Jekejeke Prolog

Minimal Logic 0.6.7

Benchmark CLP(FD) Results

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

Date: November 18th, 2013
Version: 0.2

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

All industrial property rights regarding the information - copyright and patent rights in particular - are the sole property of XLOG Technologies GmbH. If the company was not the originator of some excerpts, XLOG Technologies GmbH has at least obtained the right to reproduce, change and translate the information.

Reproduction is restricted to the whole unaltered document. Reproduction of the information is only allowed for non-commercial uses. Small excerpts can be used if properly cited. Citations must at least include the document title, the product family, the product version, the company, the date and the page. Example:

> … Defined predicates with arity>0, both static and dynamic, are indexed on the functor of their first argument [1, p.17] ...
>
> [1] Language Reference, Jekejeke Prolog 0.8.1, XLOG Technologies GmbH, Switzerland, February 22nd, 2010

Trademarks

Jekejeke is a registered trademark of XLOG Technologies GmbH.

Table of Contents

1 Introduction....4
2 Study Object....5
2.1 Clause Indexing....6
2.2 Forward Chaining....8
2.3 Attribute Variables....11
2.4 Test Scope....14
3 Available Optimizations....15
3.1 Bound Propagation....15
3.2 Refer Propagation....16
3.3 Constant Instantiation....17
3.4 Variable Instantiation....18
4 Strategies Comparison....19
4.1 Test Results....20
4.2 Discussion Bound Propagation....22
4.3 Discussion Refer Propagation....23
4.4 Discussion Constant Instantiation....24
4.5 Discussion Variable Instantiation....25
5 Interpreter Comparison....26
5.1 Test Results....27
5.2 Discussion GNU Prolog....29
5.3 Discussion B-Prolog Prolog....30
5.4 Discussion ECLiPSe Prolog....31
5.5 Discussion SWI-Prolog....32
5.6 Discussion Ciao Prolog....33
6 Appendix Harness Listings....34
6.1 Common Files....34
6.2 Jekejeke Prolog Harness....36
6.3 GNU Prolog Harness....37
6.4 B-Prolog Harness....38
6.5 ECLiPSe Prolog Harness....39
6.6 SWI Prolog Harness....41
6.7 Ciao Prolog Harness....42
7 Appendix Test Program Listings....43
7.1 grocery Test Program....44
7.2 pythago Test Program....46
7.3 queens Test Program....48
7.4 money Test Program....50
7.5 crypt Test Program....52
7.6 zebra Test Program....55
7.7 pigeon Test Program....58
8 Appendix Example Program Listings....62
8.1 add Example Program....62
8.2 addensure Example Program....69
Pictures....80
Tables....80
References....81

Change History

Jan Burse, August 22th, 2013, 0.1:

- Initial version.

Jan Burse, November 18th, 2013, 0.2:

- Little solver with union find and add constraint introduced.

1 Introduction

This document describes some benchmarking results for the Jekejeke Prolog system.

- **Study Object:** In this section we give a brief introduction into the architecture of the Jekejeke Minlog extension.

- **Available Optimizations:** We will also highlight some optimizations that were implemented for the interpreter.

- **Strategies Comparison:** We conducted a couple of performance tests. The main indicator that was measured was the elapsed time for various test cases.

- **Interpreter Comparison:** We conducted a couple of performance tests. The main indicator that was measured was the elapsed time for various test cases.

- **Appendix Harness Listings:** The full source code of the Java classes and the Prolog texts for the test harness is given.

- **Appendix Test Program Listings:** The full source code of the Prolog texts for the test programs is given.

2 Study Object

In this section we give a brief introduction into the architecture of the Jekejeke Minlog extension. We will highlight the following points:

- **Clause Indexing:** The Jekejeke Minlog extension is based on a notion of clause Java objects. We will explain how these objects come into being and how these objects are associated with the knowledge base.

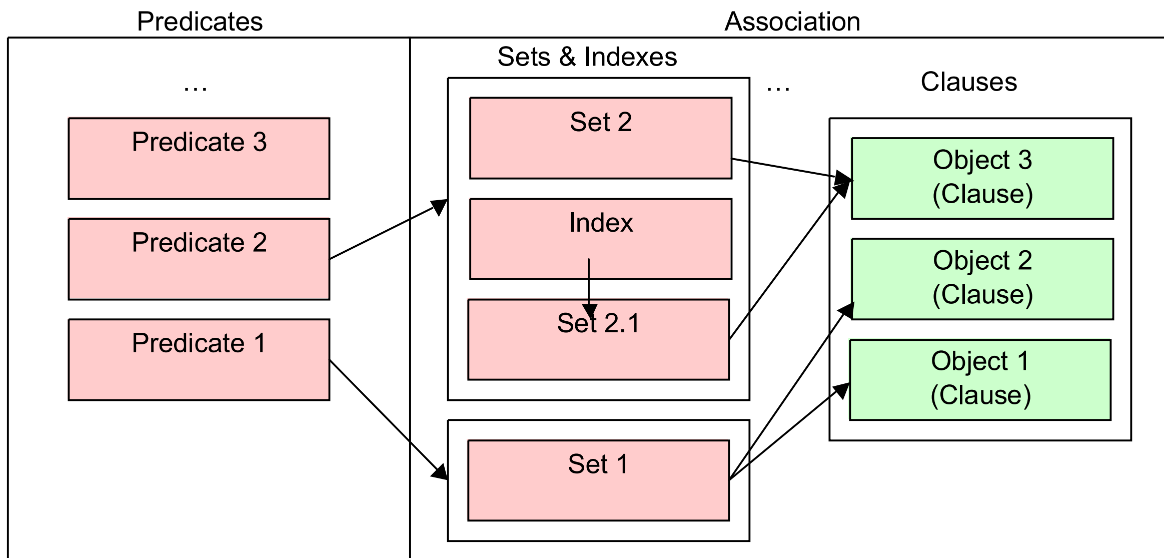
- **Forward Chaining:** The Jekejeke Minlog extension is further based on a forward chaining rule language and on a forward chaining engine. Forward chaining rules allow the declarative definition of the addition and removal of clause objects.

- **Attribute Variables:** The Jekejeke Minlog extension finally provides attribute variables. These variables provide the glue between the forward chaining engine and the native Prolog unification. Access is possible in both directions.

- **Test Scope:** t.b.d.

2.1 Clause Indexing

The Jekejeke Minlog extension is based on a notion of clause Java objects. We will explain how these objects come into being and how these objects are associated with the knowledge base. The knowledge base is divided into predicates. The predicates hold clause sets for all their clauses. But they also hold a recursive index structure. The index structure first says which arguments are currently indexed, and then provides further clause sets and index structures for each index entry.



Picture 1: Clause Indexing

The Java virtual machine treats clauses as normal Java objects in the heap. They are automatically reclaimed during the various garbage collection passes of the Java virtual machine. When a clause is removed it will not be any more referenced by the knowledge base. If an active thread does not anymore use the clause, it will then automatically be removed by the garbage collection sooner or later. A thread might use a clause via its clause reference without having it currently associated to the knowledge base.

The Jekejeke Minlog toolbox provides clause reference based operations that allow bootstrapping hypothetical and counter factual reasoning. These clause reference based operations are also used in the forward chaining component. Their main characteristic is that they are automatically undone during backtracking. This is archived by installing unbind handlers in the variable trail of the interpreter. Therefore a thread might reference a clause from the variable trail even if it does not reference it anymore in the current execution.

We did not yet spend too much time on optimizing away references from the variable trail. Instead we tried to make the toolbox operations itself as performing as possible. This means that programs that use the Jekejeke Minlog toolbox might use quite an amount of memory. This could be an issue for programs that perform a great number of operations without backtracking. On the other hand for problems where the number of operations is rather limited and a lot of backtracking occurs memory is not so much an issue. This is for example the case for our CLP(FD) solver.

A first attack vector for performing operations is to assure that the compilation of a clause doesn't take too much time. In the Jekejeke Runtime the compilation of a clause is a two-phase process. In the first phase the clause is copied and will have its own variables. In the second phase the variables are classified and the code for the head and the body is generated. These two steps have been optimized toward ground facts. Already in the first phase during copy, if a sub-term is found where some extra data structure in the compound indicates that it is ground, no traversal for copying is needed. Finally since a ground fact doesn't have variables the second phase doesn't need some work to do.

A second attack vector for performing operations is to assure that the association and de-association of a clause with the knowledge base doesn't take too much time. Here we work with custom data structures that are optimized towards representing clause sets and index structures. For clause sets we switch between arrays and a custom hash set depending on the size of the clause set. For the index structure we use a custom hash map. The custom hash set and hash map avoids some bottlenecks of the Java default implementation. For example we do not need to create an iterator object to traverse these data structures.

2.2 Forward Chaining

The Jekejeke Minlog extension is further based on a forward chaining rule language and on a forward chaining engine. Forward chaining rules allow the declarative definition of the addition and removal of clause objects. Ordinary Prolog rules can be viewed as Horn clauses in the simple case. These rules are usually executed in a backward chaining manner. Backward chaining is based on the notion of solving a sub-goal. To solve a sub-goal zero, one or many new sub-goals need to be solved. The inference step for backward chaining can be schematized as follows:

Before:
Sub-goal: A
Horn clause: A :- B_1, .., B_n

After:
Sub-goal: B_1, .., B_n

Picture 2: Backward Chaining

Our forward chaining rule language doesn't demand a considerable different rule format. It is just that that Horn clauses are executed in a different manner. This execution is not anymore based on the notion of solving a sub-goal. Instead the notion of a solved fact is used. From zero, one or many solved facts new solved facts can emerge. The inference step for forward chaining can be schematized as follows:

Before:
Fact: B_1, .., B_n
Horn clause: A :- B_1, .., B_n

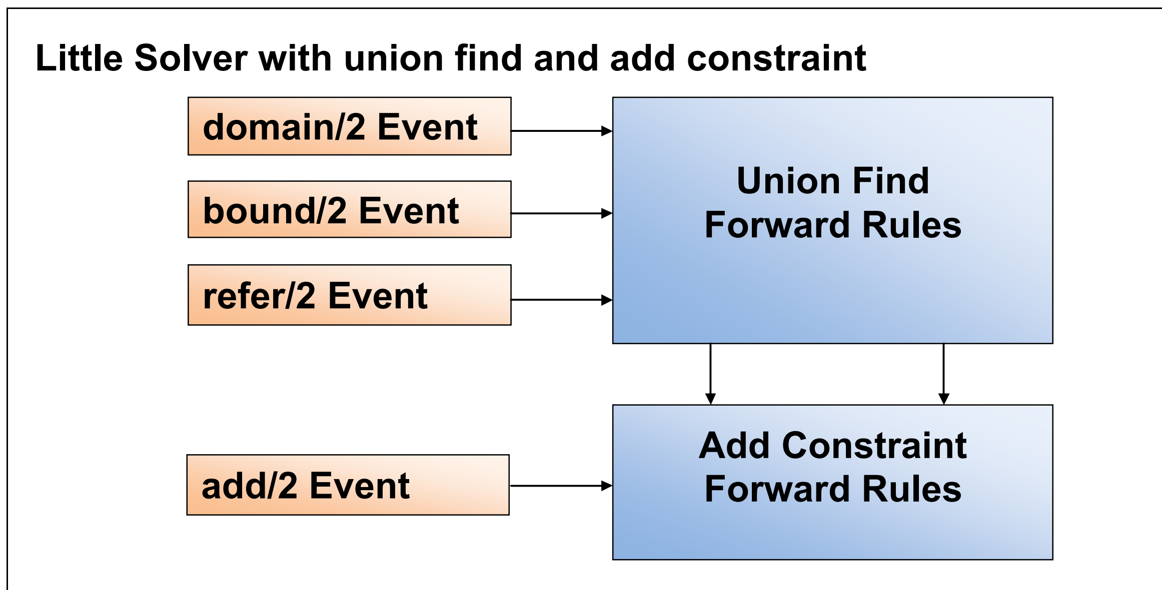
After:
Fact: A

Picture 3: Forward Chaining

In the current Jekejeke Minlog realization Prolog rules that should be executed in a forward chaining manner do use the operator (&:-)/2 instead of the operator (:-)/2. Upon seeing such a rule, the Jekejeke Minlog realization will perform a special compilation. This compilation will produce ordinary Prolog rules that can do a delta computation. The delta computation can compute new facts upon arrival of a single fact. The delta computation is executed in a greedy and exhaustive fashion.

The basic idea to use forward chaining for constraint solving is to store constraints in the forward store. When implementing a constraint solver via forward chaining there will be no separate constraint store. All the advantages of clause indexing of facts, which make up the forward store, will be available for the constraint solver. Similarly the trailing of facts will carry over to constraints, so that modifications of constraints will be automatically undone during backtracking.

The forward chaining rule language and the forward chaining engine have been enhanced so that they can be used to implement constraint solvers. During constraint solving constraints often need to be replaced by other constraints or constraints have to be eliminated right away. To allow Horn clauses to define such a behaviour we have introduced a new operator (&-)/1 which can be used to mark goals. Facts stemming from such goals will be removed during forward chaining.



Picture 4: Rule Layering

The use of the delete mechanism in our forward rule language to define constraint solver has been exemplified in the tutorial example “Little Solver” from the Jekejeke Minlog language manual. In this tutorial a constraint solver for domain intersection is realized. The constraint solver is very primitive and the main rule is the following transformation of two constraints into one constraint. The delete mechanism helps deleting the two old constraints by looking up the corresponding facts and removing them:

$$x \in S, x \in T \quad \text{-->} \; x \in S \cap T$$

The insert of the new constraint is done by the arrival of a new fact in a forward chaining manner. But this is not enough to implement the “Little Solver”. A further not yet mentioned features help implementing it. The forward chaining language allows two mix forward chaining and backward chaining rules. The set intersection of the new fact $x \in S \cap T$ is computed by invoking backward chaining predicate from within one of the forward chaining rules.

The more interesting feature is the way the forward chaining engine can be influenced in its execution. Jekejeke Minlog provides two constructs. The first construct is the asymmetric conjunction (&&)/2. It is already needed in the “Little Solver” example. Without asymmetric conjunction the above replacement rule would fire twice. If a set constraint arrives it would match with the first pattern $x \in S$ and fire the rule once, and then it would match with the second pattern $x \in T$ again and fire the rule one more time.

The second construct is the cut !/0 inside forward chaining rules. When combined with the asymmetric conjunction (&&)/2 it allows a very fine control of when the firing of forward chaining should stop or not for a fact that has just been arrived. This is used in implementing more complex constraint solvers where a layering of the transformation rules is needed. In the appendix two example extensions of the “Little Solver” are found. There are recommendations when to use the cut !/0 and when not to use the cut !/0.

The first example extension introduces a new constraint that allows the aliasing of variables. The main rule is the following transformation of one constraint into another:

$x = y, x \in S \quad \text{-->} \quad y \in S$

The aliasing is implemented by a kind of union find algorithm. If aliasing applies the newly arrived fact is transformed according to the above rule and then checked again, and so on. The result is a new version of the “Little Solver” where rules concerning variable aliasing are inserted before the other rules. The rules form a new layer in front of the old transformation rules of the simple “Little Solver”. Here the cut !/0 is needed to stop further firing as soon as an aliasing is detected.

The second example extension introduces a new constraint that allows relating three variables via addition. The main rule is basically suspension until the variables become instantiated:

$x = c, x + d = y \quad \text{-->} \quad c + d = y$

The suspension is implemented by a couple of sub constraints that model the successive instantiation of the original add constraint. Since the instantiation happens after the aliasing, again a modular organization of the forward chaining rules of the extended “Little Solver” is possible. All the add constraint rules can be placed after the aliasing rules. But since a single instantiation can affect multiple add constraints, one must see that the instantiation reaches all the relevant constraints. Here the cut !/0 is disallowed, since it would stop further firing as soon as an instantiation is detected.

Cuts that are used to organize forward chaining rules into layers behave as red cuts. If they are omitted where they should have been placed and vice versa, the forward chaining rules will not produce the right results or throw an error since an incoming fact is deleted twice. There are also situations where cuts in forward chaining rules can be used like green cuts. If the logic among subsequent forward chaining rules is exclusive, placing a cut might improve their performance.

The CLP(FD) implementation that is bundled with Jekejeke Minlog gives proof of concept that our forward chaining rule language can be used to implement complex constraint solvers. Unfortunately the forward chaining rule language hasn’t evolved yet. We have already introduced a (;)/2 in the body of a forward chaining rule. But we do not yet find the full equivalent of a (->)/2 in the body of a forward chaining rule since we haven’t yet found a way to compile it. The (->)/2 control construct would allow a more performing grouping of recurrent computations among forward chaining rules.

Evolving the forward chaining rule language further poses some interesting conceptual challenges that are non-trivial. The introduction of a (->)/2 control construct would not yet deliver enough expressiveness. It would not allow grouping recurrent computations among forward chaining rules with different rule heads. What we would need would be a control construct that switches between different rule heads. Interestingly the implication (:-)/2 in the rule head could logically allow such a reading. But again we have not yet found an appropriate compilation technique for this construct.

2.3 Attribute Variables

The Jekejeke Minlog extension finally provides attribute variables. These variables provide the glue between the forward chaining engine and the native Prolog unification. Access is possible in both directions. The forward chaining engine is based on the Jekejeke Minlog toolbox which provides trailed database updates. For the attribute variables we had to build a second toolbox. The second toolbox provides all the ingredients that are needed to make attribute variables work and to connect them with the forward chaining engine.

Attribute variables are found in various Prolog system implementations. There is currently no standard concerning attribute variables in terms of a fixed programming interface. Nevertheless there is a common understanding what functionally attribute variables should provide. In our design attribute variables have been conceived as a sub-class of ordinary variables. All our Prolog code that deals with variables went unchanged, since both classes implement the same interface. Nevertheless there are differences in the implementation.

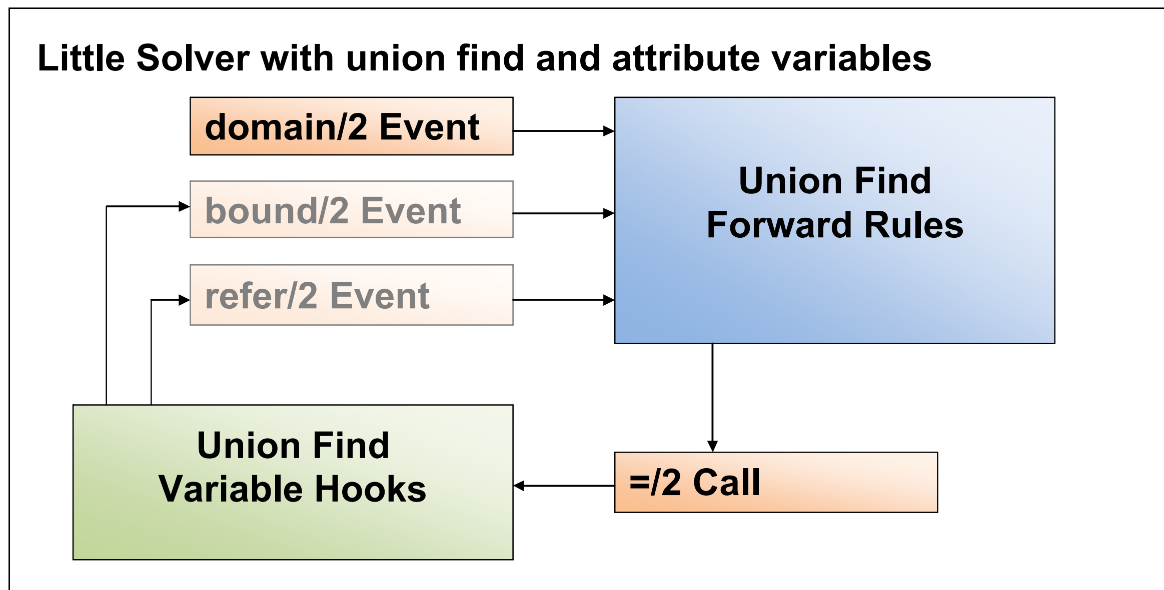
The main difference is how attribute variables react during unification compared to ordinary variables. Ordinary variables do not have some preference during variable aliasing. We just instantiate the variables based on the way the unification problem was posed to the interpreter. For better performance we choose a unification argument order during clause invocation, so that head variables are bound to goal variables. This reduces the length of instantiation chains. But when using the =/2 predicate the end-user has full control on how instantiation chains are built:

Call:	**Call:**
W = V	V = W
Result:	**Result:**
W -> V	V -> W

When aliasing with attribute variables the unification problem is solved such that attribute variables are instantiated at the least moment and the instantiation of ordinary variables is preferred over the instantiation of attribute variables. An instantiation of an attribute variable is only necessary during aliasing if the other variable is also an attribute variable. If an attribute variable instantiation attempted is made, be it by another attribute variable or a Prolog term, the hooks of the attribute variables are first called to check for a veto:

Call:	**Call:**	**Call:**
A = V	V = A	A = B
Result:	**Result:**	**Result:**
V -> A	V -> A	A.hooks(B), A -> B

Further ingredients of the attribute variable toolbox are operations to associate and deassociate hooks from an attribute variable. An attribute variable can have zero, one or many hooks, the operations allow a trailed modification. A hook is a closure for two arguments and contrary to clauses the variables of the closure are shared. Hooks are only called once and they may veto the instantiation of a variable with a term. Hooks are also allowed to have side effects. This is exemplified by the subject to occurs check example found in the tutorial section of the Jekejeke Minlog language reference.



Picture 5: Rule Interaction

The subject to occurs check example does not show how to connect attribute variables with the forward chaining engine. In the appendix there are additional versions of the little solver extension from the previous chapter. These versions show how the connection works. The connection is based on a kind of Gödel coding of attribute variables. With the help of the reference data type of Jekejeke Prolog we can find a code ⟨v⟩ for each attribute variable v. The code behaves like an atom and does not change from assert to retrieval. It is therefore well suited to be used inside forward chaining facts.

The new versions of the "Little Solver" with attribute variables still have the bound/2 and refer/2 events, but only for internal purposes. These two events are generated from native Prolog unification. As soon as a variable has been used in a constraint it will be turned into an attribute variable and a hook will be associated with the attribute variable. This hook is responsible for the translation of native Prolog unification into corresponding internal events of the constraint solver. The hook will have access to the original attribute variables v and encode them as ⟨v⟩ in the generated internal event.

Now there are situations where the old constraint solver generated new bound/2 and refer/2 events. For the new constraint solver these events should also be reflected by native Prolog unification. The idea is here that the constraint solver does generate these events indirectly by performing a call-out to native Prolog unification. The call-out is realized by advising the post/1 system predicate. The advising code will recognize internal events similar to bound/2 and refer/2 and have access to the code ⟨v⟩ of the attribute variables. It will perform according native Prolog unifications on the decoded attribute variables v.

In implementing constraint solvers, it might be necessary that the logic uses lexical comparison. For example when representing a polynomial it comes handy ordering the coefficients according to the variables of the polynomial. Implementing such algorithms is no problem when constraint variables are represented as atoms. We modified the Jekejeke Prolog runtime so that a lexical comparison is also possible when constraint variable are represented as coded attribute variables. For this purpose we have provided the possibility that the reference data type can be ordered so as to allow an ordering of the codes.

The CLP(FD) implementation that is bundled with Jekejeke Minlog gives again proof of concept that our attribute variables can be used to implement complex constraint solvers. The newest version of our CLP(FD) implementation shows a similar behavior versus variables as typical CLP(FD) implementations from other Prolog systems show. Concerning the implementation of constraint solvers, we have further experience from other applications in natural language processing and expert systems. Not all of these applications demand the introduction of attribute variables. Sometimes it is handier to have atoms as constraints variables, for example when variables names are assembled from smaller elements.

In Jekejeke Minlog the end-user has the choice to use attribute variables for his constraint solver or not. There is not automatism that compiles some attribute variable behavior into the forward chaining rules. Before we have introduced attribute variables we also experimented with what we called shared variables. Shared variables would be variables from clauses that are not universally quantified and that can thus communicate with other variables from other clauses. The experiments we conducted were based on the trailed database update toolbox from Jekejeke Minlog, so that the shared variables were temporal. The experiments were successful in that we could replicate some experiments from lambda Prolog.

But we quickly figured out that shared variables cannot be directly used for the realization of constraint solvers, since they don't share the atomic property of for example coded attribute variables. Forward chaining rules over shared variables would lead to unwanted aliasing. Nevertheless shared variables are still on our agenda, together with the concept of existentially quantified variables in clauses, which we have not yet explored much. What concerns shared variables we already figured out that the coding of attribute variables can also be used to implement these. The other concept is still open, although we expect that attribute variables will also play a role here.

The experiments we have conducted so far with shared variables were mainly based on backward chaining. Backward chaining practically goes unchanged in the presence of shared variables. Only the invocation of a clause will lead to additional unifications with the shared variables. On the other hand shared variables pose interesting challenges to the forward chaining engine. It seems that they might introduce some non-determinism into the execution of forward chaining rules. We have not yet figured out the details, but we speculate that the deterministic findall/3 used in the firing of forward chaining rules needs to be replaced by the non-deterministic bagof/3. This lead is important since some logical operators are only implementable via forward chaining and not via backward chaining [11].

2.4 Test Scope

t.b.d.

Table 1: Iterations of the Test Programs

Iterations	Name	Description
1	grocery	The modified 7-11 Problem
11	pythago	Pythagorean Triples
17	queens	Eight Queens Puzzle
21	money	Letter Arithmetic Puzzle
80	crypt	Crypt Arithmetic Puzzle
273	zebra	Zebra Riddle
10	pigeon	Boolean Pigeon Hole Problem

t.b.d.

3 Available Optimizations

t.b.d.:

- **Bound Propagation:** t.b.d.
- **Refer Propagation:** t.b.d.
- **Constant Instantiation:** t.b.d.
- **Variable Instantiation:** t.b.d.

3.1 Bound Propagation

t.b.d:

Little Solver
CLP(FD)

3.2 Refer Propagation

t.b.d:

Little Solver
CLP(FD)

3.3 Constant Instantiation

t.b.d:

Little Solver
CLP(FD)

3.4 Variable Instantiation

t.b.d:

Little Solver
CLP(FD)

4 Strategies Comparison

t.b.d.:

- **Test Results:** t.b.d.

- **Discussion Bound Propagation:** In this section we will provide a critical discussion of the impact of the bound propagation optimization.

- **Discussion Refer Propagation:** In this section we will provide a critical discussion of the impact of the refer propagation optimization.

- **Discussion Constant Instantiation:** In this section we will provide a critical discussion of the impact of the constant instantiation optimization.

- **Discussion Variable Instantiation:** In this section we will provide a critical discussion of the impact of the variable instantiation optimization.

4.1 Test Results

t,.b.d.:

Operating System: Windows 7 Professional
Processor: Intel Core i7-2620M @ 2.70GHz
Memory: 4.00 GB
Energy Settings: HP Optimized

t.b.d.:

Java Version: JDK 1.7.0_25 (64-bit)
VM Parameters: -mx512m -Duser.language=en
Module Version: Jekejeke Minlog Extension Module 0.6.6
Concurrently Idle Applications: Word, Excel, Tomcat, SQL Server

t.b.d.:

Table 2: Compared Optimization Settings

Setting	Bound Prop	Refer Prop	Const Inst	Var Inst
1) None	Off	Off	Off	Off
2) Discussion Bound Propagation	On	Off	Off	Off
3) Discussion Refer Propagation	On	On	Off	Off
4) Discussion Constant Instantiation	On	On	On	Off
5) Discussion Variable Instantiation	On	On	On	On

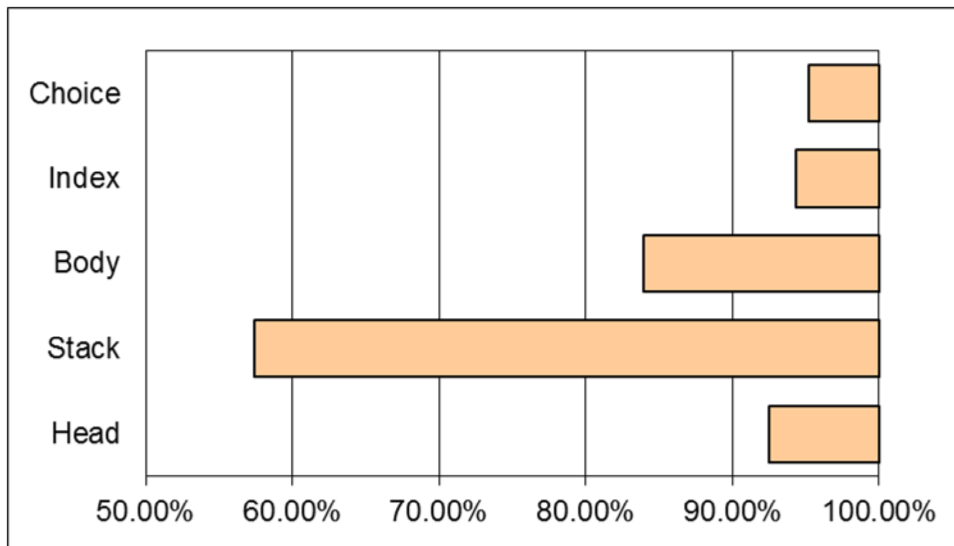
t.b.d.

The absolute raw results measured in milliseconds are displayed in the table below:

Table 3: Absolute Detailed Strategies Results (ms)

Test	1)	2)	3)	4)	5)
queens3					
money3					
einstein3					
Total					

The picture below shows the total results relative to their previous discussion:

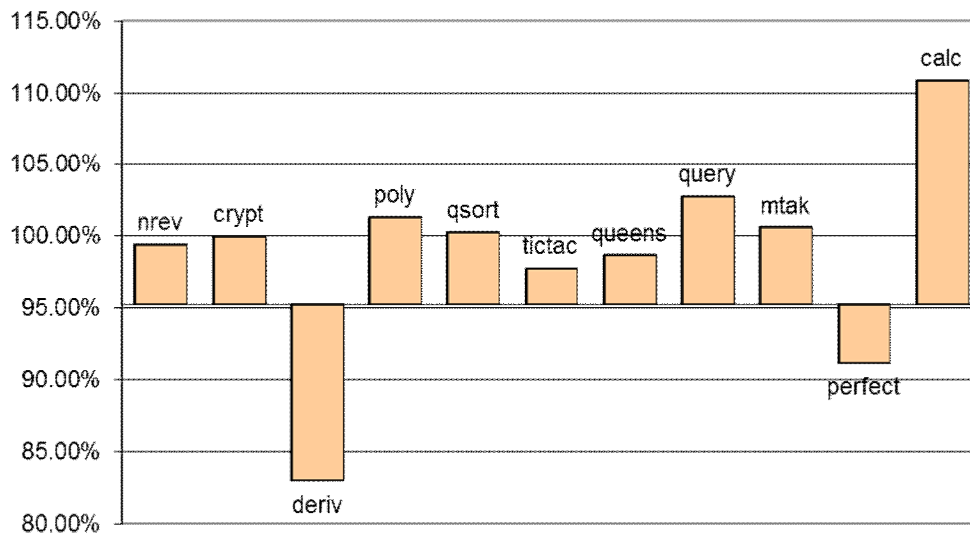


Picture 6: Incremental Relative Strategies Results

t.b.d.

4.2 Discussion Bound Propagation

t.b.d.

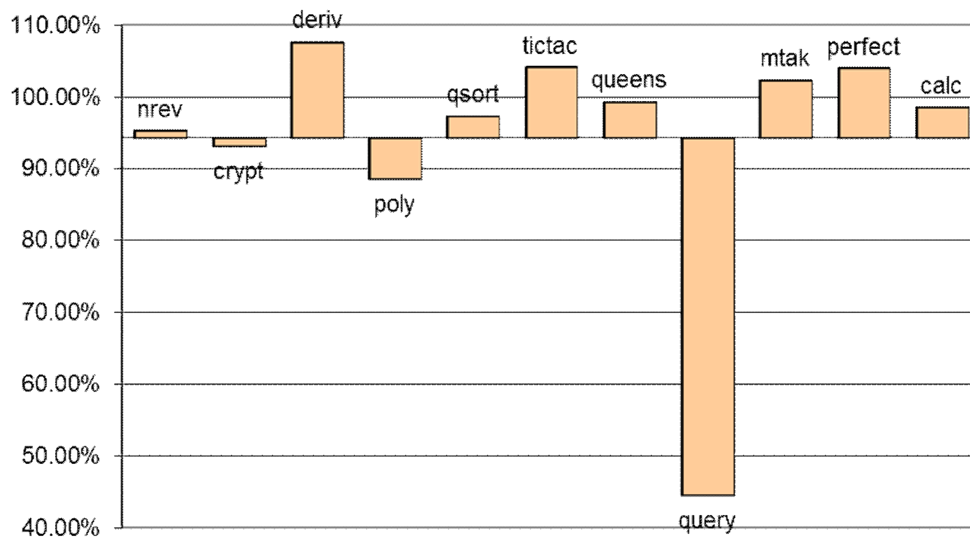


Picture 7: Bound Propagation Impact

t.b.d.

4.3 Discussion Refer Propagation

t.b.d.

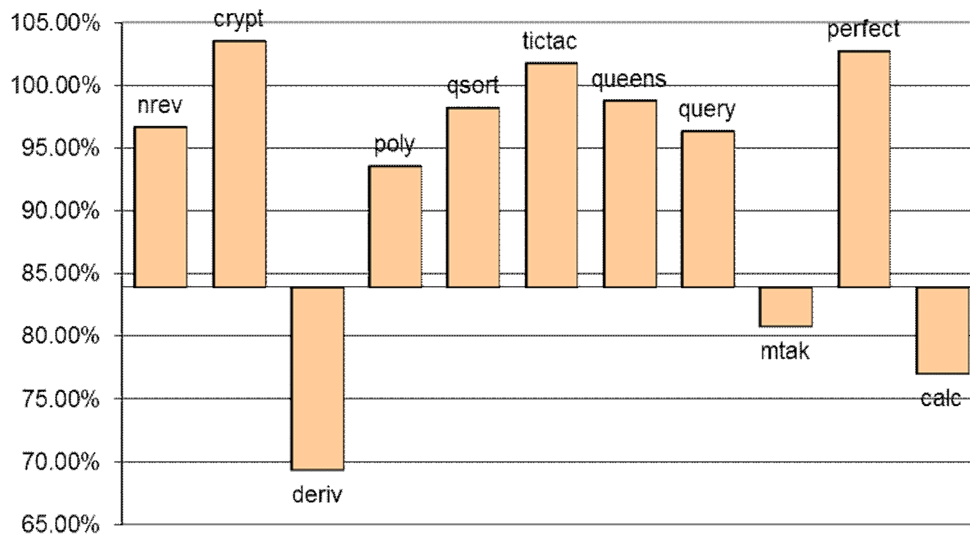


Picture 8: Refer Propagation Impact

t.b.d.

4.4 Discussion Constant Instantiation

t.b.d.

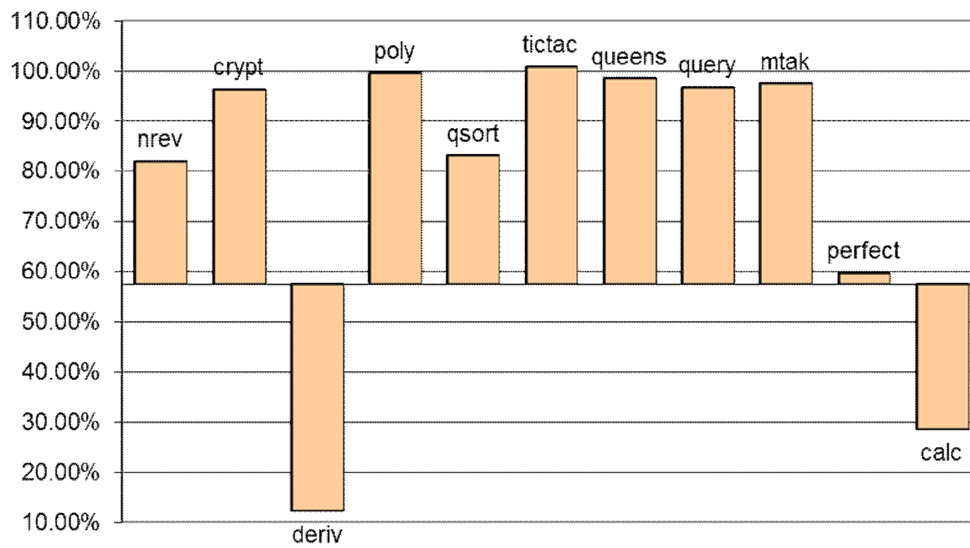


Picture 9: Constant Instantiation Impact

t.b.d.

4.5 Discussion Variable Instantiation

t.b.d.



Picture 10: Variable Instantiation Impact

t.b.d.

5 Interpreter Comparison

We conducted a couple of external performance tests. The main indicator that was measured was the elapsed time for various test cases. The test is documented along the following lines:

- **Test Results:** In this section we will present the figures that were obtained from our test runs. Figures for various Prolog systems will be given.
- **Discussion GNU Prolog:** In this section we will provide a critical discussion of the comparison with the GNU Prolog system.
- **Discussion B-Prolog Prolog:** In this section we will provide a critical discussion of the comparison with the B-Prolog Prolog system.
- **Discussion ECLiPSe Prolog:** In this section we will provide a critical discussion of the comparison with the ECLiPSe Prolog system.
- **Discussion SWI-Prolog:** In this section we will provide a critical discussion of the comparison with the SWI-Prolog system.
- **Discussion Ciao Prolog:** In this section we will provide a critical discussion of the comparison with the Ciao Prolog system.

5.1 Test Results

t.b.d:

- **GNU Prolog:** GNU Prolog 1.4.4 (64 bits)
- **B-Prolog:** B-Prolog Version 8.0 (interpreted mode)
- **ECLiPSe Prolog:** Version 6.1 #163 (x86_64_nt)
- **SWI Prolog:** SWI-Prolog (Multi-threaded, 64 bits, Version 6.5.2)
- **Ciao Prolog:** Ciao 1.15-1781 (interpreted mode)

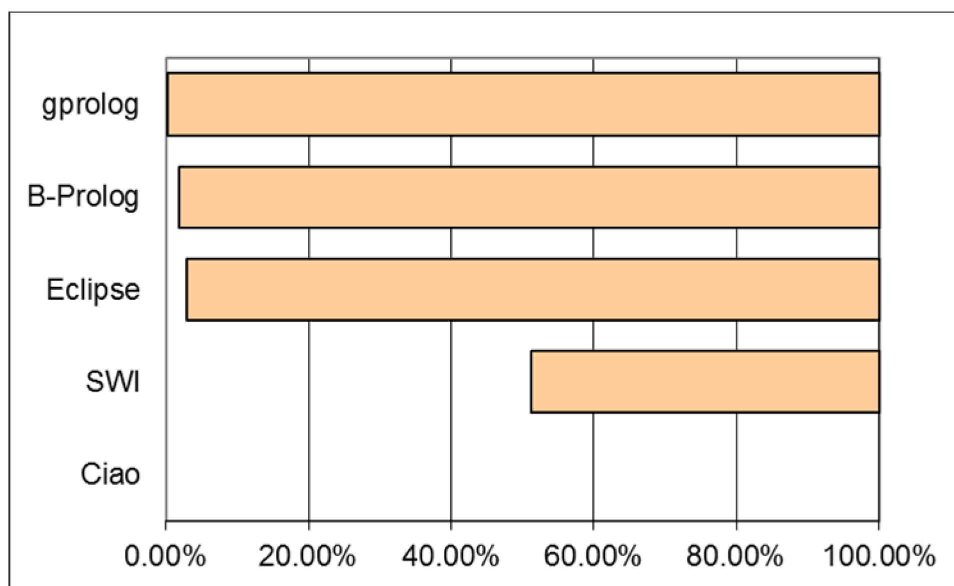
t.b.d..

The absolute raw results measured in milliseconds are displayed in the table below:

Table 4: Absolute Detailed Interpreter Results (ms)

Test	GNU	B	ECLiPSe	SWI	Ciao	Jekejeke
grocery	1	389	283	203		7'031
pythago	23	157	367	12'589		4'140
queens	11	13	102	1'155		4'198
money	1	1	2	15		4'121
crypt	2	10	26	343		4'016
zebra	6	16	33	422		4'280
pigeon	19	19	104	1'638		4'208
Total	63	605	917	16'365	-	31'994

The picture below shows the total results relative to the Jekejeke Prolog system:

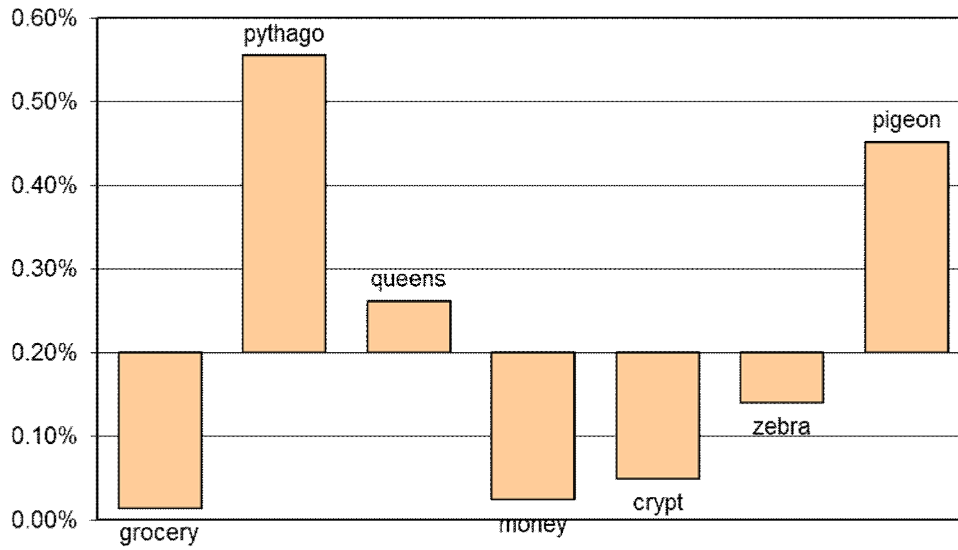


Picture 11: Relative Interpreter Results

t.b.d.

5.2 Discussion GNU Prolog

t.b.d.

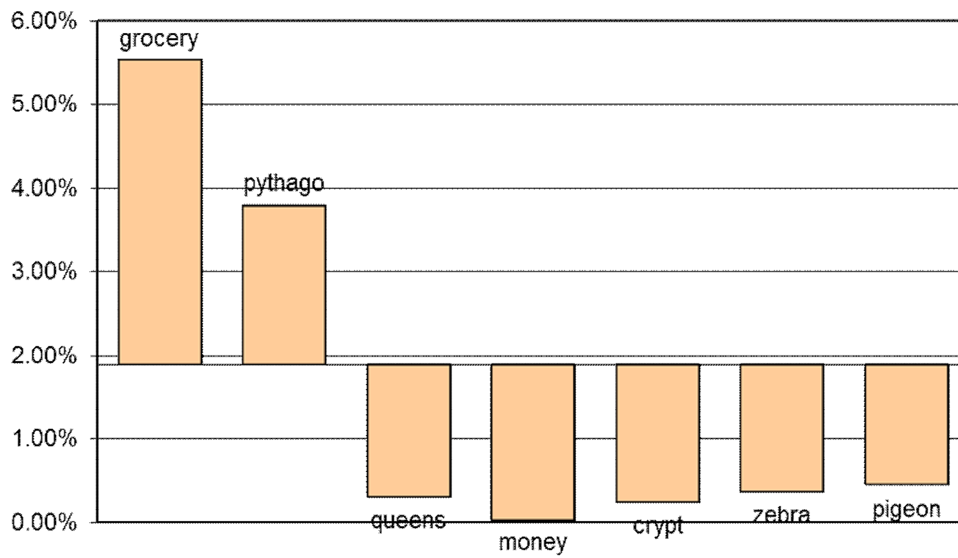


Picture 12: GNU Prolog Performance

t.b.d.

5.3 Discussion B-Prolog Prolog

t.b.d.

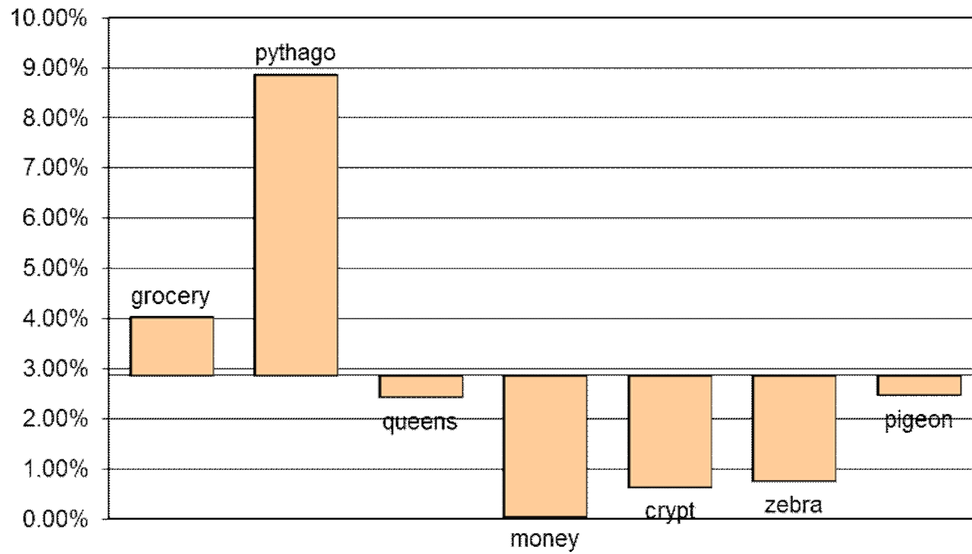


Picture 13: B-Prolog Prolog Performance

t.b.d.

5.4 Discussion ECLiPSe Prolog

t.b.d.

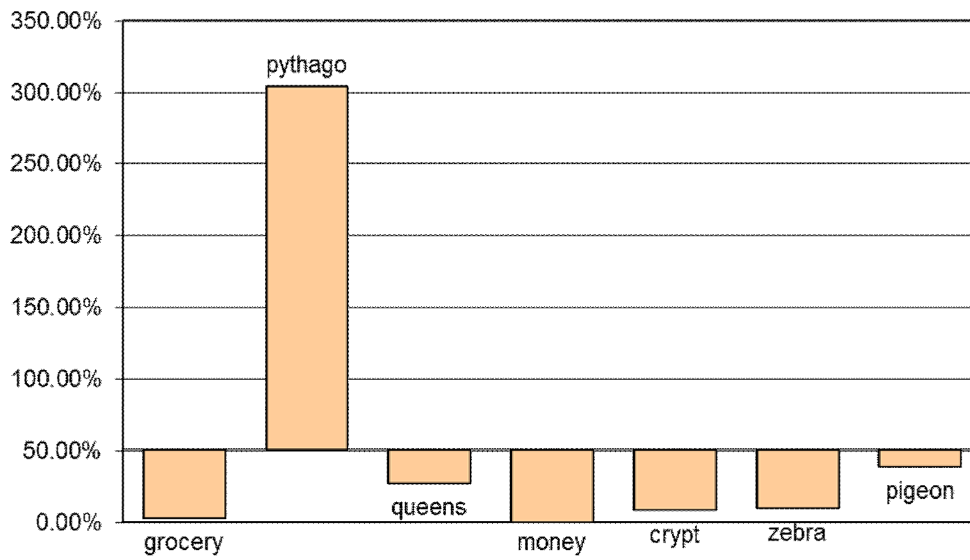


Picture 14: ECLiPSe Prolog Performance

t.b.d.

5.5 Discussion SWI-Prolog

t.b.d.

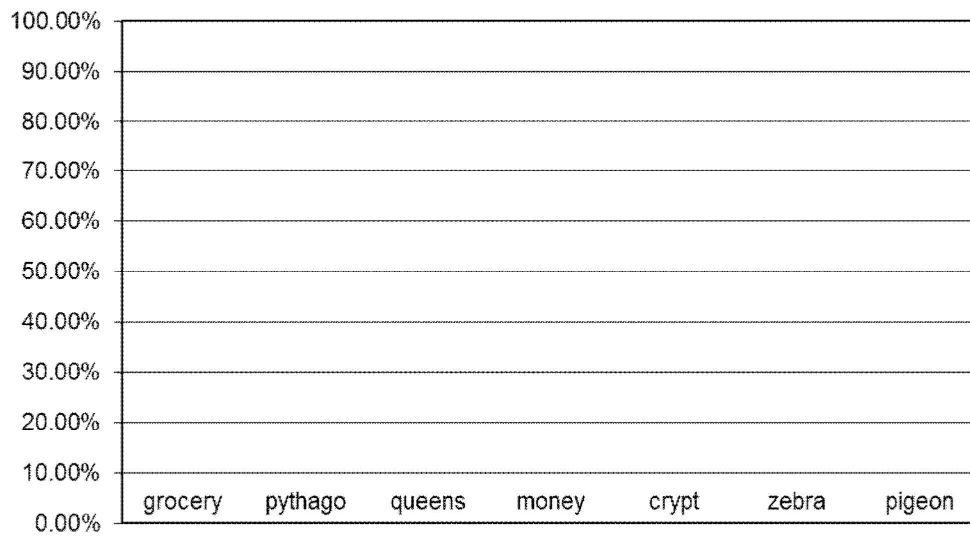


Picture 15: SWI-Prolog Performance

t.b.d.

5.6 Discussion Ciao Prolog

t.b.d.



Picture 16: Ciao Prolog Performance

t.b.d.

6 Appendix Harness Listings

The full source code of the Java classes and the Prolog texts for the test harness is given. The following source code has been included:

- Common Files
- Jekejeke Prolog Harness
- GNU Prolog Harness
- B-Prolog Harness
- ECLiPSe Prolog Harness
- SWI Prolog Harness
- Ciao Prolog Harness

6.1 Common Files

The only common file consists of a Prolog text for the invocation of the different test programs and of the measurement of the elapsed time and of the garbage collection time.

For the common files there are the following sources:

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

Prolog Text common

```
/**
 * Common Prolog code for the test harness.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

/**************************************************/
/* Utilities                                      */
/**************************************************/

for(_).
for(N) :- N > 1, M is N - 1, for(M).

test(N, X) :- for(N), X, fail.
test(_, _).

show(T, G) :-
   write('\tin '),
   write(T),
   write('\t('),
   write(G),
   write(' gc) ms'), nl.

bench(M, X, T, G) :-
   uptime(T1),
   gctime(G1),
   test(M, X),
   uptime(T2),
   gctime(G2),
   T is T2 - T1,
   G is G2 - G1,
```

```
   functor(X, F, _),
   write(F),
   show(T, G).

/**************************************************/
/* Runner                                         */
/**************************************************/

dummy.

suite4 :-
   bench(1001, dummy, _, _),
   bench(1, grocery3(_), T1, G1),
   bench(12, pythago3(_), T2, G2),
   bench(16, queens3(_), T3, G3),
   bench(35, money3(_), T4, G4),
   bench(86, crypt3(_), T5, G5),
   bench(285, zebra3(_), T6, G6),
   bench(13, pigeon3(_), T7, G7),
   T is T1+T2+T3+T4+T5+T6+T7,
   G is G1+G2+G3+G4+G5+G6+G7,
   write('Total'),
   show(T, G), nl.
```

6.2 Jekejeke Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time.

For the Jekejeke Prolog harness there are the following sources:

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

Prolog Text jekejeke

```
/**
 * Jekejeke Prolog code for the benchmark harness.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */
uptime(X) :-
   statistics(uptime, X).

gctime(X) :-
   statistics(gctime, X).

:- consult('common.p').

:- consult('grocery.p').
:- consult('pythago.p').
:- consult('queens.p').
:- consult('money.p').
:- consult('crypt.p').
:- consult('zebra.p').
:- consult('pigeon.p').

:- consult('grocery3.p').
:- consult('pythago3.p').
:- consult('queens3.p').
:- consult('money3.p').
:- consult('crypt3.p').
:- consult('zebra3.p').
:- consult('pigeon3.p').
```

6.3 GNU Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime. Replace the <base> by the test program directory. Measurement of the garbage collection time is not possible. For compatibility with our test cases we define the predicates ins/2, label/1 and all_different/2.

For the GNU Prolog harness there are the following sources:

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

Prolog Text gprolog

```
/**
 * GNU Prolog code for the test harness.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

% ?- consult('<base>\\gprolog.p').

uptime(X) :-
   real_time(X).

gctime(0).

:- op(700, xfx, ins).
:- op(100, xfx, ..).

disj_to_list(A \/ B) --> !,
   disj_to_list(A),
   disj_to_list(B).
disj_to_list(A) --> [A].

Vs ins A..B :- !, fd_domain(Vs, A, B).
Vs ins S :- phrase(disj_to_list(S), L), fd_domain(Vs, L).
label(Vs) :- fd_labelingff(Vs).
all_different(Vs) :- fd_all_different(Vs).

:- include('<base>\\common.p').

:- include('<base>\\grocery.p').
:- include('<base>\\pythago.p').
:- include('<base>\\queens.p').
:- include('<base>\\money.p').
:- include('<base>\\crypt.p').
:- include('<base>\\zebra.p').
:- include('<base>\\pigeon.p').

:- include('<base>\\grocery3.p').
:- include('<base>\\pythago3.p').
:- include('<base>\\queens3.p').
:- include('<base>\\money3.p').
:- include('<base>\\crypt3.p').
:- include('<base>\\zebra3.p').
:- include('<base>\\pigeon3.p').
```

6.4 B-Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory. For compatibility with our test cases we define the predicates ins/2, label/1 and all_different/2.

For the B-Prolog harness there are the following sources:

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

Prolog Text bprolog

```
/**
 * B-Prolog code for the test harness.
 * Since consult/1 is used code will be interpreted.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

% ?- consult('<base>\\bprolog.p').

uptime(X) :-
   statistics(runtime, [X|_]).

gctime(X) :-
   statistics(gc_time, X).

:- op(700, xfx, ins).

disj_to_list(A \/ B) --> !,
   disj_to_list(A),
   disj_to_list(B).
disj_to_list(A) --> [A].

Vs ins S :- phrase(disj_to_list(S), L), Vs :: L.
label(Vs) :- labeling(Vs).

:- consult('<base>\\common.p').

:- consult('<base>\\grocery.p').
:- consult('<base>\\pythago.p').
:- consult('<base>\\queens.p').
:- consult('<base>\\money.p').
:- consult('<base>\\crypt.p').
:- consult('<base>\\zebra.p').
:- consult('<base>\\pigeon.p').

:- consult('<base>\\grocery3.p').
:- consult('<base>\\pythago3.p').
:- consult('<base>\\queens3.p').
:- consult('<base>\\money3.p').
:- consult('<base>\\crypt3.p').
:- consult('<base>\\zebra3.p').
:- consult('<base>\\pigeon3.p').
```

6.5 ECLiPSe Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory. For compatibility with our test cases we define the predicates ins/2, label/1 and all_different/2.

For the ECLiPSe Prolog harness there are the following sources:

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

Prolog Text eclipse

```
/**
 * ECLiPSe Constraint Logic Programming System code for the test harness.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

% ?- ['<base>/eclipse.p'].

uptime(X) :-
   statistics(times, [_,_,T]),
   X is round(T*1000).

gctime(X) :-
   statistics(gc_time, T),
   X is round(T*1000).

% So that double quoted strings denote character lists.
:- lib(iso).
:- lib(ic).

:- op(700, xfx, ins).
:- op(700, xfx, in).

disj_to_list(A \/ B) --> !,
   disj_to_list(A),
   disj_to_list(B).
disj_to_list(A) --> [A].

Vs ins S :- phrase(disj_to_list(S), L), Vs :: L.
label(Vs) :- labeling(Vs).
all_different(Vs) :- alldifferent(Vs).

:- ['<base>/common.p'].

:- ['<base>/grocery.p'].
:- ['<base>/pythago.p'].
:- ['<base>/queens.p'].
:- ['<base>/money.p'].
:- ['<base>/crypt.p'].
:- ['<base>/zebra.p'].
:- ['<base>/pigeon.p'].

:- ['<base>/grocery3.p'].
:- ['<base>/pythago3.p'].
:- ['<base>/queens3.p'].
```

```
:- ['<base>/money3.p'].
:- ['<base>/crypt3.p'].
:- ['<base>/zebra3.p'].
:- ['<base>/pigeon3.p'].
```

6.6 SWI Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the SWI Prolog harness there are the following sources:

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

Prolog Text swi

```
/**
 * SWI Prolog code for the test harness.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

uptime(X) :-
   statistics(cputime, T),
   X is round(T*1000).

gctime(T) :-
   statistics(garbage_collection, [_,_,T|_]).

:- use_module(library(clpfd)).

:- consult('<base>/common.p').

:- consult('<base>/grocery.p').
:- consult('<base>/pythago.p').
:- consult('<base>/queens.p').
:- consult('<base>/money.p').
:- consult('<base>/crypt.p').
:- consult('<base>/zebra.p').
:- consult('<base>/pigeon.p').

:- consult('<base>/grocery3.p').
:- consult('<base>/pythago3.p').
:- consult('<base>/queens3.p').
:- consult('<base>/money3.p').
:- consult('<base>/crypt3.p').
:- consult('<base>/zebra3.p').
:- consult('<base>/pigeon3.p').
```

6.7 Ciao Prolog Harness

We find a Prolog text that will consult the common file, the test programs and further define the predicate for the elapsed runtime and the garbage collection time. Replace the <base> by the test program directory.

For the Ciao Prolog harness there are the following sources:

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

Prolog Text ciao

```
/**
 * Ciao Prolog code for the test harness.
 * Use consult/1 on toplevel so that code will be interpreted.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

uptime(Y) :-
   statistics(walltime, [X|_]),
   Y is round(X).

gctime(S) :-
   statistics(garbage_collection, [_,_,T]),
   S is round(T).

:- use_package(clpfd).

:- op(700, xfx, ins).
[] ins R.
[X|Y] ins R :- X in R, Y ins R.

:- include('<base>\\common.p').

:- include('<base>\\grocery.p').
:- include('<base>\\pythago.p').
:- include('<base>\\queens.p').
:- include('<base>\\money.p').
:- include('<base>\\crypt.p').
:- include('<base>\\zebra.p').
:- include('<base>\\pigeon.p').

:- include('<base>\\grocery3.p').
:- include('<base>\\pythago3.p').
:- include('<base>\\queens3.p').
:- include('<base>\\money3.p').
:- include('<base>\\crypt3.p').
:- include('<base>\\zebra3.p').
:- include('<base>\\pigeon3.p').
```

7 Appendix Test Program Listings

The full source code of the Prolog texts for the test programs is given. The following source code has been included:

- grocery Test Program
- pythago Test Program
- queens Test Program
- money Test Program
- crypt Test Program
- zebra Test Program
- pigeon Test Program

7.1 grocery Test Program

The name relates to the 7-eleven franchise of convenience stores. The franchise itself derives its name from the extended opening hours 7 a.m. until 11 p.m. One internet source attributes the problem to Doug Brumbaugh from the University of Central Florida. It must have been posted as a problem of the week as early as May 31, 2004. We adapted the problem in that we only ask for three prices instead of four, and in that we ask for a total sum of $6.42.

There is a pure Prolog and a CLP(FD) solution. The CLP(FD) solution is straight forward. The pure Prolog solution does use a special enumeration of solutions to a multiplication constraint by factorizing. Let's assume that a constant c has the following prime factors:

$$c = p_1\hat{\ }n_1 * .. * p_m\hat{\ }n_m$$

Solutions to the constraint c = x*y can be generated by assigning shares of the different prime factor exponents to the variables x and y. This is done by the predicate split/4 in the pure Prolog solution. There is still some work to do, when there are large prime factors.

Both solutions produce the same unique prices, sorted by their magnitude:

```
?- grocery(X).
X = [375, 160, 107] ;
false

?- grocery3(X).
X = [375, 160, 107] ;
false
```

For the grocery test program there are the following sources:

- **grocery.p:** The Prolog text.
- **grocery3.p:** The CLP(FD) text.

Prolog Text grocery

```
/**
 * Prolog code for the modified 7-11 problem.
 *
 * A kid goes into a grocery store and buys three items. The cashier
 * charges $6.42. The kid pays and is about to leave when the cashier
 * calls the kid back, and says "Hold on, I multiplied the three items
 * instead of adding them; I'll try again... Gosh, with adding them
 * the price still comes to $6.42"! What were the prices of the
 * three items?
 *
 * Adaptation of a problem from professor Doug Brumbaugh.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

% collect(+Integer, +Integer, -Integer, -Integer)
collect(1, _, 1, 1) :- !.
collect(N, F, A, B) :- 0 =:= N rem F, !,
```

```
   M is N // F, collect(M, F, A, C), B is F*C.
collect(N, F, A, B) :-
   G is F+1, split(N, G, A, B).

% split(+Integer, +Integer, -Integer, -Integer)
split(1, _, 1, 1) :- !.
split(N, F, A, B) :- 0 =:= N rem F,
   M is N // F, collect(M, F, A, C), B is F*C.
split(N, F, A, B) :- 0 =:= N rem F, !,
   M is N // F, split(M, F, C, B), A is F*C.
split(N, F, A, B) :-
   G is F+1, split(N, G, A, B).

% grocery(-List)
grocery(X) :-
   X = [A,B,C],
   split(6420000, 2, A, H),
   split(H, 2, B, C), A >= B,
   B >= C,
   642 =:= A+B+C.
```

CLP(FD) Text grocery3

```
/**
 * CLP(FD) code for the modified 7-11 problem.
 *
 * A kid goes into a grocery store and buys three items. The cashier
 * charges $6.42. The kid pays and is about to leave when the cashier
 * calls the kid back, and says "Hold on, I multiplied the three items
 * instead of adding them; I'll try again... Gosh, with adding them
 * the price still comes to $6.42"! What were the prices of the
 * three items?
 *
 * Adaptation of a problem from professor Doug Brumbaugh.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

% grocery3(-List)
grocery3(X) :-
   X = [A,B,C],
   X ins 0..642,
   A*B*C #= 6420000,
   A+B+C #= 642,
   A #>= B, B #>= C,
   label(X).
```

7.2 pythago Test Program

Pythagorean triples were already mentioned in Euclid's Elements. They are numbers x, y and z that form the sides of a right-angled triangle and that are integers. By the Pythagorean theorem the numbers x, y and z will satisfy the following Pythagorean equation, whereby we assume that z is the hypotenuse:

$$x^2 + y^2 = z^2$$

There is a pure Prolog and a CLP(FD) solution. The CLP(FD) solution is straight forward. The pure Prolog solution explicitly enumerates the two legs. For symmetry breaking we consider the second leg greater than the first leg. The code then uses a bisection method to see whether the sum of the squares is itself a square.

Both solutions find the same number of Pythagorean triples in the range 1 to 99:

```
?- findall(-, pythago(_), L), length(L, N).
N = 50.

?- findall(-, pythago3(_), L), length(L, N).
N = 50.
```

For the pythago test program there are the following sources:

- **pythago.p:** The Prolog text.
- **pythago3.p:** The CLP(FD) text.

Prolog Text pythago

```
/**
 * Prolog code for the pythagorean triples.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% inrange(+Integer, +Integer, -Integer)
inrange(Lo, Hi, _) :-
   Lo > Hi, !, fail.
inrange(Lo, _, Lo).
inrange(Lo, Hi, X) :-
   Lo2 is Lo+1, inrange(Lo2, Hi, X).

% bisect(+Integer, +Integer, +Integer, -Integer)
bisect(Lo, Hi, X, Y) :-
    Lo+1 < Hi, !,
    M is (Lo+Hi) // 2,
    S is M*M,
    (S > X -> bisect(Lo, M, X, Y);
     S < X -> bisect(M, Hi, X, Y);
     M = Y).
bisect(Lo, _, _, Lo).

% pythago(-List)
pythago(X) :-
```

```
   X = [A,B,C],
   inrange(1, 99, A),
   inrange(A, 99, B),
   H is A*A+B*B,
   bisect(1, H, H, C),
   C =< 99, H =:= C*C.
```

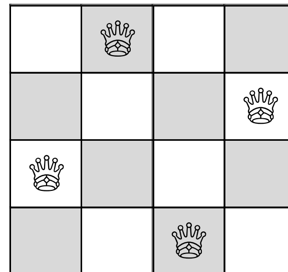
CLP(FD) Text pythago3

```
/**
 * CLP(FD) code for the pythagorean triples.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% pythago3(-List)
pythago3(X) :-
   X = [A,B,C],
   X ins 1..99,
   A*A+B*B #= C*C,
   A #=< B,
   label(X).
```

7.3 queens Test Program

The queen piece combines the power of the bishop piece and the rook piece. Our riddle asks for the placement of eight queens so that they don't attack each other. We have already introduced this problem in the benchmark for the Jekejeke Runtime. We use the same pure Prolog solution here. The solution does not represent the whole board. Only the distinct positions of the placed queens are remembered in a list, which makes the check for horizontal and vertical obsolete. The diagonal check is then implemented via arithmetic.



Picture 17: 4 Queens

The CLP(FD) solution resembles the pure Prolog solution. It uses again the integer values 1..8 to represent queen positions and imposes the horizontal and vertical conditions via a global constraint all_distinct/1 of the constraint system. The diagonal check is then implemented via the delayed arithmetic of the constraint system. The backtracking through the possible solution space if at all is then left to the constraint system.

Both solutions find the same number of solution board configurations:

```
?- findall(-, queens(_), L), length(L, N).
N = 92.

?- findall(-, queens3(_), L), length(L, N).
N = 92.
```

For the queens test program there are the following sources:

- **queens.p:** The Prolog text.
- **queens3.p:** The CLP(FD) text.

Prolog Text queens

```
/**
 * Prolog code for eight queens puzzle.
 *
 * Originally conceived in by Max Bezzel for the 8x8 checker board.
 * Used by Edsger Dijkstra to illustrate the
 * Depth-first backtracking search algorithm.
 *
 * Copyright 2010-2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.8.6 (a fast and small prolog interpreter)
 */

nodiag([], _, _).
nodiag([N|L], B, D) :-
      D =\= N - B,
```

```
      D =\= B - N,
      D1 is D + 1,
      nodiag(L, B, D1).

qdelete(L, A, A, L).
qdelete([H|T], X, A, [A|R]) :-
      qdelete(T, X, H, R).

search([], _, []).
search([H|T], History, [Q|M]) :-
      qdelete(T, Q, H, L1),
      nodiag(History, Q, 1),
      search(L1, [Q|History], M).

% queens(-List)
queens(X) :-
   search([1,2,3,4,5,6,7,8], [], X).
```

CLP(FD) Text queens3

```
/**
 * CLP(FD) Prolog code for eight queens puzzle.
 *
 * Originally conceived in by Max Bezzel for the 8x8 checker board.
 * Used by Edsger Dijkstra to illustrate the
 * Depth-first backtracking search algorithm.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% noattack_from(+List, +Variable, +Integer)
noattack_from([], _, _).
noattack_from([Y|Z], X, N) :-
   X+N #\= Y,
   Y+N #\= X,
   M is N+1,
   noattack_from(Z, X, M).

% noattack_list(+List)
noattack_list([]).
noattack_list([X|Y]) :-
   noattack_from(Y, X, 1),
   noattack_list(Y).

% queens3(-List)
queens3(X) :-
   X = [_,_,_,_,_,_,_,_],
   X ins 1..8,
   noattack_list(X),
   all_different(X),
   label(X).
```

7.4 money Test Program

There is a thin line between fun and profit in cryptography. Letter puzzles demand cryptanalysis of substitution cyphers in the form of arithmetic and logical reasoning. Our test program can be phrased as follows: A poor college student was quite frugal and sent the following telegram to his parents. How much money did he want?

```
    S E N D
  + M O R E
  ---------
  M O N E Y
```

We have already introduced this problem in the language reference for the Jekejeke Runtime library. We use the same pure Prolog solution here. The pure Prolog solution is a brute force search over the 8! possibilities. The problem was again introduced in the language reference for the Jekejeke Minlog extension. We also use the same CLP(FD) solution here. The CLP(FD) solution is a straight forward formulation of the problem. Both approaches produce the same single solution:

```
    9 5 6 7
  + 1 0 8 5
  ---------
  1 0 6 5 2
```

For the money test program there are the following sources:

- **money.p:** The Prolog text.
- **money3.p:** The CLP(FD) text.

Prolog Text money

```
/**
 * Prolog code for the money letter puzzle.
 *
 * Puzzle originally published July 1924 issue of
 * Strand Magazine by Henry Dudeney
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% oneof(+List,-Elem,-List)
oneof([X|Y], X, Y).
oneof([X|Y], Z, [X|T]) :- oneof(Y, Z, T).

% assign(-List,+List)
assign([], _).
assign([X|Y], L) :- oneof(L, X, R), assign(Y, R).

% money(-List)
money(X) :-
   X = [S,E,N,D,M,O,R,Y],
   assign(X, [0,1,2,3,4,5,6,7,8,9]),
   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.
```

CLP(FD) Text money3

```
/**
 * CLP(FD) code for the money letter puzzle.
 *
 * Puzzle originally published July 1924 issue of
 * Strand Magazine by Henry Dudeney
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% money3(-List)
money3(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).
```

7.5 crypt Test Program

The crypt riddle is a complication of the well-known SEND + MORE = MONEY riddle. Our riddle is found in W.C. Trigg's collection [1, problem 223] and it will not only involve addition but also multiplication. We have already introduced this problem in the benchmark for the Jekejeke Runtime. We use the same pure Prolog solution here. This solution is based on the graphical problem statement and implements a digit wise multiplication:

```
      O E E
        E E
    ------- *
    E O E E
    E O E
    -------
    O O E E
```

A letter E stands for an even digit. The letter O stands for an odd digit. The initial E letters are not allowed to receive the zero value. In the pure Prolog solution the corresponding generators for the digits are placed at their least possible position in the problem query so as to reduce the backtracking. The CLP(FD) solution takes another approach and uses multiplication constraints for the row multiplication. It then leaves the generator selection to the constraint system. Both approaches produce the same single solution:

```
      3 4 8
        2 8
    ------- *
    2 7 8 4
    6 9 6
    -------
    9 7 4 4
```

The crypt problem is a challenge for a CLP(FD) system since the two domains even and odd have a lot of holes, so that range bounds don't deliver much information.

For the crypt test program there are the following sources:

- **crypt.p:** The Prolog text.
- **crypt3.p:** The CLP(FD) text.

Prolog Text crypt

```
/**
 * Prolog code for the crypt riddle benchmark.
 *
 * This is problem 223 from:
 * Trigg, W. C. (1985): Mathematical Quickies,
 * Dover Publications, Inc., New York, 1985
 *
 * Copyright 2010, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.8.6 (a fast and small prolog interpreter)
 */

sum2(AL, BL, CL) :-
      sum2(AL, BL, 0, CL).
```

```
sum2([A | AL], [B | BL], Carry, [C | CL]) :-
      X is (A + B + Carry),
      C is X rem 10,
      NewCarry is X // 10,
      sum2(AL, BL, NewCarry, CL).
sum2([], BL, 0, BL) :- !.
sum2(AL, [], 0, AL) :- !.
sum2([], [B | BL], Carry, [C | CL]) :-
      X is B + Carry,
      NewCarry is X // 10,
      C is X rem 10,
      sum2([], BL, NewCarry, CL).
sum2([A | AL], [], Carry, [C | CL]) :-
      X is A + Carry,
      NewCarry is X // 10,
      C is X rem 10,
      sum2([], AL, NewCarry, CL).
sum2([], [], Carry, [Carry]).

mult(AL, D, BL) :- mult(AL, D, 0, BL).

mult([], _, Carry, [C, Cend]) :-
      C is Carry rem 10,
      Cend is Carry // 10.
mult([A | AL], D, Carry, [B | BL] ) :-
      X is A * D + Carry,
      B is X rem 10,
      NewCarry is X // 10,
      mult(AL, D, NewCarry, BL).

zero([]).
zero([0 | L]) :- zero(L).

odd(1). odd(3).
odd(5). odd(7).
odd(9).

even(0). even(2).
even(4). even(6).
even(8).

lefteven(2). lefteven(4).
lefteven(6). lefteven(8).

% crypt(-List)
crypt([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :-
      odd(A), even(B), even(C), even(E),
      mult([C, B, A], E, [I, H, G, F | X]),
      lefteven(F), odd(G), even(H), even(I), zero(X), lefteven(D),
      mult([C, B, A], D, [L, K, J | Y]),
      lefteven(J), odd(K), even(L), zero(Y),
      sum2([I, H, G, F], [0, L, K, J], [P, O, N, M | Z]),
      odd(M), odd(N), even(O), even(P), zero(Z).
```

CLP(FD) Text crypt3

```
/**
 * CLP(FD) code for the crypt riddle benchmark.
 *
 * This is problem 223 from:
```

```
 * Trigg, W. C. (1985): Mathematical Quickies,
 * Dover Publications, Inc., New York, 1985
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% crypt3(X)
crypt3(X) :-
    X = [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P],
    Odd = [A,G,K,M,N],
    Even = [B,C,E,F,H,I,D,J,L,O,P],
    Odd ins 1\/3\/5\/7\/9,
    Even ins 0\/2\/4\/6\/8,
    F #\= 0,
    D #\= 0,
    J #\= 0,
    Z #= A*100+B*10+C,
    Y #= Z*E,
    Y #= F*1000+G*100+H*10+I,
    T #= Z*D,
    T #= J*100+K*10+L,
    10*T+Y #= M*1000+N*100+O*10+P,
    label(X).
```

7.6 zebra Test Program

The Zebra riddle is an instance of so called logic grid puzzles. In these puzzles a given number of individuals hold some distinct properties from some given categories. The problem statements of the puzzle relate the properties of the individuals. The riddle then typically has a single assignment of the properties to the individuals that is consistent with the given problem statements.

In the pure Prolog solution statements about the properties of the individuals are represented as elem/2, rightTo/3 and nextTo/3 goals. The elem/3 predicate is used to express that two different properties are shared by an individual. The righto/3 and nextTo/3 predicate can be used to express a relationship of properties across neighbours. Take for example the following statement from the Zebra riddle:

> The Englishman lives in the red house.

This statement is expressed as the following pure Prolog goal:

```
elem(house(brit,_,_,_,red), Houses)
```

The CLP(FD) solution takes a little different approach. For each available property of each category a separate variable is used. The individuals are then numbered. The judgement that an individual has a certain property then corresponds to the individual's number being assigned to the properties' variable. In the end variable inequalities are used to express the statements of the riddle. For example the statement above from the Zebra riddle is expressed as follows as a CLP(FD) constraint:

```
Brit #= Red
```

Both the pure Prolog and the CLP(FD) solution can correctly figure out who owns the fish.

For the zebra test program there are the following sources:

- **zebra.p:** The Prolog text.
- **zebra3.p:** The CLP(FD) text.

Prolog Text zebra

```
/**
 * Prolog code for the Zebra riddle.
 *
 * First published as "Who owns the Zebra?" in the Life International
 * magazine on December 17, 1962 with solution given in the March 25,
 * 1963 issue. The riddle given here is not a verbatim copy.
 *
 * Copyright 2012-2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Prolog 0.9.3 (a fast and small prolog interpreter)
 */

% elem(-Elem, +List)
elem(X, [X|_]).
elem(X, [_|Y]) :-
   elem(X,Y).
```

```
% rightTo(-Elem, -Elem, +List)
rightTo(L, R, [L,R | _]).
rightTo(L, R, [_ | Rest]) :-
   rightTo(L, R, Rest).

% nextTo(-Elem, -Elem, +List)
nextTo(X, Y, List) :-
   rightTo(X, Y, List).
nextTo(X, Y, List) :-
   rightTo(Y, X, List).

% zebra(-List)
zebra(Houses) :-
   Houses = [house(norwegian,_,_,_,_),_,house(_,_,_,milk,_),_,_],
   elem(house(brit,_,_,_,red), Houses),
   elem(house(swede,dog,_,_,_), Houses),
   elem(house(dane,_,_,tea,_), Houses),
   rightTo(house(_,_,_,_,green), house(_,_,_,_,white), Houses),
   elem(house(_,_,_,coffee,green), Houses),
   elem(house(_,bird,pallmall,_,_), Houses),
   elem(house(_,_,dunhill,_,yellow), Houses),
   nextTo(house(_,_,dunhill,_,_), house(_,horse,_,_,_), Houses),
   nextTo(house(_,_,marlboro,_,_), house(_,cat,_,_,_), Houses),
   nextTo(house(_,_,marlboro,_,_), house(_,_,_,water,_), Houses),
   elem(house(_,_,winfield,beer,_), Houses),
   elem(house(german,_,rothmans,_,_), Houses),
   nextTo(house(norwegian,_,_,_,_), house(_,_,_,_,blue), Houses),
   elem(house(_,fish,_,_,_), Houses).
```

CLP(FD) Text zebra3

```
/**
 * CLP(FD) code for the Zerbra riddle.
 *
 * First published as "Who owns the Zebra?" in the Life International
 * magazine on December 17, 1962 with solution given in the March 25,
 * 1963 issue. The riddle given here is not a verbatim copy.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension)
 */

% zebra3(-List)
zebra3(Variables) :-
   Nationality = [Brit,Swede,Dane,Norwegian,German],
   Pet = [Dog,Bird,Cat,Horse,_],
   Cigarette = [Pallmall,Dunhill,Rothmans,Marlboro,Winfield],
   Drink = [Tea,Coffee,Milk,Beer,Water],
   Color = [Red,White,Green,Yellow,Blue],

   term_variables([Nationality,Pet,Cigarette,Drink,Color], Variables),
   Variables ins 1..5,

   all_different(Nationality),
   all_different(Pet),
   all_different(Cigarette),
   all_different(Drink),
   all_different(Color),

   Norwegian #= 1,
```

```
Milk #= 3,
Brit #= Red,
Swede #= Dog,
Dane #= Tea,
Green+1 #= White,
Coffee #= Green,
Bird #= Pallmall,
Dunhill #= Yellow,
Dunhill-Horse+1 #= Dist1,
Marlboro-Cat+1 #= Dist2,
Marlboro-Water+1 #= Dist3,
Winfield #= Beer,
German #= Rothmans,
Norwegian-Blue+1 #= Dist4,
[Dist1, Dist2, Dist3, Dist4] ins 0\/2,

label(Variables).
```

7.7 pigeon Test Program

The pigeonhole principle is a very basic combinatorial principle. It states that distributing n items into m boxes whereby n>m, will result in at least one box carrying two items. We use a Boolean formulation of the problem as our test case. We show that the principle holds for n=6 and m=5.

The pure Prolog solution uses a small SAT solver and generates the problem clauses via some DCG rules. The SAT solver implements a simplified variant of the Davis–Putnam algorithm. The filter/4 predicate allows to compute Φ[l=0] respectively Φ[l=1] for a clause set Φ and a literal l. Our algorithm then simply fetches always the first new literal:

```
      function sat(Φ) {
            if Φ={} then
                  return true;
            if Φ={{}, c1, .., cn} then
                  return false;
            Φ={{l, r1, .., rm}, c1, .., cn}
            return sat(Φ[l=0]) or sat(Φ[l=1]);
      }
```

The CLP(FD) solution models Boolean variables by integer variables in the range 0..1. A clause can then easily be represented as an inequality. The CPL(FD) solution leaves the choice of a solving algorithm to the constraint system. The constraint system might choose a solving algorithm that is better than our SAT solver.

For the pigeon test program there are the following sources:

- **pigeon.p:** The Prolog text.
- **pigeon3.p:** The CLP(FD) text.

Prolog Text pigeon

```
/**
 * Prolog code for the boolean pigeon hole problem.
 *
 * Clauses are respesented by assigning distinct positive
 * non-zero integers to propositional variables:
 *    x1 v .. v xn --> [x1, .. , xn]
 *    ~x           --> -x
 * State of affair is represented as:
 *    xij <=> pigeon i is placed in hole j      i in 0..n-1, j in 0..m-1
 * Clause for each pigeon that it is placed in at least one hole:
 *    xi0 v .. v xim-1       i in 0..n-1
 * Clauses for each hole that it carries maximally one pigeon:
 *    ~xij v ~xkj            i in 0..n-1, k in i+1..n-1, j in 0..m-1.
 * Should work correctly for n>=m.
 *
 * DCG used to generate the input clauses.
 * DPLL algorithm variant used to check satisfiability.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

/***********************************************************/
```

```
/* SAT Solver                                               */
/***********************************************************/

% mem(+Elem, +List)
mem(X, [X|_]).
mem(X, [_|Y]) :-
   mem(X, Y).

% sel(+Elem, +List, -List)
sel(X, [X|Y], Y).
sel(X, [Y|Z], [Y|T]) :-
   sel(X, Z, T).

% reduce(+ListOfList, +Elem, +Elem, -ListOfList)
reduce([], _, _, []).
reduce([K|F], L, M, [J|G]) :-
   sel(M, K, J), !,
   J \== [],
   reduce(F, L, M, G).
reduce([K|F], L, M, G) :-
   mem(L, K), !,
   filter(F, L, M, G).
reduce([K|F], L, M, [K|G]) :-
   filter(F, L, M, G).

% sat(+ListOfLists, -List)
sat([[L|_]|F], [L|V]):-
   M is -L,
   reduce(F, L, M, G),
   sat(G, V).
sat([[L|K]|F], [M|V]):-
   K \== [],
   M is -L,
   reduce(F, M, L, G),
   sat([K|G], V).
sat([], []).

/***********************************************************/
/* Matrix Generation                                       */
/***********************************************************/

% pair(+Integer, +Integer, -Integer)
pair(X, Y, Z) :-
   Z is (X+Y)*(X+Y+1)//2+Y.

% row(+List, -List, +Integer, +Integer)
row(X, X, _, 0).
row(L, R, I, M) :-
   M > 0, K is M-1,
   pair(I, K, H),
   X is H+1,
   row([X|L], R, I, K).

% matrix(+ListOfLists, -ListOfLists, +Integer, +Integer)
matrix(X, X, 0, _).
matrix(L, R, N, M) :-
   N > 0, K is N-1,
   row([], X, K, M),
   matrix([X|L], R, K, M).

/***********************************************************/
/* Clause Generation                                        */
```

```
/*******************************************************/

% placed(+ListOfList)
placed([]) --> [].
placed([X|Y]) --> [X], placed(Y).

% notboth(+List, +List)
notboth([], []) --> [].
notboth([X|Y], [Z|T]) -->
   {M is -X, N is -Z},
   [[M,N]],
   notboth(Y, T).

% other(+ListOfList, +List)
other([], _) --> [].
other([X|Y], Z) -->
   notboth(X, Z),
   other(Y, Z).

% carries(+ListOfList)
carries([]) --> [].
carries([X|Y]) -->
   other(Y, X),
   carries(Y).

% pigeon(+ListOfList)
pigeon(V) :-
   matrix([], X, 6, 5),
   phrase((placed(X),carries(X)), F),
   sat(F, V).
```

CLP(FD) Text pigeon3

```
/**
 * CLP(FD) code for the boolean pigeon hole problem.
 *
 * Clauses can be represented as CLP(FD) as follows:
 *    x1 v .. v xn :<=> x1+..+xn #> 0.
 *    ~x := 1-x
 * State of affair is represented as:
 *    xij <=> pigeon i is placed in hole j     i in 0..n-1, j in 0..m-1
 * Clause for each pigeon that it is placed in at least one hole:
 *    xi0 v .. v xim-1       i in 0..n-1
 * Clauses for each hole that it carries maximally one pigeon:
 *    ~xij v ~xkj            i in 0..n-1, k in i+1..n-1, j in 0..m-1.
 * Should work correctly for n>=m.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

/*******************************************************/
/* Matrice Generation                                  */
/*******************************************************/

% size(-List, +Integer)
size([], 0).
size([_|Y], N) :-
   N > 0,
   M is N-1,
```

```
   size(Y, M).

% dimension(+ListOfList, +Integer)
dimension([], _).
dimension([X|Y], N) :-
   size(X, N),
   dimension(Y, N).

% matrice(-ListOfList, +Integer, +Integer)
matrice(X, N, M) :-
   size(X, N),
   dimension(X, M).

/**********************************************************/
/* Constraint Generation                                  */
/**********************************************************/

% sum(+List, -Sum)
sum([X], X).
sum([X,Y|Z], X+T) :-
   sum([Y|Z], T).

% placed(+ListOfList)
placed([]).
placed([X|Y]) :-
   sum(X, T), T #> 0,
   placed(Y).

% notboth(+List, +List)
notboth([], []).
notboth([X|Y], [Z|T]) :-
   2-X-Z #> 0,
   notboth(Y, T).

% other(+ListOfList, +List)
other([], _).
other([X|Y], Z) :-
   notboth(X, Z),
   other(Y, Z).

% carries(+ListOfList)
carries([]).
carries([X|Y]) :-
   other(Y, X),
   carries(Y).

% pigeon3(+ListOfList)
pigeon3(X) :-
   matrice(X, 6, 5),
   term_variables(X, L),
   L ins 0..1,
   placed(X),
   carries(X),
   label(L).
```

8 Appendix Example Program Listings

The full source code of the Prolog texts for the example programs is given. The following source code has been included:

- add Example Program
- addensure Example Program

8.1 add Example Program

For the add example program there are the following sources:

- **refer.p:** The union find Prolog text.
- **add.p:** The union find and add constraint Prolog text.

Prolog Text refer

```
/**
 * Prolog code for the little solver.
 * With a union find element.
 *
 * Copyright 2012-2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6 (minimal logic extension module)
 */

/*******************************************************/
/* Little Solver with Union Find                      */
/*******************************************************/

% refer(+Atom, +Atom)
% refer(X, Y) == X = Y
:- forward refer/2.

unit &:-
   &- refer(X, X), !.
refer(Z, Y) &:-
   &- refer(X, Y) && refer(X, Z), !.
refer(X, Z) &:-
   &- refer(X, Y) && refer(Y, Z), !.

% bound(+Atom, +Elem)
% bound(X, C) == X = C
:- forward bound/2.

bound(Y, C) &:-
   &- bound(X, C) && refer(X, Y), !.
zero &:-
   &- bound(X, C) && bound(X, D), C =\= D, !.
unit &:-
   &- bound(X, _) && bound(X, _), !.
bound(Y, C) &:-
   refer(X, Y) && &- bound(X, C).

% domain(+Atom, +List)
% domain(X, A) == X in A
```

```
:- forward domain/2.

zero &:-
   domain(_, []), !.
bound(X, Y) &:-
   &- domain(X, [Y]), !.
domain(Y, A) &:-
   &- domain(X, A) && refer(X, Y), !.
zero &:-
   &- domain(X, A) && bound(X, C), \+ member(C, A), !.
unit &:-
   &- domain(X, _) && bound(X, _), !.
domain(X, C) &:-
   &- domain(X, B) && &- domain(X, A), intersect(A, B, C), C \== A, !.
unit &:-
   &- domain(X, _) && domain(X, _), !.
domain(Y, A) &:-
   refer(X, Y) && &- domain(X, A).
zero &:-
   bound(X, Y) && &- domain(X, A), \+ member(Y, A), !.
unit &:-
   bound(X, _) && &- domain(X, _).

/*****************************************************/
/* List Processing                                   */
/*****************************************************/

% member(+Elem, +List)
member(X, [X|_]).
member(X, [_|Y]) :-
   member(X, Y).

% intersect(+List, +List, -List)
intersect([], _, []).
intersect([X|Y], Z, [X|T]) :-
   member(X, Z), !, intersect(Y, Z, T).
intersect([_|Y], Z, T) :-
   intersect(Y, Z, T).

/*****************************************************/
/* Test Cases                                        */
/*****************************************************/

% ?- post(domain(x,[1])), posted.
% bound(x, 1).
% Yes

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

% ?- post(domain(x,[1,2,3])), post(domain(x,[2,3,4])), posted.
% domain(x, [2,3]).
% Yes

% ?- post(domain(x,[2,3])), post(domain(x,[2,3,4])), posted.
% domain(x, [2,3]).
% Yes

% ?- post(domain(x,[1,2,3])), post(domain(x,[2,3])), posted.
% domain(x, [2,3]).
```

```
% Yes

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

% ?- post(domain(x,[1,2,3])), post(bound(x,2)), posted.
% bound(x, 2).
% Yes

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

% ?- post(bound(x,2)), post(domain(x,[1,2,3])), posted.
% bound(x, 2).
% Yes

% ?- post(domain(x,[1,2,3])), post(refer(x,y)), post(domain(y,[2,3,4])),
posted.
% refer(x, y).
% domain(y, [2,3]).
% Yes

% ?- post(domain(x,[1,2,3])), post(domain(y,[2,3,4])), post(refer(x,y)),
posted.
% refer(x, y).
% domain(y, [2,3]).
```

Prolog Text add

```
/**
 * Prolog code for the little solver.
 * With a union find element.
 * With add constraints.
 *
 * Copyright 2012-2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6 (minimal logic extension module)
 */

/*******************************************************/
/* Little Solver with Union Find                       */
/*******************************************************/

% refer(+Atom, +Atom)
% refer(X, Y) == X = Y
:- forward refer/2.

unit &:-
   &- refer(X, X), !.
refer(Z, Y) &:-
   &- refer(X, Y) && refer(X, Z), !.
refer(X, Z) &:-
   &- refer(X, Y) && refer(Y, Z), !.

% bound(+Atom, +Elem)
% bound(X, C) == X = C
:- forward bound/2.

bound(Y, C) &:-
```

```
   &- bound(X, C) && refer(X, Y), !.
zero &:-
   &- bound(X, C) && bound(X, D), C =\= D, !.
unit &:-
   &- bound(X, _) && bound(X, _), !.
bound(Y, C) &:-
   refer(X, Y) && &- bound(X, C).

% domain(+Atom, +List)
% domain(X, A) == X in A
:- forward domain/2.

zero &:-
   domain(_, []), !.
bound(X, Y) &:-
   &- domain(X, [Y]), !.
domain(Y, A) &:-
   &- domain(X, A) && refer(X, Y), !.
zero &:-
   &- domain(X, A) && bound(X, C), \+ member(C, A), !.
unit &:-
   &- domain(X, _) && bound(X, _), !.
domain(X, C) &:-
   &- domain(X, B) && &- domain(X, A), intersect(A, B, C), C \== A, !.
unit &:-
   &- domain(X, _) && domain(X, _), !.
domain(Y, A) &:-
   refer(X, Y) && &- domain(X, A).
zero &:-
   bound(X, Y) && &- domain(X, A), \+ member(Y, A), !.
unit &:-
   bound(X, _) && &- domain(X, _).

/*******************************************************/
/* Add Constraints                                     */
/*******************************************************/

% addvv(+Atom, +Atom, +Atom)
% addvv(X, Y, Z) == X+Y = Z
:- forward addvv/3.
bound(X, 0) &:-
   &- addvv(X, Y, Y), !.
bound(Y, 0) &:-
   &- addvv(X, Y, X), !.
addvv(T, Y, Z) &:-
   &- addvv(X, Y, Z) && refer(X, T), !.
addvv(X, T, Z) &:-
   &- addvv(X, Y, Z) && refer(Y, T), !.
addvv(X, Y, T) &:-
   &- addvv(X, Y, Z) && refer(Z, T), !.
addcv(Y, C, Z) &:-
   &- addvv(X, Y, Z) && bound(X, C), !.
addcv(X, C, Z) &:-
   &- addvv(X, Y, Z) && bound(Y, C), !.
addvc(X, Y, C) &:-
   &- addvv(X, Y, Z) && bound(Z, C), !.
addvv(T, Y, Z) &:-
   refer(X, T) && &- addvv(X, Y, Z).
addvv(X, T, Z) &:-
   refer(Y, T) && &- addvv(X, Y, Z), X \== Y.
addvv(X, Y, T) &:-
   refer(Z, T) && &- addvv(X, Y, Z), X \== Z, Y \== Z.
```

```
addcv(Y, C, Z) &:-
   bound(X, C) && &- addvv(X, Y, Z).
addcv(X, C, Z) &:-
   bound(Y, C) && &- addvv(X, Y, Z), X \== Y.
addvc(X, Y, C) &:-
   bound(Z, C) && &- addvv(X, Y, Z), X \== Z, Y \== Z.

% addcv(+Atom, +Elem, +Atom)
% addcv(X, C, Y) == X+C = Y
:- forward addcv/3.
refer(X, Y) &:-
   &- addcv(X, 0, Y), !.
zero &:-
   &- addcv(X, _, X), !.
addcv(Z, C, Y) &:-
   &- addcv(X, C, Y) && refer(X, Z), !.
addcv(X, C, Z) &:-
   &- addcv(X, C, Y) && refer(Y, Z), !.
bound(Y, E) &:-
   &- addcv(X, C, Y) && bound(X, D), !, E is D+C.
bound(X, E) &:-
   &- addcv(X, C, Y) && bound(Y, D), !, E is D-C.
addcv(Z, C, Y) &:-
   refer(X, Z) && &- addcv(X, C, Y).
addcv(X, C, Z) &:-
   refer(Y, Z) && &- addcv(X, C, Y), X \== Y.
bound(Y, E) &:-
   bound(X, D) && &- addcv(X, C, Y), E is D+C.
bound(X, E) &:-
   bound(Y, D) && &- addcv(X, C, Y), X \== Y, E is D-C.

% addvc(+Atom, +Atom, +Elem)
% addvc(X, Y, C) == X+Y = C
:- forward addvc/3.
addvc(Z, Y, C) &:-
   &- addvc(X, Y, C) && refer(X, Z), !.
addvc(X, Z, C) &:-
   &- addvc(X, Y, C) && refer(Y, Z), !.
bound(Y, E) &:-
   &- addvc(X, Y, C) && bound(X, D), !, E is C-D.
bound(X, E) &:-
   &- addvc(X, Y, C) && bound(Y, D), !, E is C-D.
addvc(Z, Y, C) &:-
   refer(X, Z) && &- addvc(X, Y, C).
addvc(X, Z, C) &:-
   refer(Y, Z) && &- addvc(X, Y, C), X \== Y.
bound(Y, E) &:-
   bound(X, D) && &- addvc(X, Y, C), E is C-D.
bound(X, E) &:-
   bound(Y, D) && &- addvc(X, Y, C), X \== Y, E is C-D.

/*****************************************************/
/* List Processing                                   */
/*****************************************************/

% member(+Elem, +List)
member(X, [X|_]).
member(X, [_|Y]) :-
   member(X, Y).

% intersect(+List, +List, -List)
intersect([], _, []).
```

```
intersect([X|Y], Z, [X|T]) :-
   member(X, Z), !, intersect(Y, Z, T).
intersect([_|Y], Z, T) :-
   intersect(Y, Z, T).

/*******************************************************/
/* Test Cases I                                        */
/*******************************************************/

% ?- post(domain(x,[1])), posted.
% bound(x, 1).
% Yes

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

% ?- post(domain(x,[1,2,3])), post(domain(x,[2,3,4])), posted.
% domain(x, [2,3]).
% Yes

% ?- post(domain(x,[2,3])), post(domain(x,[2,3,4])), posted.
% domain(x, [2,3]).
% Yes

% ?- post(domain(x,[1,2,3])), post(domain(x,[2,3])), posted.
% domain(x, [2,3]).
% Yes

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

% ?- post(domain(x,[1,2,3])), post(bound(x,2)), posted.
% bound(x, 2).
% Yes

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

% ?- post(bound(x,2)), post(domain(x,[1,2,3])), posted.
% bound(x, 2).
% Yes

% ?- post(domain(x,[1,2,3])), post(refer(x,y)), post(domain(y,[2,3,4])),
posted.
% refer(x, y).
% domain(y, [2,3]).
% Yes

% ?- post(domain(x,[1,2,3])), post(domain(y,[2,3,4])), post(refer(x,y)),
posted.
% refer(x, y).
% domain(y, [2,3]).
% Yes

/*******************************************************/
/* Test Cases II                                       */
/*******************************************************/

% ?- post(addvv(x,y,z)), post(bound(x,1)), post(bound(y,2)), posted.
% bound(x, 1).
```

```
% bound(y, 2).
% bound(z, 3).
% Yes

% ?- post(bound(x,1)), post(addvv(x,y,z)), post(bound(y,2)), posted.
% bound(x, 1).
% bound(y, 2).
% bound(z, 3).
% Yes

% ?- post(bound(x,1)), post(bound(y,2)), post(addvv(x,y,z)), posted.
% bound(x, 1).
% bound(y, 2).
% bound(z, 3).
% Yes

% ?- post(addvv(x,y,z)), post(refer(z,x)), posted.
% refer(z, x).
% bound(y, 0).
% Yes

% ?- post(refer(z,x)), post(addvv(x,y,z)), posted.
% refer(z, x).
% bound(y, 0).
% Yes

% ?- post(addvc(x,y,3)), post(addvc(y,z,5)), post(addvc(x,z,6)),
post(bound(y,1)), posted.
% bound(y, 1).
% bound(z, 4).
% bound(x, 2).
% Yes

% ?- post(addvc(x,y,3)), post(addvc(y,z,5)), post(addvc(x,z,6)),
post(bound(y,2)), posted.
% No
```

8.2 addensure Example Program

For the pigeon test program there are the following sources:

- **referensure.p:** The union find Prolog text.
- **addensure.p:** The union find and add constraint Prolog text.

Prolog Text referensure

```
/**
 * Prolog code for the little solver.
 * With a union find element.
 * With user-friendly external representation.
 * With native prolog variables and unification.
 *
 * Copyright 2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6.6 (minimal logic extension module)
 */

:- op(700, xfx, ∈). /* Unicode 0x2208 */

/*******************************************************/
/* Post Extensions                                     */
/*******************************************************/

% post(+Event)
:- multifile post/1.
^ post(attr_bound(R, T)) :- !,
     sys_melt_var(R, T).
^ post(attr_refer(R, S)) :- !,
     sys_melt_var(R, T),
     sys_melt_var(S, T).

% hook(+Var, +Term)
hook(V, W) :- var(W), !,
   sys_freeze_var(V, R),
   sys_freeze_var(W, S),
   post(refer(R, S)).
hook(V, T) :- integer(T), !,
   sys_freeze_var(V, R),
   post(bound(R, T)).
hook(_, T) :-
   sys_throw_error(type_error(integer, T)).

/*******************************************************/
/* Little Solver with Union Find                       */
/*******************************************************/

% refer(+Ref, +Ref)
% refer(X, Y) == X = Y
:- forward refer/2.

unit &:-
   &- refer(_, _).

% bound(+Ref, +Elem)
% bound(X, C) == X = C
:- forward bound/2.
```

```
unit &:-
   &- bound(_, _).

% domain(+Ref, +List)
% domain(X, A) == X in A
:- forward domain/2.

zero &:-
   domain(_, []), !.
attr_bound(X, Y) &:-
   &- domain(X, [Y]), !.
domain(Y, A) &:-
   &- domain(X, A) && sys_bound_var(X), sys_melt_var(X, H), var(H), !,
sys_freeze_var(H, Y).
zero &:-
   &- domain(X, A) && sys_bound_var(X), sys_melt_var(X, C), integer(C), \+
member(C, A), !.
unit &:-
   &- domain(X, _) && sys_bound_var(X), sys_melt_var(X, C), integer(C), !.
domain(X, C) &:-
   &- domain(X, B) && &- domain(X, A), intersect(A, B, C), C \== A, !.
unit &:-
   &- domain(X, _) && domain(X, _), !.
domain(Y, A) &:-
   refer(X, Y) && &- domain(X, A).
zero &:-
   bound(X, Y) && &- domain(X, A), \+ member(Y, A), !.
unit &:-
   bound(X, _) && &- domain(X, _).

/*****************************************************/
/* External Representation                           */
/*****************************************************/

% ensure_attr(+Var)
ensure_attr(X) :- sys_attr(X), !.
ensure_attr(X) :- sys_new_attr(X).

% ensure_hook(+Freezer)
ensure_hook(R) :- sys_clause_hook(R, hook, _), !.
ensure_hook(R) :- sys_compile_hook(hook, C), assume_hook(R, C).

% +Expr ∈ +Set
X ∈ {Y} :- var(X), !,
   set_to_list(Y, A),
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, B),
   post(domain(B, A)).
X ∈ {Y} :- integer(X), !,
   set_to_list(Y, A),
   member(X, A), !.
X ∈ {_} :-
   sys_throw_error(type_error(integer, X)).

% set_to_list(+Set, -List)
set_to_list(X, _) :- var(X),
   sys_throw_error(instantiation_error).
set_to_list((X,Y), C) :- !,
   set_to_list(X, A),
```

```
   set_to_list(Y, B),
   union(A, B, C).
set_to_list(X, [X]) :- integer(X), !.
set_to_list(X, _) :- var(X),
   sys_throw_error(type_error(integer, X)).

:- multifile sys_pretty_event/1.

% sys_pretty_event(-Goal)
sys_pretty_event(X ∈ {Y}) :-
   domain(B, A),
   list_to_set(A, Y),
   sys_melt_var(B, X).

% list_to_set(+List, -Set)
list_to_set([X,Y|T], (X,C)) :- !,
   list_to_set([Y|T], C).
list_to_set([X], X).

/*******************************************************/
/* List Processing                                     */
/*******************************************************/

% member(+Elem, +List)
member(X, [X|_]).
member(X, [_|Y]) :-
   member(X, Y).

% intersect(+List, +List, -List)
intersect([], _, []).
intersect([X|Y], Z, [X|T]) :-
   member(X, Z), !, intersect(Y, Z, T).
intersect([_|Y], Z, T) :-
   intersect(Y, Z, T).

% union(+List, +List, -List)
union([], X, X).
union([X|Y], Z, T) :-
   member(X, Z), !, union(Y, Z, T).
union([X|Y], Z, [X|T]) :-
   union(Y, Z, T).

/*******************************************************/
/* Test Cases I                                        */
/*******************************************************/

% ?- X ∈ {1}, stored.
% X = 1

% ?- X ∈ {1,2,3}, Y ∈ {2,3,4}, stored.
% X ∈ {1,2,3}.
% Y ∈ {2,3,4}.
% Yes

% ?- X ∈ {1,2,3}, X ∈ {2,3,4}, stored.
% X ∈ {2,3}.
% Yes

% ?- X ∈ {2,3}, X ∈ {2,3,4}, stored.
% X ∈ {2,3}.
% Yes
```

```
% ?- X ∈ {1,2,3}, X ∈ {2,3}, stored.
% X ∈ {2,3}.
% Yes

% ?- X ∈ {1,2,3}, X = 4, stored.
% No

% ?- X ∈ {1,2,3}, X = 2, stored.
% X = 2

% ?- X = 4, X ∈ {1,2,3}, stored.
% No

% ?- X = 2, X ∈ {1,2,3}, stored.
% X = 2

% ?- X ∈ {1,2,3}, X = Y, Y ∈ {2,3,4}, stored.
% X ∈ {2,3}.
% Y = X

% ?- X ∈ {1,2,3}, Y ∈ {2,3,4}, X = Y, stored.
% X ∈ {2,3}.
% Y = X

% ?- X = a, X ∈ {1,2,3}, stored.
% Error: Argument should be an integer, found a.

% ?- X ∈ {1,2,3}, X = a, stored.
% Error: Argument should be an integer, found a.
```

Prolog Text addensure

```
/**
 * Prolog code for the little solver.
 * With a union find element.
 * With add constraints.
 * With a union find element.
 * With native prolog variables and unification.
 *
 * Copyright 2012-2013, XLOG Technologies GmbH, Switzerland
 * Jekejeke Minlog 0.6 (minimal logic extension module)
 */

:- op(700, xfx, ≡). /* Unicode 0x2261 */
:- op(700, xfx, ∈). /* Unicode 0x2208 */

/*********************************************************/
/* Post Extensions                                       */
/*********************************************************/

% post(+Event)
:- multifile post/1.
^ post(attr_bound(R, T)) :- !,
     sys_melt_var(R, T).
^ post(attr_refer(R, S)) :- !,
     sys_melt_var(R, T),
```

```
     sys_melt_var(S, T).

% hook(+Var, +Term)
hook(V, W) :- var(W), !,
   sys_freeze_var(V, R),
   sys_freeze_var(W, S),
   post(refer(R, S)).
hook(V, T) :- integer(T), !,
   sys_freeze_var(V, R),
   post(bound(R, T)).
hook(_, T) :-
   sys_throw_error(type_error(integer, T)).

/**********************************************************/
/* Little Solver with Union Find                          */
/**********************************************************/

% refer(+Ref, +Ref)
% refer(X, Y) == X = Y
:- forward refer/2.

unit &:-
   &- refer(_, _).

% bound(+Ref, +Elem)
% bound(X, C) == X = C
:- forward bound/2.

unit &:-
   &- bound(_, _).

% domain(+Ref, +List)
% domain(X, A) == X in A
:- forward domain/2.

zero &:-
   domain(_, []), !.
attr_bound(X, Y) &:-
   &- domain(X, [Y]), !.
domain(Y, A) &:-
   &- domain(X, A) && sys_bound_var(X), sys_melt_var(X, H), var(H), !,
sys_freeze_var(H, Y).
zero &:-
   &- domain(X, A) && sys_bound_var(X), sys_melt_var(X, C), integer(C), \+
member(C, A), !.
unit &:-
   &- domain(X, _) && sys_bound_var(X), sys_melt_var(X, C), integer(C), !.
domain(X, C) &:-
   &- domain(X, B) && &- domain(X, A), intersect(A, B, C), C \== A, !.
unit &:-
   &- domain(X, _) && domain(X, _), !.
domain(Y, A) &:-
   refer(X, Y) && &- domain(X, A).
zero &:-
   bound(X, Y) && &- domain(X, A), \+ member(Y, A), !.
unit &:-
   bound(X, _) && &- domain(X, _).

/**********************************************************/
/* Add Constraints                                        */
/**********************************************************/
```

```
% addvv(+Atom, +Atom, +Atom)
% addvv(X, Y, Z) == X+Y = Z
:- forward addvv/3.
attr_bound(X, 0) &:-
   &- addvv(X, Y, Y), !.
attr_bound(Y, 0) &:-
   &- addvv(X, Y, X), !.
addvv(T, Y, Z) &:-
   &- addvv(X, Y, Z) && sys_bound_var(X), sys_melt_var(X, H), var(H), !,
sys_freeze_var(H, T).
addvv(X, T, Z) &:-
   &- addvv(X, Y, Z) && sys_bound_var(Y), sys_melt_var(Y, H), var(H), !,
sys_freeze_var(H, T).
addvv(X, Y, T) &:-
   &- addvv(X, Y, Z) && sys_bound_var(Z), sys_melt_var(Z, H), var(H), !,
sys_freeze_var(H, T).
addcv(Y, C, Z) &:-
   &- addvv(X, Y, Z) && sys_bound_var(X), sys_melt_var(X, C), integer(C),
!.
addcv(X, C, Z) &:-
   &- addvv(X, Y, Z) && sys_bound_var(Y), sys_melt_var(Y, C), integer(C),
!.
addvc(X, Y, C) &:-
   &- addvv(X, Y, Z) && sys_bound_var(Z), sys_melt_var(Z, C), integer(C),
!.
addvv(T, Y, Z) &:-
   refer(X, T) && &- addvv(X, Y, Z).
addvv(X, T, Z) &:-
   refer(Y, T) && &- addvv(X, Y, Z), X \== Y.
addvv(X, Y, T) &:-
   refer(Z, T) && &- addvv(X, Y, Z), X \== Z, Y \== Z.
addcv(Y, C, Z) &:-
   bound(X, C) && &- addvv(X, Y, Z).
addcv(X, C, Z) &:-
   bound(Y, C) && &- addvv(X, Y, Z), X \== Y.
addvc(X, Y, C) &:-
   bound(Z, C) && &- addvv(X, Y, Z), X \== Z, Y \== Z.

% addcv(+Atom, +Elem, +Atom)
% addcv(X, C, Y) == X+C = Y
:- forward addcv/3.
attr_refer(X, Y) &:-
   &- addcv(X, 0, Y), !.
zero &:-
   &- addcv(X, _, X), !.
addcv(Z, C, Y) &:-
   &- addcv(X, C, Y) && sys_bound_var(X), sys_melt_var(X, H), var(H), !,
sys_freeze_var(H, Z).
addcv(X, C, Z) &:-
   &- addcv(X, C, Y) && sys_bound_var(Y), sys_melt_var(Y, H), var(H), !,
sys_freeze_var(H, Z).
attr_bound(Y, E) &:-
   &- addcv(X, C, Y) && sys_bound_var(X), sys_melt_var(X, D), integer(D),
!, E is D+C.
attr_bound(X, E) &:-
   &- addcv(X, C, Y) && sys_bound_var(Y), sys_melt_var(Y, D), integer(D),
!, E is D-C.
addcv(Z, C, Y) &:-
   refer(X, Z) && &- addcv(X, C, Y).
addcv(X, C, Z) &:-
   refer(Y, Z) && &- addcv(X, C, Y), X \== Y.
attr_bound(Y, E) &:-
```

```
   bound(X, D) && &- addcv(X, C, Y), E is D+C.
attr_bound(X, E) &:-
   bound(Y, D) && &- addcv(X, C, Y), X \== Y, E is D-C.

% addvc(+Atom, +Atom, +Elem)
% addvc(X, Y, C) == X+Y = C
:- forward addvc/3.
addvc(Z, Y, C) &:-
   &- addvc(X, Y, C) && sys_bound_var(X), sys_melt_var(X, H), var(H), !,
sys_freeze_var(H, Z).
addvc(X, Z, C) &:-
   &- addvc(X, Y, C) && sys_bound_var(Y), sys_melt_var(Y, H), var(H), !,
sys_freeze_var(H, Z).
attr_bound(Y, E) &:-
   &- addvc(X, Y, C) && sys_bound_var(X), sys_melt_var(X, D), integer(D),
!, E is C-D.
attr_bound(X, E) &:-
   &- addvc(X, Y, C) && sys_bound_var(Y), sys_melt_var(Y, D), integer(D),
!, E is C-D.
addvc(Z, Y, C) &:-
   refer(X, Z) && &- addvc(X, Y, C).
addvc(X, Z, C) &:-
   refer(Y, Z) && &- addvc(X, Y, C), X \== Y.
attr_bound(Y, E) &:-
   bound(X, D) && &- addvc(X, Y, C), E is C-D.
attr_bound(X, E) &:-
   bound(Y, D) && &- addvc(X, Y, C), X \== Y, E is C-D.

/*****************************************************************/
/* External Representation                                       */
/*****************************************************************/

% ensure_attr(+Var)
ensure_attr(X) :- sys_attr(X), !.
ensure_attr(X) :- sys_new_attr(X).

% ensure_hook(+Freezer)
ensure_hook(R) :- sys_clause_hook(R, hook, _), !.
ensure_hook(R) :- sys_compile_hook(hook, C), assume_hook(R, C).

% +Expr ∈ +Set
X ∈ {Y} :-  var(X), !,
   set_to_list(Y, A),
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, B),
   post(domain(B, A)).
X ∈ {Y} :- integer(X), !,
   set_to_list(Y, A),
   member(X, A), !.
X ∈ {_} :-
   sys_throw_error(type_error(integer, X)).

% set_to_list(+Set, -List)
set_to_list(X, _) :- var(X),
   sys_throw_error(instantiation_error).
set_to_list((X,Y), C) :- !,
   set_to_list(X, A),
   set_to_list(Y, B),
   union(A, B, C).
set_to_list(X, [X]) :- integer(X), !.
```

```
set_to_list(X, _) :- var(X),
   sys_throw_error(type_error(integer, X)).

% +Expr + +Expr ≡ +Expr
X ≡ _ :- var(X),
   sys_throw_error(instantiation_error).
X + Y ≡ Z :- var(X), var(Y), var(Z), !,
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, A),
   ensure_attr(Y),
   ensure_hook(Y),
   sys_bind_serno(Y),
   sys_freeze_var(Y, B),
   ensure_attr(Z),
   ensure_hook(Z),
   sys_bind_serno(Z),
   sys_freeze_var(Z, C),
   post(addvv(A,B,C)).
X + Y ≡ Z :- var(X), var(Y), integer(Z), !,
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, A),
   ensure_attr(Y),
   ensure_hook(Y),
   sys_bind_serno(Y),
   sys_freeze_var(Y, B),
   post(addvc(A,B,Z)).
X + Y ≡ Z :- var(X), integer(Y), var(Z), !,
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, A),
   ensure_attr(Z),
   ensure_hook(Z),
   sys_bind_serno(Z),
   sys_freeze_var(Z, B),
   post(addcv(A,Y,B)).
Y + X ≡ Z :- var(X), integer(Y), var(Z), !,
   ensure_attr(X),
   ensure_hook(X),
   sys_bind_serno(X),
   sys_freeze_var(X, A),
   ensure_attr(Z),
   ensure_hook(Z),
   sys_bind_serno(Z),
   sys_freeze_var(Z, B),
   post(addcv(A,Y,B)).
X + Y ≡ Z :- var(X), integer(Y), integer(Z), !,
   T is Z-Y,
   X = T.
Y + X ≡ Z :- var(X), integer(Y), integer(Z), !,
   T is Z-Y,
   X = T.
X + Y ≡ Z :- integer(X), integer(Y), var(Z), !,
   T is X+Y,
   Z = T.
X + Y ≡ Z :- integer(X), integer(Y), integer(Z), !,
   T is X+Y,
   Z == T.
```

```
X + _ ≡ _ :- \+ integer(X), \+ var(X),
   sys_throw_error(type_error(integer, X)).
_ + Y ≡ _ :- \+ integer(Y), \+ var(Y),
   sys_throw_error(type_error(integer, Y)).
_ + _ ≡ Z :-
   sys_throw_error(type_error(integer, Z)).

:- multifile sys_pretty_event/1.

% sys_pretty_event(-Goal)
sys_pretty_event(X ∈ {Y}) :-
   domain(B, A),
   list_to_set(A, Y),
   sys_melt_var(B, X).
sys_pretty_event(X+Y ≡ Z) :-
   addvv(A, B, C),
   sys_melt_var(A, X),
   sys_melt_var(B, Y),
   sys_melt_var(C, Z).
sys_pretty_event(X+Y ≡ Z) :-
   addcv(A, Y, C),
   sys_melt_var(A, X),
   sys_melt_var(C, Z).
sys_pretty_event(X+Y ≡ Z) :-
   addvc(A, B, Z),
   sys_melt_var(A, X),
   sys_melt_var(B, Y).

% list_to_set(+List, -Set)
list_to_set([X,Y|T], (X,C)) :- !,
   list_to_set([Y|T], C).
list_to_set([X], X).

/******************************************************/
/* List Processing                                    */
/******************************************************/

% member(+Elem, +List)
member(X, [X|_]).
member(X, [_|Y]) :-
   member(X, Y).

% intersect(+List, +List, -List)
intersect([], _, []).
intersect([X|Y], Z, [X|T]) :-
   member(X, Z), !, intersect(Y, Z, T).
intersect([_|Y], Z, T) :-
   intersect(Y, Z, T).

% union(+List, +List, -List)
union([], X, X).
union([X|Y], Z, T) :-
   member(X, Z), !, union(Y, Z, T).
union([X|Y], Z, [X|T]) :-
   union(Y, Z, T).

/******************************************************/
/* Test Cases I                                       */
/******************************************************/

% ?- X ∈ {1}, stored.
% X = 1
```

```
% ?- X ∈ {1,2,3}, Y ∈ {2,3,4}, stored.
% X ∈ {1,2,3}.
% Y ∈ {2,3,4}.
% Yes

% ?- X ∈ {1,2,3}, X ∈ {2,3,4}, stored.
% X ∈ {2,3}.
% Yes

% ?- X ∈ {2,3}, X ∈ {2,3,4}, stored.
% X ∈ {2,3}.
% Yes

% ?- X ∈ {1,2,3}, X ∈ {2,3}, stored.
% X ∈ {2,3}.
% Yes

% ?- X ∈ {1,2,3}, X = 4, stored.
% No

% ?- X ∈ {1,2,3}, X = 2, stored.
% X = 2

% ?- X = 4, X ∈ {1,2,3}, stored.
% No

% ?- X = 2, X ∈ {1,2,3}, stored.
% X = 2

% ?- X ∈ {1,2,3}, X = Y, Y ∈ {2,3,4}, stored.
% Y ∈ {2,3}.
% Y = X

% ?- X ∈ {1,2,3}, Y ∈ {2,3,4}, X = Y, stored.
% Y ∈ {2,3}.
% Y = X

/*******************************************************/
/* Test Cases II                                       */
/*******************************************************/

% ?- X + Y ≡ Z, X = 1, Y = 2, stored.
% X = 1,
% Y = 2,
% Z = 3.

% ?- X = 1, X + Y ≡ Z, Y = 2, stored.
% X = 1,
% Y = 2,
% Z = 3.

% ?- X = 1, Y = 2, X + Y ≡ Z, stored.
% X = 1,
% Y = 2,
% Z = 3.

% ?- X + Y ≡ Z, Z = X, stored.
% Y = 0,
% Z = X
```

```
% ?- Z = X, X + Y ≡ Z, stored.
% Z = X,
% Y = 0

% ?- X + Y ≡ 3, Y + Z ≡ 5, X + Z ≡ 6, Y = 1, stored.
% X = 2,
% Y = 1,
% Z = 4

% ?- X + Y ≡ 3, Y + Z ≡ 5, X + Z ≡ 6, Y = 2, stored.
% No
```

Pictures

Picture 1: Clause Indexing....6
Picture 2: Incremental Relative Strategies Results 21
Picture 3: Bound Propagation Impact 22
Picture 4: Refer Propagation Impact....23
Picture 5: Constant Instantiation Impact 24
Picture 6: Variable Instantiation Impact 25
Picture 7: Relative Interpreter Results 28
Picture 8: GNU Prolog Performance....29
Picture 9: B-Prolog Prolog Performance....30
Picture 10: ECLiPSe Prolog Performance 31
Picture 11: SWI-Prolog Performance....32
Picture 12: Ciao Prolog Performance 33
Picture 13: 4 Queens....48

Tables

Table 1: Iterations of the Test Programs....14
Table 2: Compared Optimization Settings 20
Table 3: Absolute Detailed Strategies Results (ms)....21
Table 4: Absolute Detailed Interpreter Results (ms)....28

References

[1] Morales et al. (2012): The Ciao CLP(FD) Library. A Modular CLP Extension for Prolog. In, N. Angelopoulos and R. Bagnara, editors, Proceedings of CICLOPS 2012, Budapest, Hungary, September 4, 2012
http://arxiv.org/abs/1301.7702v1

[2] Tikovsky, J. R. (2012): Integration eines Finite-Domain-Constraint-Solvers in KiCS2, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, August 2012
http://www.informatik.uni-kiel.de/~mh/lehre/abschlussarbeiten/msc/tikovsky.pdf

[3] Carlsson, M., Ottosson, G. and Carlson, B. (1997). An Open-Ended Finite Domain Constraint Solver, In H. Glaser, P. Hartel, and H. Kuchen, editors, Programming Languages: Implementations, Logics, and Programming, volume 1292 of LNCS, pages 191–206. Springer-Verlag, 1997
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3107

[4] Hanák, D., Szeredi, T. and Szeredi, P. (2004): FDBG, the CLP(FD) Debugger Library of SICstus Prolog. In B. Demoen and V. Lifschitz, editors, Proc. of ICLP'04, Poster. LNCS 3132, 2004
http://clip.dia.fi.upm.es/Conferences/WLPE04/papers/hanak-szeredi.ps

[5] Triska, M. (2012): The Finite Domain Constraint Solver of SWI-Prolog, In Schrijvers, T. and Thiemann, P, editors, Proceedings of FLOPS'12, 307-316, LNCS 7294, 2012
http://web.student.tuwien.ac.at/~e0225855/swiclpfd.pdf

[6] Zhou, NF. (2010): What I Have Learned From All These Solver Competitions, In U. Geske and A. Wolf, editors, Proceedings of the 23rd Workshop on (Constraint) Logic Programming 2009, 17 – 34, Potsdam, 2010
http://opus.kobv.de/ubp/volltexte/2010/4143/pdf/wlp09_17_34.pdf

[7] Schimpf, J. and Shen. K. (2011): ECLiPSe - from LP to CLP, In Theory and Practice of Logic Programming / Volume 12 / Special Issue on Prolog Systems 1-2, 127 - 156. Copyright Cambridge University Press 2011, Published online 12 September 2011
http://eclipseclp.org/reports/tplp2012-eclipse.pdf

[8] Le Berre, D. (2009): Understanding and using SAT solvers, A practitioner perspective, Summer School 2009: Verification Technology, Systems & Applications, Nancy, October 12-16, 2009
http://www.mpi-inf.mpg.de/vtsa09/slides/leberre2.pdf

[9] Howe, J.M. and King, A. (2010): A Pearl on SAT Solving in Prolog, In M. Blume, N. Kobayashi, and G. Vidal, editors, Proceedings FLOPS'10, 165-174, LNCS 6009. Springer, 2010.
http://www.soi.city.ac.uk/~jacob/solver/flops10talk.pdf

[10] Creignou, N. and Vollmer, H. (2008): Boolean constraint satisfaction problems : When does post's lattice help ? In Complexity of Constraints — An Overview of Current Research Themes [Result of a Dagstuhl Seminar], volume 5250 of Lecture Notes in Computer Science, pages 3-37. Springer, 2008.
http://link.springer.com/chapter/10.1007/978-3-540-92800-3_2

[11] Negri, S. (2003): Contraction-free sequent calculi for geometric theories, with an application to Barr's theorem, Archive for Mathematical Logic, vol. 42, pp. 389-401, 2003
http://www.helsinki.fi/~negri/articles.html/barrfin.pdf