

Realization of Forward Checking in Logic Programming via Extended Unification

Christian Holzbaur
(email: christian@ai-vie.uucp)

Department of Medical Cybernetics and AI
University of Vienna, and
Austrian Research Institute for Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria*

TR-90-11

Abstract

Forward checking is one of the most promising *a priori* pruning techniques for Constraint Satisfaction Problems [Mackworth 77, Mackworth and Freuder 85]. In the past some schemes for the incorporation of forward checking and other consistency techniques over finite domains into Prolog have been introduced in [Hentenryck 89]. This work resulted in the implementation of a new prototype Prolog interpreter. In this paper we present an implementation of forward checking in Prolog. This implementation is based on a very general scheme for the incorporation of semantic unification into Prolog [Holzbaur 90, Neumerkel 90]. Constraint Satisfaction Problems can be formulated declaratively in Prolog. The problem with the standard backtracking evaluation strategy however is that it does not execute this specification efficiently. Therefore, the Prolog implementation of forward checking serves at least two purposes: the set of effectively executable declarative specifications in Prolog is enlarged; the choice of Prolog as implementation language results in more declarative, reliable, maintainable, accessible, and extendible code.

Keywords: Constraints, Metaprogramming, Unification, Implementation.

*This work was supported by the Federal Ministry of Science and Research. Thanks to P. Petta and G. Widmer for their comments.

Contents

1	Introduction	3
2	Logical background	3
2.1	Domains	3
2.2	Forward checking	4
3	The user's view of the implementation	5
4	Examples	7
4.1	Map coloring	7
4.2	Scene labeling	8
4.3	Inequalities	12
4.4	A combinatorial problem	14
4.5	Another old puzzle	16
5	Implementation	23
5.1	The general mechanism	23
5.2	Domains	23
5.3	Forward checking	27
5.3.1	Optimizations	35
5.4	Inequality	37
5.4.1	An exercise in partial evaluation	40
5.5	The combination of the theories for forward checking and inequality	46
5.6	Labeling	48
5.7	Set implementations	51
5.7.1	Direct representation of sets as ordered lists	51
5.7.2	Indirect representation of small sets as integer numbers	54
5.7.3	Indirect representation of large sets, realized as bitvectors	59
5.7.4	Folding the set representation into programs	60
5.8	Compilation	61
6	Summary	66
	References	66
	Glossary	68
	List of Figures	71

1 Introduction

The outline of the paper is the following: We start with a sketch of the theoretical framework for embedding consistency techniques into logic languages. Next, the user's view of these extensions in a Prolog environment is condensed into a handful of new predicates. In the subsequent section these predicates are applied in the formulation of some examples. Finally, the main body of the paper deals with the implementation of forward checking. As the use of semantic unification as an implementation mechanism is not very common yet, the final solution is derived step by step. Hereby it is shown how the application of partial evaluation as a constructive means for the derivation of efficient programs from general specifications is facilitated through the choice of Prolog as the implementation language.

2 Logical background

This section presents the theoretical framework for embedding consistency techniques. A declarative semantics that *preserves* the attractive semantic properties of the standard theory is defined. In addition, a sound and complete procedural semantics is introduced. The definitions were taken from [Hentenryck 89].

2.1 Domains

The domain concept allows for the specification of the range of a variable. Domains carry the same information as monadic predicates in logic languages. This information is used during unification as opposed to the use at resolvent level.

Definition 1 *A domain is a non-empty finite set of constants.*

A variable x with domain d is written as x^d . Several domains may be used in the same logic program.

Definition 2 *A first-order language with domain variables is composed of the following alphabet:*

1. *a set of simple variables*
2. *a set of domain variables*
3. *sets of constants, functions, predicates, connectives, quantifiers and a set of punctuation symbols*

Terms, definite programs, and goals can be constructed as usual [Lloyd 88, Chang and Lee 73]. The declarative semantics of a logic program is given by the model theoretic semantics of first order logic with domain variables. The notions of logical consequences, Herbrand interpretations, and models as well as the least Herbrand model can be defined as usual [Hentenryck 89].

The procedural semantics has to deal with domains, too. The required modification consists in a redefinition of the unification algorithm.

Definition 3 *There are three additions to the usual definition [Lloyd 88] of the unification algorithm:*

1. *If a domain variable and a constant have to be unified, the unification succeeds if the constant is a member of the domain of the domain variable and the domain variable is instantiated to the constant. Otherwise the unification fails.*
2. *If two domain variables have to be unified, the unification succeeds if the intersection of their domains is non-empty and binds both variables to a new domain variable ranging over this intersection. Otherwise the unification fails.*
3. *If a domain variable and a simple variable have to be unified, the unification succeeds and binds the simple variable to the domain variable.*

The modified unification algorithm accepts a finite set of expressions and produces either a most general unifier (mgu) or an indication of failure. Proving the soundness and completeness of SLD-Resolution with the extended unification is not too difficult, because the standard proofs do not rely on the assumption that the unification takes place in an empty equational theory. If we use the domain concept described so far, only equality constraints¹ are handled yet. Forward checking is a new inference rule that operates upon the domain concept; it allows the use of general constraints in an active way.

2.2 Forward checking

In forward checking, constraints are used as soon as only one variable of an atom is left uninstantiated. The forward checking inference rule (FCIR) removes inconsistent values from the domain of this last variable.

Definition 4 *Let p be an n -ary predicate. p is a constraint iff for any ground terms t_1, \dots, t_n $p(t_1, \dots, t_n)$ has a successful refutation or only finitely failed derivations.*

Definition 5 *Let $p(t_1, \dots, t_n)$ be an atom. $p(t_1, \dots, t_n)$ is forward checkable iff*

1. p/n is a constraint.
2. *there exists exactly one t_i that is a domain variable, all others being ground.*

This last variable is called the forward variable.

¹unifications between terms are equality constraints

Definition 6 (FCIR) Let P be a program, $G_i = \leftarrow A_1, \dots, A_m, \dots, A_k$ a goal. G_{i+1} is derived by the FCIR from G_i and P using the substitution θ_{i+1} if the following conditions hold:

1. A_m is forward checkable and x^d is the forward variable.
2. $e = \{a \in d \mid P \models A_m\{x^d/a\}\} \neq \emptyset$.
3. θ_{i+1} is
 - $\{x^d/c\}$ if $e = \{c\}$;
 - $\{x^d/z^e\}$ where z^e is a new domain variable otherwise.
4. $G_{i+1} = \leftarrow (A_1, \dots, A_{m-1}, A_{m+1}, \dots, A_k)\theta_{i+1}$.

The inference rule reduces the search space a priori as inconsistent values from the domain of the forward variable are removed once and for all. According to the definition of a constraint the derivation of a ground instance of a constraint in step 2 of the above definition either succeeds or finitely fails. Furthermore, this inference rule *instantiates* the forward variable when only one consistent value remains. This instantiation possibly makes the FCIR applicable to other atoms. The FCIR can also be viewed as a mechanism enforcing node consistency [Mackworth and Freuder 85]. Since only one variable appears in the forward checking atom, this atom can be considered a unary predicate. The domain of the variable is reduced to satisfy it. The soundness and completeness of the FCIR has been proved in [Hentenryck 89].

3 The user's view of the implementation

Just a small number of predicates are needed to enable the user to work with finite domains and constraints acting upon them in a forward checking manner:

1. `domain/2` is used to attach a domain, i.e., a set of possible values, to a variable. The potential values must be ground terms.

Example:

```
1 [Clp] ?- domain( X, [1,2,3]).
2 X = _17 <Constraint(s): [domain(_17,[1,2,3])]>
```

2. `forward/1` is used to declare a goal as forward checkable. The current implementation does not allow for structures in the arguments of the goal submitted to `forward/1`.

Example:

```
1 [Clp] ?- forward( A < 3).
2 A = _12 <Constraint(s): [forward(_12<3)]>
```

3. `neq/2` specifies that the two arguments differ. Allowed arguments are variables, domain variables, atoms and numbers, but not structures.

Example:

```
1 [Clp] ?- domain( A, [1,2,3]), neq( A, B), B=2.
2 A = _214 <Constraint(s): [domain(_214,[1,3])]>
3 B = 2
```

4. `indomain/1` makes a domain variable ground. During backtracking the domain variable is instantiated to the elements of its domain.

Example:

```
1 [Clp] ?- domain( X, [1,2,3]), indomain( X).
2 X = 1 ;
3 X = 2 ;
4 X = 3 ;
```

5. `labeling/1` is used to label a list of domain variables. This is but the application of `indomain/1` to each element of the list.

Example:

```
1 [Clp] ?- domain( X, [1,2]), domain( Y, [a,b]),
2           L=[X,Y,z], labeling( L).
3 X = 1
4 Y = a
5 L = [1,a,z] ;
6
7 X = 1
8 Y = b
9 L = [1,b,z] ;
...

```

6. `dump/3` produces a copy of a term with all metaterms² removed and a list that *describes* the metastructures, i.e., the constraints on variables that were found in the original term. Via `dump/3` the user can get an *external* description of yet unsatisfied constraints in an answer substitution delivered by the Prolog system. The description of the constraints together with the copy of the term can be put into the dynamic Prolog database or written to a file.

Example:

```
1 [Clp] ?- domain( X, [1,2,3]), forward( X < Y),
2           dump( f(X,Y,pos), Copy, Constraints).
3
4 Copy      = f(_171,_172,pos)
5 Constraints = [domain(_171,[1,2,3]),forward(_171<_172)]
```

²Metaterms are introduced in section 5.1. They are the basic data structure upon which forward checking is implemented.

4 Examples

This section lists a few examples to which forward checking was applied. Some of the examples serve as benchmark programs in the later sections on the implementation of forward checking. All execution times were gathered on an otherwise idling Apollo DN3000 workstation. Comparisons between **Metaprolog** and **Quintus Prolog** were made on the same machine, under the same conditions. **Metaprolog** is an implementation of the specification from [Holzbaur 90] in **C-Prolog** [Pereira 82], which is a Prolog interpreter written in C for 32 bit machines.

4.1 Map coloring

The following problem statement is taken from [Coelho et al. 80]: ‘Write a program for coloring any planar map with at most four colors, such that no two adjacent regions have the same color.’ The program consists of a list of connections between regions and of a collection of admissible pairs of colors.

```

1  sample(Cs) :-
2      Cs=[Albania,Greece,Yugoslavia],
3      next(Albania,Greece),
4      next(Albania,Yugoslavia),
5      next(Greece,Yugoslavia).
6
7  next(blue,yellow).    next(red,yellow).
8  next(blue,red).       next(red,blue).
9  next(blue,green).     next(red,green).
10 next(yellow,blue).    next(green,yellow).
11 next(yellow,red).     next(green,red).
12 next(yellow,green).   next(green,blue).
```

Figure 1: Map coloring with Prolog

This implementation (see figure 1) works pretty well for small maps. However, *compiled* **Quintus Prolog** was trying to color the 32 countries of Europe for more than five weeks. The process was stopped prior to termination. We are aware that the instantiation order of variables plays a critical role in such a task, but we did not put any effort in finding a better one. The point is that with forward checking the problem was solved in 1.75 seconds by *interpreted* **Metaprolog**.

Of course, this mainly proves the superiority of forward checking over chronological backtracking. On the other hand, the reformulation effort was minor, and a previously practically intractable problem was solved instantly, *without* forcing the user to become an expert in graph theory. The adapted (sample) program is shown in figure 2.

```

1 fwd_sample(Cs) :-
2   Cs=[Albania,Greece,Yugoslavia],
3   domain(Albania, [red,green,blue,yellow]),
4   domain(Greece, [red,green,blue,yellow]),
5   domain(Yugoslavia,[red,green,blue,yellow]),
6   forward( next(Albania,Greece)),
7   forward( next(Albania,Yugoslavia)),
8   forward( next(Greece,Yugoslavia)),
9   labeling( Cs).

```

Figure 2: Map coloring with forward checking

4.2 Scene labeling

Scene labeling problems are usually solved with some arc-consistency algorithms [Mackworth and Freuder 85, Mohr and Henderson 86], together with a backtracking component. Forward checking is applicable to such domains, too. An introduction to scene interpretation in a world of crack-free polyhedra without shadows, where all vertices are the intersection of exactly three object faces is given in [Winston 84, chapter 3]. Figure 3 depicts such a body, and its Prolog description is given in figure 4.

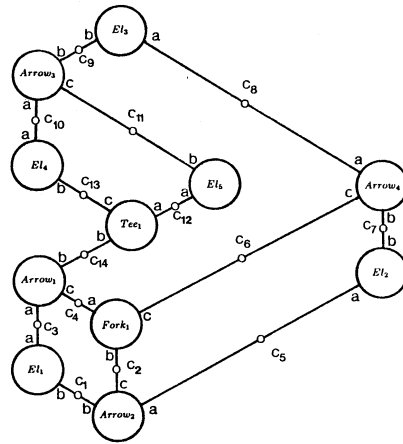


Figure 3: L-shaped body

The predicates `l_junction/2` (figure 5), `fork_junction/3` (figure 6), `tee_junction/3` (figure 7) and `arrow_junction/3` (figure 8) describe the possible junction types.


```

1  el([AB,BC,CD,DE,EF,FG,GH,HI,IA,IJ,GJ,FK,DK,BK]) :-
2    l_junction(AB,AI),
3    arrow_junction(BA,BC,BK),
4    l_junction(CD,CB),
5    arrow_junction(DC,DE,DK),
6    l_junction(EF,ED),
7    arrow_junction(FE,FG,FK),
8    tee_junction(GJ,GF,GH),
9    l_junction(HI,HG),
10   arrow_junction(IH,IA,IJ),
11   l_junction(JG,JI),
12   fork_junction(KF,KD,KB),
13   inversions( [AB,BC,CD,DE,EF,FG,GH,HI,IA,IJ,GJ,FK,DK,BK] ,
14               [BA,CB,DC,ED,FE,GF,HG,IH,AI,JI,JG,KF,KD,KB]
15               ).

```

Figure 4: Prolog description of the L-shaped body

```

1  l_junction(>,<).
2  l_junction(<,>).
3  l_junction(+,>).
4  l_junction(<,+).
5  l_junction(-,<).
6  l_junction(>,-).

```

Figure 5: Predicate `l_junction/2`

```

1  fork_junction(+,+,+).
2  fork_junction(-,-,-).
3  fork_junction(<,>,-).
4  fork_junction(-,<,>).
5  fork_junction(>,-,<).

```

Figure 6: Predicate `fork_junction/3`

```

1  tee_junction(>,<,+).
2  tee_junction(>,<,-).
3  tee_junction(>,<,<).
4  tee_junction(>,<,>).

```

Figure 7: Predicate `tee_junction/3`

```

1  arrow_junction(<,>,+).
2  arrow_junction(-,-,+).
3  arrow_junction(+,+,-).

```

Figure 8: Predicate `arrow_junction/3`

The junctions of a scenario are connected by so-called inversions (figure 9).

```

1  inversion(+,+).
2  inversion(-,-).
3  inversion(<,>).
4  inversion(>,<).
5
6  inversions([],[]).
7  inversions([A|As],[B|Bs]) :-
8      inversion(A,B),
9      inversions(As,Bs).
```

Figure 9: Predicate `inversion/2`

In analogy to the map coloring example, this program was also augmented with domain declarations and forward definitions (figure 10). In this example, labeling is done by `inversions/2`.

```

1  el([AB,BC,CD,DE,EF,FG,GH,HI,IA,IJ,GJ,FK,DK,BK]) :-
2      domain_vars([AB,BC,CD,DE,EF,FG,GH,HI,IA,IJ,GJ,FK,DK,BK],[+,-,<,>]),
3      domain_vars([BA,CB,DC,ED,FE,GF,HG,IH,AI,JI,JG,KF,KD,KB],[+,-,<,>]),
4      forward( l_junction(AB,AI) ),
5      forward( arrow_junction(BA,BC,BK) ),
6      forward( l_junction(CD,CB) ),
7      forward( arrow_junction(DC,DE,DK) ),
8      forward( l_junction(EF,ED) ),
9      forward( arrow_junction(FE,FG,FK) ),
10     forward( tee_junction(GJ,GF,GH) ),
11     forward( l_junction(HI,HG) ),
12     forward( arrow_junction(IH,IA,IJ) ),
13     forward( l_junction(JG,JI) ),
14     forward( fork_junction(KF,KD,KB) ),
15     inversions( [AB,BC,CD,DE,EF,FG,GH,HI,IA,IJ,GJ,FK,DK,BK],
16                 [BA,CB,DC,ED,FE,GF,HG,IH,AI,JI,JG,KF,KD,KB]
17                 ).
```

Figure 10: Metaprolog description of the L-shaped body

The original program was run in compiled **Quintus Prolog** with a varying number of instantiated variables, i.e., it was called as shown in figure 11. The augmented version was run in **Metaprolog**. The execution times for the determination of *all* solutions are summarized in figure 11. A more appealing summary is in figure 12. The light bars correspond to compiled **Quintus Prolog**, the black bars are for **Metaprolog**.

Call	Solutions	Quintus Prolog, compiled	Metaprolog, interpreted
<code>e1(L).</code>	8	5596	7.05
<code>e1([> L]).</code>	6	1881	4.33
<code>e1([>,> L]).</code>	6	711	4.20
<code>e1([>,>,> L]).</code>	3	243	2.37
<code>e1([>,>,>,> L]).</code>	3	85	2.28
<code>e1([>,>,>,>,> L]).</code>	3	29	2.00
<code>e1([>,>,>,>,>,> L]).</code>	3	10	1.90
<code>e1([>,>,>,>,>,>,> L]).</code>	2	2.6	1.00
<code>e1([>,>,>,>,>,>,>,> L]).</code>	1	0.89	0.73

Figure 11: Execution times in seconds for the scene labeling example

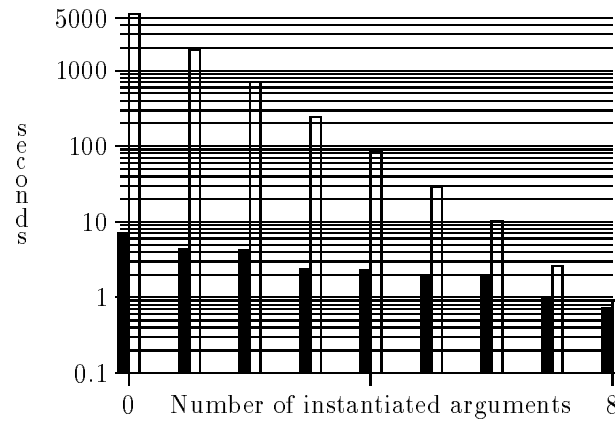


Figure 12: Execution times for the scene labeling example

4.3 Inequalities

Inequalities are a kind of constraints that are found frequently in problem formulations that work with domain variables. In particular, the predicate `alldifferent/1` is used in places where one would employ a permutation predicate in ‘ordinary’ Prolog problem formulation. A pattern as in figure 13 is likely to be found in generate-and-test programs. The crux with this is that the generator (`permutation/2`) is not steerable. If an arrange-

```

1  arrange( L, Perm) :-
2      permutation( L, Perm),
3      apply_constraints( Perm).
4
5  permutation( [], []).
6  permutation( Rest, [Ci|Cs]) :-
7      delete( Ci, Rest, Nrest),
8      permutation( Nrest, Cs).
9
10 delete( A,[A | L], L).
11 delete( A,[B | L], [B | L1]) :- delete( A, L, L1).
```

Figure 13: Generating and testing permutations

ment does not pass the validation, Prolog’s usual chronological backtracking strategy is used to generate alternative permutations. In *Metaprolog* we can do better by making use of active constraints. The formulation in figure 14 uses active inequality constraints (`neq/2`) to generate permutations. The new generator is realized in three steps:

1. A domain of ‘positions’ is assigned to the list of objects that are to be permuted.
2. Constraints stating the pairwise disjointness of the objects in the list are generated. In lines 1 to 5 in figure 15 the disjointness condition was omitted. Consequently labeling generates all n^n assignments instead of the $n!$ permutations, where n is the length of the list of objects. In lines 7 to 11 in figure 15 we used the correct formulation.
3. Labeling is applied to the elements of the list.

Note that for a list of length n , the application of `alldifferent/1` creates $\frac{n(n-1)}{2}$ inequality constraints. Thus it is a suitable benchmark for the implementation of inequality constraints.

```

1  arrange( L, Domain) :-
2    domain_vars( L, Domain),
3    alldifferent( L),
4    apply_constraints( L),
5    labeling( L).
6
7  domain_vars( [], _).
8  domain_vars( [X|Xr], Dom) :-
9    domain( X, Dom),
10   domain_vars( Xr, Dom).
11
12 alldifferent( []).
13 alldifferent( [X|R]) :-
14   alld_others( X, R),
15   alldifferent( R).
16
17 alld_others( _, []).
18 alld_others( X, [F|T]) :-
19   neq( X, F),
20   alld_others( X, T).

```

Figure 14: Predicate `alldifferent/1`, computing permutations with domains and inequalities

```

1  [Clp] ?- L=[A,B], domain_vars( L, [1,2]), labeling( L).
2  L = [1,1] ;
3  L = [1,2] ;
4  L = [2,1] ;
5  L = [2,2] ;
6
7  [Clp] ?- L=[A,B], domain_vars( L, [1,2]),
9          alldifferent( L), labeling( L).
10 L = [1,2] ;
11 L = [2,1] ;

```

Figure 15: Computing permutations with `alldifferent/1`

4.4 A combinatorial problem

The following problem can be solved by using just inequality constraints. It was taken from [Hentenryck and Dincbas 86], where it is attributed to [Lauriere 78]. It is reproduced here to emphasize the mentioned relation between the generation of permutations and the use of `alldifferent/1`.

Problem statement: Six couples took part in a tennis match. Their names were Howard, Kress, McLean, Randolph, Lewis and Rust. The first names of their wives were Margaret, Susan, Laura, Diana, Grace and Virginia. Each of the ladies hailed from a different city: Fort Worth, Wichita, Mt. Vernon, Boston, Dayton, Kansas City. Finally, each of the women had a different hair color, namely black, brown, gray, red, auburn and blond. Informations are given to state doubles and singles which were played. For instance, Howard and Kress played against Grace and Susan or the gray haired lady played against Margaret. There is only one other fact we ought to know to be able to find the last names, home towns, and hair colors of all six wives, and it is a fact that “No married couple ever took part in the same game”.

The Prolog program in figure 16 can be used to determine the unique solution to the problem. The translation to a forward checking program that uses just inequality con-

```

1  tennis( L ) :-
2      L = [Ho,Ke,Mc,Ra,Le,Ru,
3           Fo,Wi,Mt,Bo,Da,Ka,
4           Bl,Br,Gr,Re,Au,Blo],
5      permutation( [ma,su,la,di,gr,vi], [Ho,Ke,Mc,Ra,Le,Ru]),
6      Ho \== gr, Ho \== su, Ke \== gr, Ke \== su, Mc \== la,
7      Mc \== su, Ra \== la, Ra \== su, Mc \== gr, Ra \== gr,
8      Le \== gr, Ke \== la, Ke \== vi, Mc \== di, Mc \== vi,
9      permutation( [ma,su,la,di,gr,vi], [Bl,Br,Gr,Re,Au,Blo]),
10     Br \== vi, Br \== Ho, Br \== Mc, Ra \== Gr, Gr \== la,
11     Blo \== la, Blo \== di, Le \== Blo, Blo \== ma,
12     permutation( [ma,su,la,di,gr,vi], [Fo,Wi,Mt,Bo,Da,Ka]),
13     Fo \== Ho, Fo \== Mc, Fo \== Ra, Wi \== Ho, Wi \== Mc,
14     Da \== ma, Mt \== ma, Mt \== di, Da \== di, Mt \== vi,
15     Wi \== Ra, Wi \== Ke, Ru \== Fo, Fo \== Ke, Gr \== Bo,
16     Re \== Da, Gr \== Fo, Re \== Mt, Blo \== Da,
17     Bl \== Bo, Bl \== Da, Ka \== ma.
18
19     % solution: [la,di,ma,vi,su,gr,su,gr,la,ma,vi,di,la,su,di,ma,vi,gr]
```

Figure 16: Prolog description of a tennis match

straints is straightforward (figure 17). This program does not even need labeling — the constraints are strong enough that all domains are reduced to single values. The unique

```

1  tennis(L) :-
2    L = [Ho,Ke,Mc,Ra,Le,Ru,
3         Fo,Wi,Mt,Bo,Da,Ka,
4         Bl,Br,Gr,Re,Au,Blo],
5    domain_vars( L, [ma,su,la,di,gr,vi]),
6    neq(Ho,gr), neq(Ho,su), neq(Ke,gr), neq(Ke,su), neq(Mc,la),
7    neq(Mc,su), neq(Ra,la), neq(Ra,su), neq(Mc,gr), neq(Ra,gr),
8    neq(Le,gr), neq(Ke,la), neq(Ke,vi), neq(Mc,di), neq(Mc,vi),
9    neq(Mt,ma), neq(Mt,di), neq(Da,di), neq(Mt,vi),
10   neq(Blo,la), neq(Blo,di), neq(Da,ma), neq(Ka,ma),
11   neq(Br,vi), neq(Gr,la), neq(Blo,ma),
12   alldifferent( [Ho,Ke,Mc,Ra,Le,Ru]),
13   neq(Fo,Ho), neq(Fo,Mc), neq(Fo,Ra), neq(Wi,Ho), neq(Wi,Mc),
14   neq(Wi,Ra), neq(Wi,Ke), neq(Ru,Fo), neq(Br,Ho),
15   neq(Br,Mc), neq(Le,Blo), neq(Ra,Gr), neq(Fo,Ke),
16   alldifferent( [Bl,Br,Gr,Re,Au,Blo]),
17   neq(Gr,Bo), neq(Re,Da), neq(Gr,Fo), neq(Re,Mt),
18   neq(Blo,Da), neq(Bl,Bo), neq(Bl,Da),
19   alldifferent( [Fo,Wi,Mt,Bo,Da,Ka]).

```

Figure 17: Forward checking version of the Tennis puzzle

ground solution is generated without any choice for the value of a domain variable and therefore without any backtracking in making such choices. The execution times for the tennis example are summarized in figure 18.

	Quintus Prolog, compiled	C-Prolog	Forward checking in Metaprolog
First solution	22.1	135.5	1.75
All solutions	55.2	338.4	1.77

Figure 18: Execution times in seconds for the Tennis puzzle

4.5 Another old puzzle

In this section we solve another problem that has been in use as an example in slight variations³ for quite some time [Dechter 86, Hentenryck 89].

Problem statement: There are five houses of different colors, inhabited by different nationals, with different pets, drinks, and cigarettes. The following facts are known:

1. The Englishman lives in the red house.
2. The Spaniard owns a dog.
3. Coffee is drunk in the green house.
4. The Ukranian drinks tea.
5. The green house is to the right of the ivory house.
6. The Old-gold smoker owns snails.
7. Kools are being smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The Chesterfield smoker lives next to the fox owner.
11. Kools are smoked next to the house with the horse.
12. The Lucky-Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

The question is: who owns the zebra and who is drinking water?

The Prolog program in figure 19 assigns each of the 25 variables⁴ a domain of the house numbers one to five. The facts are encoded in the same sequence as they were stated above. The comments at the end of the lines point to the facts. The subsidiary predicates `right_of/2` and `next_to/2` are given in figure 20, `alldifferent/1` was defined in figure 14.

³renamed variables

⁴five colors, nationals, pets, drinks, and cigarettes


```

1  houses( Vars) :-
2      Vars = [Englishman,Spaniard,Ukranian,Japanese,Norwegian,
3              Red,Green,Blue,Yellow,Ivory,
4              Tea,Water,Coffee,Orange_juice,Milk,
5              Dog,Snails,Fox,Horse,Zebra,
6              Kools,Parliament,Lucky_strike,Chesterfield,Old_gold],
7      domain_vars( Vars, [1,2,3,4,5]),
8
9      Red = Englishman,                % 1
10     Spaniard = Dog,                  % 2
11     Coffee = Green,                  % 3
12     Ukranian = Tea,                  % 4
13     forward( right_of(Green,Ivory)), % 5
14     Old_gold = Snails,                % 6
15     Kools = Yellow,                  % 7
16     Milk = 3,                        % 8
17     Norwegian = 1,                  % 9
18     forward( next_to(Chesterfield,Fox)), % 10
19     forward( next_to(Kools,Horse)),   % 11
20     Lucky_strike = Orange_juice,      % 12
21     Japanese = Parliament,            % 13
22     forward( next_to(Norwegian,Blue)), % 14
23
24     alldifferent([Englishman,Spaniard,Ukranian,Japanese,Norwegian]),
25     alldifferent([Red,Green,Blue,Yellow,Ivory]),
26     alldifferent([Tea,Water,Coffee,Orange_juice,Milk]),
27     alldifferent([Dog,Snails,Fox,Horse,Zebra]),
28     alldifferent([Kools,Parliament,Lucky_strike,Chesterfield,Old_gold]),
29
30     labeling( Vars).

```

Figure 19: Forward checking formulation of the Zebra puzzle

```

1  right_of(A,B) :- A is B+1.
2
3  next_to(A,B) :- 1 is B-A.
4  next_to(A,B) :- 1 is A-B.

```

Figure 20: Predicates `right_of/2` and `next_to/2`

The table in figure 21 lists the variables of the problem together with their domains at eight junctures. Digits indicate possible house number assignments for the variables, dots indicate removed values. Labeling processes the variables top down and the domains left to right. Readers that are primarily interested in the implementation of forward checking might want to skip the rather detailed execution trace. It is provided for those who would like to compare it to their own line of reasoning on this puzzle.

Step 1: This is the situation when labeling starts. Facts eight and nine already instantiated the variables ‘Milk’ and ‘Norwegian’, the constraint from fact 14 instantiated ‘Blue’ to 2. The remaining dropouts are due to inequality constraints.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,japanese), neq(chesterfield,orange_juice),
 neq(chesterfield,snails), neq(chesterfield,yellow), neq(englishman,japanese),
 neq(englishman,spaniard), neq(englishman,ukranian), neq(fox,horse),
 neq(fox,snails), neq(fox,spaniard), neq(fox,zebra), neq(green,englishman),
 neq(green,ivory), neq(green,orange_juice), neq(green,ukranian),
 neq(green,water), neq(green,yellow), neq(horse,snails), neq(horse,spaniard),
 neq(horse,zebra), neq(ivory,englishman), neq(ivory,yellow),
 neq(japanese,orange_juice), neq(japanese,snails), neq(orange_juice,snails),
 neq(spaniard,japanese), neq(spaniard,snails), neq(spaniard,zebra),
 neq(ukranian,japanese), neq(ukranian,orange_juice), neq(ukranian,spaniard),
 neq(ukranian,water), neq(water,orange_juice), neq(yellow,englishman),
 neq(yellow,japanese), neq(yellow,orange_juice), neq(yellow,snails),
 neq(zebra,snails)

The choice of 3 for ‘Englishman’ leads us to the next step.

Step 2: Inequality constraints remove the value 3 from ‘Spaniard’ and ‘Japanese’. ‘Spaniard’ is equal to ‘Dog’ and therefore the value is removed there, too. The variable ‘Japanese’ is equal to ‘Parliament’ which loses 3. ‘Red’ is equal to ‘Englishman’ and therefore simultaneously instantiated to 3. This instantiation removes 3 from ‘Yellow’ and ‘Ivory’. The equality between ‘Yellow’ and ‘Kools’ drops 3 from ‘Kools’.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,japanese), neq(chesterfield,orange_juice),
 neq(chesterfield,snails), neq(chesterfield,yellow), neq(fox,horse), neq(fox,snails),
 neq(fox,spaniard), neq(fox,zebra), neq(green,ivory), neq(green,orange_juice),
 neq(green,ukranian), neq(green,water), neq(green,yellow), neq(horse,snails),
 neq(horse,spaniard), neq(horse,zebra), neq(ivory,yellow),
 neq(japanese,spaniard), neq(japanese,ukranian), neq(orange_juice,japanese),
 neq(orange_juice,snails), neq(orange_juice,ukranian), neq(snails,japanese),
 neq(snails,spaniard), neq(ukranian,spaniard), neq(water,orange_juice),
 neq(water,ukranian), neq(yellow,japanese), neq(yellow,orange_juice),
 neq(yellow,snails), neq(zebra,snails), neq(zebra,spaniard)

Our next choice is 2 for ‘Spaniard’.

Step 3: Inequality constraints remove the value 2 from ‘Ukranian’ and ‘Japanese’, via the equality between ‘Tea’ and ‘Ukranian’ from ‘Tea’, and via the equality between ‘Parliament’ and ‘Japanese’ from ‘Parliament’. The equality between ‘Spaniard’ and ‘Dog’ instantiates ‘Dog’ to 2, which removes this value from the other pets. The equality between ‘Old_gold’ and ‘Snails’ drops the value 2 from the latter.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,japanese), neq(chesterfield,orange_juice),
 neq(chesterfield,snails), neq(chesterfield,yellow), neq(fox,horse), neq(fox,snails),
 neq(fox,zebra), neq(green,ivory), neq(green,orange_juice), neq(green,ukranian),
 neq(green,water), neq(green,yellow), neq(horse,snails), neq(horse,zebra),
 neq(ivory,yellow), neq(japanese,ukranian), neq(orange_juice,japanese),
 neq(orange_juice,snails), neq(orange_juice,ukranian), neq(snails,japanese),
 neq(water,orange_juice), neq(water,ukranian), neq(yellow,japanese),
 neq(yellow,orange_juice), neq(yellow,snails), neq(zebra,snails)

We proceed with 4 for ‘Ukranian’.

Step 4: Inequality constraints remove the value 4 from ‘Japanese’ which instantiates this variable to the only remaining value, 5. The equality between ‘Ukranian’ and ‘Tea’ instantiates ‘Tea’ to 4, which removes this value from the other drinks. The instantiation of ‘Japanese’ propagates the value 5 via an equality to ‘Parliament’, which in turn removes this value from the other cigarette brands. The equality between ‘Yellow’ and ‘Kools’ drops 5 from ‘Yellow’. Another equality between ‘Orange_juice’ and ‘Lucky_strike’ removes 5 from the former, too. The equality between ‘Old_gold’ and ‘Snails’ drops 5 from the latter. ‘Lucky_strike’ loses the value 4 through an equality with ‘Orange_juice’.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,orange_juice), neq(chesterfield,snails), neq(chesterfield,yellow),
 neq(fox,horse), neq(fox,snails), neq(fox,zebra), neq(green,ivory),
 neq(green,orange_juice), neq(green,water), neq(green,yellow), neq(horse,snails),
 neq(horse,zebra), neq(ivory,yellow), neq(snails,orange_juice),
 neq(water,orange_juice), neq(yellow,orange_juice), neq(yellow,snails),
 neq(zebra,snails)

The assignment of 1 to ‘Green’ fails because of the constraint from fact 5. The instantiation of ‘Green’ to 5 fails because the constraint from fact 5 instantiates ‘Ivory’ to 4. Inequalities between colors remove 4 from ‘Yellow’, leaving the only value 1. As ‘Yellow’ equals ‘Kools’, this triggers the constraint from fact 11, which fails as there is no suitable value for ‘Horse’. Having no further value for ‘Green’, we must backup to the previous variable ‘Ukranian’ and try the value 5.

Step 5: Again the choices 1 and 4 for ‘Green’ fail. Backtracking to ‘Ukranian’ propagates to ‘Spaniard’ as we exhausted the choices for ‘Ukranian’, too. We alter ‘Spaniard’ to 4.

Step 6: The previous instantiation removes 4 from ‘Ukranian’, ‘Japanese’, ‘Tea’, all pets but ‘Dog’, ‘Parliament’, and via ‘Snails’ from ‘Old_gold’.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,japanese), neq(chesterfield,orange_juice),
 neq(chesterfield,snails), neq(chesterfield,yellow), neq(fox,horse), neq(fox,snails),
 neq(fox,zebra), neq(green,ivory), neq(green,orange_juice), neq(green,ukranian),
 neq(green,water), neq(green,yellow), neq(horse,snails), neq(horse,zebra),
 neq(ivory,yellow), neq(japanese,ukranian), neq(orange_juice,japanese),
 neq(orange_juice,snails), neq(orange_juice,ukranian), neq(snails,japanese),
 neq(water,orange_juice), neq(water,ukranian), neq(yellow,japanese),
 neq(yellow,orange_juice), neq(yellow,snails), neq(zebra,snails)

We choose 2 for ‘Ukranian’.

Step 7: The previous instantiation removes 2 from ‘Japanese’ which leaves the only value 5. ‘Tea’ and ‘Parliament’ get also instantiated to 2 and 5. The remaining dropouts are due to inequality constraints.

Residual constraints: forward(next_to(chesterfield,fox)),
 forward(next_to(yellow,horse)), forward(right_of(green,ivory)),
 neq(chesterfield,orange_juice), neq(chesterfield,snails), neq(chesterfield,yellow),
 neq(fox,horse), neq(fox,snails), neq(fox,zebra), neq(green,ivory),
 neq(green,orange_juice), neq(green,water), neq(green,yellow), neq(horse,snails),
 neq(horse,zebra), neq(ivory,yellow), neq(snails,orange_juice),
 neq(water,orange_juice), neq(yellow,orange_juice), neq(yellow,snails),
 neq(zebra,snails)

Once again the values 1 and 4 for ‘Green’ fail because of the constraint from fact 5, but we succeed with the value 5. The constraint from fact 5 instantiates ‘Ivory’ to 4. An inequality removes 4 from ‘Yellow’, instantiating it to 1. ‘Coffee’ equals ‘Green’ and is instantiated simultaneously to 5, removing this value from ‘Water’. ‘Yellow’ equals ‘Kools’ and is instantiated simultaneously to 1, removing this value from the other cigarette brands, ‘Snails’ and ‘Orange_juice’, instantiating the latter to 4, and instantiating ‘Water’ to 1 in turn. The instantiation of ‘Kools’ triggers the constraint from fact 11, instantiating ‘Horse’ to 2. Removal of this value from the other pets instantiates ‘Snails’ and ‘Old_gold’ to 3. The removal of 1 from the cigarette brands instantiates ‘Lucky_strike’ to 4, which in turn removed 4 from ‘Chesterfield’. Now the constraint from fact 10 is enabled to instantiate ‘Fox’ to 1, leaving the only value 5 for ‘Zebra’.

Step 8: The previous assignment instantiated all remaining variables. This is the solution state.

Variable	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
englishman	..345	..3..	..3..	..3..	..3..	..3..	..3..	..3..
spaniard	.2345	.2.45	.2...	.2...	.2...	...4.	...4.	...4.
ukranian	.2.45	.2.45	...45	...4.5	.2..5	.2...	.2...
japanese	.2345	.2.45	...455	...4.	.2..555
norwegian	1....	1....	1....	1....	1....	1....	1....	1....
red	..345	..3..	..3..	..3..	..3..	..3..	..3..	..3..
green	1..45	1..45	1..45	1...5	1..4.	1..45	1..455
blue	.2...	.2...	.2...	.2...	.2...	.2...	.2...	.2...
yellow	1.345	1..45	1..45	1..4.	1...5	1..45	1..4.	1....
ivory	1.345	1..45	1..45	1..45	1..45	1..45	1..45	...4.
tea	.2.45	.2.45	...45	...4.5	.2..5	.2...	.2...
water	12.45	12.45	12.45	12..5	12.4.	12.45	1..45	1....
coffee	1..45	1..45	1..45	1...5	1..4.	1..45	1..455
orange_juice	12.45	12.45	12.45	12...	12...	12.45	1..4.	...4.
milk	..3..	..3..	..3..	..3..	..3..	..3..	..3..	..3..
dog	.2345	.2.45	.2...	.2...	.2...	...4.	...4.	...4.
snails	12345	12345	1.345	1.34.	1.3.5	123.5	123..	..3..
fox	12345	12345	1.345	1.345	1.345	123.5	123.5	1....
horse	12345	12345	1.345	1.345	1.345	123.5	123.5	.2...
zebra	12345	12345	1.345	1.345	1.345	123.5	123.55
kools	1.345	1..45	1..45	1..4.	1...5	1..45	1..4.	1....
parliament	.2345	.2.45	...455	...4.	.2..555
lucky_strike	12.45	12.45	12.45	12...	12...	12.45	1..4.	...4.
chesterfield	12345	12345	12345	1234.	123.5	12345	1234.	.2...
old_gold	12345	12345	1.345	1.34.	1.3.5	123.5	123..	..3..

Figure 21: Zebra puzzle execution trace

Metaprolog computes the first and only solution of the program in figure 19 in 1.9 seconds. The proof for the uniqueness of the solution requires another 3.3 seconds. Although forward checking programs tend to be less sensitive to the instantiation ordering of the variables than generate and test programs, there is some dependence. Therefore we used 200 random permutations of the 25 variables for the labeling stage of the program to gather the statistics of figure 22.

Execution time	First solution	All solutions
Minimum	0.45	1.15
Average	4.25	9.33
Standard Deviation	4.96	7.83
Maximum	32.57	38.87

Figure 22: Execution time statistics in seconds for the Zebra puzzle

5 Implementation

This section deals with the *implementation* of domains and the forward checking inference rule. The reader might prefer to have a look at the examples first and return to the teasing issues of implementation details after being convinced of the utility of the extensions.

In the following specification the representation of sets is kept abstract. Later, realizations of sets and the corresponding operations will be presented.

5.1 The general mechanism

The implementation of domains and forward checking is based on semantic unification as described in [Holzbaur 90]. One particular property of the ‘forward checking theory’⁵ is that variables have an associated domain which is narrowed subsequently. This narrowing is reflected by ‘reassignments’ of new domains.

As our forward checking mechanism is implemented in Prolog, there is no way to actually reassign values to logical variables. The escape from this problem is the concept of ‘open’ data structures. Instead of representing the value of a variable directly, we package it into a structure. The special property of this structure is that it can be extended, i.e., updated with subsequent values. A familiar example of this technique are open tailed lists [Sterling and Shapiro 86]. Such lists can be extended indefinitely by appending any number of open tailed lists. If we assume the convention that the last element of such a list represents its ‘value’ we have a mechanism for nondestructive (re)assignments. Of course, the data structure does not have to be a list — any structure that is extended by a value *and* an unbound variable for subsequent updates works. The process of scanning such a data structure to find its ‘value’ is called ‘dereferencing’.

Metaterms as introduced in [Holzbaur 90] are such open data structures. The first argument⁶ of a metaterm might itself be a metaterm — we proceed the scan for its value recursively in analogy to the open tailed lists. If the first argument is an unbound variable, the value of the metaterm is undetermined. Any other object in the first argument position of a metaterm represents the value of this metaterm. As a result of this coding scheme, a very frequent operation in extended theories is dereferencing such chains of metaterms, for all operations usually apply to the current value of an object, which is found at the end of the chain. For this reason, the convention about the special meaning of the first argument of a metaterm and the process of dereferencing has been included in the specification of *Metaprolog*.

5.2 Domains

Domains are attached to variables by binding them to a metaterm which represents the domain. This new sort of metaterms is introduced in figure 23, line 1. After the sort is

⁵purists read ‘method’

⁶this is just another convention — it could be any argument

introduced, we have to specify how this sort unifies with any other sort. Unifications of ordinary Prolog terms with the new sort are covered by `metatermunify/2` in lines 3 to 5 in figure 23: the predicate succeeds if the term is an element of the domain. In addition, the term is bound to the first position of the metaterm `dom/2`. According to the conventions of *Metaprolog* [Holzbaur 90] this assigns the term as ‘final’ value to the constrained variable for which this metaterm stands. If two domain-variables are to be unified, a new metaterm with the intersection of the two domains is bound to both metaterms, according to section 2. If the intersection consists of just one element this very element is assigned as the final value to both domain-variables (line 11 in figure 23). Empty intersections indicate non-satisfiability, i.e., the unification of the two domain-variables has to fail (line 12 in figure 23).

```

1  :- meta_functor( '$dom'/2).
2
3  metatermunify( '$dom'(Term,Dom), Term) :-
4      set_valid_elem( Term),
5      set_test_membership( Term, Dom).
6
7  metametaunify( '$dom'(V,Dom1), '$dom'(V,Dom2)) :-
8      set_intersection( Dom1, Dom2, Dom3),
9      newdom( Dom3, V).
10
11 newdom( Dom, V ) :- set_singleton( Dom, V), !.
12 newdom( Dom, '$dom'(_,Dom)) :- set_nonempty( Dom).
```

Figure 23: Domains in Metaprolog

The user defines domains and refers to domains via the predicate `domain/2`, which succeeds if the second argument is a list of domain-elements of the first argument of the predicate. There are some examples in figure 24. Line 1 in figure 24 is probably the most

Example:

```

1  | ?- domain(X,[c,a,b]).
2  X = $dom(_159,[a,b,c])
3
4  | ?- domain(X,[c,a,b]),domain(X,Dom).
5  X = $dom(_171,[a,b,c])
6  Dom = [a,b,c]
7
8  | ?- domain(X,[c,a,b]),domain(X,[c,d]).
9  X = c
```

Figure 24: Examples of the use of `domain/2`

common use of `domain/2`: a domain, i.e., a set of possible values, is associated with an unbound variable. The predicate `domain/2` takes care of representing the domain. After

the attachment of the domain, the variable is bound to a metaterm (line 2). Please note that the actual representation of the domain inside the metaterm need not always be as readable as in this example. The predicate `domain/2` may also be used the other way round to refer to the domain of a variable (lines 4 to 6). If the first argument to `domain/2` does already have a domain then it is restricted further by subsequent domain-definitions (line 8). The atom 'c' is the only element in the intersection of the given domains. Figure 25 gives the definition of the predicate `domain/2`. The predicate `domain/2` dispatches on its second argument:

1. The caller of `domain/2` provided a list of domain elements. This list is converted into the internal representation for sets (line 6 in figure 25). Depending on the type of the first argument to `domain/2` the equation in line 7 in figure 25 might give rise to a metaunification.
2. The caller left the second argument uninstantiated. Therefore we try to extract a domain from the first argument (lines 3 and 4 in figure 25). There are five cases that correspond to the five kinds of results of dereferencing a (meta)term [Holzbaur 90] can yield.

```

1  domain( X, List) :-
2      (var( List) ->
3          meta_deref(X,Tx,Dx,Ax),
4          extract_domain( Tx, Dx, Ax, List)
5      );
6      list_to_set( List, Set),
7      X = '$dom'(_,Set)
8  ).
9
10 extract_domain( 1, _, V, [V]) :- set_valid_elem( V).
11 extract_domain( 4, _, V, [V]) :- set_valid_elem( V).
12 extract_domain( T, X, _, Set) :-
13     extract_domain_internal( T, X, Dom),
14     set_to_list( Dom, Set).
15
16 :- syntactic_headunification( extract_domain_internal/3).
17
18 extract_domain_internal( 2, '$dom'(_,Dom), Dom).
```

Figure 25: Predicate `domain/2`

type 1: There was a nonempty chain of metaterms. The last metaterm in the chain has its first argument bound to something other than a metaterm. We call such a metaterm *instantiated*. The predicate `extract_domain/4` receives the first argument of the last metaterm in the chain as its third argument. A list with this argument as its only element is returned if the element is representable as

an element of a set (line 10 in figure 25). Some set implementations might not allow for arbitrary terms as set elements. The astute reader might wonder why this test is needed, as all assignments to the first argument of a metaterm are made via `metatermunify/2` or `metametaunify/2`. If we had just one sort of metaterm as presented so far, the test would be truly redundant, but there are more to come.

type 2: There was a nonempty chain of metaterms. The last metaterm in the chain has its first argument unbound. The ‘value’ of the metaterm is yet undetermined. The domain is extracted from the metaterm via syntactic unification (note line 16 in figure 25) and converted from the internal set representation into a list.

type 3: The first argument to `meta_deref/4` was an unbound variable. There is no clause in `extract_domain/4` covering this case. Therefore a call to `domain/2` with both arguments uninstantiated fails. This is what we want, as unbound⁷ variables cannot possibly have an associated domain.

type 4: The first argument to `meta_deref/4` was neither metaterm nor variable. This is resolved like type 1.

type 5: After skipping an arbitrary number (possibly zero) of metaterms a non-dereferable metaterm was encountered. This type is not used in this application.

This concludes the description of the implementation of finite domains in `Metaprolog`.

⁷as opposed to constrained variables

5.3 Forward checking

It was pointed out in section 2 that the realization of finite domains alone is not yet very powerful. The only constraint covered so far is equality between domains and other terms. This section extends the implementation such that arbitrary⁸ predicates can be used as constraints over variables with finite domains. From the user's point of view, there is one new predicate **forward/1** which accepts a goal as its only argument. This goal is subject to the forward checking inference rule (FCIR).

In order to implement forward checking, it makes sense to analyze the events that can happen to a generalized Prolog variable in some detail. A variable starts out as an unbound variable with no associated domain. Later it might get bound to another variable, to a term, or to a constrained variable (one that has an associated domain). Figure 26 summarizes the possible state transitions. The operational semantics of the forward checking inference

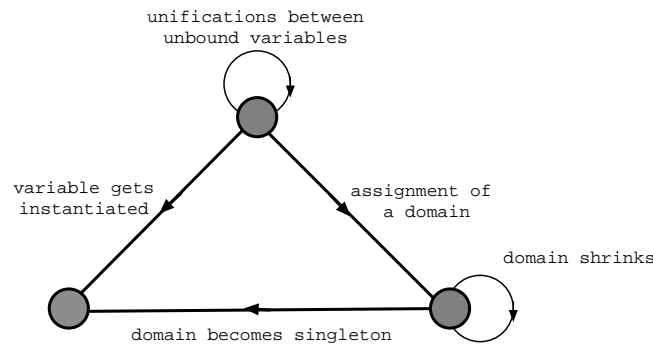


Figure 26: State transition of variables

rule makes it necessary to delay the application until the submitted goal is sufficiently instantiated. These delayed FCIR applications have to be reconsidered on state transition events of the associated variables.

The task of the predicate **forward/1** is to analyze the goal that should be forward checked. In order to verify the applicability of the FCIR and to provide some means to delay and to reconsider FCIR applications, we introduce the new metaterm **\$fwd/3**. It associates goals submitted to **forward/1** with their arguments. The first argument of **\$fwd/3** is the usual placeholder for the final value of the metaterm, the second and third arguments are the parts of a difference list of goals subject to the FCIR in which the metaterm occurs as an argument. The example in figure 27 clarifies why we need a *list* of goals: the variable 'A' occurs in two goals submitted to **forward/1**.

```
1 | ?- forward( A \== 3 ), forward( A < 10 ).
```

Figure 27: **forward/1** application

⁸refer to section 2

The implementation of `forward/1` in figure 28 unifies each argument of the goal with a `$fwd/3` metaterm (line 10). The previous example yields the binding for ‘A’ shown in

```

1 forward( Goal) :-
2   nonvar( Goal),
3   functor( Goal, N, A),
4   functor( Copy, N, A),
5   forward_args( 0, A, Goal, fwdgoal(_Mutex,A,Copy,Goal)).
6
7 forward_args( N, N, _, _ ) :- !.
8 forward_args( N, A, G, Mg) :-
9   N1 is N+1,
10  arg( N1, G, '$fwd'( _, [Mg|Tail], Tail)),
11  forward_args( N1, A, G, Mg).
```

Figure 28: Predicate `forward/1`

figure 29. Actually, a cyclic term structure was created, but `Metaprolog` has means to deal with them. In line 2 in figure 29 we see that the former *variable* ‘A’ is now bound to a *metaterm* `$fwd/3`. The first argument of the metaterm is unbound and the metaterm participates in two goals that are subject to the FCIR. As one can deduce from line 5 of figure 28, this is not the whole, but the essential truth. Additional information that is assembled into the list *facilitates* future computations, from the logical point of view it is redundant.

```

1 | ?- forward( A \== 3 ), forward( A < 10 ).
2 A = $fwd(_57,[_57\==3,_57<10)|_59],_59)
```

Figure 29: `forward/1` application, continued

Each time we introduce new metaterms we have to specify how they socialize with other terms and metaterms. In figure 30 we have the definition of `metatermunify/2` and `metametaunify/2` for `$fwd/3` metaterms to start with. If two `$fwd/3` metaterms

```

1 metatermunify( '$fwd'(Term,Fgs1,_), Term) :-
2   reconsider( Fgs1).
3
4 metametaunify( '$fwd'(NewMeta,Fgs1,FgsTail1),
5   '$fwd'(NewMeta,Fgs2,FgsTail2)) :-
6   append_otl( Fgs1, FgsTail1, Fgs2, FgsTail2, Fgs3, FgsTail3),
7   NewMeta = $fwd(_,Fgs3,FgsTail3).
8
9 append_otl( H, T1, T1, T, H, T).
```

Figure 30: Unification of `$fwd/3` metaterms

meet, their difference lists of pending FCIR applications are appended in constant time

by `append_otl/6`. A new `$fwd/3` metaterm with the resulting list is bound to the first arguments of both metaterms, which effectively replaces the old metaterms by the new one. The connection between the forward checking mechanism and domain variables is introduced through an additional clause for `metametaunify/2` in figure 31. The example

```

1  metametaunify( '$dom'(NewMeta,Dom1),
2                  '$fwd'(NewMeta,Fgs2,FgsTail2)) :-
3      NewMeta = '$dom$fwd'(_,Dom1,Fgs2,FgsTail2),
4      reconsider( Fgs2).
```

Figure 31: Unification of `$dom/2` with `$fwd/3` metaterms

```

1  | ?- domain( A, [1,2,3]),
2      forward( B < 2),
3      A=B.
```

Figure 32: Example: unification of `$dom/2` with `$fwd/3` metaterms

in figure 32 presents the new `metametaunify/2` clause at work. Just before the execution of line 3 in figure 32 we have the following bindings (figure 33). The equation in line 3

```

1  A = $dom(_32,[1,2,3])
2  B = $fwd(_162,[fwdgoal(_151,2,_152<_153,_162<2)|_164],_164)
```

Figure 33: Example: unification of `$dom/2` with `$fwd/3` metaterms

of figure 32 causes a metaunification between the two metaterms ‘A’ and ‘B’. The corresponding clause of `metametaunify/2` from figure 31 replaces both metaterms with a new one that consists of the arguments of the `$dom/2` and the `$fwd/3` metaterm. The resulting metaterm is shown in figure 34.

```

1  A = B =
2      $dom$fwd(_206,[1,2,3],
3              [fwdgoal(_151,2,_152<_153,_206<2)|_164],_164)
```

Figure 34: Example: unification of `$dom/2` with `$fwd/3` metaterms

In our example this combined metaterm has a short life. The reconsideration of FCIR applications that is started by line 4 in figure 31 just after the construction of the new metaterm instantiates it to 1. This is the only element from the domain of ‘A’ that satisfies the constraint $A < 2$. For the user the matters are somewhat simpler and less procedural — she/he observes what is shown in figure 35. And of course, she/he would get the same results for all six permutations of the three goals.

In general, the new metaterm will not be instantiated immediately after its creation. For this reason we have to specify how the new metaterm unifies with other terms and metaterms. The definitions are in figure 36, figure 37 and in figure 38.

```

1 | ?- domain(A,[1,2,3]), forward( B < 2), B=A.
2
3 A = 1
4 B = 1

```

Figure 35: Example: unification of $\$dom/2$ with $\$fwd/3$ metaterms

```

1 metatermunify( '$dom$fwd'(Term,Dom1,Fgs1,FgsTail1), Term) :-
2   metatermunify( '$dom'(_,Dom1), Term),
3   metatermunify( '$fwd'(_,Fgs1,FgsTail1), Term).

```

Figure 36: Unification of $\$dom\$fwd/4$ metaterms with ordinary terms

The fact that the new metaterm $\$dom\$fwd/4$ combines the properties of the metaterms that led to its composition is mirrored by the `metatermunify/2` clauses in figure 36. The metaterm is decomposed and each of the two component metaterms is unified with the term through other clauses of `metatermunify/2` that we saw in figure 23 lines 3 to 5 and in figure 30 lines 1 and 2, respectively. In fact, this formulation is already somewhat optimized. The definition in figure 37 would work just as well. The derivation of the optimized form is left as an exercise to the reader.

```

1 metatermunify( '$dom$fwd'(Term,Dom1,Fgs1,FgsTail1), Term) :-
2   '$dom'(_,Dom1) = Term,
3   '$fwd'(_,Fgs1,FgsTail1) = Term.

```

Figure 37: Unification of $\$dom\$fwd/4$ metaterms with ordinary terms

The definition of `metametaunify/2` has to deal with the new metaterm, too. The new metaterm might be unified with every other metaterm, including itself. The treatment of the combined forms of metaterms during metaunification reduces to their decomposition and the treatment of the resulting already specified base cases. The code in figure 38 follows this idea. The most general case is covered by lines 1 to 5 in figure 38. Both $\$dom\$fwd/4$ metaterms are decomposed into their $\$dom/2$ and $\$fwd/3$ components which are equated in turn (lines 3 and 4).

Although logically sound and concise, this formulation has a few computational disadvantages. In lines 7 to 10 for example, a $\$dom\$fwd/4$ metaterm is decomposed. The $\$dom/2$ component is unified with the other argument to `metametaunify/2` and the equation in line 10 leads to a call to `metatermunify/2` or to `metametaunify/2`, depending on the outcome of the equation in line 9. The bad thing about this is that this final, potential `metametaunify/2` call triggers useless reconsiderations of FCIR applications. The reconsiderations are useless because they were executed once already just after the $\$dom\$fwd/4$ metaterm was composed. All that happened to the $\$dom\$fwd/4$ metaterm now, was that its domain was reduced. This makes FCIRs potentially applicable only if the domain is reduced to a single element. Further, in lines 12 to 15 such FCIR reconsiderations are triggered for both metaterms involved, whereas only one truly received a domain. For these reasons, the *actual* implementation takes care of these special cases.

```

1  metametaunify( '$dom$fwd'(NewMeta,Dom1,Fgs1,FgsTail1),
2                '$dom$fwd'(NewMeta,Dom2,Fgs2,FgsTail2)) :-
3    '$dom'(_,Dom1) = '$dom'(NewMeta,Dom2),
4    '$fwd'(_,Fgs1,FgsTail1) = '$fwd'(NM1,Fgs2,FgsTail2),
5    NM1 = NewMeta.
6
7  metametaunify( '$dom'(NewMeta,Dom1),
8                '$dom$fwd'(NewMeta,Dom2,Fgs2,FgsTail2)) :-
9    '$dom'(_,Dom1) = '$dom'(NewMeta,Dom2),
10   NewMeta = '$fwd'(_,Fgs2,FgsTail2).
11
12 metametaunify( '$dom$fwd'(NewMeta,Dom1,Fgs1,FgsTail1),
13               '$fwd'(NewMeta,Fgs2,FgsTail2)) :-
14   '$fwd'(_,Fgs1,FgsTail1) = '$fwd'(NewMeta,Fgs2,FgsTail2),
15   NewMeta = '$dom'(_,Dom1).

```

Figure 38: Unification of `$dom/2` with `$fwd/3` metaterms, continued

The predicate `reconsider/1` in figure 39 browses through the associated goals of `$fwd/3` and `$dom$fwd` metaterms and tries to apply forward checking to each of the goals in turn. Line 1 recognizes the end of the list, lines 8 and 9 continue the reconsideration if the applicability test in line 4 failed. A goal is forward checkable if there is at most

```

1  reconsider( L ) :- var( L ), !.
2  reconsider( [fwdgoal(Mutex,A,C,G)|Xs] ) :-
3    var( Mutex ),
4    applicable( 0, A, G, C, D, Dd, Ds ),
5    !,
6    apply_fc( D, Dd, Ds, C, Mutex ),
7    reconsider( Xs ).
8  reconsider( [_|Xs] ) :-
9    reconsider( Xs )

```

Figure 39: Predicate `reconsider/1`

one domain variable left among its arguments. The predicate `applicable/7` in figure 40 verifies this condition and builds a copy of the goal in parallel. The need for copying stems from the desired generality of the operation of `forward/1`: any predicate should be usable as a constraint. *Arbitrary* predicates do not know anything about metaterms and the associated conventions — therefore metaterms are stripped off. An alternative convention would put the burden of dealing⁹ with metaterms on the constraint predicates. The chosen convention keeps the corresponding code local to one place in the program. The predicate `applicable/7` is just the iterator for `applicable_one/7` in figure 41. The arguments are the passed on results from `meta_deref/4` of the n-th argument of the goal,

⁹essentially dereferencing

```

1 applicable( N, N, _, _, _, _ ) :- !.
2 applicable( N, A, G, C, D, Ds, Dd) :-
3   N1 is N+1,
4   arg( N1, G, Ga),
5   arg( N1, C, Ca),
6   meta_deref( Ga, Tga, Dga, Aga),
7   applicable_one( Tga, Dga, Aga, Ca, D, Ds, Dd),
8   applicable( N1, A, G, C, D, Ds, Dd).

```

Figure 40: Predicate `applicable/7`

the corresponding n -th argument from the copy of the goal, the final domain variable, the surrogate variable that is used in place of the final domain variable in the copy of the goal, and the domain of the final domain variable. Constants and instantiated metaterms instantiate the corresponding arguments of the copy of the goal (lines 1 and 2 in figure 41). Upon the first occurrence of an uninstantiated metaterm (lines 3 to 7) we verify that it has indeed an associated domain (line 6) and remember it preliminarily as the final domain variable of the goal. If subsequent uninstantiated metaterms are encountered, they must be equal to the one seen already (lines 8 and 9). As we introduced the `domfwd/4` metaterm

```

1 applicable_one( 1, _, X, X, _, _, _).
2 applicable_one( 4, _, X, X, _, _, _).
3 applicable_one( 2, M, X, X, D, X, Dom) :-
4   var( D),
5   !,
6   extract_domain_internal( 2, M, Dom),
7   M = D.
8 applicable_one( 2, _, X, X, _, Ds, _ ) :-
9   Ds == X.

```

Figure 41: Predicate `applicable_one/7`

with a domain component, we have to extend the definitions for the extraction of a domain from a domain variable (figure 42). A goal submitted to `forward/1` can be forward

```

1 :- syntactic_headunification( extract_domain_internal/3).
2
3 extract_domain_internal( 2, '$dom'(_,Dom), Dom).
4 extract_domain_internal( 2, '$dom$fwd'(_,Dom,_,_), Dom).

```

Figure 42: Predicate `extract_domain_internal/3`

checkable because of being ground. In this case, the predicate `applicable/7` succeeds, but the arguments 5 to 7 are left unbound. This is detected by `apply_fc/5` in figure 43 (lines 1 and 2). A missing domain variable indicates a ground goal which is executed via `call/1`. The fifth argument of `apply_fc/5` instantiates the mutual exclusion variable that

is a component of every delayed FCIR application. It is used to inhibit subsequent reconsiderations of FCIRs (line 3 in figure 39). Lines 3 to 5 in figure 43 actually realize the FCIR

```

1  apply_fc( Domvar, _, _, Goal, done) :-
2    var( Domvar), !, call( Goal).
3  apply_fc( Domvar, Dom, Su, Goal, done) :-
4    set_filter( Dom, Su, Goal, NewDom),
5    Domvar = '$dom'(_,NewDom).
```

Figure 43: Predicate `apply_fc/5`

via `set_filter/4`. This predicate successively binds its second argument to the elements from the set in the first argument during backtracking. The second argument also occurs in the third argument, the goal. The enumeration makes the goal ground. It is executed via `call/1`. Instantiations that let the call succeed are collected. The filtered set constitutes the new domain of the final domain variable of the goal submitted to `forward/1`. The

```

1  | ?- domain(A,[1,2,3]), forward( A < 3).
2    ...
3    Call:
4      applicable( 0, 2,
5        $dom$fwd(_186,[1,2,3],[fwdgoal(_153,2,_154<_155,_186<3)|_160],_160)<3,
6        _154<_155,
7        _65922, _65923, _65924)
8    Exit:
9      applicable( 0, 2,
10       $dom$fwd(_154,[1,2,3],[fwdgoal(_153,2,_154<3,_154<3)|_160],_160)<3,
11       _154<3,
12       $dom$fwd(_154,[1,2,3],[fwdgoal(_153,2,_154<3,_154<3)|_160],_160),
13       [1,2,3], _154)
14    Call:
15      apply_fc(
16        $dom$fwd(_154,[1,2,3],[fwdgoal(_153,2,_154<3,_154<3)|_160],_160),
17        [1,2,3], _154,
18        _154<3, _153)
19    Call: set_filter([1,2,3],_154,_154<3,_65934)
20    Exit: set_filter([1,2,3],_154,_154<3,[1,2])
21    ...
22  A = $dom$fwd(_223,[1,2],[fwdgoal(done,2,_223<3,_223<3)|_160],_160)
```

Figure 44: Example: `apply_fc/5`

execution trace in figure 44 shows how `applicable/7` works together with `apply_fc/5`: the goal `A < 3` is determined to be forward checkable. Upon return from `applicable/7` (lines 8 to 13), the second argument of the copy of the goal `_154 < _155` has been instantiated to 3, ‘A’ has been determined as the final domain variable of the goal, its domain is `[1,2,3]`, and the surrogate used in the copy of the goal is the variable `_154`. This variable is instantiated to 1,2 and 3 in turn. The set `[1,2]` passes the filter, which is the resulting

domain of ‘A’. The mutual exclusion variable `_153` is instantiated to ‘done’ (line 22). This prevents reconsiderations of the goal `A < 3` — the constraint was solved once and for all.

5.3.1 Optimizations

In the previous section forward checking was achieved through mere unifications of the arguments of goals submitted to `forward/1` with `$fwd/3` metaterms (figure 28). The ‘real’ actions took place somewhere inside `metatermunify/2` and `metametaunify/2`. The only drawback of this construction is that it triggers too many reconsiderations per goal. Imagine an n-ary constraint with constants or distinct domain variables as arguments. The code in figure 28 leads to a reconsideration of the constraint for each of the arguments. This is clearly suboptimal, as one initial check per goal submitted to `forward/1` is sufficient to determine the applicability of the FCIR. The code in figure 45 eliminates this flaw.

```

1 forward( Goal) :-
2   nonvar( Goal),
3   functor( Goal, N, A),
4   functor( Copy, N, A),
5   forward_n( A, Goal, fwdgoal(_Mutex,A,Copy,Goal)).
6
7 forward_n( 0, Goal, _) :- !, call( Goal).
8 forward_n( A, Goal, fwdgoal(Mutex,_,Copy,_)) :-
9   applicable( 0, A, Goal, Copy, D, Dd, Ds),
10  !,
11  apply_fc( D, Dd, Ds, Copy, Mutex).
12 forward_n( A, Goal, Mg) :-
13  forward_args( 0, A, Goal, Mg).
```

Figure 45: Predicate `forward/1`, new version

A goal submitted to `forward/1` is checked for the applicability of the FCIR only once. Goals or arity zero are trivially executable (line 7). Otherwise the applicability is determined by `applicable/7` and the constraint is then enforced through `apply_fc/5`. In the case of non-applicability, we have to attach `$fwd/3` metaterms to some arguments of the goal. In figure 46 we have an iterator and the predicate `forward_arg/3` for this purpose. Constants and instantiated metaterms can be ignored (lines 11 and 12), free variables are bound to a `$fwd/3` metaterm (line 13). Uninstantiated metaterms are extended according to the corresponding clauses in `metametaunify/2` — except that no reconsiderations are run (lines 15 to 20). The explicit equations are for readability only. In the actual implementation they have been folded into the heads of the clauses. Note that this code produces the same data structures as the earlier version if the FCIR is not yet applicable to a goal. For a forward checkable goal the construction of this data structures is avoided altogether. In figure 47 we have a comparison of the two versions of `forward/1` on some examples from section 4¹⁰ As the arities of the constraints involved are pretty small, the merits of the optimization materialize at higher numbers of constraints only.

¹⁰for those of you who have read ahead a few pages: for this comparison the first version for `neq/2` from section 5.4.1, page 40 was used.

```

1 forward_args( N, N, _, _ ) :- !.
2 forward_args( N, A, G, Mg) :-
3   N1 is N+1,
4   arg( N1, G, Arg),
5   meta_deref( Arg, Tx, Dx, _),
6   forward_arg( Tx, Dx, Mg),
7   forward_args( N1, A, G, Mg).
8
9 :- syntactic_headunification( forward_arg/3).
10
11 forward_arg( 1, _, _).
12 forward_arg( 4, _, _).
13 forward_arg( 3, '$fwd'(_, [Mg|T], T), Mg).
14
15 forward_arg( 2, '$dom'(V, Dom), Mg) :-
16   V = '$dom$fwd'(_, Dom, [Mg|T], T).
17 forward_arg( 2, '$fwd'(V, Fgs, [Mg|T]), Mg) :-
18   V = '$fwd'(_, Fgs, T)
19 forward_arg( 2, '$dom$fwd'(V, Dom, Fgs, [Mg|T]), Mg) :-
20   V = '$dom$fwd'(_, Dom, Fgs, T).

```

Figure 46: Predicate `forward_arg/3`

Benchmark	a) First ver.	b) Optimized ver.	Ratio a/b	Constraints
cube	2.71	2.62	1.03	7
el	7.37	7.07	1.04	11
zebra	13.53	11.09	1.22	54
tennis	6.59	5.21	1.26	91

Figure 47: Execution times in seconds for some benchmarks for the first and the optimized version of `forward/1`

5.4 Inequality

Although inequality constraints can be coded as forward checkable constraints, there are some difficulties with this approach. Check the example in figure 48:

```

1  [Clp] ?- forward( A \== B), A=B.
2
3  A = _24 <Constraint(s): [forward(_24\==_24)]>
4  B = _24 <Constraint(s): [forward(_24\==_24)]>

```

Figure 48: A shortcoming of the forward checking implementation of inequalities

Despite the fact that the variables ‘A’ and ‘B’ are constrained to be distinct, their subsequent equation succeeds. This is because the FCIR is not yet applicable. As soon as a domain or a constant is assigned to one of the two variables, the unsatisfiability will be detected. This situation is somewhat contrary to the philosophy of making use of constraints as early as possible, apart from the inconvenience for the user. In order to compensate for this, we give an implementation of a simplified version `dif/2`. The predicate `dif/2` asserts the inequality between its two arguments which are arbitrary Prolog terms. Classical implementations of `dif/2` have been described by [Boizumault 86, Carlsson 87], [Neumerkel 90] gives an implementation of `dif/2` through metaterms.

For our purposes we will ignore the complications that result from structured arguments to `dif/2` and the possible binding of variables that were arguments to `dif/2` to structures. This simplification is compatible with the restriction for forward checkable constraints. Our simplified version of `dif/2` will be called `neq/2`. Inequality will be implemented via metaterms, the corresponding metaterm is `$ne/2`. The first argument of a `$ne/2` metaterm is the conventional place holder for the final value. The second argument is a list of terms¹¹ from which the variable the metaterm stands for should be distinct. In figure 49 and figure 50 we have the necessary definitions for this simple theory. Two variables are made

```

1  neq( A, B) :-
2    A = '$ne'(_, [B|_]),
3    B = '$ne'(_, [A|_]).
4
5  metatermunify( '$ne'(Term,Rivals1), Term) :-
6    no_such_var( Rivals1, Term, _).
7
8  metametaunify( '$ne'(NewMeta,Rivals1), '$ne'(NewMeta,Rivals2)) :-
9    no_such_var( Rivals1, NewMeta, Last),
10   no_such_var( Rivals2, NewMeta, _),
11   Last = Rivals2,
12   NewMeta = $ne(_,Rivals1).

```

Figure 49: Predicate `neq/2` and corresponding metaterm unifications

¹¹variables, constants and metaterms to be exact

unequal by unifying each of them with a metastructure that has the respective opposite variable in the list of objects from which it should be distinct. If the arguments to `neq/2` are free variables, the equations in lines 2 and 3 of figure 49 simply bind the metaterms to the variables. If on the other hand, one of the arguments is a constant, the equation leads to a call to `metatermunify/2`. The metaterm is instantiated to the constant and the predicate `no_such_var/3` asserts that the constant is *not* among the objects from which the metaterm should be distinct.

Unifications between two `$ne/2` metaterms are treated quite similarly. Through syntactic unification of their first arguments the two metaterms are made equal. This unification could lead to the situation where a metaterm is among it's own list of objects it should be distinct of. The predicate `no_such_var/3` in figure 50 is used to verify this for both metaterms. As a by-product, `no_such_var/3` returns the open tail of the list of distinct objects in case of success (if the new metaterm is not among its own list of objects it should be distinct). Having the tail of one of the two lists, we can append them in constant time (line 11 in figure 49). The lists have to be scanned anyway, therefore an additional argument in the metaterm holding the tail of the list would be a waste of memory. A new metaterm with the union¹² of distinct objects conceptually replaces the two old metaterms that were just unified. The dialog in figure 51 exhibits the new intended behavior of `neq/2`

```

1 no_such_var( L,      _, Last) :- var( L), !, Last = L.
2 no_such_var( [E|Rest], V, Last) :-
3     meta_deref( E, _, _, Ev),
4     Ev \== V,
5     no_such_var( Rest, V, Last).
```

Figure 50: Predicate `no_such_var/3`

in combination with equations.

```

1 | ?- neq( A, B).
2 A = $ne(_16,[_18|_17])
3 B = $ne(_18,[_16|_19])
4
5 | ?- neq( A, B), A=B.
6 no
```

Figure 51: Example: inequalities

In the next step we combine this theory for inequality with the theory for domains. As usual, this is accomplished by additional clauses in `metatermunify/2` and `metametaunify/2` and the introduction of a new, combined metaterm. The code in figure 52 allows us to express constraints as in figure 53. We succeeded allowing for a variable to have a domain and associated inequality constraints simultaneously. But what happened if we instantiated the variable ‘B’ from our example to ‘2’, say? The answer is simple: almost nothing.

¹²Actually, we do not remove duplicates. This operation would be too expensive and duplicates have no influence on the correctness of the theory

```

1 metatermunify( '$dom$ne'(Term,Dom1,Rivals1), Term) :-
2   metatermunify( '$dom'(_,Dom1), Term),
3   metatermunify( '$ne'(_,Rivals1), Term).
4
5 metametaunify( '$dom'(NewMeta,Dom1), '$ne'(NewMeta,Rivals2)) :-
6   no_such_var( Rivals2, NewMeta, _),
7   NewMeta = '$dom$ne'(_,Dom1,Rivals2).

```

Figure 52: Unification of $\$dom/2$ metaterms with $\$ne/2$ metaterms

```

1 | ?- domain(A,[1,2,3]), neq( A, B).
2
3 A = $dom$ne(_158,[1,2,3],[_154|_161])
4 B = $ne(_154,[_158|_155])

```

Figure 53: Combining domain variables and inequalities

In particular, the domain of ‘A’ is *not* reduced by removing the element ‘2’. Nevertheless, the fact that ‘A’ is different from ‘2’ is kept in evidence. This knowledge applies when ‘A’ is eventually instantiated.

To obtain inequality constraints that behave actively in the presence of domain variables, we have to enhance the clause in `metametaunify/2` that describes this very interaction. In figure 54 we have the new version of the essential metaunifications that lead to active inequalities. The predicate `no_such_var_and_cc/6` is used to split the list of inequal

```

1 metatermunify( '$ne'(Term,Rivals1), Term) :-
2   remove_constant( Rivals1, Term).
3
4 metametaunify( '$dom'(NewMeta,Dom1), '$ne'(NewMeta,Rivals2)) :-
5   no_such_var_and_cc( Rivals2, NewMeta, [], Consts, _, Nriv),
6   list_to_set( Consts, CS),
7   set_difference( Dom1, CS, NewDom),
8   newdom_plus_ne( NewDom, NewMeta, Nriv).

```

Figure 54: Active unification of $\$dom/2$ metaterms with $\$ne/2$ metaterms

objects of the $\$ne/2$ metaterm into a list of constants and a list of variables and metaterms when a $\$dom/2$ metaterm is unified with a $\$ne/2$ metaterm (line 5 in figure 54). Additionally, it applies the same check as `no_such_var/3` from figure 50. The constants are removed from the domain via `set_difference/3`. The predicate `newdom_plus_ne/3` fails for empty domains, leads to a `metatermunify/2` call for singleton domains and creates a $\$dom\$ne/3$ metaterm otherwise.

The predicate `metatermunify/2` also needs a slight enhancement. When a $\$ne/2$ metaterm gets instantiated, this very constant has to be removed from the domain variables among the objects from which the $\$ne/2$ metaterm is supposed to be different. This is the task of `remove_constant/2` (figure 55) called from line 2 of figure 54. Additionally, it

applies the same check as `no_such_var/3` (figure 50).

```

1  remove_constant( X, _ ) :- var( X ), !.
2  remove_constant( [X|Rest], Constant ) :-
3      meta_deref( X, Tx, Dx, Ax),
4      remove_constant( Tx, Dx, Ax, Constant),
5      remove_constant( Rest, Constant).
6
7  remove_constant( 1, _, C1, C2 ) :- C1 \== C2.
8  remove_constant( 4, _, C1, C2 ) :- C1 \== C2.
9  remove_constant( 3, _, _, _).
10 remove_constant( 2, Meta, _, C ) :-
11     set_valid_elem( C),
12     extract_domain_internal( 2, Meta, Dom),
13     !,
14     set_remove_elem( Dom, C, NewDom),
15     Meta = $dom( _, NewDom).
16 remove_constant( 2, _, _, _).

```

Figure 55: Predicate `remove_constant/2`

The example in figure 56 shows that the new inequality constraints are actively applied to domain variables. After the instantiation of ‘B’ to ‘2’, this constant is removed from the domain of ‘A’.

```

1  [Clp] ?- domain(A,[1,2,3]), neq( A, B ), B=2.
2
3  A = _214 <Constraint(s): [domain(_214,[1,3])]>
4  B = 2

```

Figure 56: Combining domain variables and ‘active’ inequalities

5.4.1 An exercise in partial evaluation

In this section we are going to partially evaluate the definition of `neq/2` with respect to specification of the implementation of semantic unification ([Holzbaur 90]) and the particular clauses for `metametaunify/2` that have been developed above. Partial evaluation does not change the semantics of programs. Therefore we cannot expect more than constant improvements in terms of algorithmic complexity. On the other hand, when the number of constraints grows with a polynomial of sufficient degree, we are grateful for any tiny constant improvement in basic operations, as this simply stretches our mileage. In the experiment below we compare three versions of `neq/2`:

1. Forward checking is used to implement `neq/2`:

```
neq( A, B ) :- forward( A \== B ).
```


2. The definition from the last section is used:

```

neq( A, B) :-
  A = '$ne'(_, [B|_]),
  B = '$ne'(_, [A|_]).

```

3. A partially evaluated version of the definition from the last section is used.

These three versions of **neq/2** are applied to the following examples:

1. Application of **alldifferent/1** to lists of *unbound* variables of increasing length. The number of inequality constraints is given by $\frac{n(n-1)}{2}$ where n is the length of the list.
2. Application of **alldifferent/1** to lists of *domain* variables of increasing length. The number of inequality constraints is given by $\frac{n(n-1)}{2}$ where n is the length of the list.
3. Computation of permutations of increasing numbers of objects via **alldifferent/1** and **labeling/1**. The number of inequality constraints is given by $\frac{n(n-1)}{2}$ where n is the number of objects. The number of solutions is $n!$. This benchmark is intended to measure how quickly constants can be propagated across inequality constraints.
4. The Tennis puzzle from section 4.4.

The execution times for the three versions of **neq/2** are summarized in figure 57, together with the number of **neq/2** constraints and the number of domain variables and their domain sizes. The table in figure 57 shows that the implementation of **neq/2** via **forward/2** (version

Benchmark	neq/2 ver. 1	neq/2 ver. 2	neq/2 ver. 3	neq/2 Constraints	Domains/Size
alldifferent_10	0.471	1.133	0.233	45	0/0
alldifferent_20	2.058	8.950	1.050	190	0/0
alldifferent_40	9.150	75.850	4.867	780	0/0
alldifferent_80	42.333	-	22.433	3160	0/0
dom_alldifferent_10	0.688	1.721	0.267	45	10/10
dom_alldifferent_20	2.967	10.792	1.133	190	20/20
dom_alldifferent_40	13.050	77.050	5.150	780	40/40
dom_alldifferent_80	59.483	-	23.767	3160	80/80
neq_labeling_3	0.429	0.158	0.154	3	3/3
neq_labeling_4	2.325	0.883	0.875	6	4/4
neq_labeling_5	13.067	5.517	5.510	10	5/5
tennis	5.212	3.533	1.727	91	18/6

Figure 57: Execution times in seconds for some **neq/2** benchmarks

1) is about half as fast in collecting constraints as the third, partially evaluated version. If the variables upon the the constraints are specified are domain variables, there is some overhead in version 1 due to the reconsiderations of the FCIR. The real difference comes with labeling. The forward checking version (1) has to enumerate the elements from the

domains and to filter them. The specialized versions of `neq/2` (2 and 3) use predicates that operate directly on the representations of sets (e.g. `set_difference/3` in figure 54).

Let us now turn to the derivation of the partially evaluated version of `neq/2`. From the definition

```
neq( A, B ) :-
  A = '$ne'(_, [B|_]),
  B = '$ne'(_, [A|_]).
```

we can deduce that `metatermunify/2` or `metametaunify/2` calls may result from the equations between 'A' and 'B' and the `$ne/2` metaterms. Given the specification from [Holzbaur 90], we can set up the calls to the metaunifications ourselves. This explicit treatment allows us to take advantage of some special cases. In figure 58 we have the first part of the unfolded version of `neq/2`. The first step is to dereference both arguments. The

```
1 neq( A, B ) :-
2   meta_deref( A, Ta, Da, Aa),
3   meta_deref( B, Tb, Db, Ab),
4   neq( Ta, Tb, Da, Db, Aa, Ab).
5
6 neq( 1, 1, _, _, A, B ) :- A \== B.
7 neq( 4, 4, _, _, A, B ) :- A \== B.
8 neq( 1, 4, _, _, A, B ) :- A \== B.
9 neq( 4, 1, _, _, A, B ) :- A \== B.
```

Figure 58: Partially evaluated version of `neq/2`, part 1

predicate `neq/6` dispatches on the pairs of types of the dereferenced arguments. If `neq/2` was called with a pair of constants (type 4) or instantiated metaterms (type 1), we just verify the inequality.

Derivation: As both arguments are constants or instantiated metaterms, this would have led to `metatermunify/2` calls in the original version. For example, the call `neq(1,2)` would have led to the calls:

```
metatermunify( '$ne'(_, [2|_]), 1),
metatermunify( '$ne'(_, [1|_]), 2)
```

which reduce in turn to:

```
remove_constant( [2|_], 1),
remove_constant( [1|_], 2)
```

Applying the definition of `remove_constant/2` from figure 55 we arrive at the unfolded version from above. The symmetry of the inequality constraint allows us to drop the second inequality with flipped arguments.

What did we gain through partial evaluation? The construction of intermediate metaterms was avoided. As the length of the list of different objects for both metaterms was known

at partial evaluation time, the recursion for the traversal of this list could be unfolded. A symmetry was uncovered and used to remove a redundant test. In figure 59 we have the clauses of `neq/6` that deal with unbound variables (type 3).

Derivation: For the equation between the unbound variable and the `$ne/2` metaterm we know from [Holzbaur 90] that the variable is just bound to the metaterm. The equation between the constant and the second `$ne/2` metaterm leads to a metatermunification. As in the previous case, the length of the list of different objects is known. The only resulting inequality test can be dropped, as a constant and a variable are always different under the Prolog builtin inequality predicate.

If the inequality relates two unbound variables (lines 6 to 9 in figure 59), this test has to be made, however.

```

1 neq( 3, 1, _, _, '$ne'(_, [C|_]), C).
2 neq( 3, 4, _, _, '$ne'(_, [C|_]), C).
3 neq( 1, 3, _, _, C, '$ne'(_, [C|_])).
4 neq( 4, 3, _, _, C, '$ne'(_, [C|_])).
5
6 neq( 3, 3, _, _, A, B) :-
7     A \== B,
8     A = '$ne'(_, [B|_]),
9     B = '$ne'(_, [A|_]).

```

Figure 59: Partially evaluated version of `neq/2`, part 2

Figure 60 treats inequalities between uninstantiated metaterms (type 2) and constants (types 1 and 4). The constant is removed from metaterms that have an associated domain (lines 8 to 12 and 15 to 19) if the constant is a possible set element¹³. If the metaterm is a `$ne/2` structure, the constant is added to the list of different objects (lines 13 and 14). The last clause of `neq_remove_constant/2` is not to catch other metaterms, but to cover the cases where the constant is not a representable set element.

Derivation: In a first step, `neq_remove_constant/2` could be rewritten as:

```

neq_remove_constant( Meta, Const) :-
    metatermunify( '$ne'(_, [M|_]), C),
    M = '$ne'(_, [C|_]).

```

The first equation is guaranteed to provoke a `metatermunify/2` call. For the second equation we cannot tell at partial evaluation time whether it would lead to a `metatermunify/2` or a `metametaunify/2` call, as this depends on what the first equation does to the metaterm. In any case the second equation would not

¹³if the set implementation would allow only integers as set elements, there would be no need (and possibility) to remove symbols from such sets

provide any new information to the metaterm — therefore it can be omitted. In a second step, we use the definition of `metatermunify/2`. This results in the case analysis of figure 60.

```

1 neq( 2, 1, M, _, _, C) :- neq_remove_constant( M, C).
2 neq( 2, 4, M, _, _, C) :- neq_remove_constant( M, C).
3 neq( 1, 2, _, M, C, _) :- neq_remove_constant( M, C).
4 neq( 4, 2, _, M, C, _) :- neq_remove_constant( M, C).
5
6 :- syntactic_headunification( neq_remove_constant/2).
7
8 neq_remove_constant( '$dom'(V,Dom1), Const) :-
9   set_valid_elem( Const),
10  !,
11  set_remove_elem( Dom1, Const, NewDom),
12  newdom( NewDom, V).
13 neq_remove_constant( '$ne'(V,Rivals1), Const) :- !,
14  V = '$ne'(_, [Const|Rivals1]).
15 neq_remove_constant( '$dom$ne'(V,Dom1,Rivals1), Const) :-
16  set_valid_elem( Const),
17  !,
18  set_remove_elem( Dom1, Const, NewDom),
19  '$dom$ne'(V,Dom1,Rivals1) = '$dom'(_,NewDom).
20 neq_remove_constant( _, _).
```

Figure 60: Partially evaluated version of `neq/2`, part 3

In figure 61 we have the cases that treat inequalities between free variables and uninstantiated metaterms and finally, inequalities between uninstantiated metaterms. The code

```

1 neq( 2, 3, M, _, V1, V2) :-
2   V2 = '$ne'(_, [V1|_]),
3   neq_metoo( M, V2).
4 neq( 3, 2, _, M, V2, V1) :-
5   V2 = '$ne'(_, [V1|_]),
6   neq_metoo( M, V2).
7
8 neq( 2, 2, M1, M2, V1, V2) :-
9   V1 \== V2,
10  neq_metoo( M1, V2),
11  neq_metoo( M2, V1).
```

Figure 61: Partially evaluated version of `neq/2`, part 4

for `neq_metoo/2` is given in figure 62. The explicit equations in lines 4, 6 and 8 are for readability only. In the actual implementation they have been folded into the heads of the clauses.

```

1  :- syntactic_headunification( neq_metoo/2).
2
3  neq_metoo( '$dom'(V,Dom1), Other) :-
4    V = '$dom$ne'(_,Dom1,[Other|_]).
5  neq_metoo( '$ne'(V,Rivals1), Other) :-
6    V = '$ne'(_,[Other|Rivals1]).
7  neq_metoo( '$dom$ne'(V,Dom,Rivals1), Other) :-
8    V = '$dom$ne'(_,Dom,[Other|Rivals1]).

```

Figure 62: The predicate `neq_metoo/2`

Derivation: In the case of an inequality between two uninstantiated meta-terms we can rewrite

```

neq( 2, 2, '$ne'(V1n,R1), '$ne'(V2n,R2), _, _ ) :-
  '$ne'(V1n,R1) = '$ne'(V3n,['ne'(V2n,R2)|_]),
  '$ne'(V2n,R2) = '$ne'(V4n,['ne'(V1n,R1)|_]).

```

into:

```

1  V1n = V3n,
2  no_such_var( R1, V1n, Last1),
3  no_such_var( ['ne'(V2n,R2)|_], V1n, _),
4  Last1 = ['ne'(V2n,R2)|_],
5  V1n = '$ne'(_,['ne'(V2n,R2)|R1]),
6  V2n = V4n,
7  no_such_var( R2, V2n, Last2),
8  no_such_var( ['ne'(V1n,R1)|_], V2n, _),
9  Last2 = ['ne'(V1n,R1)|_],
10 V2n = '$ne'(_,['ne'(V1n,R1)|R2]).

```

Lines 1 to 5 are from the unfolded body of the `metametaunify/2` (figure 49) call for the first equation, the remaining lines result from unfolding the second equation. The statement in line 2 resolves into `true` if we apply the invariant that a metaterm is kept different from the objects it should differ from¹⁴. The same argument applies to line 3. The remaining code for line 3 is `V1n \== V2n`. Line 4 would have appended the new `$ne/2` metaterm to list 'R1' by binding it to the tail that would have been determined in line 2. This statement is replaced by `true` — therefore we do not know the tail of 'R1'. Luckily, the append operation is commutative in our application. The tail of the other list has not even to be searched for, it is known statically. We can therefore drop line 4 with the result of the append operation given in line 5. The reduction of the statements in lines 6 to 10 is analogous. Thus:

¹⁴the metaterm cannot possibly be introduced in its own list of differing objects through the unification in line 1, because 'V3n' is a fresh variable

```

neq( 2, 2, '$ne'(V1n,R1), '$ne'(V2n,R2), _, _) :-
    V1n \== V2n,
    V1n = '$ne'(_, [V2n|R1]),
    V2n = '$ne'(_, [V1n|R2]).

```

The derivation for an inequality between an unbound variable (type 2) and an uninstantiated metaterm is quite similar. The derivations for the remaining metaterms (`$dom/2` and `$dom$ne`) lead us to the cases summarized in figure 62.

5.5 The combination of the theories for forward checking and inequality

So far we created the two theories for forward checking and inequalities in isolation. We now have to combine the two theories so that forward checking *and* inequalities can act *simultaneously* on any variable. With the current definitions just taken together, the example in figure 63 would otherwise simply fail. We already had an example for the combination

```

1  [Clp] ?- neq( A, 1), forward( A < 3).
2
3  A = _32 <Constraint(s): [neq(_32,1),forward(_32<3)]>

```

Figure 63: Combining constraints

of semantic theories: the theory for inequalities was joined with the theory of domains. In figure 64 we have the `metametaunify/2` clause that connects forward checking with inequalities. The combination is rather passive. The new metaterm `fwdne/4` consists of the fields of its constituent metaterms. The introduction of this metaterm leads to a further sort `domfwd$ne/5` which has a domain in addition. To deal with all the combined metaterms we have to specify seven clauses for `metatermunify/2` and 28 clauses for `metametaunify/2`. The skeletons for these clauses were generated by a little program and

```

1  metametaunify( '$fwd'(NewMeta,Fgs1,FgsTail1),
2                '$ne'(NewMeta,Rivals2)) :-
3      no_such_var( Rivals2, NewMeta, _),
4      NewMeta = '$fwd$ne'(_,Fgs1,FgsTail1,Rivals2).

```

Figure 64: Joining forward checking with inequalities

some of the bodies of the clauses were taken from the basic cases such as figure 64.

In general, the combination of n basic metatermsorts yields $2^n - 1$ clauses for `metatermunify/2`. This is just the cardinality of the powerset of the set of basic metaterms¹⁵. Each pair of elements from the powerset has to be covered by clauses of `metametaunify/2`.

$$\text{Number of metametaunify/2 clauses} = \binom{2^n - 1}{2} + 2^{n-1} = (2^n - 1)2^{n-1}$$

n	mtuc	mmuc
1	1	1
2	3	6
3	7	28
4	15	120
5	31	496
6	63	2016
7	127	8128
8	255	32640
9	511	130816
10	1023	523776

Figure 65: Number of `metatermunify/2` clauses (mtuc) and `metametaunify/2` clauses (mmuc) for n metaterms

The numbers for some small n are in figure 65. This clearly shows that it is not very adequate to combine more than about four basic metaterm sorts. The parole for programming with metaterms is therefore: ‘keep the number of metaterm sorts small’. This can be achieved by using more general metaterms and introducing neutral elements for the fields of the general metaterm. In our current scenario this would amount to dropping all but the `dom fwd$ne/5` metaterm and using the universal domain and empty lists as neutral elements.

¹⁵excluding the empty set — therefore ‘−1’

5.6 Labeling

By now the language extensions allow us to talk about domains and constraints. The constraints will in general reduce the search space, but choices might still be left. Besides choices in the execution state of the Prolog implementation, there might be remaining choices for the values of domain variables that manifest themselves as non-singleton domains. In order to compute ground solutions, values for these variables have to be selected. We call this process *labeling*. Labeling could be coded with the predicates we already have defined: `domain/2` gives us the domain of a domain variable and `member/2` can be used to enumerate the elements in a nondeterministic way. Line 3 in figure 66 looks quite harmless. The subtle point is that ‘X’ is not just bound to elements from the domain — ‘X’ is a domain variable! Therefore, we have unifications between metaterms and constants. In figure 67 is a fragment of the execution trace of this example. Logically there is nothing

```

1  | ?- domain(X,[a,b,c]),
2      domain(X,Dom),
3      member(X,Dom).
4
5  X = a
6  Dom = [a,b,c] ;
7
8  X = b
9  Dom = [a,b,c] ;
10
11 X = c
12 Dom = [a,b,c]
```

Figure 66: Labeling example

ing wrong with this situation, i.e., these metaunifications. From the computational point of view it is somewhat annoying that the metaunifications verify something we already know: that the elements are valid members of the domain of the variable. For this reason,

```

(13) 1 Call: member( $dom(_137,[a,b,c]), [a,b,c])
(13) 1 Exit: member( $dom(_137,[a,b,c]), [a,b,c])
(14) 1 Call: metatermunify( $dom(_137, [a,b,c]), a)
      % what happens here depends on the implementation of sets
(14) 1 Exit: metatermunify( a, a)
```

Figure 67: Labeling example trace

and to ease labeling for the user, we introduce the predicate `indomain/1`. This predicate `indomain/1` provides a little bit more than labeling as described above. It does not only instantiate single domain variables, but accepts general terms that might contain domain variables. These domain variables are instantiated nondeterministically one by one, resulting in term instances that are ground with respect to the domain variables. The application of `indomain/1` in figure 68 generates all instances of the `turtle/2` term.


```

1  ?- domain( Direction, [north,east,south,west]),
2      domain( Distance, [5, 10, 100]),
3      X = turtle( turn(Direction), move(Distance)).
4
5  X = turtle(turn($dom(_227,[east,north,south,west])),
6             move($dom(_336,[5,10,100])))
7
8  ?- domain( Direction, [north,east,south,west]),
9      domain( Distance, [5, 10, 100]),
10     X = turtle( turn(Direction), move(Distance)),
11     indomain( X).
12
13 X = turtle(turn(east),move(5)) ;
14 X = turtle(turn(east),move(10)) ;
15 X = turtle(turn(east),move(100)) ;
16 X = turtle(turn(north),move(5)) ;
17 X = turtle(turn(north),move(10)) ;
18 ...

```

Figure 68: Predicate `indomain/1` example

`indomain/1` is defined in figure 69. The argument is dereferenced and handed to `structure_labeling/3`. If the argument is an instantiated metaterm we recurse on its value, which might be a structure containing domain variables (lines 7 to 9). The same procedure applies to non-metaterms (lines 10 to 12). Unbound variables are not considered during labeling (line 18). The elements from variables with an associated domain are enumerated via `set_enumerate/2`. In figure 69 we show only the most complex case for `dom fwd$ne/5` metaterms (lines 14 to 17). The code for lines 16 and 17 has been taken from `metatermunify/2` (figure 30 and figure 54). Uninstantiated metaterms that do not have an associated domain are ignored during labeling (line 18). The iterator predicate for structures is listed in figure 70.

In many applications one has a list of domain variables that all have to be labeled. The predicate `labeling/1` can easily be formulated in terms of `indomain/1` (figure 71).

```

1  indomain( X) :-
2    meta_deref( X, Tx, Dx, Ax),
3    structure_labeling( Tx, Dx, Ax).
4
5  :- syntactic_headunification( structure_labeling/3).
6
7  structure_labeling( 1, _, X) :- !,
8    functor( X, _, A),
9    structure_labeling_args( 0, A, X).
10 structure_labeling( 4, _, X) :- !,
11   functor( X, _, A),
12   structure_labeling_args( 0, A, X).
13 % some clauses removed
14 structure_labeling( 2, '$dom$ fwd$ne'(V,Dom,Fds,_,Rivals), _) :- !,
15   set_enumerate( Dom, V),
16   remove_constant( Rivals, V),
17   reconsider( Fds).
18 structure_labeling( _, _, _).

```

Figure 69: Predicate `indomain/1`

```

1  structure_labeling_args( N, N, _) :- !.
2  structure_labeling_args( N, A, F) :-
3    N1 is N+1,
4    arg( N1, F, X),
5    meta_deref( X, Tx, Dx, Ax),
6    structure_labeling( Tx, Dx, Ax),
7    structure_labeling_args( N1, A, F).

```

Figure 70: Predicate `indomain/1`, continued

```

1  labeling([]).
2  labeling([V|Vs]) :-
3    indomain(V),
4    labeling(Vs).

```

Figure 71: Predicate `labeling/1`

5.7 Set implementations

In the previous sections we gratuitously ignored the essential question of how to represent sets. The basic decision to be made is, whether we ought to use a direct representation of sets. In a direct representation, the set elements represent themselves and every Prolog term is a possible set element. If we alternatively choose an indirect representation as bitvectors, we have to provide a mapping between the elements of the sets to their bit-positions. This mapping can be a global one that maps *all* constants from the Herbrand model to integers. Finally, we can require set elements to be integers in the first place. This relieves the implementor from managing a global mapping.

5.7.1 Direct representation of sets as ordered lists

In the direct representation, we can use ordinary lists to represent sets. The disadvantage of this simple approach is, that the complexity of the computation of the intersection of such sets is proportional to the product of the cardinalities. We can do better than that, under the premise that the sets are kept sorted according to the order induced by the built-in predicate `compare/3`¹⁶. The computation of intersections can then be done in time proportional to the sum of the cardinalities of the sets.

The procedure for the computation of the intersection of two ordered sets is shown in figure 72. The predicate succeeds when the third argument is the ordered representation of the intersection of the first and second argument, provided that the first two arguments are themselves ordered sets. As it turns out, this definition is equivalent to a `merge/3`

```

1  set_intersection( _, [], [] ) :- !.
2  set_intersection( [], _, [] ) :- !.
3  set_intersection( [Head1|Tail1], [Head2|Tail2], Intersection) :-
4      compare( Order, Head1, Head2),
5      set_intersection( Order, Head1, Tail1, Head2, Tail2, Intersection).
6
7  set_intersection( =, Head, Tail1, _, Tail2, [Head|Intersection]) :-
8      set_intersection( Tail1, Tail2, Intersection).
9  set_intersection( <, _, Tail1, Head2, Tail2, Intersection) :-
10     set_intersection( Tail1, [Head2|Tail2], Intersection).
11 set_intersection( >, Head1, Tail1, _, Tail2, Intersection) :-
12     set_intersection( [Head1|Tail1], Tail2, Intersection).
```

Figure 72: Predicate `set_intersection/3`

predicate that removes duplicates.

How do we create sets in the first place? This task is taken care of by `domain/2` (figure 25). The definition of `list_to_set/2` in figure 73 utilizes the built-in predicate `sort/2` to sort the list of domain values and to get rid of duplicates.

¹⁶this excludes variables as set elements as their instantiation might invalidate the ordering

```

1 list_to_set( L, Set) :-
2   sort( L, Set).

```

Figure 73: Predicate `list_to_set/2`

The reverse conversion from sets to lists is accordingly simple. The predicate `set_to_list/2` is in figure 74.

```

1 set_to_list( List, List).

```

Figure 74: Predicate `set_to_list/2`

The fact that every Prolog term can be an element of a set is expressed via `valid_set_elem/1` in figure 75.

```

1 set_valid_elem( _).

```

Figure 75: Predicate `valid_set_elem/1`

The recognition of non-empty and singleton sets can be achieved by mere unification, too (figure 76). Equality of sets can be tested via the built-in predicate `==/2`, due to the fact

```

1 set_nonempty( [_|_]).
2 set_singleton( [E1], E1).

```

Figure 76: Predicates `set_nonempty/1` and `set_singleton/2`

that they are sorted.

The definition of `set_test_membership/2` is given in figure 77. The predicate succeeds if the first argument equals the first element of the second argument. If the first argument is greater than the head of the second argument, we recurse on the tail of the latter. As the list is sorted, an element which is smaller than the first element cannot be a member of the tail – therefore the predicate `set_test_membership/2` fails as there is no clause covering this case.

```

1 set_test_membership( Elem, [Car|Cdr]) :-
2   compare( Order, Elem, Car),
3   set_test_membership( Order, Elem, Cdr).
4
5 set_test_membership(=, _, _).
6 set_test_membership(>, E, S) :- set_test_membership(E,S).

```

Figure 77: Predicate `set_test_membership/2`, first version

This implementation of `set_test_membership/2` thus also exploits the fact that our sets are ordered. Nevertheless, the simpler version in figure 78 is faster¹⁷ despite the fact that it makes no use of the ordering. Note that the two versions are not fully equivalent, as the second version in figure 78 succeeds if the first argument is a variable, even if the variable is not a member of the list.

```
1 set_test_membership( E, [E|_] ) :- !.
2 set_test_membership( E, [_|T] ) :-
3   set_test_membership( E, T ).
```

Figure 78: Predicate `set_test_membership/2`, second version

For the same reasons of performance, we prefer the second version of `set_remove_elem/3` in figure 80 to the version in figure 79.

```
1 set_remove_elem( [], _, [] ).
2 set_remove_elem( [Car|Cdr], Elem, Res ) :-
3   compare( Order, Elem, Car ),
4   set_remove_elem( Order, Elem, Car, Cdr, Res ).
5
6 set_remove_elem( =, _, _, Res, Res ).
7 set_remove_elem( >, Elem, Car, Cdr, [Car|Res] ) :-
8   set_remove_elem( Cdr, Elem, Res ).
9 set_remove_elem( <, _, Car, Cdr, [Car|Cdr] ).
```

Figure 79: Predicate `set_remove_elem/3`, first version

```
1 set_remove_elem( [], _, [] ).
2 set_remove_elem( [E|T], E, T ) :- !.
3 set_remove_elem( [X|T], E, [X|Ts] ) :-
4   set_remove_elem( T, E, Ts ).
```

Figure 80: Predicate `set_remove_elem/3`, second version

The code for nondeterministic enumeration of elements from a set is in figure 81.

```
1 set_enumerate( [E1|_] , E1 ).
2 set_enumerate( [_|T] , E1 ) :- set_enumerate( T, E1 ).
```

Figure 81: Predicate `set_enumerate/2`

The predicate for filtering sets is `set_filter/4` (figure 82). The arguments are: a set, a surrogate variable, a goal and the resulting filtered set. The predicate is derived from `bagof/3`. The difference is that there is no need to check for free variables in the goal, as the only free variable is the surrogate variable, according to the calling conventions of this predicate. The surrogate variable is instantiated to successive set elements through

`set_enumerate/2` in line 3 of figure 82. An example for the application of `set_filter/4` is in figure 83. The surrogate variable is bound to successive elements from the set in a failure driven loop. Bindings that allow the goal `Su < 3` to succeed are collected in the database. Together they constitute the resulting set `[1,2]`.

5.7.2 Indirect representation of small sets as integer numbers

In the introduction to this section we mentioned that one possible indirect set representation are bitvectors. Set elements are mapped to bit-positions and the corresponding bit indicates the presence or absence of the element in the set. It would be convenient if bitvectors were supported as a basic data type in Prolog implementations. Currently this is still rare, so we restrict ourselves to small sets with a maximal cardinality of 29. The magic number 29 comes from the fact that **C-Prolog** has that many bits available in its representation of integer numbers. This leads to the definition of `valid_set_elem/1` in figure 84.

Using to this representation, the computation of intersections is of constant complexity (figure 85).

On the other hand, the construction of a set from a list of elements is slightly more involved (figure 86). We start with an empty set in line 2 and add valid elements (line 8) by setting the corresponding bit to one (line 12). If there is a non-representable element in the list, the predicate `list_to_set/2` fails.

A nonempty set has at least one bit set, i.e., the corresponding number is different from zero¹⁷ (figure 87).

Sets of a single element are enumerated by `set_singleton/2` (figure 88), together with the corresponding element.

Testing for membership and removing an element from a set are reduced to simple arithmetic evaluations (figure 89 and figure 90).

The nondeterministic enumeration of elements from a set proceeds by testing corresponding bits.

¹⁷at least for our examples

¹⁸in the current representation of integers in **C-Prolog**

```

1  set_filter( Set, X, P, _ ) :-
2    recorda( '$bag', '$bag', _),
3    set_enumerate( Set, X),
4    call( P),                      % once( P)
5    recorda( '$bag', X, _),
6    fail.
7  set_filter( _, _, _, NewSet) :-
8    reap( [], NewSet).
9
10 reap( L0, L) :-
11   recorded( '$bag', X, Ref), erase( Ref), !,
12   reap1( X, L0, L).
13
14 reap1( '$bag', L0, L) :- !, L0 = L.
15 reap1( X, L0, L) :- reap( [X|L0], L).

```

Figure 82: Predicate `set_filter/4`

```

1  | ?- set_filter( [1,2,3], Su, Su < 3, New).
2  Su = _8
3  New = [1,2]

```

Figure 83: Example for the use of `set_filter/4`

```

1  set_valid_elem( X) :-
2    integer( X),
3    X >= 0,
4    X <= 28.

```

Figure 84: Predicate `valid_set_elem/1`

```

1  set_intersection( A, B, C) :- C is A /\ B.

```

Figure 85: Predicate `set_intersection/3`

```

1  list_to_set( L, Set) :-
2    set_empty( S),
3    list_to_set( L, S, Set).
4
5  list_to_set( [], S, S).
6  list_to_set( [X|Xs], In, Out) :-
7    set_valid_elem( X),
8    set_add_elem( In, X, In1),
9    list_to_set( Xs, In1, Out).
10
11 set_add_elem( Set, Elem, NewSet) :-
12   NewSet is Set \/ 1 << Elem.
13
14 set_empty(0).

```

Figure 86: Predicate `list_to_set/2`

The enumeration in figure 92 shows that the integer 6 represents the set with the elements 1 and 2.

```

1  | ?- set_enumerate( 2'110, E1).
2  E1 = 1 ;
3  E1 = 2

```

Figure 92: Enumeration example

The new `set_filter/4` predicate differs from the version of the last section in the way the elements of the filtered set are collected. At any time, only one entry of the database is used. This entry represents the whole filtered set. The advantage is that the space requirements are constant. This solution is adequate, because the sets, i.e., integer numbers, are directly represented by the Prolog implementation, as opposed to lists in the former case.

```

1  set_filter( _, _, _, _ ) :-
2      set_empty( S),
3      recorda( '$set', S, _),
4      fail.
5  set_filter( S, X, P, _ ) :-
6      set_enumerate( S, X),
7      call( P),
8      set_filter_add_solution( X),
9      fail.
10 set_filter( _, _, _, NewSet) :-
11     recorded( '$set', NewSet, Ref),
12     erase(Ref),
13     !.
14
15 set_filter_add_solution( X) :-
16     recorded( '$set', Set, Ref),
17     erase( Ref),
18     set_add_elem( Set, X, Set1),
19     recorda( '$set', Set1, _),
20     !.

```

Figure 93: Predicate `set_filter/4`

A `set_filter/4` example equivalent to the one in figure 83 is given in figure 94. We see that the integer 14 represents the set of numbers from 1 to 3. The filtered set is represented by the integer 6 (see figure 92).

```
1 | ?- list_to_set( [1,2,3], Set),  
2     set_filter( Set, Su, Su < 3, New).  
3  
4 Set = 14  
5 Su = _13  
6 New = 6
```

Figure 94: Example for the use of `set_filter/4`

5.7.3 Indirect representation of large sets, realized as bitvectors

Currently, bitvectors are not frequently present as a basic data type in Prolog implementations. What one can do in the meantime is to implement larger sets from integers as basic building blocks. Available data structures to this purpose are lists of integers, terms of higher arity with integers as arguments, or trees with integer leaves. Lists do not allow indexed access to the elements, access time is of $O(n)$, space requirements for updates are of $O(n)$. Terms allow for indexed access via `arg/3`, therefore we have the favorable access time of $O(1)$, but still update space requirements of $O(n)$. Logarithmic arrays¹⁹ [Warren 83] have access costs of $O(\log n)$ and update space requirements of $O(\log n)$. In addition, such arrays work reasonably well with sparse indices. The predicate in figure 95 shows the general pattern for operations with this realization of bitvectors. An index is split into a bit-offset within an integer and an index of the integer in the logarithmic array. Operations upon integers can be taken from the last section, everything else is provided by the implementation of logarithmic arrays.

```

1  set_add_elem( Set, Elem, NewSet) :-
2    Bit is Elem mod 29,
3    Word is Elem // 29,
4    aref_zero( Word, Set, Theword),
5    New_word is Theword \ 1 << Bit,
6    aset( Word, Set, New_word, NewSet).
```

Figure 95: Predicate `set_add_elem/3` for bitvectors

¹⁹a kind of radix tree, as this data structure could descend from a radix sorting algorithm[Knuth 75]

5.7.4 Folding the set representation into programs

After having made a commitment for one particular set representation, we can fold this representation into the programs that implement forward checking. For example, take the implementation of sets via ordered lists and the code fragment from figure 23. It can be rewritten to what we have in figure 96. Both clauses of `newdom/2` reduce to unit clauses. Set

```

1  :- meta_functor( '$dom'/2 ).
2
3  metatermunify( '$dom'(Term,Dom), Term) :-
4    set_test_membership( Term, Dom ).
5
6  metametaunify( '$dom'(V,Dom1), '$dom'(V,Dom2)) :-
7    set_intersection( Dom1, Dom2, Dom3 ),
8    newdom( Dom3, V ).
9
10 newdom( [V], V ).
11 newdom( [E1,E2|Es], '$dom'(_, [E1,E2|Es])) .

```

Figure 96: Domains in `Metaprolog`, unfolded set representation

properties such as singularity and non-emptiness are determined through mere unification. Two predicate calls and a cut are saved.

5.8 Compilation

At the time of writing, **Metaprolog** is implemented in **C-Prolog**. For the purpose of performance comparisons with a special purpose prototype implementation ([Hentenryck and Dincbas 86, Hentenryck 89]) this is a very bad starting point as **C-Prolog** is ‘just’ a Prolog *interpreter*, whereas the prototype encodes domain operations in C. Although this paper is primarily motivated by the software technological advantages of an executable Prolog specification for forward checking, we will not dismiss the reader without at least giving an estimate of the achievable performance when employing an optimizing Prolog compiler.

Our forward checking code does not contain any unusual Prolog constructs. Therefore one can expect the common speedups from compilation. The other half of the story is that the Prolog environment hosting the compiler has to be modified as **C-Prolog** actually was, in order to provide the functionality of **Metaprolog**. There is an executable specification for this part, too ([Holzbaur 90]), but it is not meant and suited for direct execution. The global idea for the provision of **Metaprolog** functionality is to take a decent host Prolog implementation, apply the relatively small patches to allow for user defined semantic unification and stop hacking in C at this point. This is somewhat contrary to the construction of a special purpose (prototype) interpreter from scratch. The problem with high performance Prolog systems is that they are commercialized, i.e., the source code is not available.

What will be presented in the sequel is an *emulation* of **Metaprolog** in compiled **Quintus Prolog** Prolog. As it turns out, the operation of dereferencing metaterm chains, which is a kernel functionality of **Metaprolog**, soon dominates the emulation. The costs for dereferencing metaterm chains are of linear complexity. The implementation constants are summarized in figure 97. The differences between the **C-Prolog Metaprolog** kernel and the emulation in **Quintus Prolog** stem from the fact that in **Metaprolog** `meta_deref/4` is a builtin predicate call which executes a tight loop, comparable to the loop that dereferences ‘ordinary’ Prolog variables. In the emulation, however, every dereferencing step necessitates a procedure call.

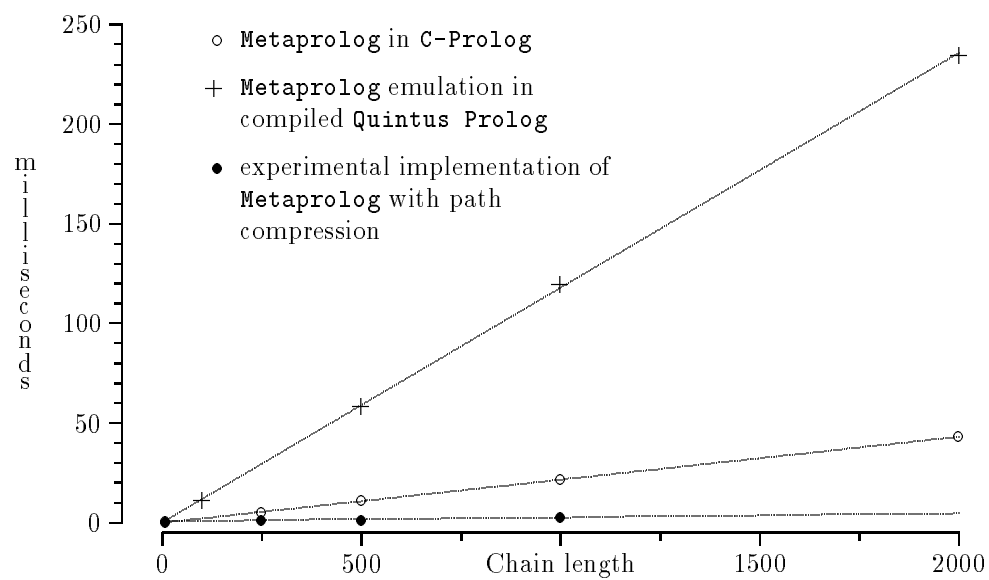


Figure 97: Dereference times in milliseconds vs. metaterm chain length

Given this model for dereferencing costs, we can split the execution time of programs into the portion spent in the kernel emulation (`kernel_time`) and the remaining ‘pure’ Prolog execution, if we know how many dereference operations on chains of certain lengths arise. This ‘*deref profile*’ is characteristic for a program and independent from the kernel implementation. The kernel execution time for one particular deref profile is computed as:

$$\text{kernel_time} = \sum_{\text{len}=1}^{\infty} \text{Calls}_{\text{len}}(k\text{len} + d)$$

The constants k and d are the parameters of the first order polynomials that describe the linear relationships. They are summarized in figure 98.

<code>kernel_time(Len)</code>	C-Prolog implementation of Metaprolog	Metaprolog with path compression	Metaprolog emulation with compiled Quintus Prolog
$k\text{Len} + d$	$0.0216\text{Len} + 0.0308$	$0.0023\text{Len} + 0.0599$	$0.1179\text{Len} + 0.0427$

Figure 98: Kernel execution time model

In figure 99 and figure 100 we find the deref profiles for the Tennis example and for an application of `alldifferent/1` (figure 14) with version 2 of `neq/2` from section 5.4.1 applied to a list of 40 domain variables (780 inequality constraints).

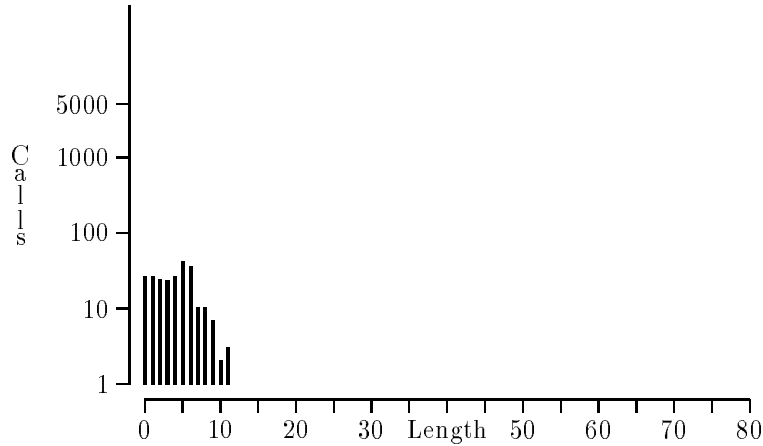
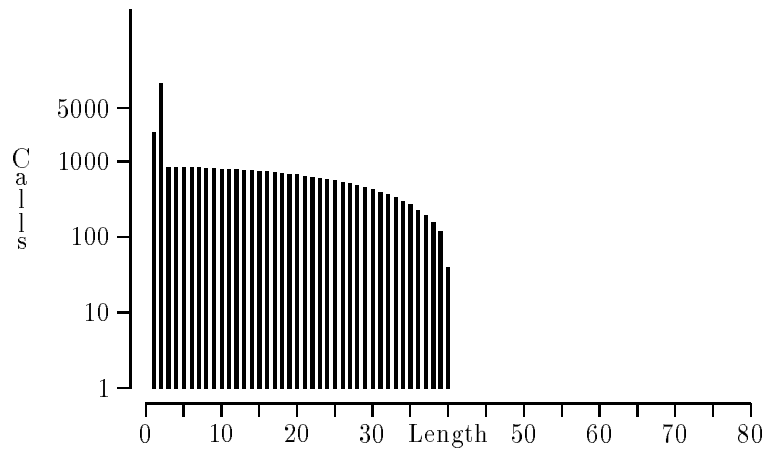


Figure 99: Deref profile for the Tennis example

From this data we compute the results of figure 101 and figure 102. If we compare the total execution times for the second example, we might get the impression that the Quintus Prolog compiler is not worth the money. A second look reveals that kernel emulation consumes as much as 85% of the execution time. If we compare the ‘pure’ execution times, have the usual average speedups confirmed again. Even in the first example, which

Figure 100: Deref profile for the `dom_alldifferent_40` example

is relatively ‘flat’ in terms of metaterm chains, almost half of the time is spent in the kernel emulation. The middle column, an implementation of **Metaprolog** that uses a path compression algorithm [Aho et al. 83] to keep the chains short, is provided as another reference point for this timing model. It also should give an idea of the achievable performance of **Metaprolog** with path compression in a compiling Prolog. The model confirms that it is beneficial to compress metaterm chains in terms of kernel time. On the other hand, path compression requires a modified trailing mechanism, which slightly penalizes normal Prolog execution — at least in the current implementation.

Runtime (msec.)	C-Prolog implementation of Metaprolog	Metaprolog with path compression	Metaprolog emulation with compiled Quintus Prolog
Total	1727.00	1733.00	300.00
Kernel	27.79 2%	16.25 1%	123.07 41%
Rest	1699.21 98%	1716.75 99%	176.93 59%

Figure 101: Execution time partition estimates for the Tennis example

Runtime (msec.)	C-Prolog implementation of Metaprolog		Metaprolog with path compression		Metaprolog emulation with compiled Quintus Prolog	
Total	75850.00		79966.00		55851.00	
Kernel	9414.43	12%	2961.05	4%	47251.42	85%
Rest	66435.57	88%	77004.95	96%	8599.58	15%

Figure 102: Execution time partition estimates for the `dom_alldifferent.40` example

6 Summary

Earlier works on the incorporation of forward checking and other consistency techniques over finite domains into Prolog resulted in C implementations of the extensions. Our Prolog implementation of forward checking is based on semantic unification, which has proven useful as a powerful implementation technique for mechanisms that one would not customarily associate with unification. The paper was primarily motivated by the software technological advantages of an executable Prolog specification for forward checking.

References

- Aho A.V., Hopcroft J.E., Ullman J.D. (1983): *Data Structures and Algorithms*, Addison-Wesley, Reading, MA.
- Boizumault P. (1986): A General Model to Implement DIF and FREEZE, in Shapiro E.(ed.), *Third International Conference on Logic Programming*, Springer, Berlin.
- Carlsson M. (1987): Freeze, Indexing, and Other Implementation Issues in the WAM, in Lassez J.(ed.), *Logic Programming - Proceedings of the 4th International Conference - Volume 1*, MIT Press, Cambridge, MA.
- Chang C., Lee R. (1973): *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.
- Coelho H., Cotta J.C., Pereira L.M. (1980): How to Solve it with Prolog, Laboratorio Nacional de Engenharia Civil, Centro de Informatica, Lisboa, Portugal, Techn. Report.
- Dechter R. (1986): Learning While Searching in Constraint-Satisfaction-Problems, in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Morgan Kaufmann, Los Altos, CA.
- Haralick R.M., Elliott G.L. (1980): Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, 14(3)263.
- Hentenryck P.van, Dincbas M. (1986): Domains in Logic Programming, in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Morgan Kaufmann, Los Altos, CA.
- Hentenryck P.van (1989): *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA.
- Holzbaur C. (1990): Integration of Extended Unification into a Prolog Interpreter, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, TR-90-9.
- Knuth D.E. (1975): *Sorting and Searching - The Art of Computer Programming*, Addison-Wesley, New York.
- Lauriere J.-L. (1978): A Language and a Program for Stating and Solving Combinatorial Problems, *Artificial Intelligence*, 10, 29-127.

- Lloyd J.W. (1988): *Foundations of Logic Programming*, Springer, Wien, New York.
- Mackworth A.K. (1977): Consistency in Networks of Relations, *Artificial Intelligence*, 8, 99-118.
- Mackworth A.K., Freuder E.C. (1985): The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence*, 25(1)65-74.
- Mohr R., Henderson T.C. (1986): Arc and Path Consistency Revisited, *Artificial Intelligence*, 28(2)225-233.
- Neumerkel U. (1990): Extensible Unification by Metastructures, *Proc. META90*.
- Pereira F. (1982): C-Prolog 1.5 Users Manual, SRI International, Menlo Park, CA.
- Sterling L., Shapiro E. (1986): *The Art of Prolog*, MIT Press, Cambridge, MA.
- Warren D.H.D. (1983): Logarithmic Access Arrays for Prolog, *Unpublished Program*.
- Winston P.H. (1984): *Artificial Intelligence (2nd ed.)*, Addison-Wesley, Reading, MA.

Glossary

bound variable:

An instantiated variable.

circular term:

An instance of a term containing a variable that is instantiated to the term it occurs in.

constant:

An atom or a number.

constraint:

A restriction in the form of a predicate that is enforced on one or more variables. In the context of domain variables a constraint usually limits the set of possible values of domain variables.

dereferencing:

Following a chain of indirections.

domain variable:

A variable with an associated domain. The values to which the variable can possibly be instantiated must be elements from the domain.

domain:

A set of terms.

FCIR:

Forward checking inference rule.

forward checking:

A technique to establish consistency across the assignments of (domain) variables which are related by constraints.

free variable:

Uninstantiated variable.

functor:

The name of a Prolog structure with an associated arity.

ground term:

A term that contains no variables.

instance:

A term with partially instantiated variables.

instantiated variable:

A variable bound to a value.

instantiation:

Binding of a free (possibly constrained) variable to a term.

interpreted terms:

Terms that are treated differently during unification. The exact treatment of such terms is described by the user-defined predicates `metatermunify/2` and `metametaunify`.

label:

An element from a domain. Labeling is the process of assigning labels, i.e., values from the domain, to domain variables.

metafunctor:

Alternative name for 'interpreted functor'.

metaterm:

Alternative name for 'interpreted term'.

mutual exclusion:

A control mechanism that prevents multiple events from taking place at the same instant.

shared variable:

A variable that appears more than once in a clause.

structure:

A term consisting of a functor and a corresponding number of arguments. The arguments are terms.

surrogate:

Replacement, substitute, placeholder.

term:

A term is either a constant or a variable or a structure.

trigger:

An association between an event and a Prolog goal to be executed at an instance of the event.

unbound variable:

A variable having no value yet.

unification:

Computation of the substitutions for variables in two terms that makes them equal. We

distinguish between *syntactic* and *semantic* unification. Semantic unification deals with the equality of interpreted terms.

uninstantiated variable:

Opposite of instantiated, i.e., unbound. An uninstantiated interpreted term represents a constraint over a variable.

List of Figures

1	Map coloring with Prolog	7
2	Map coloring with forward checking	8
3	L-shaped body	8
4	Prolog description of the L-shaped body	9
5	Predicate <code>l_junction/2</code>	9
6	Predicate <code>fork_junction/3</code>	9
7	Predicate <code>tee_junction/3</code>	9
8	Predicate <code>arrow_junction/3</code>	9
9	Predicate <code>inversion/2</code>	10
10	Metaprolog description of the L-shaped body	10
11	Execution times in seconds for the scene labeling example	11
12	Execution times for the scene labeling example	11
13	Generating and testing permutations	12
14	Predicate <code>alldifferent/1</code> , computing permutations with domains and inequalities	13
15	Computing permutations with <code>alldifferent/1</code>	13
16	Prolog description of a tennis match	14
17	Forward checking version of the Tennis puzzle	15
18	Execution times in seconds for the Tennis puzzle	15
19	Forward checking formulation of the Zebra puzzle	17
20	Predicates <code>right_of/2</code> and <code>next_to/2</code>	17
21	Zebra puzzle execution trace	21
22	Execution time statistics in seconds for the Zebra puzzle	22
23	Domains in Metaprolog	24
24	Examples of the use of <code>domain/2</code>	24
25	Predicate <code>domain/2</code>	25
26	State transition of variables	27
27	<code>forward/1</code> application	27
28	Predicate <code>forward/1</code>	28
29	<code>forward/1</code> application, continued	28
30	Unification of <code>\$fwd/3</code> metaterms	28
31	Unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms	29
32	Example: unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms	29
33	Example: unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms	29
34	Example: unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms	29
35	Example: unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms	30
36	Unification of <code>\$dom\$fwd/4</code> metaterms with ordinary terms	30
37	Unification of <code>\$dom\$fwd/4</code> metaterms with ordinary terms	30
38	Unification of <code>\$dom/2</code> with <code>\$fwd/3</code> metaterms, continued	31
39	Predicate <code>reconsider/1</code>	31
40	Predicate <code>applicable/7</code>	32

41	Predicate <code>applicable_one/7</code>	32
42	Predicate <code>extract_domain_internal/3</code>	32
43	Predicate <code>apply_fc/5</code>	33
44	Example: <code>apply_fc/5</code>	33
45	Predicate <code>forward/1</code> , new version	35
46	Predicate <code>forward_arg/3</code>	36
47	Execution times in seconds for some benchmarks for the first and the optimized version of <code>forward/1</code>	36
48	A shortcoming of the forward checking implementation of inequalities	37
49	Predicate <code>neq/2</code> and corresponding metaterm unifications	37
50	Predicate <code>no_such_var/3</code>	38
51	Example: inequalities	38
52	Unification of <code>\$dom/2</code> metaterms with <code>\$ne/2</code> metaterms	39
53	Combining domain variables and inequalities	39
54	Active unification of <code>\$dom/2</code> metaterms with <code>\$ne/2</code> metaterms	39
55	Predicate <code>remove_constant/2</code>	40
56	Combining domain variables and 'active' inequalities	40
57	Execution times in seconds for some <code>neq/2</code> benchmarks	41
58	Partially evaluated version of <code>neq/2</code> , part 1	42
59	Partially evaluated version of <code>neq/2</code> , part 2	43
60	Partially evaluated version of <code>neq/2</code> , part 3	44
61	Partially evaluated version of <code>neq/2</code> , part 4	44
62	The predicate <code>neq_metoo/2</code>	45
63	Combining constraints	46
64	Joining forward checking with inequalities	46
65	Number of <code>metatermunify/2</code> clauses (mtuc) and <code>metametaunify/2</code> clauses (mmuc) for n metaterms	47
66	Labeling example	48
67	Labeling example trace	48
68	Predicate <code>indomain/1</code> example	49
69	Predicate <code>indomain/1</code>	50
70	Predicate <code>indomain/1</code> , continued	50
71	Predicate <code>labeling/1</code>	50
72	Predicate <code>set_intersection/3</code>	51
73	Predicate <code>list_to_set/2</code>	52
74	Predicate <code>set_to_list/2</code>	52
75	Predicate <code>valid_set_elem/1</code>	52
76	Predicates <code>set_nonempty/1</code> and <code>set_singleton/2</code>	52
77	Predicate <code>set_test_membership/2</code> , first version	52
78	Predicate <code>set_test_membership/2</code> , second version	53
79	Predicate <code>set_remove_elem/3</code> , first version	53
80	Predicate <code>set_remove_elem/3</code> , second version	53
81	Predicate <code>set_enumerate/2</code>	53

82	Predicate <code>set_filter/4</code>	55
83	Example for the use of <code>set_filter/4</code>	55
84	Predicate <code>valid_set_elem/1</code>	55
85	Predicate <code>set_intersection/3</code>	55
86	Predicate <code>list_to_set/2</code>	55
87	Predicate <code>set_nonempty/1</code>	56
88	Predicate <code>set_singleton/2</code>	56
89	Predicate <code>set_test_membership/2</code>	56
90	Predicate <code>set_remove_elem/3</code>	56
91	Predicate <code>set_enumerate/2</code>	56
92	Enumeration example	57
93	Predicate <code>set_filter/4</code>	57
94	Example for the use of <code>set_filter/4</code>	58
95	Predicate <code>set_add_elem/3</code> for bitvectors	59
96	Domains in <i>Metaprolog</i> , unfolded set representation	60
97	Dereference times in milliseconds vs. metaterm chain length	62
98	Kernel execution time model	63
99	Deref profile for the Tennis example	63
100	Deref profile for the <code>dom_alldifferent_40</code> example	64
101	Execution time partition estimates for the Tennis example	64
102	Execution time partition estimates for the <code>dom_alldifferent_40</code> example	65