-level we do not need to load the clpfd library again, since querying the puzzle/1 doesn't require any predicates or syntax operators from the finite domain constraint solver:

```
?- puzzle(Z).
Z = [9,5,6,7,1,0,8,2] ;
No
```

For certain problems the constraint solver can considerably reduce the search space and thus perform much quicker than the standard backtracking solution. For the letter puzzle the standard Jekejeke Prolog backtracking solution takes around 3000ms, whereas the Jekejeke Minlog constraint solving solution only needs around 30ms in the average.

2.5 Little Solver

We present a sketch on how a little custom constraint solver can be implemented via forward chaining rules. The basic idea of the sketch can be used as a boiler plate to implement various constraint solvers. But more able and efficient solvers probably need additional ideas. Our little solver will allow elementary reasoning for the following constraints:

$X = c$,	c is a number
$X \in D$,	D is a finite set of numbers

Our little solver will use atoms to represent constraint variables. Many existing Prolog systems use Prolog variables to represent constraint variables. We might also do so in the future, but atoms are currently easier to deal with in forward chaining rules. We will represent the constraints by forward facts. The forward chaining rules will implement the desired elementary reasoning.

We start with the rules for the X = c constraints. The constraint will be represented by bound/2 forward facts. We find only two rules:

$X = c, X = c$	$\rightarrow$	$X = c$	
$X = c, X = d$	$\rightarrow$	$\bot$,	$c \neq d$

To implement these rules we will need to draw upon all features of our forward chaining engine. In particular we will make use of the condition annotations posted/1, phaseout/1 and phaseout_posted/1. We will also make use of the actions true/0 and fail/0. In the first rule we need an arrival and delete phaseout_posted/1 annotation since two equalities are replaced by a single equality. In the second rule we need a fail since the result is bottom. We also make use of the cut (!):

```
:- multifile bound/2.
:- thread_local bound/2.

true <=
   phaseout_posted(bound(X, C)), bound(X, C), !.
fail <=
   posted(bound(X, _)), bound(X, _).
```

Let's turn to the rules for the $X \in D$ constraints. The constraint will be represented by domain/2 forward facts. We find rules that transform a domain/2, that combine a domain/2 with a bound/2 and that combine a domain/2 with another domain/2:

$X \in \{\}$	$\rightarrow$	$\bot$	
$X \in \{c\}$	$\rightarrow$	$X = c$	
$X \in D, X = c$	$\rightarrow$	$X = c$,	$c \in D$
$X \in D, X = c$	$\rightarrow$	$\bot$,	$c \notin D$
$X \in D, X \in E$	$\rightarrow$	$X \in D \cap E$	

The above rules need additional computations. The rules that combine a domain/2 and a bound/2 will make use of a member/2 predicate to check the member ship (∈). The rules that combine a domain/2 with another domain/2 will make use of an intersection/2 predicate to compute the set intersection (∩). The implementation of these predicates is drawn from some standard modules. The forward rules read as follows:

```
:- multifile domain/2.
:- thread_local domain/2.

fail <=
   posted(domain(_, [])), !.
post(bound(X, C)) <=
   phaseout_posted(domain(X, [C])), !.
true <=
   phaseout_posted(domain(X, D)), posted(bound(X, C)),
   member(C, D), !.
fail <=
   posted(domain(X, _)), posted(bound(X, _)).
post(domain(X, F)) <=
   phaseout_posted(domain(X, D)), phaseout(domain(X, E)),
   intersection(D, E, F).
```

The forward store and thus constraint store can be inspected with the predicate listing/1. The forward chaining can be triggered via the predicate post/1. Let's run some tests:

```
?- post(domain(x, [1])), listing(bound/2), listing(domain/2).
bound(x, 1).

?- post(domain(x, [1,2,3])), post(domain(y, [2,3,4])),
   listing(bound/2), listing(domain/2).
domain(x, [1,2,3]).
domain(y, [2,3,4]).

?- post(domain(x, [1,2,3])), post(domain(x, [2,3,4])),
   listing(bound/2), listing(domain/2).
domain(x, [2,3]).

?- post(domain(x, [1,2,3])), post(bound(x,4)),
   listing(bound/2), listing(domain/2).
No
```

2.6 Type Inference

We show how a sound type inference algorithm can be implemented in using a subject to occurs check constraint. The occurs-check is a check that is needed for a sound unification. It verifies that a variable is not bound to a term which contains the variable itself. Since this check is costly it is often omitted from Prolog systems. As result a Prolog system which omits the occurs-check might produce wrong results.

Consider the following problem of inferring a simple type for a lambda expression. Assume lambda expressions are built from applications app/2, abstractions lam/1 and variables. Simple types are either variables or functions spaces ->/2. Consider contexts $T_1 : X_1, .., T_n : X_n$ that enumerate the variables of a lambda expression and their types. Let Γ |- T : E denote that the lambda expression E provides context Γ and has the simple type T. We can readily write down the following inference rules:

```
----------------- (Id)
A : X |- A : X

Γ |- A -> B : S      Δ |- A : T                Γ, A : X |- B : T
---------------------------------- (MP)        -------------------------- (->R)
     Γ, Δ |- B : app(S,T)                      Γ |- A -> B : lam(X,T)
```

It is straight forward to produce a predicate typed/3 that takes a context Γ, a lambda expression E and then derives the type T if a smaller context Δ subset Γ exists such that Δ |- T : E. Unfortunately without a sound unification, we might assign types to lambda expressions we do not want them to have a simple type. For example we might get an erroneous affirmative for the self-application app(F,F):

```
/* SWI-Prolog */
?- typed(app(F,F), [F-A], B).
A = (A->B).
```

Or it might even happen that the Prolog system crashes:

```
/* Jekejeke Prolog */
?- typed(app(F,F), [F-A], B).
java.lang.StackOverflowError
```

There are a number of options to avoid either behaviour. Some Prolog systems provide a global flag that switches on the occurs-check for all unifications. Other Prolog systems implement the ISO predicate unify_with_occurs_check/2. Both approaches have their disadvantage. The former might slowdown unifications that don't need the occurs-check, and the latter might need some cumbersome code rewriting. We present an alternative here by means of the Jekejeke Minlog attribute variables.

The idea is to use the sto/1 constraint that allows imposing an occurs check on the supplied variables. The sto/1 constraint is provided as part of the module herbrand. The detailed source code of this module can be found in the open source code corner of the Jekejeke Minlog web site. The sto/1 constraint is implemented with the help of attribute variables and eagerly checks the condition before the attribute variable is even instantiated. Here are some example uses of the sto/1 constraint:

```
?- use_module(library(term/herbrand)).

?- sto(X), X=f(Y).
X = f(Y),
sto(Y)

?- sto(X), X=f(Y), Y=g(X).
No
```

The sto/1 constraint is now deployed like a variable declaration. Whenever one finds a fresh variable that needs an occurs check the sto/1 constraint should be used. In the case of our typed/3 predicate we only find the (MP) rule that needs an additional sto/1 constraint. The (MP) rule is non-analytic it introduces a new formula A in backward chaining. We therefore define a revised typed2/3 predicate with the following modified clause:

```
typed2(app(E,F), C, T) :-
   sto(S),
   typed2(E, C, (S->T)),
   typed2(F, C, S).
```

The sto/1 constraint needs also to be deployed for Prolog queries in the top-level. The general rule is the same here as for clauses. Whenever one finds a fresh variable that needs an occurs check the sto/1 constraint should be used. It is important that the sto/1 constraint is used before the argument receives a cyclic term. Since Jekejeke Prolog doesn't support cyclic terms, the sto/1 constraint will hang if it receives an already cyclic term.

Here are some examples for an accordingly modified predicate typed2/3:

```
?- sto((A,B,C)), typed2(app(E,F), [E-A,F-B], C).
A = (B->C),
sto(C),
sto(B)

?- sto((A,B)), typed2(app(F,F), [F-A], B).
No

?- sto(A), typed2(lam(X,lam(Y,app(Y,X))), [], A).
A = (_A->(_A->_B)->_B),
sto(_B),
sto(_A)
```

The predicate sto/1 might be a viable alternative to the other two approaches of introducing an occurs check into a Prolog text. The sto/1 predicate has the advantage that it can be selectively applied to variables where necessary. On the other hand it should be noted that the sto/1 predicate introduces an additional memory footprint in the form of allocated attribute variables and hooks.

3 Minlog Conversations

When used in connection with the Jekejeke Prolog development environment, the Jekejeke Minlog module will as well provide some interaction.

- **Backward Debugging:** Backward chaining shows a top-down pattern. We recapitulate the tracing of ordinary Prolog interpreter execution.

- **Forward Debugging:** Forward chaining shows a bottom up pattern. We highlight in which way our forward chaining engine execution is traced.

- **Chart Debugging:** t.b.d.

- **Finite Debugging:** t.b.d.

3.1 Backward Debugging

Ordinary Prolog interpreter execution works with backward chaining goal search. In this execution schema a rule is read as "Goal :- Sub-goals". A common debugging model are Byrd Boxes named after Lawrence Byrd. Our debugger framework from the development environment supports this model with a few extensions and adaptions.

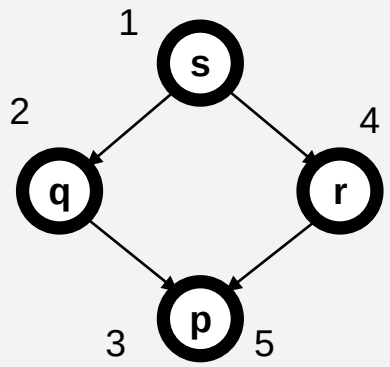
The Byrd Boxes model features the ports "call", "exit", "redo" and "fail". Among the extensions in our debugger framework, we find the additional ports "head" and "chop" which are not visible by default. Further, deterministic execution of a goal is detected and then the ports "redo" and "fail" of that goal become invisible as well.

Backward chaining has a distinct top-down pattern from goals to sub-goals. We will first demonstrate this pattern and later compare it with the pattern of our forward chaining engine. As an example problem, we will use a propositional problem, which can be represented in Prolog. The problem consists of the following Prolog facts, rules and query:

```
p.
r :- p.
q :- p.
s :- q, r.
?- s.
Yes
```

When we run the query in our default debugger, we get the following trace. We have also depicted the dependency graph among the facts, rules and query. The numbering in the dependency graph shows the order of the Byrd Box calls during tracing. Since the dependency shows sub-goals further down, the tracing also happens in a top-down fashion:

```
?- trace.
Yes.
?- s.
    0 Call s ?
    1 Call q ?
    2 Call p ?
    2 Exit p ?
    1 Exit q ?
    1 Call r ?
    2 Call p ?
    2 Exit p ?
    1 Exit r ?
    0 Exit s ?
```



The debugger framework and the default debugger is not part of the Minlog module. The debugger allows issuing debugger directives by mouse and keyboard. For further information about the debugger and the available directives, please consult the language reference manual of the development environment.

3.2 Forward Debugging

We now turn our attention to the forward chaining engine. The forward chaining engine allows forward chaining rules in the form "New-facts <= Facts". We also use the Byrd Box model to debug forward chaining rules. A call to the forward chaining instruction post/1 will indicate that a new fact is to be added to the forward store.

The forward chaining instruction post/1 will in turn check the forward chaining rules for new facts and put them on an agenda. These checks might involve some ordinary backward chaining calls, which will then be seen by ordinary Byrd Box calls. The forward chaining instruction post/1 itself will only exit when the agenda has been worked off.

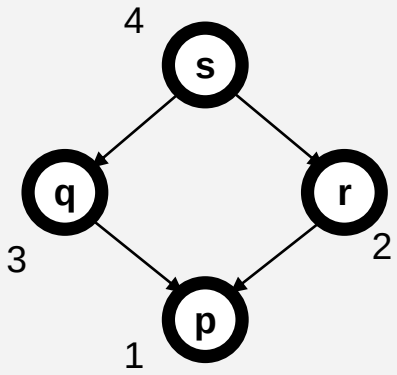
Forward chaining has a distinct bottom-up pattern from facts to new-facts. We will now demonstrate this pattern. As an example problem, we will use the same propositional problem again. This time we will use forward chaining rules to represent the problem. We will not use a fact but post a fact as port of the query to start the forward chaining engine:

```
:- use_module(library(minimal/delta)).
:- thread_local s/0, q/0, r/0, p/0.
:- forward s/1.

post(r) <= posted(p).
post(q) <= posted(p).
post(s) <= posted(q), posted(r).
?- post(p), s.
Yes
```

When we run the forward chaining query in our default debugger, we get the following trace. We have also again depicted the dependency graph among the facts, rules and query. The numbering in the dependency graph now shows the order of the Byrd Box for the calls of the forward chaining instruction post/1 during tracing.

```
?- trace.
Yes
?- post(p), s.
    0 Call post(p) ?
    1 Call post(r) ?
    2 Call q ?
    2 Fail q ?
    1 Exit post(r) ?
    1 Call post(q) ?
    2 Call r ?
    2 Exit r ?
    2 Call post(s) ?
    2 Exit post(s) ?
    1 Exit post(q) ?
    0 Exit post(p) ?
    0 Call s ?
    0 Exit s ?
```



Since the dependency shows sub-goals and therefore facts further down, the tracing happens this time in a bottom-up fashion. Otherwise, default debugger offers the same debugging as for ordinary backward chaining. It is possible to set spy points or break points and it is possible to have different debug modes.

3.3 Chart Debugging

t.b.d.

3.4 Finite Debugging

t.b.d.

4 Minlog Syntax

The Jekejeke Minlog module extends the syntax of Prolog texts by forward clauses and chart rules. We show what additional syntax the Jekejeke Minlog module accepts.

- **Minimal Logic:** The Jekejeke Minlog modules enhance the syntax ordinary Prolog texts and ordinary Prolog sessions.
- **Miscellaneous Definitions:** The interpreter keeps track of flags and properties.

4.1 Miscellaneous Definitions

The interpreter also needs to keep track of flags and properties definitions. The following flags and properties are provided by the Jekejeke Prolog Minimal Logic extension:

- **Prolog Flags:** The predefined Prolog flags.
- **Predicate Properties:** The predefined predicate properties.
- **Source Properties:** The predefined source properties.
- **Atom Properties:** The predefined atom properties.

Prolog Flags

Prolog flags can be accessed via the system predicates current_prolog_flag/2 and set_prolog_flag/2. The following Prolog flags are supported by the Jekejeke Prolog Minimal Logic extension:

Predicate Properties

Predicate properties can be accessed via the system predicates predicate_property/2, set_predicate_property/2 and reset_predicate_property/2. The following predicate properties are supported by the Jekejeke Prolog Minimal Logic extension:

cosmetic: See the module delta section.

Source Properties

Source properties can be accessed via the system predicates source_property/2, set_source_property/2 and reset_source_property/2. The following source properties are supported by the Jekejeke Prolog Minimal Logic extension:

Atom Properties

Atom properties can be accessed via the system predicates atom_property/2 and updated copies of atoms can be obtained via the predicates set_atom_property/3 and reset_atom_property/3. The following atom properties are supported by the Jekejeke Prolog Minimal Logic extension:

5 Minlog Extension Packages

The Jekejeke Minlog module comes with a set of new predicates. Predicates can be grouped into theories and we present them as such:

- **Minimal Logic Package:** This theory groups the predicates that extend the expressiveness of normal Prolog.
- **Term Domain Package:** This theory groups the predicates for constraint solvers over term domains.
- **Finite Domain Package:** This theory groups the predicates of the finite domain constraint solver.
- **Miscellaneous Package:** This theory is concerned with miscellaneous predicates.
- **System Package:** This theory is concerned with accessing the Prolog system.

5.1 Minimal Logic Package

This theory groups the predicates that extend the expressiveness of ordinary Prolog. We find the following topics:

- **Module assume:** This module provides an assumption toolbox for clause references and variable hooks.
- **Module hypo:** This module provides a couple of connectives for hypothetical reasoning.
- **Module delta**: This module provides a couple of directives and predicates to control the forward chaining component.
- **Module chart**: This module also allows executing definite clause grammars (DCGs) in a forward manner.
- **Module chr:** This module provides logical constraint handling rules (CHR) via forward chaining rules.
- **Module asp:** This module provides an answer set programming (ASP) choice operator for satisfiability search.

Module assume

Clauses and attribute variable hooks are identified by their reference data type. The predicates deposita_ref/1 respectively depositz_ref/1 will assume the given clause or hook for the duration of the continuation, whereas the predicate withdrawa_ref/1 respectively withdrawz_ref/1 will retire the given clause or hook for the duration of the continuation.

Example:

```
hook(V, T) :-
   write('bind '), write(V),
   write(' to '), write(T), nl.

?- sys_ensure_hook(X, hook), X = 99.
bind _A to 99
X = 99.
```

The predicate sys_ensure_hook/2 takes care of the compilation of a hook into a hook reference in case the hook is not yet associated with an attribute variable for the duration of the continuation. The variant sys_ensure_hook/3 does the same job during the execution of the given goal.

The predicate sys_assume_cont/1 temporarily pushes the given goal on the continuation queue. For more information on the continuation queue see the module cont.

The following assumption toolbox predicates are provided:

deposita_ref(R):
: The predicate temporarily inserts the clause or hook referenced by R at the top for the duration of the continuation.

depositz_ref(R):
: The predicate temporarily inserts the clause or hook referenced by R at the bottom for the duration of the continuation.

withdrawa_ref(R):
: The predicate temporarily removes the clause or hook referenced by R for the duration of the continuation. The undo will happen at the top.

withdrawz_ref(R):
: The predicate temporarily removes the clause or hook referenced by R for the duration of the continuation. The undo will happen at the bottom.

sys_ensure_hook(A, H, G):
: The predicate temporarily ensures that the hook H is in the hook list of the attribute variable A for the duration of the goal G and succeeds whenever G succeeds.

sys_ensure_hook(A, H):
: The predicate temporarily ensures that the hook H is in the hook list of the attribute variable A for the duration of the continuation.

sys_assume_cont(G):
: The predicate temporarily pushes the goal G on the continuation queue.

Module hypo

This module provides a couple of primitives for hypothetical reasoning. The primitives come in two flavours. The first flavour is the continuation variant where the effect of the primitive persists for the continuation. The second flavour is the temporary variant where the effect of the primitive only persist for the duration of a given goal.

Syntax:

```
<verb>(<arguments>)                % Continuation Variant
<verb>(<arguments>, <goal>)        % Temporary Variant
```

Since the temporary variant uses an additional goal argument, given the continuation variant the temporary variant can be easily invoked with the help of call/2. The clause primitives are implemented with the help of clause references and the module assume. The other primitives extend some logical meta-predicates to the temporary variant.

Examples:

```
?- assumez(p), p.
Yes
?- assumez(p), retire(p), p.
No
```

The continuation variants of true/1, fail/1, (',')/3 and (;)/3 are already defined in the runtime library. By default the assume predicate will create a thread local predicate, if the predicate of the head of the given clause was not yet defined. Further the retire predicate will silently fail if the predicate of the head of the given clause was not yet defined.

The following hypothetical reasoning predicates are provided:

true(G):
The construct does nothing before further solving.

fail(G):
The construct prevents further solving.

assumea(C):
assumea(C, G):
The construct assumes the clause C at the top before further solving.

assumez(C):
assumez(C, G):
The construct assumes the clause C at the bottom before further solving.

retire(C):
retire(C, G):
The construct retires the clause C before further solving. Need not preserve the input order.

retireall(H):
retireall(H, G):
The construct retires all the clauses with head H before further solving. Need not preserve the input order.

','(A, B, G):
The construct does A and then B before further solving.

;(A, B, G):
The construct does A before further solving or it does B before further solving.

Module delta

This module provides a couple of operators and predicates to define forward chaining clauses. A forward chaining clause is recognized by the (<=)/2 operator. A forward chaining clause has the form "Action <= Condition" where the action part can be arbitrary Prolog goals. A forward chaining clause will be rewritten into multiple delta computation rules.

Syntax:

```
Action <= Condition     % Forward Chaining Clause.
```

The module provides a new hypothetical reasoning verbs post/1. This verb first invokes the delta computation rules to determine a new agenda, then uses assumez/1 to assume the given fact and finally continues with the new agenda. By delete set inclusion the delta computation can also yield counter factual reasoning.

Example:

```
:- forward q/2.
post(q(X)) <= posted(p(X)).

?- post(p(a)), q(X).
X = a
```

The delta computation has functor F/N+1 for an arriving fact with a functor F/N. Each literal in the body of a forward chaining clause has to be annotated either for delete set inclusion phaseout/1, for delta computation posted/1 or for both phaseout_posted/1. Non-annotated literals are condition goals that are executed in backward chaining fashion.

The following delta computation predicates are provided:

forward P, ...:
The predicate sets the predicate P to discontiguous, sys_notrace and static.

post(F):

post(F, G):
The fact F is not directly assumed. Instead, it is first treated as a new fact for delta computation.

The following forward chaining condition goals are recognized:

phaseout(F):
The condition succeeds when F is an old fact and it then gets included for deletion.

posted(F):
The condition reacts if F is a new fact.

phaseout_posted(F):
The condition reacts if F is a new fact and it then gets included for deletion.

A, B:
The condition reacts when A or B react, and the other succeeds, or both react.

A; B:
The condition reacts when A or B react.

P:
Whenever P succeeds in backward chaining the condition succeeds.

The following forward chaining rule term expansion is recognized:

A <= C:

The construct defines a forward chaining clause with action A and condition C. The forward chaining clause is rewritten into delta computations.

Module chart

This Jekejeke Minlog module allows executing definite clause grammars(DCGs) in a forward manner. The ordinary DCG rules of Jekejeke Prolog are not suitable for this purpose since they model terminals by [T|O] = I. We therefore provide special chart DCG rules as part of which model terminals by ‘D’(T,I,O). Chart DCG rules are identified by the (==:)/2 operator:

Syntax:

```
P ==: Q.      % chart DCG rule
```

Chart DCG rules do currently not allow for push backs. The term expansion augments the head and body by two additional parameters that are to represent the sentence position before and after the parsing. A predicate identifier p/n will thus be turned into a predicate identifier p/n+2. Further the DCG chart operator (==:)/2 is replaced by the forward chaining operator (<=)/2:

Translation:

```
chart_post(P, I, O) <= chart_posted(Q, I, O).
```

The expansion will then go to work and tackle the head and the body. Compared to ordinary DCG rules, the chart DCG rules support fewer constructs. For example we do not yet support the conditional (->)/2 and the higher order calls call/n. On the other hand the look-ahead negation (\+) is already supported.

Let us consider the following example chart DCG rule:

Example:

```
p(X) ==: "a", q(X), {r(X)}.  % chart DCG rule
```

As an intermediate results the chart DCG rule will be turned into:

Result:

```
post(p(X, I, O)) <= posted('D'(97, I, H)), q(X, H, O), r(X)
```

The above rule will then be turned into a delta computation for the predicate ’D’/3. In general only the first literal of a DCG chart rule will be translated into a delta computation rule, improving efficiency. The words/3 construct can be used to generate the ‘D’/3 facts and the chart/3 construct can be used to query the result of parsing. See the palindrom example for more details.

The following chart DCG predicates are provided:

words([A_1, ..., A_n], I, O):
words([A_1, ..., A_n], I, O, G):
Post the words A_1,..,A_n from index I to index O before further solving.
chart(A, I, O):
Succeeds when there is a phrase A from index I to index O.

The following chart DCG goal expansions are provided:

P:
The non-terminal P is checked.
fail:
The grammar fails.
A, B:
The output of A is conjoined with the input of B.
A; B:
The grammar succeeds when A succeeds or when B succeeds.
[A_1, ..., A_n]:
The terminals A_1, ..., A_n are checked.
!:
The choice points are removed.
{A}:
The auxiliary condition is checked.
\+ A:
The negation of A is checked. The output of A is left loose.

The following chart DCG term expansions are provided:

H ==: B:
Chart DCG rule with chart head H and chart body B.

Module chr

This module provides constraint handling rules rewriting to forward chaining rules from the module "delta". The following rule format is provided. The vertical bar ('|')/2 is according to the ISO core standard. The guard G can be omitted.

Syntax:

```
H ==> G | B.          % Propagation Rule
H \ J <=> G | B.      % Simpagation Rule
J <=> G | B.          % Simplification Rule
```

During translation what is called CHR head H respectively J becomes forward chaining body, and what is called CHR body B becomes forward chaining head. The translation of the above rules reads as follows:

Translation:

```
post(B) <= posted(H), G.
post(B) <= posted(H), phaseout_posted(J), G.
post(B) <= phaseout_posted(J), G.
```

A current restriction is that the resulting forward chaining rules should produce ground facts. Further, the semantics is logical and not chronological as in the usual CHR implementations. Our implementation is not based on attribute variables as in [9].

The following constraint handling term expansion is provided:

H ==> B:
Propagation rules.
J <=> B:
Simpagation and simplification rules.

Module asp

This module provides some choice operators for answer set programming via forward chaining [10]. Since we use the (:-)/2 operator already for backward chaining rules, the operator (<=)/2 form the module "delta" needs to be used to write answer set programming rules. Take this example:

Example:

```
:- p, q, r.
{p,r}.
{q} :- p.
{r} :- p.
```

This would need to be written as follows. For answer set programming constraints a forward chaining rule with a fail action is suggested. Further answer set programming ordinary facts and disjunctive facts need a start condition such as "init", which can then be used to produce an anwser set:

Example:

```
fail <= posted('p'), posted('q'), posted('r').
choose(['p','r']) <= posted(init).
choose(['q']) <= posted('p').
choose(['r']) <= posted('p').
?- post(init).
```

Our approach to answer set programming allows explicit mixing of forward chaining and backward chaining in that a forward chaining rule allows backward chaining goals in its condition part. The only predicate that this module currently provides is a choose/1 operator that allows satisfiability search.

The following answer set programming predicates are provided:

choose(L):
choose(L, G):

If a positive literal from L is already satisfied, the construct does nothing before further solving. Otherwise, the construct posts each positive literal from L via backtracking before further solving.

5.2 Term Domain Package

This theory groups the predicates for constraint solvers over term domains. We find the following topics:

- **Module herbrand:** This module provides Herbrand constraints.
- **Module suspend:** This module provides the delay of goals.
- **Module unify:** Type 1 attributed variables interface.
- **Module verify:** Type 2 attributed variables interface.
- **Module state:** This module provides trailed named values.

Module herbrand

This Jekejeke Minlog module provides a subject to occurs check constraint sto/1 and a term inequality constraint neq/2. To allow good performance both constraints perform an attribute variable verification before their attribute variables are instantiated and new sub constraints are registered:

Example:

```
?- sto(X), X = f(X).
No
?- neq(f(X,X), f(Y,Z)), X = Y.
Y = X,
neq(X, Z)
```

The subject to occurs check has to be initially called with an acyclic term, but it will subsequently assure the subject to occurs check as required. There is no interaction of the subject to occurs check with the term inequality, and the term inequality neither decided for or against the occurs check..

The following herbrand predicates are provided.

sto(T):

The predicate continues the subject to occurs check for the term T. The term T has to be acyclic when calling this predicate.

neq(S, T):

The predicate checks S and T for inequality and establishes variable constraints, so that this inequality is maintained in the continuation. The inequality neither decides for or against the occurs check.

Module suspend

This Jekejeke Minlog module provides the delay of goals until certain variable conditions are satisfied. The predicate freeze/2 delays until the first argument is instantiated. The predicate when/2 delays until the first argument succeeds.

Example:

```
?- freeze(X, (write(foo), nl)).
freeze(X, (write(foo), nl))
?- freeze(X, (write(foo), nl)), X = a.
foo
X = a
```

The delayed goal is allowed to fail or to succeed multiple times. The when/2 predicate currently understands as conditions conjunction (C1; C2), disjunction (C1; C2), variable instantiation nonvar(V) and ground-ness ground(V).

The following suspend predicates are provided:

freeze(V, G):

If V is a variable further checks are delayed until the variable is instantiated with a non-variable or another variable. Otherwise the goal G is directly called.

when(C, G):

If C simplifies to a non-trivial condition further simplifications are delayed until a variable in C is instantiated with a non-variable or another variable. Otherwise the goal G is directly called.

Module unify

This module provides a type 1 interface to attributed variables. The trailed state of the attributed variable is modelled as key value pairs and can be accessed and modified by the predicates put_attr/3, get_attr/3 and del_attr/2. When an attributed variable with type 1 state gets instantiated a call to the attributed variable unify hooks gets scheduled.

Examples:

```
?- [user].
foo:attr_unify_hook(L, _) :- write('L='), write(L), nl.
^D
Yes
?- put_attr(X, foo, [X,Y]), put_attr(Y, foo, [X,Y]), [X,Y]=[1,2].
L=[1,2]
L=[1,2]
?- put_attr(X, foo, [X,Y]), put_attr(Y, foo, [X,Y]), X=Y.
L=[_A,_A]
```

The unify hook attr_unify_hook/2 has to be declared inside the module of the key. The scheduled hook is called after the variable has been instantiated and at the next calls port if the surrounding unification was successful. The hook is allowed to fail or to succeed non-deterministically.

The goals hook attribute_goals/3 has to be optionally declared inside the module of the key. When needed the hook is called only once. If the hook is missing or if it fails a single goal for a put_attr/3 call is generated. The goals are used in the top-level display of answers and they can be retrieved by the call_residue/2 predicate from the module residue.

The following unify predicates are provided:

put_attr(V, K, W):
: The predicate assigns the value W to the key K of the variable V. The assignment is automatically undone upon backtracking.

get_attr(V, K, W):
: The predicate succeeds for the value W of the key K of the variable V.

del_attr(V, K):
: The predicate de-assigns the key K from the variable V.
The de-assignment is automatically undone upon backtracking.

K:attr_unify_hook(W, T) (hook):
: This predicate has to be implemented as a hook for a key K. It will be called with the value W and the term T after the unification.

K:attribute_goals(V, I, O) (hook):
: This predicate has to be optionally implemented as a hook for a key K. It will be called when the goals by the variable V are needed. It should return a list of goals in I. The list uses the end O.

Module verify

This module provides a type 2 interface to attributed variables. The trailed state of the attributed variable is modelled as key value pairs and can be accessed and modified by the predicates put_atts/3, get_atts/3 and del_atts/2. When an attributed variable with type 2 states gets instantiated the attributed variable verify hooks are called immediately.

Examples:

```
?- [user].
foo:verify_attributes(V, _, true) :-
    get_atts(V, foo, L), write('L='), write(L), nl.
^D
Yes
?- put_atts(X, foo, [X,Y]), put_atts(Y, foo, [X,Y]), [X,Y]=[1,2].
L=[_A,_B]
L=[1,_B]
?- put_atts(X, foo, [X,Y]), put_atts(Y, foo, [X,Y]), X=Y.
L=[_A,_B]
```

The verify hook verify_attributes/3 has to be declared inside the module of the key. The hook is called before the variable has been instantiated and it should return a goal which gets scheduled. The hook is allowed to fail or succeed, but it is called only once. If the hook fails the surrounding unification will also fail.

The goals hook portray_attributes/3 has to be optionally declared inside the module of the key. When needed the hook is called only once. If the hook is missing or if it fails a single goal for a put_atts/3 call is generated. The goals are used in the top-level display of answers and they can be retrieved by the call_residue/2 predicate from the module residue.

The following verify predicates are provided:

put_atts(V, K, W):
: The predicate assigns the value W to the key K of the variable V. The assignment is automatically undone upon backtracking.

get_atts(V, K, W):
: The predicate succeeds for the value W of the key K of the variable V.

del_atts(V, K):
: The predicate de-assigns the key K from the variable V.
: The de-assignment is automatically undone upon backtracking.

K:verify_attributes(V, T, G) (hook):
: This predicate has to be implemented as a hook for a key K. It will be called with the variable V and the term T before the unification. It should return a goal in G which will be called after the unification.

K:portray_attributes(V, I, O) (hook):
: This predicate has to be optionally implemented as a hook for a key K. It will be called when the goals by the variable V are needed. It should return a list of goals in I. The list uses the end O.

Module state

This module provides trailed named values.

The following state predicates are provided:

b_setval(K, W):
: The predicate assigns the value W to the key K. The assignment is automatically undone upon backtracking.

nb_current(K, W):
: The predicate succeeds for the value W of the key K.

b_delete(K):
: The predicate de-assigns the key K. The de-assignment is automatically undone upon backtracking.

5.3 Finite Domain Package

This theory groups the predicates of the finite domain constraint solver and the SAT solver. The finite domain constraint solver is available as a module library(finite/clpfd). The SAT solver is available as a module library(finite/clpb). The modules implement instances of the CLP(X) scheme, where X is a domain [7].

For the finite domain solver the domain is the negative and positive integers, for the SAT solver the domain are the Boolean values {0,1}. For the finite domain solver there is also reification of constraints, which isn't necessary for the SAT solver. Our algorithms are deterministic, for theoretical underpinnings and current trends see [8].

We find the following topics:

- **Integer Values:** The finite domain solver allows denoting integer value expressions. These expressions can contain variables.
- **Integer Sets:** The finite domain solver allows denoting integer set expressions. These expressions cannot contain variables.
- **Integer Comparison:** As a convenience the finite domain solver provides a couple of comparisons between integers.
- **Constraint Reification:** As a further convenience we also provide reification of finite domain constraints.
- **Domain Search:** Finally we provide a couple of solving techniques for finite domain constraints and their reification.
- **Boolean Values:** The SAT solver allows denoting Boolean value expressions. These expressions can contain variables.
- **Boolean Satisfaction:** Finally we provide a couple of solving techniques for SAT solver constraints.

Integer Values

The finite domain solver allows denoting integer value expressions. These expressions can contain native Prolog variables. Integer expressions are posted to the finite domain solver via the predicate #=/2. Internally integer equations are broken down into elementary integer equations with the help of new native Prolog variables. The resulting elementary integer equations are automatically shown by the top-level.

Example:

```
?- X*Y*Z #= T.
_C*Z #= T,
X*Y #= _C
```

Forward chaining rules and attribute variable hooks guard the interaction between the elementary integer equations. A minimal logic reading of forward chaining rules with delete has been presented in [1]. It has also been shown there how forward chaining rules with delete can do simplifications if the constraint store is held in the forward store. Currently the following inference rule sets have been implemented for integer value expressions:

- Forward Checking
- Duplicate Detection

The forward checking consists of the two inference rules constant elimination and constant back propagation, depending on whether the constants equations have first arrived in the forward store or the elementary integer equations. The forward checking also provides the inference rules of union find and variable renaming. Where permitted the #=/2 integer equations are replaced by native Prolog unification. It is also allowed mixing native Prolog unification =/2 with integer equations:

Examples:

```
?- X = Y, 4 #= X+Y.
X = 2,
Y = 2
?- 4 #= X+Y, X = Y.
X = 2,
Y = 2
```

The duplicate detection has only been implemented for elementary integer equations that are the scalar product of a constant vector and a variable vector. The duplicate detection does not yet work for arbitrary products. It is handy in that it reduces the number of elementary equations and might also detect inconsistencies early on:

Examples:

```
?- 2*X #= 4*Y, 3*X #= 6*Y.
X #= 2*Y
?- X #= Y+1, Y #= X+1.
No
```

The following integer value expressions are provided:

V (finite):
A native Prolog variable V represents an integer variable.

I (finite):
An integer I represents an integer constant.

A + B (finite):
If A and B are expressions then the sum A+B is also an expression.
A - B (finite):
If A and B are expressions then the subtraction A-B is also an expression.
- A (finite):
If A is an expression then the change sign -A is also an expression.
A * B (finite):
If A and B are expressions then the multiplication A*B is also an expression.
abs(A) (finite):
If A is a value expression then the absolute value abs(A) is also an expression.
T [E_1, .., E_n] (finite):
If T is a term and E_1, .., E_n are expressions for $1 \leq n \leq 7$ then the array subscript T [E_1, .., E_n] is also an expression.
C (finite):
A callable C is also an expression.

The following integer value predicates are provided:

A #= B:
If A and B are expressions then their equality is posted.

Integer Sets

Set expressions are posted via the predicate in/2 respectively ins/2 and are similarly handled to equality. Further forward checking and duplicate detection is also applied. But element hood provides additional functionality. Namely we find the following additional inference rule for the element hood:

- Interval Consistency

The interval consistency has been both implemented for the scalar product and multiplication. A restricted form of interval consistency has been implemented. Namely intervals only propagate to the lexical head of a scalar product respectively multiplication. Directed interval consistency also works for the recently introduced abs/1 function.

Examples:

```
?- X in 0..9, Y #= X+1, Z #= Y+1, X #= Z+1.
Z #= X-1,
Z in 2..8,
...

?- X in 0..sup, Y #= X+1, Z #= Y+1, X #= Z+1.
Z #= X-1,
Z #> 1,
...
```

This directed form of interval consistency is weaker than the full interval consistency. The directed form might accept equations that would otherwise be detected as inconsistent. But it has the advantage that it never loops and therefore also works for unbounded domains..

The following integer set expressions are provided:

I (finite):
: An integer I represents a singleton set {I}.

I..J (finite):
: If I and J are integers then the expression represents the interval [I..J]. It is also possible that I takes the value inf or that J takes the value sup. The expression then denotes the corresponding half interval or even the full integer domain.

A \/ B (finite):
: If A and B are set expressions then the union is also a set expression.

A /\ B (finite):
: If A and B are set expressions then the intersection is also a set expression.

\ A (finite):
: If A a set expressions then the complement is also a set expression.

The following integer value predicates are provided:

A in S:
: If A is a value expression and S is a set expression then the element hood is posted.

[A_1, .., A_n] ins S:
: If A_1, .., A_n are value expression and S is a set expression then the element hood is posted for each value expression.

Integer Comparison

As a convenience the finite domain solver provides a couple of comparisons between integers. The following features are provided in connection with integer comparison:

- Constraint Factoring
- Global Constraints

Our finite domain solver is probably unique in that it allows posting element hood for arbitrary expressions. This feature is used to internally implement integer comparison. We find the usual comparisons such as #\=/2, #</2, #>/2, #=</2 and #>=/2. Comparisons are reconstructed from element hood when displaying constraints.

Example:

```
?- Y - X in 0..sup.
X #=< Y
```

The constraint solver also attempts to combine element hood constraints. Element hood constraints over the same expression are intersected similarly to domain constraints. Consequently comparisons can be contracted, subsumed or conflict. Also interaction with equations is possible, which are then treated as singleton element hood constraints.

Example:

```
?- X #> Y, X #= Y.
No
```

The integer comparisons can be used to define more complex conditions. A recurring problem is stating the inequality of a couple of value expressions. The predicate all_different/2 has been defined as a corresponding convenience.

The following integer comparison predicates are provided:

A #\= B:
If A and B are value expressions then their inequality is posted.

A #< B:
If A and B are value expressions then is posted that A is less than B.

A #> B:
If A and B are value expressions then is posted that A is greater than B.

A #=< B:
If A and B are value expressions then is posted that A is less or equal than B.

A #>= B:
If A and B are value expressions then is posted that A is greater or equal than B.

all_different([A_1, .., A_n]):
If A_1, .., A_n are value expressions then their inequality is posted.

Constraint Reification

As a further convenience we also provide reification. Reification comes with a set of Boolean operators (#\)/1, (#\/)/2, etc.. and the possibility to embed membership (in)/2 and equalities respectively inequalities (#=)/2, (#>)/2, etc.. in Boolean expressions. The constraint solver main use case is as follows. The reified variable should on one hand allow firing the constraint, but it should also reflect the entailed of the constraint.

Examples:

```
?- X #< 100-Y #<==> B, B = 0.
B = 0,
X #> -Y+99

?- X #< 100-Y #<==> B, Y = 74, X = 25.
X = 25,
Y = 74,
B = 1
```

Under the hood the reification is implemented via a couple of new guarded constraints. We find the guarded domain range, the guarded equality and the guarded membership. The latter two guarded constraints are based on the scalar product. Each reified constraint is modelled by a pair of guarded constraints. At the moment only the main use case is implemented, so we don't find yet interactions of the guarded constraints.

The following Boolean value predicates are provided:

A #==> B:
If A and B are Boolean expressions then the implication is posted.

A #<== B:
If A and B are Boolean expressions then the converse implication is posted.

A #<==> B:
If A and B are Boolean expressions then the bi-implication is posted.

A #\/ B:
If A and B are Boolean expressions then the disjunction is posted.

A #/\ B:
If A and B are Boolean expressions then the conjunction is posted.

#\ A:
If A is a Boolean expression then the negation is posted.

Domain Search

The constant values in the domain of a variable can be enumerated via the predicate indomain/1. This predicate is capable of enumerating finite and infinite domains. For infinite domains it is also possible to enumerate domains that are open ended on both sides, resulting in an alternating enumeration towards inf and sup:

Examples:

```
?- X in 10..15, indomain(X).
X = 10 ;
...
X = 15
?- indomain(X).
X = 0 ;
X = -1 ;
...
```

As a convenience the finite domain solver provides a couple of solving techniques. We provide the following solving techniques along domain ranges when there is an attempt to label multiple variables at once. The predicate for this search is label/1:

- Brute Infinite Search
- Heuristic Finite Search

Infinite domains are filtered out first and then cantor paired. For finite domains we have implemented a search strategy, which prefers those variables with a smaller cardinality of the domain first. In certain cases this can reduce the search space. Further notions of consistency and search are discussed in [2].

Examples:

```
?- [X,Y] ins 0..9, 3*X+5*Y #= 11, label([X,Y]).
X = 2,
Y = 1 ;
No
?- 3*X+5*Y #= 11, label([X,Y]).
X = 2,
Y = 1 ;
X = -3,
Y = 4 ;
...
```

The predicates indomain/1 and label/1 have randomized equivalents random_indomain/1 and random_label/1. For a full enumeration the randomized versions would be slower, more memory intensive and not give a random sequence, but they are still helpful in picking a first random solution.

The following domain search predicates are provided:

indomain(V):
The predicate succeeds for every element in the domain of the variable V and instantiates the variable V with this element. The domain of the variable can be finite or infinite. A missing domain is interpreted as the full domain.

random_indomain(V):
The predicate succeeds randomly for every constant I that is in the domain of the variable V. The domain of the variable can be only finite.

label([V_1, .., V_n]):
The predicate posts all the assignments of constants I_1, .., I_n to the variables V_1, .., V_n from their domains. Infinite domains are filtered out and cantor paired. Then smaller domains are enumerated first.

random_label([V_1, .., V_n]):
The predicate posts randomly all the assignments of constants I_1, .., I_n to the variables V_1, .., V_n from their domains. Infinite domains are filtered out and cantor paired. Then smaller domains are enumerated first.

Boolean Values

The SAT solver allows denoting Boolean value expressions. These expressions can contain native Prolog variables. Boolean expressions are posted to the SAT solver via the predicate sat/1. Internally the SAT constraint is normalized into a BDD tree. The resulting BDD tree is automatically shown by the top-level:

Example:

```
?- sat(X#Y).
sat((X->(Y->0;1);Y->1;0))
```

BDD tree reductions and attribute variable hooks guard the interaction between SAT constraints. Currently the following inference rule sets have been implemented for Boolean value expressions:

- Forward Checking

The forward checking consists of the two inference rules constant elimination and constant back propagation. Where permitted SAT constraints are replaced by native Prolog unification. It is also allowed mixing native Prolog unification =/2 with SAT constraints:

Examples:

```
?- X=Y, sat(X+Y).
X = 1,
Y = 1
?- sat(X+Y), X=Y.
X = 1,
Y = 1
```

The following Boolean value expressions are provided:

V (SAT):
A native Prolog variable V represents a Boolean variable.
0 (SAT):
1 (SAT):
The constant 0 or 1 represents a Boolean constant.
~ A (SAT):
If A is an expression then the negation ~A is also an expression.
A + B (SAT):
If A and B are expressions then the disjunction A+B is also an expression.
A * B (SAT):
If A and B are expressions then the conjunction A*B is also an expression.
A =< B (SAT):
If A and B are expressions then the implication A=<B is also an expression.
A >= B (SAT):
If A and B are expressions then the implication B=<A is also an expression.
A > B (SAT):
If A and B are expressions then the difference A>B is also an expression.
A < B (SAT):
If A and B are expressions then the difference B>A is also an expression.
A =:= B (SAT):
If A and B are expressions then the equality A=:=B is also an expression.
A # B (SAT):
If A and B are expressions then the xor A#B is also an expression.

A -> B; C (SAT):
If A, B and C are expressions then the if-then-else A->B;C is also an expression.

V^A (SAT):
If V is a native Prolog variable and A is an expression then the Boolean existential quantification V^A is also an expression.

The following Boolean value predicates are provided:

sat(A):
If A is an expression then its satisfiability is posted.

Boolean Satisfaction

As a convenience the SAT solver provides a single solving technique in two incarnations. We provide the following solving technique when there is an attempt to label multiple variables at once or to count the number of solutions.

- Brute Finite Search

Although variable rankings are found in the literature, we didn't implement some special search strategy, since we did not yet find a solution to overcome the ranking overhead. The given variables are tried in the given input order. Counting further depends on labelling, since it is no yet able to use counts derived from a single BDD tree.

Examples:

```
?- sat(X=<Y), sat(Y=<Z), sat(Z=<X), labeling([X,Y,Z]).
X = 0,
Y = 0,
Z = 0 ;
X = 1,
Y = 1,
Z = 1
?- sat(X=<Y), sat(Y=<Z), sat(Z=<X), sat_count([X,Y,Z], N).
N = 2,
sat((X->1;Z->0;1)),
sat((X->(Y->1;0);1)),
sat((Y->(Z->1;0);1))
```

The Boolean constraint can be used to define more complex conditions. A recurring problem is stating the cardinality of a couple of Boolean expressions. The predicate card/2 has been defined as a corresponding convenience.

The following satisfaction predicates are provided:

labeling(L):
The predicate labels the variables in L.

random_labeling(L):
The predicate randomly labels the variables in L.

sat_count(L, N):
The predicate silently labels the variables in L and succeeds in N with the count of the solutions.

card(N, L):
If N is an integer and L is a variable list then the constraint that the number of true variables amounts exactly to N is posted.

5.4 Package misc

This theory is concerned with miscellaneous predicates. The predicates mainly deliver additional operations for the standard Prolog data types. These operations might be useful when implementing chart parsers, constraint solvers, etc..

- **Module elem:** We provide a couple of additional elementary operations.
- **Module bits:** We provide a couple of additional bitwise operations.
- **Module struc:** We provide a couple of additional structure predicates.
- **Module aggregate:** This module provides aggregate predicates.
- **Compatibility Matrix:** t.b.d.

Module elem

We provide a couple of additional elementary operations. The ulp evaluable function is defined for integer, float and decimal. The function returns a result of the same type as its argument. The gcd and the evaluable function operation are currently only defined for integers. The functions return a result of type integer:

```
ulp: integer -> integer          isqrt: integer -> integer
ulp: float -> float              sqrtrem: integer -> integer^2
ulp: decimal -> decimal          iroot: integer^2 -> integer
gcd: integer^2 -> integer        rootrem: integer^2 -> integer^2
lcm: integer^2 -> integer        divmod : number^2 -> number^2
```

The ulp evaluable function makes use of the ulp() function of the Java Math library. The gcd evaluable function implements a binary gcd algorithm for 32-bit integers and otherwise delegates to the Java BigInteger gcd function implementation. The lcm evaluable function is bootstrapped from the gcd evaluable function.

Examples:

```
ulp(0)          --> 1              isqrt(7)        --> 2
ulp(0.0)        --> 4.9E-324       sqrtrem(7)      --> (2,3)
ulp(0d0.00)     --> 0d0.01         iroot(77,5)     --> 2
gcd(36,24)      --> 12             rootrem(77,5)   --> (2, 45)
lcm(36,24)      --> 72             divmod(12,-7)   --> (-1,5)
```

The evaluable functions isqrt/1 and iroot/2 as well as the predicates sqrtrem/3 and rootrem/4 use the fast Hacker method of finding an integer root. The predicate divmod/4 returns both the quotient and remainder of a division. This is faster than invoking the ISO core standard evaluable functions (//)/2 and (rem)/2 separately.

The following elementary evaluable functions are provided:

ulp(X, Y):
Predicate succeeds in Y with the unit of least precision of the number X.

gcd(X, Y, Z):
Predicate succeeds in Z with the greatest common divisor of the integer X and the integer Y.

lcm(X, Y, Z):
Predicate succeeds in Z with the least common multiple of the integer X and the integer Y.

isqrt(X, Y):
The predicate succeeds in Y with the integer square root of X.

sqrtrem(X, Y, Z):
The predicate succeeds in Y with integer square root of X and in Z with the corresponding remainder.

iroot(X, Y, Z):
The predicate succeeds in Z with the Y-th root of X.

rootrem(X, Y, Z, T):
The predicate succeeds in Z with the Y-th root of X with and in T with the corresponding remainder.

divmod(X, Y, Z, T):
The predicate succeeds in Z with the division of X by Y, and in T with the modulo of X by Y.

Module bits

We provide a couple of additional bitwise operations. The evaluable functions bitcount/1, bitlength/1 and lowestsetbit/1 deal with the determination of certain bits of the given integer.

Examples:

```
bitlength(333)              --> 9
```

The evaluable functions setbit/2 and clearbit/2 update the given integer in a more efficient way than would be possible with existing logical and shift operations. The predicate testbit/2 tests a particular bit in a given integer, again the implementation is more efficient than would be possible with existing logical, shift and test operations.

The following bitwise evaluable functions are provided:

bitcount(X, N):
Predicate succeeds in N with the number of non-zero bits of X.
bitlength(X, N):
Predicate succeeds in N with the highest non-zero bit of X.
lowestsetbit(X, N):
Predicate succeeds in N with the lowest non-zero bit of X.
setbit(X, Y, Z):
The predicate succeeds in Z with Y \/ (1 << X).
clearbit(X, Y, Z):
The predicate succeeds in Z with Y /\ \ (1 << X).

The following built-in predicates are provided for bitwise extensions. The built-ins arithmetically evaluate their arguments before performing their tests:

testbit(X, Y):
The predicate succeeds when Y /\ (1 << X) =\= 0.

Module struc

We provide a couple of additional variables predicates. Prolog already provides an existential quantification operator in the form of (^)/2. This operator is only needed in certain circumstance such as the frequent predicates from the module abstract or the module bags.

Examples:

```
?- sys_term_kernel(X#p(X,Y),K).
K = p(X,Y)
?- sys_term_globals(X#p(X,Y),L).
L = [Y]
```

We introduce a further operator. We use the form (#)/2 for the universal quantification operator. For performance reasons we have introduced the predicates sys_term_kernel/2 and sys_term_globals/2 which are the analogues to the corresponding goal predicates.

The following variable predicates are provided:

sys_term_kernel(G, K):
The predicate succeeds when K unifies with the kernel of the goal G.

sys_term_globals(G, L):
The predicate succeeds when L unifies with the global variables of the goal G.

Module aggregate

The aggregate predicates take a set of solutions and compute an aggregate on it. The predicate accumulate/3 aggregates the solution that is produced by findall/3. The predicate aggregate/3 respectively sys_collect/3 aggregates the solutions that are produced by bagof/3 respectively sys_heapof/3. The current implementation is not yet optimal, since the solution lists are always materialized.

Examples:

```
?- [user].
p(4,5).
p(1,2).
p(1,3).

Yes
?- aggregate((sum(X),count),p(Y,X),R).
Y = 4,
R = (5,1) ;
Y = 1,
R = (5,2)
```

The following aggregate predicates are provided:

accumulate(A, G, R):

The predicate finds all the solutions to the goal G. It then computes the aggregate A from the solutions. The predicate then succeeds when R unifies with the result. The following aggregates are recognized:

count:	The result is the number of solutions.
sum(X):	The result is the sum of the X values.
min(X):	The result is the minimum of the X values.
max(X):	The result is the maximum of the X values.
bag(X):	The result is the list of the X values.
nodup(X):	The result is the list of the distinct X values.
set(X):	The result is the list of the sorted X values.
(A,B):	The result is the pairing of the aggregate A and B.

aggregate(A, X_1^…^X_n^G, R):

The predicate determines all the solutions to the goal G, whereby computing the aggregate A of the solutions grouped by the witnesses. The predicate then repeatly succeeds by unifying the witnesses and when R unifies with the result.

sys_collect(S, X_1^…^X_n^G, R):

The predicate determines the same results as the predicate aggregate/3. But the results are sorted by the witnesses instead of grouped by the witnesses.

foreach(G, H):

Calls the conjunction of those instances of H where G succeeds. Variables occuring in H and not occuring in G are shared across the conjunction.

Compatibility Matrix

t.b.d.

Table 1: Compatibility Matrix for the Miscellaneous Package

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

5.5 Package experiment

This theory is concerned with accessing the Prolog system. Internally the forward chaining component uses clause references to undo insertions and removals.

- **Module attr:** Attribute variables trigger during unification.
- **Module trail:** Clauses can be temporarily asserted or retracted.
- **Module cont:** Goals can be pushed on the continuation queue.
- **Compatibility Matrix:** t.b.d.

Module attr

Attribute variables trigger during unification. From the viewpoint of the interpreter attribute variables are simply variables. The main functionality is that binding an attribute variable triggers the hooks that are associated with the attribute variable. If a hook fails the unification fails. If a hook succeeds the binding effects of the hook are kept.

Example:

```
hook(V, T) :-
   write('bind '), write(V),
   write(' to '), write(T), nl.
```

A hook is just a closure that takes the attribute variable and the term that is attempted to unify with the attribute variable. It can be compiled with the predicate sys_compile_hook/3. The resulting reference can be recorded and erased like a clause reference. The hooks of a variable can be enumerated via the predicate sys_clause_hook/3.

Example:

```
?- sys_compile_hook(V, hook, R),
   recordz_ref(R), V = 99.
bind _A to 99
V = 99.
```

In contrast to ordinary variables, attribute variables are reordered during unification so that two attribute variables are only unified as a last resort. As a result attribute variables are less often instantiated than ordinary variables. The predicate sys_ensure_serno/1 can be used to force the assignment of a serial number. The serial number of a variable is used in writing and lexical comparison.

The following attribute variable predicates are provided:

sys_ensure_serno(V):
: The predicate promotes the first argument to an attribute variable when V is an ordinary variable. The predicate then ensures that the first argument has a serial number.

sys_compile_hook(V, H, R):
: The predicate promotes the first argument to an attribute variable when V is an ordinary variable. The predicate then succeeds when the compiled reference of the hook H unifies with R.

sys_clause_hook(V, H, R):
: The predicate fails when V is an ordinary variable. Otherwise the predicate succeeds for each hook H and reference R that unifies with the hooks and references of the attribute variable V.

Module trail

The system predicate sys_unbind/1 installs an unbind handler and immediately succeeds. The unbind handler is invoked during a redo or a close. In contrast to a cutter the unbind handler is not invoked when choice points are removed. When the unbind handlers are executed exceptions are accumulated that are possibly thrown by the unbind handlers. When unbind handler fails a directive failed error is thrown.

The predicate sys_freeze_var/2 will create a new reference object that captures the given variable. The reference object will have a stable hash, equal and lexical ordering for the duration of the continuation. The predicate sys_melt_var/2 allows retrieving the term the variable is currently instantiated to. The predicate sys_bound_var/1 allows checking whether the variable is currently instantiated or not.

The following trailed update predicates are provided:

sys_unbind(A):
: The predicate installs an unbind handler A and immediately succeeds. The unbind handler is invoked during a redo or a close.

sys_freeze_var(V, R):
: The predicate succeeds when R unifies with a new references object that captures the variable V.

sys_melt_var(R, T):
: The predicate succeeds when T unifies with what the variable captured by the reference object R is currently instantiated.

sys_bound_var(R):
: The predicate succeeds when the variable captured by the reference object R is currently instantiated.

sys_freezer(R):
: The predicate succeeds when R is a reference to a variable.

Module cont

This module allows accessing the continuation queue and the verify flag. New goals can be pushed and popped from the continuation queue via the predicates cont_push/1 and cont_pop/0. The predicate sys_ripple/1 allows disabling the verify flag during the execution of a goal.

It is recommended to push a delayed goal by the predicate sys_assume_cont/1 from the module assume. This predicate will also place an undo handler on the trail that will pop the delayed goal upon backtracking.

When the verify flag allows it the whole continuation queue is executed either on the next exit port of a built-in or the next unification port of a defined predicate. The verify flag is automatically disabled during execution of attribute variable hook.

The following continuation queue predicates are provided:

cont_push(G):
The predicate pushes the goal G on the continuation queue.

cont_pop:
The predicate pops a goal from the continuation queue.

sys_ripple(A):
The predicate succeeds whenever A succeeds. The goal A is invoked with the verify flag temporarily set to off.

The following Prolog flags for continuation queue are provided:

sys_verify:
Legal values are on and off. The flag indicates whether the interpreter currently executes continuations. Default value is on. The value can be changed.

Compatibility Matrix

t.b.d.

Table 2: Compatibility Matrix for the Miscellaneous Package

Nr	*Description*	*System*
1	t.b.d.	
2	t.b.d.	
3	t.b.d.	
4	t.b.d.	

6 Appendix Example Listings

The full source code of the Jekejeke Minlog module examples is given. The following source code has been included:

6.1 Bonner's Examples

For the Bonner's examples there are the following sources:

- **grade.p:** The Prolog text of the embedded implication example.
- **grade2.p:** The Prolog text of the embedded contraction example.

Prolog Text grade

```
/**
 * Bonner's first hypothetical reasoning example.
 */

:- use_module(library(minimal/hypo)).

/* must take german and can choose between french and italian */
grade(S) :- take(S, german), take(S, french).
grade(S) :- take(S, german), take(S, italian).

/* hans has already taken french */
:- multifile take/2.
:- thread_local take/2.
take(hans, french).

% ?- use_module(library(minimal/hypo)).
% % 0 consults and 0 unloads in 0 ms.
% Yes

% /* hans would not grade if he takes also italian */
% ?- assumez(take(hans, italian)), grade(hans).
% No

% /* hans would grade if he takes also german */
% ?- assumez(take(hans, german)), grade(hans).
% Yes ;
% No
```

Prolog Text grade2

```
/**
 * Bonner's second counter factual example.
 */

/* anna has already taken german, french and italian*/
:- multifile take/2.
:- thread_local take/2.
take(anna, german).
take(anna, french).
take(anna, italian).

% /* anna would grade if she would not have taken italian */
% ?- retire(take(anna, italian)), grade(anna).
% Yes ;
% No

% /* anna would not grade if she would not have taken german */
% ?- retire(take2(anna, german)), grade(anna).
% No
```

6.2 Animals Revisited

For the animals revisited there is the following source:

- **animals3.p:** The Prolog text.

Prolog Text animals3

```
/**
 * Animals expert system via forward chaining.
 */

:- use_module(library(minimal/delta)).

:- multifile motion/1, skin/1, diet/1.
:- thread_local motion/1, skin/1, class/1, diet/1, animal/1.

post(class(mamal)) <= posted(motion(walk)), posted(skin(fur)).
post(class(fish)) <= posted(motion(swim)), posted(skin(scale)).
post(class(bird)) <= posted(motion(fly)), posted(skin(feather)).

post(animal(rodent)) <= posted(class(mamal)), posted(diet(plant)).
post(animal(cat)) <= posted(class(mamal)), posted(diet(meat)).
post(animal(salmon)) <= posted(class(fish)), posted(diet(meat)).
post(animal(eagle)) <= posted(class(bird)), posted(diet(meat)).

write('The animal is '), write(X), nl <= posted(animal(X)).

% ?- use_module(library(minimal/delta)).
% % 0 consults and 0 unloads in 0 ms.
% Yes

% ?- post(motion(walk)), post(skin(fur)).
% Yes

% ?- post(motion(walk)), post(skin(fur)), post(diet(meat)).
% The animal is cat
% Yes

% ?- post(motion(walk)), post(skin(fur)), post(diet(plant)).
% The animal is rodent
% Yes
```

6.3 Palindrome Revisited

For the palindrome revisited there is the following source:

- **palin3.p:** The Prolog text.

Prolog Text palin3

```
/**
 * Palindromes in via chart DCG.
 */

:- use_module(library(minimal/chart)).

:- multifile 'D'/3.
:- thread_local 'D'/3, palin/4.
:- forward palin/5.

palin([], [Middle]) ==:
   [Middle].
palin([Middle], []) ==:
   [Middle, Middle].
palin([Border | List], Middle) ==:
   [Border], palin(List, Middle), [Border].

% ?- use_module(library(minimal/chart)).
% % 0 consults and 0 unloads in 0 ms.
% Yes

% ?- words("bert", 0, _), listing('D'/3).
% :- thread_local 'D'/3.
% 'D'(116, 3, 4).
% 'D'(114, 2, 3).
% 'D'(101, 1, 2).
% 'D'(98, 0, 1).

% ?- words("racecar", 0, N), chart(palin(X,Y), 0, N).
% X = [114,97,99],
% Y = [101]

% ?- words("bert", 0, N), chart(palin(X,Y), 0, N).
% No
```

6.4 Money Revisited

For the money revisited there is the following source:

- **money3.p:** The Prolog text.

Prolog Text money3

```
/**
 * Prolog code for the revisited backtracking example.
 *
 * Puzzle originally published July 1924 issue of
 * Strand Magazine by Henry Dudeney
 *
 * Copyright 2012-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.5 (minimal logic extension module)
 */

:- use_module(library(finite/clpfd)).

% puzzle(-List)
puzzle(X) :-
   X = [S,E,N,D,M,O,R,Y],
   X ins 0..9,
   all_different(X),
   M #\= 0,
   S #\= 0,
             1000*S + 100*E + 10*N + D +
             1000*M + 100*O + 10*R + E #=
   10000*M + 1000*O + 100*N + 10*E + Y,
   label(X).

% ?- puzzle(Z).
% Z = [9,5,6,7,1,0,8,2] ;
% No
```

6.5 Little Solver

For the little solver there is the following source:

- **domain.p:** The Prolog text.

Prolog Text domain

```
/**
 * Prolog code for the little solver.
 */

:- use_module(library(minimal/delta)).
:- use_module(library(basic/lists)).
:- use_module(library(advanced/sets)).

% bound(+Atom, +Elem)
:- multifile bound/2.
:- thread_local bound/2.

true <=
   phaseout_posted(bound(X, C)), bound(X, C), !.
fail <=
   posted(bound(X, _)), bound(X, _).

% domain(+Atom, +List)
:- multifile domain/2.
:- thread_local domain/2.

fail <=
   posted(domain(_, [])), !.
post(bound(X, C)) <=
   phaseout_posted(domain(X, [C])), !.
true <=
   phaseout_posted(domain(X, D)), posted(bound(X, C)),
   member(C, D), !.
fail <=
   posted(domain(X, _)), posted(bound(X, _)).
post(domain(X, F)) <=
   phaseout_posted(domain(X, D)), phaseout(domain(X, E)),
   intersection(D, E, F).

% ?- use_module(library(minimal/delta)).
% Yes

% ?- post(domain(x, [1])), listing(bound/2), listing(domain/2).
% bound(x, 1).
% Yes

% ?- post(domain(x, [1,2,3])), post(domain(y, [2,3,4])),
%    listing(bound/2), listing(domain/2).
% domain(x, [1,2,3]).
% domain(y, [2,3,4]).
% Yes

% ?- post(domain(x, [1,2,3])), post(domain(x, [2,3,4])),
%   listing(bound/2), listing(domain/2).
% domain(x, [2,3]).
% Yes
```

```
% ?- post(domain(x, [1,2,3])), post(bound(x,4)),
%    listing(bound/2), listing(domain/2).
% No
```

6.6 Subject to Occurs Check

For the subject to occurs check there are the following sources:

- **typed.p:** The Prolog text for the type inference without subject to occurs check.
- **typed2.p:** The Prolog text for the type inference with subject to occurs check.

Prolog Text typed

```
/**
 * Prolog code for the type inference
 * without subject to occurs check.
 *
 * Copyright 2013-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

:- use_module(library(experiment/pairlist)).

/*****************************************************/
/* Type Inference for Simple Types                   */
/*****************************************************/

% typed(+Expression, +Context, -Type)
typed(X, C, T) :- var(X), !, lookup_eq(C, X, T).
typed(lam(X,E), C, (S->T)) :- typed(E, [X-S|C], T).
typed(app(E,F), C, T) :- typed(E, C, (S->T)), typed(F, C, S).

% ?- typed(app(E,F), [E-A,F-B], C).
% A = (B->C)

% ?- typed(app(F,F), [F-A], B).
% A = (A->B) or crash, both undesired

% ?- typed(lam(X,lam(Y,app(Y,X))), [], A).
% A = (_C->(_C->_I)->_I
```

Prolog Text typed2

```
/**
 * Prolog code for the type inference
 * with subject to occurs check.
 *
 * Copyright 2013-2014, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

:- use_module(library(experiment/pairlist)).
:- use_module(library(term/herbrand)).

/*****************************************************/
/* Type Inference for Simple Types                   */
/*****************************************************/

% typed(+Expression, +Context, -Type)
typed2(X, C, T) :- var(X), !, lookup_eq(C, X, T).
typed2(lam(X,E), C, (S->T)) :- typed2(E, [X-S|C], T).
typed2(app(E,F), C, T) :- sto(S), typed2(E, C, (S->T)), typed2(F, C, S).
```

```
% ?- sto((A,B,C)), typed2(app(E,F), [E-A,F-B], C).
% A = (B->C),
% sto(C),
% sto(B)

% ?- sto((A,B)), typed2(app(F,F), [F-A], B).
% No

% ?- sto(A), typed2(lam(X,lam(Y,app(Y,X))), [], A).
% A = (_A->(_A->_B)->_B,
% sto(_B),
% sto(_A)
```

Acknowledgements

We are in great debt to Markus Triska, Technische Universität Wien, Austria for putting on display the two attribute variable interfaces on the SWI-Prolog mailing list in 2017.

We are also grateful to Kuniaki Mukai, Keio University, Japan for exchanging code snippets of a prototype SAT solver and encouraging comments.

Finally we would like to thank Jan Wielemaker, Vrije Universiteit Amsterdam for patiently answering questions about the SWI-Prolog system.

Indexes

Public Predicates

Predicate	Module
#/\ /2	finite/reify
#< /2	finite/clpfd
#<== /2	finite/reify
#<==> /2	finite/reify
#= /2	finite/linform
#=< /2	finite/clpfd
#==> /2	finite/reify
#> /2	finite/clpfd
#>= /2	finite/clpfd
#\ /1	finite/reify
#\/ /2	finite/reify
#\= /2	finite/clpfd
(+)/1	minimal/delta
(+)/3	minimal/chart
(-)/1	minimal/hypo
--: /2	minimal/chart
<= /1	minimal/hypo
hypo: <= /1	minimal/delta
hypo: <= /1	minimal/chart
<= /2	minimal/delta
= /1	minimal/delta
==> /2	minimal/chart
=> /2	minimal/hypo
hypo: => /2	minimal/delta
hypo: => /2	minimal/chart
all_different/1	finite/clpfd
bitcount/2	misc/bits
bitlength/2	misc/bits
chart/2	minimal/chart
chart/3	minimal/chart
clearbit/3	misc/bits
cosmetic/1	minimal/delta
delta_abnormal/1	minimal/delta
forward/1	minimal/delta
freeze/2	term/suspend
gcd/3	misc/elem
user:goal_expansion/2	minimal/chart
user:goal_exposing/3	minimal/delta
hypo_abnormal/1	minimal/hypo
hypo:hypo_abnormal/1	minimal/delta
hypo:hypo_abnormal/1	minimal/chart
in/2	finite/intset
indomain/1	finite/intset
ins/2	finite/intset
label/1	finite/clpfd
last_atom_concat/3	misc/atom
last_sub_atom/4	misc/atom
last_sub_atom/5	misc/atom
lowercase/2	misc/char
lowestsetbit/2	misc/bits

minus_abnormal/1	minimal/hypo
neq/2	term/herbrand
setbit/3	misc/bits
sto/1	term/herbrand
sys_assume_cont/1	minimal/assume
sys_assume_ref/1	minimal/assume
sys_bound_var/1	experiment/trail
sys_clause_hook/3	experiment/attr
sys_compile_hook/3	experiment/attr
user:sys_current_eq/2	term/herbrand
user:sys_current_eq/2	term/suspend
user:sys_current_eq/2	finite/clpfd
sys_ensure_hook/2	minimal/assume
sys_ensure_hook/3	minimal/assume
sys_ensure_serno/1	experiment/attr
sys_find_goal/4	minimal/delta
sys_freeze_var/2	experiment/trail
simp:sys_goal_simplification/2	minimal/delta
sys_melt_var/2	experiment/trail
sys_pop_cont/0	experiment/cont
sys_post_goal/3	minimal/delta
sys_push_cont/1	experiment/cont
sys_ref_eq/2	finite/clpfd
sys_retire_ref/1	minimal/assume
sys_ripple/1	experiment/cont
sys_term_globals/2	misc/struc
sys_term_kernel/2	misc/struc
simp:sys_term_simplification/2	minimal/delta
sys_test_bit/2	misc/bits
sys_unbind/1	experiment/trail
user:sys_unwrap_eq/2	term/herbrand
user:sys_unwrap_eq/2	term/suspend
user:sys_unwrap_eq/2	finite/clpfd
user:term_expansion/2	minimal/delta
user:term_expansion/2	minimal/chart
ulp/2	misc/elem
uppercase/2	misc/char
when/2	term/suspend
zero/0	minimal/hypo

Package Local Predicates

Predicate	Module
sys_abs_range/2	finite/intset
sys_absv/2	finite/clpfd
sys_absv/3	finite/clpfd
sys_add_prod/3	finite/linform
sys_add_range/3	finite/intset
sys_add_set/3	finite/intset
sys_bisect/4	finite/intset
sys_blur_range/3	finite/intset
sys_bound_set/2	finite/intset
sys_card_set/2	finite/intset
sys_comp_set/2	finite/intset

sys_conc_range/2	finite/intset
sys_cross_range/3	finite/intset
sys_div_prod/3	finite/linform
sys_div_range/3	finite/intset
sys_div_set/3	finite/intset
sys_elem_set/2	finite/intset
sys_expr_set/2	finite/intset
sys_flip_prod/2	finite/linform
sys_flip_range/2	finite/intset
sys_flip_set/3	finite/intset
sys_fresh_var/2	finite/linform
sys_gcd_prod/2	finite/linform
sys_hook/2	finite/clpfd
sys_in/3	finite/clpfd
sys_in/4	finite/clpfd
sys_inter_range/3	finite/intset
sys_inter_set/3	finite/intset
sys_invabs_range/2	finite/intset
sys_lin/2	finite/clpfd
sys_lin/3	finite/clpfd
sys_make_prod/4	finite/linform
sys_mem_set/2	finite/intset
sys_mul_prod/3	finite/linform
sys_mulv/3	finite/clpfd
sys_mulv/4	finite/clpfd
sys_prem_range/2	finite/intset
sys_pretty_lin/3	finite/linform
sys_pretty_set/2	finite/intset
sys_pump_range/3	finite/intset
sys_root_range/2	finite/intset
sys_set/2	finite/clpfd
sys_set/3	finite/clpfd
sys_slash_range/3	finite/intset
sys_square_range/2	finite/intset
sys_value_expr/3	finite/linform
sys_value_expr_inv/3	finite/linform

Non-Private Meta-Predicates

Predicate	Exp	Body	Rule	Module
+0	yes	no	no	minimal/delta
- 0	yes	no	no	minimal/hypo
<=(-1)	yes	no	no	minimal/hypo
hypo: <=(-1)	yes	no	no	minimal/delta
hypo: <=(-1)	yes	no	no	minimal/chart
<=(0,-1)	yes	no	no	minimal/delta
=(0)	yes	no	no	minimal/delta
=>(-1,0)	yes	no	no	minimal/hypo
hypo: =>(-1,0)	yes	no	no	minimal/delta
hypo: =>(-1,0)	yes	no	no	minimal/chart
cosmetic(0)	yes	no	no	minimal/delta
forward(0)	yes	no	no	minimal/delta
freeze(?,0)	yes	no	no	term/suspend
user:goal_expansion(0,0)	no	no	no	minimal/chart

user:sys_current_eq(?,0)	yes	no	no	term/herbrand
user:sys_current_eq(?,0)	yes	no	no	term/suspend
user:sys_current_eq(?,0)	yes	no	no	finite/clpfd
simp:sys_goal_simplification(0,0)	no	no	no	minimal/delta
sys_push_cont(0)	yes	no	no	experiment/cont
sys_ref_eq(?,0)	yes	no	no	finite/clpfd
sys_ripple(0)	yes	no	no	experiment/cont
simp:sys_term_simplification(-1,-1)	no	no	no	minimal/delta
sys_unbind(0)	yes	no	no	experiment/trail
user:sys_unwrap_eq(0,0)	yes	no	no	term/herbrand
user:sys_unwrap_eq(0,0)	yes	no	no	term/suspend
user:sys_unwrap_eq(0,0)	yes	no	no	finite/clpfd
user:term_expansion(-1,-1)	no	no	no	minimal/delta
user:term_expansion(-1,-1)	no	no	no	minimal/chart
when(?,0)	yes	no	no	term/suspend

Non-Private Closure-Predicates

Predicate	Module
+(2,?,?)	minimal/chart
--:(2,-3)	minimal/chart
==>(2,-3)	minimal/chart
chart(2,?)	minimal/chart
chart(2,?,?)	minimal/chart
sys_clause_hook(?,2,?)	experiment/attr
sys_compile_hook(?,2,?)	experiment/attr
sys_ensure_hook(?,2)	minimal/assume
sys_ensure_hook(?,2,0)	minimal/assume

Non-Private Syntax Operators

Level	Mode	Operator	Module
1200	xfx	<=	minimal/delta
1200	xfx	--:	minimal/chart
1200	xfx	==>	minimal/chart
1150	fy	forward	minimal/delta
1150	fy	cosmetic	minimal/delta
760	xfx	#<==	finite/reify
760	xfx	#<==>	finite/reify
760	xfx	#==>	finite/reify
740	yfx	#\/	finite/reify
720	yfx	#/\	finite/reify
700	fx	<=	minimal/hypo
700	fy	#\	finite/reify
700	xfx	#<	finite/clpfd
700	xfx	#>	finite/clpfd
700	xfx	#>=	finite/clpfd
700	xfx	#\=	finite/clpfd
700	xfx	#=	finite/linform
700	xfx	in	finite/intset
700	xfx	=>	minimal/hypo

700	xfx	ins	finite/intset
700	xfx	#=<	finite/clpfd
200	fy	=	minimal/delta
100	fx	..	finite/intset
100	xfx	..	finite/intset
100	xf	...	finite/intset

Pictures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tables

References

[1] Burse, J. (2012): Bedeutungsextraktion als Deduktion, Präsentation an der 22. Tagung der Computerlinguistik-Studierenden, June, 2012, University Trier, German http://www.xlog.ch/webapps/sitetab/doclet/en/docs/1_newsandevents/2_events/098_tacos/package.html

[2] Apt, K.R. (2003): Principles of Constraint Programming, Cambridge University Press, 2003 https://www.amazon.de/Principles-Constraint-Programming-Krzysztof-Apt/dp/0521125499

[3] Bonner, A.J. (1988): A Logic for Hypothetical Reasoning, Technical Report DCS-TR-230, Department of Computer Science, Rutgers University, August, 1988 http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1451

[4] Bonner, A.J. (1990): Hypothetical Datalog: Complexity and Expressibility, Theoretical Computer Science, 76:3-51, 1990 http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8941

[5] Halpern , J. (1999): Hypothetical Knowledge and Counterfactual Reasoning, International Journal of Game Theory, 28:315-330, 1999 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.7343

[6] Lindgren, T. (1994): A Continuation-Passing Style for Prolog, International Symposium on Logic programming, 603-617, 1994 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.3062

[7] Codognet, P. and Diaz, D. (1993): Boolean Constraint Solving Using clp(FD), The Journal of Logic Programming, Volume 27, Issue 3, June 1996, Pages 185-226 http://cri-dist.univ-paris1.fr/diaz/publications/CLP-FD/ilps93.pdf

[8] Vardi, M.Y. (2015): The SAT Revolution: Solving, Sampling, and Counting, Special Talk at Highlights of Logic, Games and Automata, Prague, 15–18 September 2015 https://www.cs.rice.edu/~vardi/papers/highlights15.pdf

[9] Holzbauer, C. and Frühwrith, T. (1999): Compiling Constraint Handling Rules into Prolog with Attributed Variables, Principles and Practice of Declarative Programming pp 117-133, G. Nadathur, Ed., 1999 http://pdfs.semanticscholar.org/fa3b/4fc380c2c387a17d56f5dfd9faaf3de4a155.pdf

[10] Niemelä, I. (1999): Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4) (1999) 241-273 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.8713