FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Correctness Considerations in CLP(FD) Systems

Dissertation

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Markus Triska
Registration number 0225855

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Dr. Nysret Musliu

This dissertation has been reviewed by:

Dr. Nysret Musliu	Jan Wielemaker, Ph.D.

Vienna, December 3rd 2013

To Ludwig Moser, whose commitment to elegance guided me.

How infinitely good that Providence is, which has provided, in its government of mankind, such narrow bounds to his sight and knowledge of things; and though he walks in the midst of so many thousand dangers, the sight of which, if discovered to him, would distract his mind and sink his spirits, he is kept serene and calm, by having the events of things hid from his eyes, and knowing nothing of the dangers which surround him.

Daniel Defoe, *The Life and Adventures of Robinson Crusoe*

Abstract

Constraint logic programming (CLP) is a declarative formalism for describing conditions a solution must satisfy. Constraint logic programming over finite domains, denoted as CLP(FD), considers problems involving integers, including combinatorial problems such as planning, scheduling and allocation tasks. Given a problem description, a constraint solver tries to find valid solutions via constraint propagation and search.

Constraint solvers are complex programs, and many existing and widely used CLP(FD) systems suffer from limitations and mistakes that can cause them to miss valid solutions or give wrong answers.

In this thesis, we show examples of common limitations and mistakes of several widely used CLP(FD) systems. We then present a new CLP(FD) system that overcomes some of these issues. Our system has several unique features such as monotonicity, reasoning over arbitrarily large integers and always terminating propagation. This yields new application opportunities for constraint solvers which we also present. We develop new domain-specific languages that let us concisely and declaratively express parts of our system whose encoding would otherwise be difficult and error prone, such as parsing, propagator selection and constraint reification. We present two methods for testing our solver: systematic test cases, and automated analysis of individual propagators. Our contributions are applicable to other constraint systems as well and may improve their correctness.

Zusammenfassung

Constraint Logic Programming (CLP) ist ein deklarativer Formulismus zur Beschreibung von Bedingungen, die von gesuchten Lösungen erfüllt werden müssen. Constraint Logic Programming über endlichen Wertebereichen, abgekürzt als CLP(FD), befasst sich mit Problemstellungen mit ganzen Zahlen und umfasst kombinatorische Probleme wie Planung, Scheduling und Allokationsaufgaben. Ein Constraint Solver sucht unter Anwendung von Constraint Propagierung nach Lösungen, welche die angegebenen Bedingungen erfüllen.

Constraint Solver sind komplexe Programme, und viele existierende und in vielen Bereichen eingesetzte CLP(FD) Systeme weisen Einschränkungen und Fehler auf, durch die sie gültige Lösungen nicht finden oder falsche Antworten geben.

In dieser Dissertation zeigen wir Beispiele für typische Einschränkungen und Fehler von weit verbreiteten CLP(FD) Systemen. Anschließend präsentieren wir ein neues CLP(FD) System, das einige dieser Einschränkungen aufhebt. Mehrere Eigenschaften wie Monotonie, Schließen über beliebig großen ganzen Zahlen und immer terminierende Propagierung werden von unserem System erstmals garantiert. Dadurch entstehen neue Anwendungsmöglichkeiten für Constraint Solver, die wir ebenfalls beschreiben. Wir entwickeln neue domänenspezifische Sprachen, mit denen wir kompakt und deklarativ Teile unseres Systems ausdrücken können, die sonst schwierig und fehleranfällig wären, wie Parsen, Propagator-Auswahl und Reifikation von Constraints. Wir präsentieren zwei Methoden, mit denen wir unseren Constraint Solver testen: systematische Testfälle, und automatisierte Analyse einzelner Propagatoren. Unsere Beiträge sind auch auf andere Constraint Systeme anwendbar und können ihre Korrektheit verbessern.

Acknowledgements

First and foremost, I thank my advisor Nysret Musliu for supervising this thesis. His interest in combinatorial problems motivated me to finally implement the CLP(FD) system that is the main subject of this thesis.

Further, I thank Ulrich Neumerkel for introducing me to constraint logic programming, and drawing my attention to PostScript. My gratitude also goes to Mats Carlsson, whose exceptionally elegant CLP(FD) formulation of the so-called "social golfer problem" was my main motivation for implementing several important global constraints.

Jan Wielemaker provides a robust, feature-rich and free Prolog system, which I used extensively. Tom Schrijvers generously contributed several libraries to SWI-Prolog, from which I learned a lot. Thank you!

I thank all my colleagues in the Austrian Federal Ministry of Finance for making each working day so enjoyable. I thank all users of my constraint solver for their feedback and encouragement. I thank my friends for their support. Last, not least, I thank my parents and family for making it all possible in the first place.

Contents

1 Introduction

Constraint logic programming (CLP) is a declarative formalism for describing conditions a solution must satisfy. Constraint logic programming over finite domains, denoted as CLP(FD), considers problems involving integers, including combinatorial problems such as planning, scheduling and allocation tasks. Given a problem description, a constraint solver tries to find valid solutions via constraint propagation and search.

Constraint solvers are complex programs, and many existing and widely used CLP(FD) systems suffer from limitations and mistakes that can cause them to miss valid solutions or give wrong answers. One frequent source of erroneous answers in common constraint systems are their – either implicit or explicit – restrictions to quite small values. Consider for example the following interaction with GNU Prolog 1.4.0 (32-bit):

```
| ?- X #> 200, X #\= 2.

X = _#2(201..268435455)

yes
```

As expected, the conjunction of these two constraints succeeds, since there clearly are integers that are greater than 200 and not equal to 2. However, if we exchange the two goals by commutativity of conjunction, we get the *incorrect* answer "no" instead of "yes":

```
| ?- X #\= 2, X #> 200.
Warning: Vector too small - maybe lost solutions (...)

no
```

In this case, at least a warning is emitted that solutions *may* have been lost (as they indeed have been), but according to the manual, this may not be detected in all cases, and indeed the system fails without qualification for example in:

```
| ?- X #= Y*Z.

no
```

As we show in this thesis, similar problems are easily found in other constraint systems as well. Since CLP(FD) is applied in many industrial settings like systems verification, it is natural to ask: How can we implement constraint solvers that are more reliable?

In this thesis, we approach this question by first asking: What do we guarantee in our systems? Clearly, it would be ideal if we could for example answer with "We guarantee correctness.", or "If there is a solution, our system will always find it.", or "Our system will never emit a wrong answer.". However, these guarantees are very hard to ensure, and to the best of our knowledge, no CLP(FD) system can justifiably give them yet. We therefore aim for useful properties that are easier to ensure, and which help us to establish further guarantees towards the ultimate goal of ensuring correctness of our systems.

We will describe a new CLP(FD) system that gives strong guarantees with respect to:

- *domains*
  Our system reasons over arbitrarily large integers.
- *termination*
  In our system, constraint propagation *always* terminates.
- *monotonicity*
  Our system is *monotonic* when a specific flag is set to `true`.

We will demonstrate the importance of each of these guarantees throughout this thesis. To the best of our knowledge, ours is the first widely available CLP(FD) system that gives any of these guarantees.

As we will show in this thesis, mistakes in CLP(FD) systems can be very subtle and occur only rarely. We will see an example of a mistake that occurs only after weeks of computation time. Very isolated mistakes are hard to find with black-box tests. We will therefore develop a way to make further guarantees about individual propagators, using *abstract interpretation*. This technique is easily applicable for a homoiconic language like Prolog, which we use for implementing our system. It is harder to make comparable guarantees for systems that are written in other languages.

Based in the observation that isolated mistakes are hard to find, the second question we ask is: How can we make mistakes *less* isolated? In our view, which may appear counter-intuitive at first and which we will justify with specific examples throughout this thesis, an ideal CLP(FD) system satisfies the following property:

If there is a single mistake *anywhere* in the implementation of the system, then the system does not work *at all*.

In our system, we are aiming to satisfy this property as far is we can with the development and use of new *domain-specific languages* that are also applied to new domains such as compactified arithmetic and propagator selection. These languages let us generate large portions of our system from concise and declarative specifications. If there is any mistake in the expansion phase, it is likely to affect several parts of our system at once. Mistakes in these portions are therefore less isolated and thus easier to find.

1.1 Goals of this thesis

The goals of this thesis are:

1. to show examples of common limitations and mistakes of several widely used CLP(FD) systems to increase awareness of these issues among authors and users of these systems
2. to present a new CLP(FD) system that overcomes some of these issues and gives guarantees that other systems do not give
3. to outline systematic test cases based on these guarantees
4. to apply the system on benchmarks and real-life problems.

1.2 Main results of this thesis

We briefly summarize the main results of this thesis:

1. We identify limitations and mistakes in several widely used CLP(FD) systems and show that these limitations prevent the use of these systems on interesting classes of programs that have hitherto received little attention from the constraint community.
2. We develop and present a new CLP(FD) system with several unique features and guarantees: monotonicity, reasoning over arbitrarily large integers and always terminating propagation.
3. We show new domain-specific languages that are used for parsing arithmetic expressions, propagator selection, compactified arithmetic and constraint reification to improve correctness and efficiency.
4. We present systematic test cases which are also applicable to other systems, and describe how we ensure several important properties in our system via automated analysis of individual propagators.
5. We apply parts of our CLP(FD) system in real-life *rotating workforce scheduling* instances with competitive results.

Our CLP(FD) system is entirely written in Prolog and available in the free Prolog system SWI-Prolog as `library(clpfd)`. In this thesis, we include several pages of the concrete Prolog source code of this library. The reason is that we consider it essential to refer to *concretely executable* code when discussing correctness considerations. Readers are encouraged to read and verify for themselves all portions of code, which we consider an integral part of our contribution. Little would be gained for our purpose by studying code that is not actually executed, since who could then tell whether what is actually executed is correct?

1.3 Publications

Some results of this thesis appear in the following publications:

Books

- **SWI Prolog Reference Manual 6.2.2**
  Jan Wielemaker, Thom Fruehwirth, Leslie De Koninck, Markus Triska, Marcus Uneson
  Books on Demand, ISBN: 3848226170

Journal papers

- **SWI-Prolog**
  Jan Wielemaker, Tom Schrijvers, Markus Triska and Torbjörn Lager
  TPLP 12 (2012), pp. 67–96

Conference papers

- **The Finite Domain Constraint Solver of SWI-Prolog**
  Markus Triska
  FLOPS 2012, LNCS 7294, pp. 307–316
- **Domain-specific Languages in a Finite Domain Constraint Programming System**
  Markus Triska
  Proceedings of INAP 2011, Technical report
- **A Constraint Programming Application for Rotating Workforce Scheduling**
  Markus Triska and Nysret Musliu
  EA/AIE 2011, Studies in Computational Intelligence 363 (2011), pp. 83–88
- **Generalising Constraint Solving over Finite Domains**
  Markus Triska
  ICLP 2008, LNCS 5366, pp. 820–821

The introduction to CLP(FD) that appears in this thesis is an extended and improved version of a chapter published in the author's Masters thesis, *Solution Methods for the Social Golfer Problem.*

2 Constraint Programming and CLP(FD)

2.1 Introduction

Constraint programming (CP) is a declarative formalism that lets users specify conditions a solution must satisfy. Based on that description, a constraint *solver* can then search for solutions.

The first ideas for CP date back to the sixties and seventies ([Sut63]), with the *scene labelling* problem ([Wal75]) being one of the first constraint satisfaction problems (CSPs) that were formalised. A CSP consists of:

- a set X of variables, $X = \{x_1, \ldots, x_n\}$
- for each variable x_i, a set $D(x_i)$ of values that x_i can assume, which is called the *domain* of x_i
- a set of *constraints*, which are relations among variables in X, and which can further restrict their domains.

One key observation, made by Jaffar, Lassez ([JL87a]), Gallaire ([Gal85]) and others, was the insight that pure *logic programming* (LP) can be regarded as an instance of constraint solving, namely as solving constraints over variables whose domains are Herbrand terms. In addition, LP and CP share an important intention, which is to make users less concerned about *how* a problem should be solved, and instead let them focus on a clear description of *what* should be solved. From that description, a logic engine or constraint solver can, in principle, compute a solution without additional instructions. Logic programming languages like Prolog are therefore among the most important host platforms for constraint solvers, and most Prolog implementations nowadays ship with several libraries for constraint programming. When CP is used with a logic programming language as its host, it is referred to as constraint *logic* programming (CLP). However, constraint programming is not restricted to CLP: It is possible to embed constraint solvers in other host languages, even if they might not blend in as seamlessly as they do with Prolog.

2.2 CLP(FD)

In connection with combinatorial optimisation or completion problems, one of the most frequently used instances of constraint programming is constraint logic programming over finite domains, denoted as CLP(FD). This means that all domains are sets of *integers*, and the available constraints include at least the common arithmetic relations between integer expressions.

One advantage when reasoning over integers is that many known laws of arithmetic can be used to further reduce the domains of variables that participate in the provided relations. Another advantage is that there is a

predefined total order over the integers, which can often help to eliminate uninteresting symmetries between solutions and reduce the search space.

CLP(FD) can also help to solve problems over rational numbers. The following example is known as the "7-11 problem" ([PG83]):

Example 2.1. The total price of 4 items is €7.11. The product of their prices is €7.11 as well. What are the prices of the 4 items?
Answer. The prices are €3.16, €1.50, €1.25 and €1.20. A CLP(FD) solution for this problem is shown in Fig. 2.1. Line 1 states the domain of all variables, lines 2 and 3 post the two known constraints. The product of all variables equals a quite large constant, which is beyond the capabilities of several current constraint systems on still common 32-bit platforms. Line 4 imposes additional constraints to break symmetries between values. Finally, line 5 searches for valid ground instantiations of all variables, using a strategy which we explain in Section 2.6. Notice that other constraints could be imposed as well. For example, due to the fundamental theorem of arithmetic, the factorisation of one of the variables must contain one of the prime factors of 711×100^3. However, imposing such a constraint would in general also require weakening the ordering relation, and it is not a priori clear which of the formulations is better.

```
1  ?- Vs = [A,B,C,D], Vs ins 0..711,
2     A * B * C * D #= 711*100^3,
3     A + B + C + D #= 711,
4     A #>= B, B #>= C, C #>= D,
5     labeling([ff], Vs).
```

Figure 2.1: Solving the 7-11 problem with a single query

2.3 Example: Sudoku

In the recent past, a combinatorial number puzzle called *Sudoku* has attracted significant attention. Sudoku puzzles are commonly found in newspapers and periodicals and are naturally modelled as CSPs. A Sudoku Latin square is a particular kind of Latin square ([CD96b]):

Definition 2.1. A *Latin square* of order n is an $n \times n$ array in which each cell contains a single symbol from a set S with n elements, such that each symbol occurs exactly once in each row and exactly once in each column.

Definition 2.2. Let a, b and n be positive integers with $a \times b = n$. Partition an $n \times n$ array into $a \times b$ rectangles. An (a, b)-*Sudoku Latin square* is a Latin square on the symbol set $\{1, \ldots, n\}$ where each (a, b)-rectangle contains all symbols. A *Sudoku Latin square* is a $(3, 3)$-Sudoku Latin square.

```
1  sudoku(Rows) :-
2          length(Rows, 9), maplist(length_(9), Rows),
3          append(Rows, Vs), Vs ins 1..9,
4          maplist(all_different, Rows),
5          transpose(Rows, Columns), maplist(all_different, Columns),
6          Rows = [A,B,C,D,E,F,G,H,I],
7          blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).
8
9  length_(N, Ls) :- length(Ls, N).
10
11 blocks([], [], []).
12 blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
13         all_different([A,B,C,D,E,F,G,H,I]),
14         blocks(Bs1, Bs2, Bs3).
```

Figure 2.2: A CLP(FD) description of Sudoku Latin squares

Definition 2.3. An (a, b)-*Sudoku critical set* is a partial Latin square P that is completable in exactly one way to an (a, b)-Sudoku Latin square, and removal of any of the filled cells from P destroys the uniqueness of completion.

Fig. 2.2 shows a CLP(FD) formulation for Sudoku Latin squares. Here, a Sudoku Latin square is modelled as a list of rows, with each row being a list of variables with domain $\{1, \ldots, 9\}$. Line 2 ensures the correct list structure, which makes it possible to use the predicate in all directions: One can use the specification to test and complete partially filled squares as well as to enumerate all possible squares. The only constraint used in this formulation is the built-in constraint `all_different/1`, which imposes pairwise disequalities between all variables occurring in a list. The constraint is imposed for each row (line 4), column (line 5), and 3×3-subsquare (lines 6, 7 and 11–14).

A valid Sudoku puzzle as commonly found in contemporary newspapers and periodicals is a partial Latin square that is completable in exactly one way to a Sudoku Latin square. Fig. 2.3 (a) shows an example of a valid Sudoku puzzle, which is simultaneously a $(3, 3)$-Sudoku critical set. In fact, the figure shows one of the "hardest" Sudoku puzzles with respect to the number of hints: There is no $(3, 3)$-Sudoku critical set with fewer than 17 given numbers ([MTC12]). Fig. 2.3 (b) shows the Sudoku Latin square that is uniquely determined by this Sudoku critical set.

2.4 Consistency

A CSP is called *consistent* if it has a solution. An element v of a domain $D(x)$ is said to be *inconsistent* with respect to a given CSP if there is *no* solution in which x assumes the value v. Consistency techniques were introduced in [Wal75] and are derived from graph notions (see [Bar99]).

A domain $D(x)$ is called *domain consistent* with respect to a constraint c if $D(x)$ contains all valid values of x with respect to c, and no proper subset

1								
		2	**7**	**4**				
			5					**4**
	3							
7	**5**							
					9	**6**		
	4				**6**			
							7	**1**
					1		**3**	

(a)

1	**8**	**4**	**9**	**6**	**3**	**7**	**2**	**5**
5	**6**	**2**	**7**	**4**	**8**	**3**	**1**	**9**
3	**9**	**7**	**5**	**1**	**2**	**8**	**6**	**4**
2	**3**	**9**	**6**	**5**	**7**	**1**	**4**	**8**
7	**5**	**6**	**1**	**8**	**4**	**2**	**9**	**3**
4	**1**	**8**	**2**	**3**	**9**	**6**	**5**	**7**
9	**4**	**1**	**3**	**7**	**6**	**5**	**8**	**2**
6	**2**	**3**	**8**	**9**	**5**	**4**	**7**	**1**
8	**7**	**5**	**4**	**2**	**1**	**9**	**3**	**6**

(b)

Figure 2.3: (a) A $(3,3)$-Sudoku critical set, and (b) the induced Sudoku Latin square

of $D(x)$ contains all valid values.

Different notions of *bounds consistency* appear in the literature (see for example [CHLS06] for an overview). In this thesis, we use the following definition throughout: A set of integers $D(x)$ is *bounds consistent* with respect to a constraint c if it contains all valid values of x with respect to c and it is a subset of the smallest *interval* of integers that contains all valid values. Note that the interval need not be finite.

2.5 Constraint propagation and search

A *constraint propagator* is a method for filtering inconsistent elements from a domain. The process of deterministically ensuring some form of consistency is called constraint *propagation*.

Since propagation alone is in general insufficient to reduce all domains to singleton sets and thus produce concrete solutions, some form of *search* is necessary in addition to constraint propagation. Systematically trying out values for variables is called *labeling*, and we discuss it in the next section. As soon as a variable is labeled, constraint propagation is used to further prune the search space. Conversely, propagation can in itself yield a singleton set for a variable's domain, thus causing the variable to be instantiated to a ground value. Search and propagation are therefore interleaved when solving a CSP. Clearly, a trade-off must be reached between strong propagation, implying great reduction of the search space for some problems, and computational tractability.

As an example for different consistency notions, consider again Sudoku puzzles. In this case, the search space is often quite large when traversed naively. However, a constraint solver is typically able to delete many values

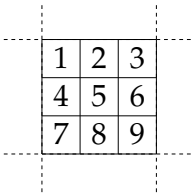
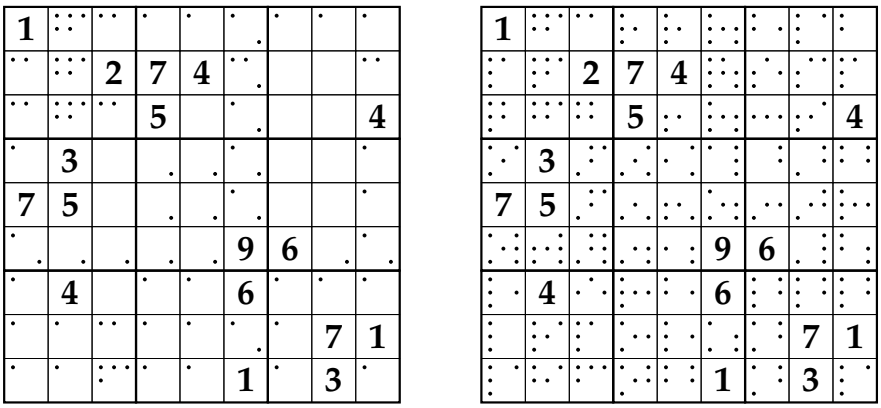


Figure 2.4: Subdivision of a single cell



(a) (b)

Figure 2.5: Domain elements that can be removed after posting the Sudoku puzzle with (a) a bounds consistent constraint solver and (b) a stronger solver

from the domains of those variables that correspond to free cells before the search even begins.

To give a visual impression of the values that can be removed from domains, we proceed as follows: First, we subdivide all free cells into 9 small regions as shown in Fig. 2.4. Each region corresponds to the domain element that it contains in this figure. Then, a dot is drawn in those regions that correspond to domain elements which can be excluded due to the given constraints. Fig. 2.5 shows which values can be excluded by two different constraint solvers without performing any search. Fig. 2.5 (a) was created with a bounds consistent solver, and Fig. 2.5 (b) was created with a solver with stronger filtering.

Since Sudoku puzzles only have a single solution, a solver with perfect filtering would reduce all domains to singleton sets in this case, making further search unnecessary while expending more computation time on the propagation itself. However, note that even if we use a domain consistent variant of `all_different/1`, we do not necessarily obtain such a strong filtering, because the consistency notions only apply to individual constraints in isolation and do not take combinations of constraints into account.

2.6 Selection strategies for variables and values

When searching for solutions of a CSP by trying ground values for variables, there are at least two degrees of freedom: First, the instantiation order of variables. Second, the order in which values are tried for each variable. Choosing good orders can significantly reduce computation time.

We first discuss the impact of variable instantiation orders. Fig. 2.6 depicts two possible search tree shapes arising from complete enumerations of two unconstrained variables, X and Y, with domains of size 2 and 5, respectively. The order or type of actual values that are tried for each variable is currently of no concern, as we focus on the order in which the variables themselves are instantiated. Inner nodes of the search tree, which are the variables, are shown as circles, and leaves are shown as boxes. When a leaf is reached in the search process, all variables are instantiated. Clearly, the number of leaves must be the same for all possible shapes of the search tree, while the number of inner nodes can obviously differ significantly.

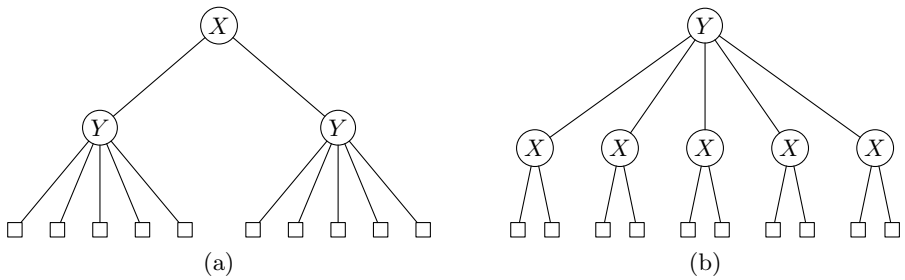


Figure 2.6: Search tree shapes arising from different instantiation orders

In typical CSPs, many values can turn out to be infeasible. In fact, a significant number of subtrees of the search tree will often turn out to be of no interest at all. We expect the greatest reduction of inner nodes that must still be visited by first trying to instantiate the variable with the *fewest* domain elements left. The strategy of instantiating the variables in order of increasing size of domains is called "first-fail", and often performs very well in practice. The intention here is twofold: First, variables with small domains are likely to run out of domain elements, causing their instantiation to fail. Clearly, it is advantageous to detect inevitable failure as early as possible. Second, instantiating variables can only further constrain the domains of remaining variables. Therefore, we want to instantiate variables with small domains while that is still possible, since the situation can only become worse for them. For a probabilistic analysis of the impact of this strategy, see [HE80].

Constraint solvers typically provide several pre-defined variable selection strategies that users can choose from, and which can influence computation time considerably. For example, SICStus Prolog provides the following

strategies to instantiate a list of variables (ties are broken by selecting the leftmost variable in the list), which are also available in most other constraint solvers:

- **leftmost**
  Instantiate the variables from left to right in the order they occur in the given list.
- **ff** ("first-fail")
  Instantiate a variable with smallest domain next.
- **ffc**
  Of the variables having smallest domains, one involved in most constraints is instantiated next.
- **min**
  Instantiate a variable whose lower bound is the lowest next.
- **max**
  Instantiate a variable whose upper bound is the highest next.

For most of these options, it is important to accurately assess a variable's current domain, and thus solvers with different propagation strengths can lead to very different instantiation orders of variables. Somewhat counter-intuitively, stronger propagation can even have an adverse effect in this case. This was first pointed out in [SF94] and can be explained by the fact that stronger propagation can also lead an instantiation strategy *away* from a "good" ordering, since propagation affects the variables' domains and thus the selected variable for many of these options.

After having selected a variable x for instantiation, a constraint solver must choose a value from $D(x)$ that should be assigned to x. A good strategy is often to instantiate x to a value of its domain which constrains the remaining variables *the least*. However, determining which of the values have this property can be costly, and many constraint solvers therefore do not provide this option. Two examples for value selection strategies are:

- **up**
  The values of each domain are tried in ascending order.
- **down**
  The values are tried in descending order.

In addition to these pre-defined selection strategies and value ordering options, users are free to implement their own allocation strategies. We regard this as one of the great advantages of constraint-based approaches over other methods: Once all constraints are stated, variables can be instantiated in any order and to any values, and infeasible choices are automatically rejected.

2.7 Visualising the constraint solving process

In many cases, it is very interesting to visualise the constraint solving process graphically. At the very least, one can get an impression of how the search progresses. Based on that observation, one can then try different allocation strategies, which sometimes work much better than others.

Transparent constraint animations have not received much attention in the literature so far: In [NRS97], Neumerkel et al. explain the importance of visualisations in the context of GUPU, a teaching environment for Prolog. However, they do not mention the potential usefulness of visualisations for deducing alternative strategies. Fages et al. present a graphical user interface for CLP in [FSC04]. Their approach typically requires several changes in the actual program code to obtain visualisations. In addition, it is hard to customise towards problem-specific visualisations. Finally, Ducassé and Langevine present abstract visualisations generated from an automated analysis of execution traces in [DL02]. This requires a rather involved event filtering and transformation scheme.

We now adapt the approach proposed in [NRS97] to the free Prolog system SWI-Prolog and explain it in more detail than the authors themselves. Our intention is to make their very transparent and portable approach more widely accessible and understandable also for casual users of constraint programming systems.

To focus on the main points involved when producing animations, we use the so-called N-queens problem as a self-contained and simple example, which is also presented in [NRS97]. The task is to place N queens on an $N \times N$ chess board in such a way that no two queens attack each other, which we call a *consistent* placement. Fig. 2.7 shows a consistent placement of 8 queens.

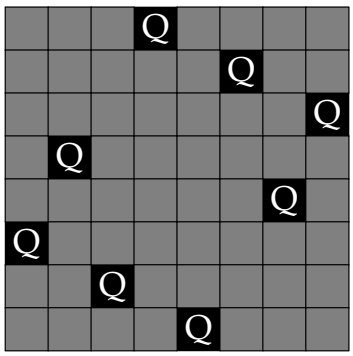


Figure 2.7: A consistent placement of 8 queens

Fig. 2.8 shows a CLP(FD) formulation for the N-queens problem: We use N variables $Q_1, \ldots, Q_N$, where Q_i denotes the row number of the queen in column i. Lines 13 and 14 impose the necessary constraints: The queens'

rows must be pairwise distinct to forbid horizontal attacks, and diagonal attacks are prohibited as well.

Fig. 2.9 shows how the CLP(FD) formulation can be transparently extended to emit PostScript instructions that visualise the constraint solving process: For each value n_i of the domain of queen Q_j, a so-called *reified* constraint of the form $(Q_j = n_i) \leftrightarrow B_{ij}$ is posted. Constraint reification is a common feature of constraint solvers and lets us reflect the truth value of many constraints into Boolean variables. When n_i vanishes from the domain of Q_j, B_{ij} becomes 0. In that case, PostScript instructions for graying out the corresponding square are emitted. When B_{ij} becomes 1, the equality holds, and instructions for placing the queen are emitted. On backtracking, the square is cleared in both cases. To make the example completely self-contained, we show the necessary PostScript definitions in Fig. 2.10. Fig. 2.11 (a) shows an example of its usage and Fig. 2.11 (b) shows the resulting picture. To obtain a real-time animation of the constraint solving process, the PostScript instructions that are generated can be directly fed into a PostScript interpreter.

Fig. 2.12 shows an animation for 50 queens. The labeling strategy is *first-fail*, modified as proposed by Ertl in [Ert90]: In case of ties, we try to distribute the queens across the two horizontal halves of the board. In [Ert90], this strategy is proposed without further explanation, and it is not mentioned how this heuristic could be improved for board sizes where it does not perform well. However, when an animation of the process is available, alternative strategies are often apparent. For example, in Fig. 2.12, one can see that the strategy does not distribute the queens as evenly as intended towards the end.

```
1  n_queens(N, Qs) :-
2          length(Qs, N),
3          Qs ins 1..N,
4          safe_queens(Qs).
5
6  safe_queens([]).
7  safe_queens([Q|Qs]) :-
8          safe_queens(Qs, Q, 1),
9          safe_queens(Qs).
10
11 safe_queens([], _, _).
12 safe_queens([Q|Qs], Q0, D0) :-
13         Q0 #\= Q,
14         abs(Q0 - Q) #\= D0,
15         D1 #= D0 + 1,
16         safe_queens(Qs, Q0, D1).
```

Figure 2.8: A CLP(FD) formulation for the N-queens problem

```
1  animate(Qs) :- animate(Qs, Qs, 1).
2
3  animate([], _, _).
4  animate([_|Rest], Qs, N) :-
5          animate_(Qs, 1, N),
6          N1 #= N + 1,
7          animate(Rest, Qs, N1).
8
9  animate_([], _, _).
10 animate_([Q|Qs], C, N) :-
11         freeze(B, queen_value_truth(C,N,B)),
12         Q #= N #<==> B,
13         C1 #= C + 1,
14         animate_(Qs, C1, N).
15
16 queen_value_truth(Q, N, 1) :- format("~w ~w q\n", [Q,N]).
17 queen_value_truth(Q, N, 0) :- format("~w ~w i\n", [Q,N]).
18 queen_value_truth(Q, N, _) :- format("~w ~w c\n", [Q,N]), false.
```

Figure 2.9: Observing the constraint solving process for N-queens

```
1  /init {  /N exch def 322 N div dup scale -1 -1 translate
2      /Palatino-Roman 0.8 selectfont 0 setlinewidth
3      1 1 N { 1 1 N { 1 index c } for pop } for } bind def
4
5  /showtext { 0.5 0.28 translate
6      dup stringwidth pop -2 div 0 moveto 1 setgray show } bind def
7  /r { translate 0 0 1 1 4 copy rectfill 0 setgray rectstroke } bind def
8  /i { gsave .5 setgray r grestore } bind def
9  /q { gsave r (Q) showtext grestore } bind def
10 /c { gsave 1 setgray r grestore } bind def
```

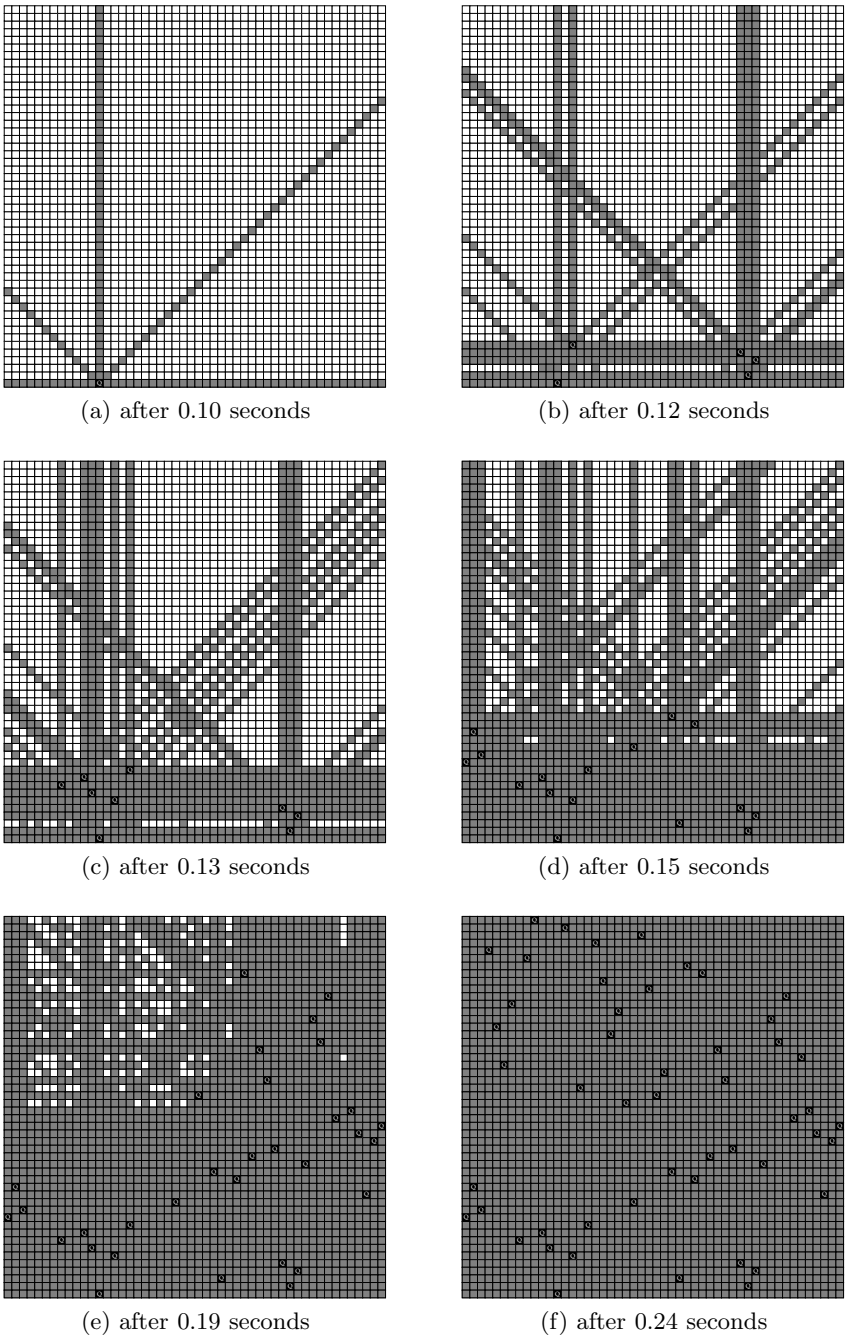
Figure 2.10: PostScript definitions for visualising N-queens

```
2 init
2 1 q
1 1 q
1 1 c
1 2 i
```

(a)

(b)

Figure 2.11: (a) PostScript instructions and (b) the resulting picture



(a) after 0.10 seconds

(b) after 0.12 seconds

(c) after 0.13 seconds

(d) after 0.15 seconds

(e) after 0.19 seconds

(f) after 0.24 seconds

Figure 2.12: 50 queens, strategy *first-fail*, breaking ties as proposed in [Ert90], using SWI-Prolog 6.5.2 on a 2.66 GHz iMac

3 Current CLP(FD) systems and their properties

In this chapter, we give a brief overview of a several prominent existing constraint solvers over finite domains. They are all widely used in commercial and academic environments and are well known in the constraint programming community.

Throughout this chapter, we present characteristic properties of these systems. We give special attention to known shortcomings and correctness issues, to raise awareness of these issues among the users and authors of these systems. Shortcomings in these CLP(FD) systems are almost invariably due to conscious decisions, for example, because performance is deemed more important than declarative correctness, or because certain use cases are deemed more important than others. However, we also present examples of unintended and previously unknown mistakes, which were typically corrected after we reported them to the authors of these systems.

Despite all shortcomings and mistakes, these systems deserve to be admired for their great practical value and gracious designs. They have been useful to practitioners, researches and students for many years and have set a standard against which every future CLP(FD) system will be measured.

3.1 Terminology

We first define a few concepts that are applicable to all CLP(FD) systems that we discuss in this chapter. A CLP(FD) **system** is given a logic **program**. Users run the program by posting a **query**. The textual response of the system is called an **answer**. An answer is for example:

- a *solution*, which is a set of bindings of the query's variables to concrete values that turn the query into a logical consequence of the given program.
- an *exception*, indicating that an error arose in the course of the computation.
- `false`, indicating that there is no solution.
- a *conditional solution*, indicating that there is a solution *if* some conditions, which are called *residual constraints* or *residual goals*, hold.

There are also other kinds of output, for example, because the program itself emits text during the computation.

3.2 Kinds of mistakes

When considering pure logic programs that terminate without raising an exception, there are two ways in which a system can yield wrong results:

1. it fails to find a solution that actually exists. We call such a system *incomplete*.

2. it emits an answer that is not a consequence of the given logic program. We call such a system *unsound*.

From a user's perspective, an *incomplete* system is typically the worse case. This is because it is typically easy to verify any answer the system emits, and to discard invalid answers. It is much harder to detect the case that valid solutions are *not* emitted, because they are lost internally in the system and never appear outside of it.

Note that we are not talking about mistakes in the user-supplied logic *program*, but about mistakes in the CLP(FD) *system* itself. That is, we assume that a given problem is *correctly* formulated via CLP(FD) constraints, but the system does not behave according to its documentation.

Also note that raising an *exception* is not considered a mistake in this sense, since it is then clear that there is some kind of problem, and no *unconditional* answer is reported to the user.

3.3 GNU Prolog

GNU Prolog ([DC01], [DAC12]) is one of the most established and well known Prolog systems. It is available as *free software* and used in teaching and research at several universities.

GNU Prolog's finite domain constraint solver is an integral part of the system, and all its predicates are available even without loading any libraries. The solver does not support negative integers, and could therefore be called a CLP($\mathbb{N}$) solver. According to the manual, there are two internal representations for the domain of a finite domain variable:

- *interval representation*, which maintains the end-points of an integer interval. It can represent values between 0 and the greatest value that any finite domain variable can take, called `fd_max_integer`.

- *sparse representation*, where an additional bit-vector is used to represent the domain. It can represent values between 0 and an internal value called `vector_max`, which is 127 by default and adjustable via the built-in predicate `fd_set_vector_max/1`.

Quoting from GNU Prolog's manual: "The initial representation for an FD variable X is always an interval representation and is switched to a sparse representation when a 'hole' appears in the domain (e.g. due to an inequality constraint). Once a variable uses a sparse representation it will not switch back to an interval representation even if there are no longer holes in its domain. When this switching occurs some values in the domain of X can be lost since `vector_max` is less than `fd_max_integer`."

Further, the manual states: "As seen before, the solver tries to display a message when a failure occurs due to a too short `vector_max`. Unfortunately, in some cases it cannot detect the lost of values and no message is emitted. So the user should always take care to this parameter to be sure that it is large to encode any vector."

3.3.1 Toplevel interaction

We present a few examples of interacting with GNU Prolog 1.4.1 (32-bit). Let us first ask if there are finite domain elements less than 0:

```
| ?- X #< 0.

no
```

The system answers "no", indicating that *no* term `X` satisfies this constraint. However, it also says that -1 *does* satisfy the constraint:

```
| ?- X = -1, X #< 0.

X = -1

yes
```

GNU Prolog's constraint solver is thus not *monotonic*, since adding constraints (in this case, `X = -1`) can yield additional solutions. This is not a problem in most practical applications, but you have to keep it in mind when testing or reasoning about the system. Moreover, even the constraint solver itself can yield negative integers, since integer overflow is not detected for performance reasons:

```
| ?- X #= 268435455 + 1.

X = -268435456

yes
```

It is understandable that GNU Prolog omits overflow checks for better performance, especially because the system is mainly used to solve combinatorial tasks that involve only quite small values in practice. On the other hand, users who are interested in other kinds of tasks may not even consider using GNU Prolog due to this trade-off.

The disequality in the following conjunction causes GNU Prolog to use its *sparse representation* for the domain of `X`, limiting it to integers up to at most 127 by default:

```
| ?- X #\= 2, X #> 200.
Warning: Vector too small - maybe lost solutions (...)

no
```

In this case, at least a warning is emitted that solutions *may* have been lost. If we exchange the two constraint goals, we get the opposite and *correct* answer "yes" instead of "no", since there clearly are integers greater than 200 that do not equal 2:

```
| ?- X #> 200, X #\= 2.

X = _#2(201..268435455)

yes
```

As the manual notes, the solver cannot detect lost values in some cases. For example, the system *incorrectly* fails without qualification in:

```
| ?- X #= Y*Z.

no
```

On the other hand, the following case which also involves only variables works as one may expect:

```
| ?- X #= Y*Y.

X = _#21(0..268402689)
Y = _#2(0..16383)

yes
```

Again, this trade-off has little impact for most practical applications for which GNU Prolog is being used. However, more predictable behaviour of propagators is very desirable for automated testing and declarative debugging, which rely on properties like monotonicity and commutativity.

3.3.2 Indexicals

Internally, GNU Prolog uses *indexicals* ([CD96a], [HSD98]) to implement propagators. The main idea of indexicals is to declaratively describe the domains of variables as functions of the domains of related variables. The indexical language consists of the constraint `in/2` and expressions such as `min(X)..max(X)`. It also includes specialized constructs that make it applicable to describe a large variety of arithmetic and combinatorial constraints. Fig. 3.1 shows several examples of indexicals, cited from the actual source code of GNU Prolog. Notice how compact and elegant these definitions are.

```
1          /*------------*
2           * Partial AC *
3           *------------*/
4
5
6   x_eq_y(fdv X,fdv Y)
7
8   {
9    start X in min(Y) .. max(Y)
10   start Y in min(X) .. max(X)
11  }
12
13
14
15
16  x_plus_c_eq_y(fdv X,int C,fdv Y)
17
18  {
19   start X in min(Y) - C .. max(Y) - C
20   start Y in min(X) + C .. max(X) + C
21  }
22
23
24         /*------------*
25          *  Full AC   *
26          *------------*/
27
28
29  x_eq_y_F(fdv X,fdv Y)
30
31  {
32   start X in dom(Y)
33   start Y in dom(X)
34  }
35
36
37
38
39  x_plus_c_eq_y_F(fdv X,int C,fdv Y)
40
41  {
42   start X in dom(Y) - C
43   start Y in dom(X) + C
44  }
45
46
47  xy_eq_z_F(fdv X,fdv Y,fdv Z)
48
49  {
50   start Z in dom(X)**dom(Y)
51   wait_switch
52      case min(Z)>0
53              start Y in 1..max_integer
54              start X in dom(Z)//dom(Y)
55              start Y in dom(Z)//dom(X)
56
57      case max(Z)==0 && min(Y)>0
58              start X in { 0 }
59
60      case max(Z)==0 && min(X)>0
61              start Y in { 0 }
62  }
63
```

Figure 3.1: Examples of indexicals in GNU Prolog

3.3.3 Application: Constructing 2-(16,4,2) designs

To illustrate some of the phenomena one encounters when working with a typical CLP(FD) system, we now use GNU Prolog to search for specific combinatorial objects from *design theory*, a subfield of discrete mathematics:

Definition 3.1. A t-(v, k, λ) *design* is a collection of k-element subsets (called *blocks*) of a v-element set V, such that every t-element subset of V is contained in exactly λ blocks.

In this section, our goal is to find 2-(16,4,2) designs that are *decomposable*, that is, the union of two 2-(16,4,1) designs, and *supersimple*, which means that every two blocks share at most two points. We actually faced this task in a real-life situation: We wanted to reproduce the results described by Colbourn in [Col99] for reasons that are completely unrelated to the present thesis, and one of the steps outlined in that paper is the construction of such designs.

Fig. 3.2 shows a GNU Prolog program that describes such designs declaratively with finite domain constraints, using the integers $0, \ldots, 15$ as the set V. Each block is a list containing 4 finite domain variables, which are constrained to be strictly ascending to break symmetries within blocks (line 27). To express that every pair of numbers occurs in at most one block, we build a *multiplication table* (lines 56 and 57) consisting of triples of the form (n_i, n_j, p_{ij}) for each pair of numbers n_i, n_j, $n_i < n_j$. The value p_{ij} is computed as $n_1 \times 16 + n_2$ and is thus unique for each such pair of numbers. Next, we collect all pairs of variables that occur in the same block, and extend them to triples by adding one new variable to each of them.

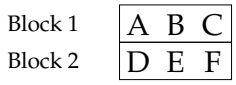


Figure 3.3: Two blocks of a design containing variables

For example, given the two blocks of a (different) design shown in Fig. 3.3, this yields the triples (A,B,x_1), (A,C,x_2), (B,C,x_3), (D,E,x_4), (D,F,x_5) and (E,F,x_6), where x_i denotes a free variable that does not occur anywhere else in the formulation. Each of these triples is constrained to be an element of the previously built multiplication table (using GNU Prolog's built-in `fd_relation/2` constraint in line 58), and the variables x_i are constrained to be pairwise distinct (using the `fd_all_different/1` constraint in line 60). These steps guarantee that every pair of numbers occurs in at most one block:

Proof. Suppose integers a and b, $a < b$, occur together in two different blocks, and let (a, b, x_i) and (a, b, x_j) be triples that were built from such

blocks. Then, by the `fd_relation/2` constraint, $x_i = x_j = a \times 16 + b$. But by the `fd_all_different/1` constraint, $x_i \neq x_j$. Contradiction. □

This example shows that quite large numbers can arise even when formulating tasks that involve only small values on the surface. In the present case, we describe designs that consist of numbers between 0 and 15, using a multiplication table that contains numbers as high as $14 \times 16 + 15 = 239$. This already exceeds the default value of GNU Prolog's `vector_max` variable, and we therefore set it to a higher value in line 25.

We exhaustively search for such designs and emit them via *forced backtracking* with the following query:

```
?- supersimple_decomposable_16_4_2(Solution, Vs),
   fd_labelingff(Vs),
   portray_clause(Solution),
   fail.
```

The *first-fail* search strategy in the above query works well in this case: On a 2.66 GHz iMac, GNU Prolog 1.3.1 emits its first solution (Fig. 3.4) after a few seconds. After about 2 weeks of computation time and exactly 22,219 correct solutions, it emits its first *invalid* answer, shown in Fig. 3.5, followed by many other answers that are no solutions, indicating a mistake in the internal mechanism of GNU Prolog.

In this case, the solver's failure to emit a valid design is so obvious that it is easy to spot. In more complex applications, it is easy to miss such mistakes especially when one does not expect them.

We reported this problem on the GNU Prolog mailing list. The source of the problem was found to be an overflow in the system's time-stamp mechanism that is used for scheduling propagators. The author of GNU Prolog included a change that resets an internal counter of the solver to 1 when it has become negative due to overflow in the next release of GNU Prolog, version 1.4.0. In that version, the code that assesses the size of a domain was also changed. The new code contains a mistake which affects the variable selection strategy used in our query (which chooses a variable with the *smallest* domain and therefore uses the code) so heavily that not a single solution is emitted with the above query even after several days of computation time. This independent mistake was corrected in the next release of GNU Prolog, version 1.4.1. After about 16 days of computation time, this version emits a different, but also invalid, answer after again emitting exactly 22,219 correct solutions with the above query. We again reported this problem on the GNU Prolog mailing list, and the author made another change to the constraint solver that is included in GNU Prolog as of version 1.4.2. After about 27 days of computation time that version emits more than 22,219 solutions, all of which are correct.

```
supersimple_decomposable_16_4_2(D1-D2, Vs) :-
        d_16_4_1(D1, Vs1), d_16_4_1(D2, Vs2),
        leq2_in_common(D1, D2),
        D1 = [[_,_,X,_]|_], D2 = [[_,_,Y,_]|_], X #< Y,
        append(Vs1, Vs2, Vs).

leq2_in_common([], _).
leq2_in_common([Block|Blocks], D2) :-
        leq2_each(D2, Block), leq2_in_common(Blocks, D2).

leq2_each([], _).
leq2_each([B|Bs], Block) :-
        phrase(in_common(B, Block), Cs), sum(Cs, #=, N), N #=< 2,
        leq2_each(Bs, Block).

in_common([], _) --> [].
in_common([X|Xs], Bs) --> in_common_(Bs, X), in_common(Xs, Bs).

in_common_([], _) --> [].
in_common_([X|Xs], Y)  --> { X #= Y #<=> B }, [B], in_common_(Xs, Y).

length_(L, Ls) :- length(Ls, L).

d_16_4_1(Blocks, Vars) :-
        fd_set_vector_max(500),
        length(Blocks, 20),
        maplist(length_(4), Blocks), maplist(block, Blocks),
        maplist(nth0(0), Blocks, Firsts),
        chain(Firsts, #=<), length(Five, 5),
        append(Five, [F1,F2,F3,F4,F5,F6,F7|_], Blocks),
        Blocks = [[0,1,_,_]|_], ordered_by_second(Blocks),
        maplist(first(0), Five), maplist(first(1), [F1,F2,F3,F4]),
        maplist(first(2), [F5,F6,F7]), append(Blocks, Vars),
        numlist(0, 15, Players),
        maplist(fd_exactly(5, Vars), Players), unique_pairs(Blocks),
        rests_diff(Five),
        rests_diff([F1,F2,F3,F4]),
        rests_diff([F5,F6,F7]).

rests_diff(Ls) :- maplist(arg(2), Ls, Rests), append(Rests, Diff),
        fd_all_different(Diff).

ordered_by_second([]).
ordered_by_second([_]) :- !.
ordered_by_second([[A,B|_],Second|Rest]) :-
        Second = [C,D|_], A #= C #==> B #< D,
        ordered_by_second([Second|Rest]).

block(Block) :- chain(Block, #<).

first(N, [N|_]).

unique_pairs(Blocks) :-
        phrase(blocks_triples(Blocks), Triples),
        findall([A,B,P], (fd_domain([A,B], 0, 15), A #< B,
                          P #= A*16+B, fd_labeling([A,B])), Table),
        maplist(fd_relation(Table), Triples),
        maplist(nth0(2), Triples, Ps),
        fd_all_different(Ps).

blocks_triples([])     --> [].
blocks_triples([B|Bs]) --> block_triples(B), blocks_triples(Bs).

block_triples([])     --> [].
block_triples([L|Ls]) --> block_triples(Ls, L), block_triples(Ls).

block_triples([], _)     --> [].
block_triples([Y|Ys], X) --> [[X,Y,_]], block_triples(Ys, X).
```

Figure 3.2: GNU Prolog CLP(FD) formulation of supersimple decomposable 2-(16,4,2) designs (omitted predicates are defined as in SWI-Prolog)

2-(16,4,1):

Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9	Block 10	Block 11	Block 12	Block 13	Block 14	Block 15	Block 16	Block 17	Block 18	Block 19	Block 20
0	0	0	0	0	1	1	1	1	2	2	2	3	3	4	4	5	5	6	9
1	2	3	4	6	2	3	7	11	3	7	11	4	6	5	8	6	8	7	10
5	8	7	9	10	4	8	10	13	5	9	12	14	12	7	10	9	12	8	13
14	13	11	12	15	6	9	12	15	10	15	14	15	13	13	11	11	15	14	14

2-(16,4,1):

0	0	0	0	0	1	1	1	1	2	2	2	3	3	4	4	4	5	5	6
1	2	3	8	12	2	3	5	9	3	6	7	5	7	6	7	10	7	8	10
6	4	9	11	13	8	4	11	14	11	9	13	6	8	8	9	12	10	9	11
7	5	10	14	15	10	13	12	15	15	12	14	14	12	15	11	14	15	13	13

Figure 3.4: A supersimple decomposable 2-(16,4,2) design found by GNU Prolog, displayed as two 2-(16,4,1) designs

Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9	Block 10	Block 11	Block 12	Block 13	Block 14	Block 15	Block 16	Block 17	Block 18	Block 19	Block 20
0	0	0	0	0	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3
1	2	3	4	6	2	3	7	11	3	7	10	4	4	4	4	4	4	4	4
5	8	7	9	10	4	8	10	13	5	9	11	5	5	5	5	5	5	5	5
14	13	11	12	15	6	9	12	15	12	15	14	6	6	6	6	6	6	6	6

0	0	0	0	0	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3
1	2	3	4	10	2	3	5	6	3	4	5	6	5	5	5	5	5	5	5
7	12	5	6	11	10	4	8	11	7	9	6	10	7	7	7	7	7	7	7
9	14	15	8	13	15	13	12	14	8	11	13	12	10	10	10	10	10	10	14

Figure 3.5: Erroneous answer emitted by GNU Prolog 1.3.1 after about 2 weeks of computation time. It contains several identical blocks, for example blocks 14 through 19 are (erroneously) the same even in *both* of the two substructures.

3.4 SICStus Prolog

SICStus Prolog ([CM12]) is well-known by academic and industrial users alike as one of the most advanced CLP systems that is currently available. In SICStus Prolog, CLP(FD) is available as `library(clpfd)`, which must be explicitly loaded before it can be used. In comparison with many other Prolog systems, the CLP(FD) solver of SICStus Prolog stands out particularly both due to the large number of semantically rich constraints that it provides and its high performance on many problems of practical relevance.

In SICStus Prolog, most global constraints are implemented in the programming language C. Like GNU Prolog, SICStus uses *indexicals* to implement several arithmetic constraints.

3.4.1 Symbolic infinities as domain boundaries

SICStus is notable as the first widely available Prolog system that uses a symbolic notion for *infinities* as domain boundaries in CLP(FD): The atoms `inf` and `sup` denote negative and positive infinity, respectively. These atoms can be used when constraining the domain of a variable (`in/2`) and are also used in residual goals, for example, with SICStus 4.2.3 (32-bit):

```
| ?- Y in inf..0, X #> 3.
Y in inf..0,
X in 4..sup ?
```

However, these atoms only mask underlying and still comparatively small numbers which are the actual domain boundaries for finite domain variables in SICStus Prolog. When these boundaries are exceeded, a *representation error* is thrown:

```
| ?- X #= 1000000000 + 1.
! Representation error in argument 2 of user: #= /2
```

This is a declaratively valid way to denote that a system limit has been breached. In cases like this, it is clearly preferable to silent failure, which indicates that no solution exists.

In contrast, arbitrary precision integers are supported in SICStus Prolog when evaluating arithmetic expressions with `is/2`:

```
| ?- X is 10000000000000000000000000000000000000000 + 1.
X = 10000000000000000000000000000000000000001 ?
```

Clearly, it would be ideal if finite domain constraints could be used uniformly for all integer calculations that arise in programs. However, since finite domain constraints cannot handle arbitrarily large numbers in SICStus Prolog, its lower-level arithmetic predicates need to be taught, learned and used until finite domain constraints become sufficiently general to be the only predicates that are needed to perform integer arithmetic.

3.4.2 Termination properties

Since SICStus provides notions for infinities, it is natural to ask: Does constraint propagation in SICStus Prolog always terminate? In practice, we can observe terminating propagation on still common 32-bit platforms even when SICStus takes some time for apparently simple queries, such as:

```
| ?- X #> abs(X).
! Representation error in user:'t=<u+c'/3

| ?- X #> Y, Y #> X, X #> 0.
! Representation error in user:'t>=u+c'/3
```

On 32-bit platforms, these queries terminate as shown in that they yield a *representation error* after a quite noticeable delay. On increasingly more common 64-bit platforms, this can no longer be observed. We will say more about desirable termination properties of propagation in Section 4.6.

3.4.3 Semantic inconsistencies

Although the CLP(FD) solver of SICStus Prolog is widely used throughout academia and industry alike and thus benefits from a lot of testing, one can readily find semantic inconsistencies in the system's answers that make it hard to tell what the constraints actually mean. For example, using the most recent version of SICStus Prolog (4.2.3) at the time of this writing, we see that `#=/2` does not constrain its arguments to integers in cases like:

```
| ?- 0 #= X - X.
true ?
```

Notice that the query succeeds unconditionally, meaning that it is true for any term `X`. As a consequence, we have for example:

```
| ?- X = f(a), 0 #= X - X.
X = f(a) ?
```

From this and elementary syntactic substitution, we should be able to deduce that `0 #= f(a) - f(a)` holds as well. But it does not hold:

```
| ?- 0 #= f(a) - f(a).
! Existence error in user:f/1
```

Moreover, applying a syntactic substitution can also yield new and unintended solutions. For example, the following yields a *type error*, as intended:

```
| ?- Y = 0, X #= Y*a.
! Type error in argument 2 of user:'x*y=z'/3
```

but after substituting `0` for `Y`, we obtain:

```
| ?- X #= 0*a.
X = 0 ?
```

In the above cases, one may argue that the system's behaviour is justifiably undefined because the input is ill-typed in the first place. However, such phenomena also occur with valid arithmetic expressions, as in:

```
| ?- 1 + 2 #= 3.
yes
```

whereas if we simply introduce an additional unification, we get:

```
| ?- X = (1 + 2), X #= 3.
! Type error in argument 1 of user: #= /2
! expected an integer, but found 1+2
! goal:  1+2#=3
```

In contrast, using `setof/3` to collect all solutions works as intended even in the case above:

```
| ?- X = (1 + 2), setof(X, X #= 3, Solutions).
X = 1+2,
Solutions = [1+2] ? ;
no
```

Unexpectedly though, it again raises an error if we combine both goals:

```
| ?- setof(X, (X = (1 + 2), X #= 3), Solutions).
! Type error in argument 1 of user: #= /2
! expected an integer, but found 1+2
! goal:  1+2#=3
```

The lack of such elementary algebraic properties in a constraint system significantly complicates reasoning about queries and their results and thus makes it harder to apply declarative debuggers and black-box tests. For practical scheduling tasks, these properties are usually less relevant.

In our tests of SICStus Prolog, we also found a category of mistakes that involve only finite domain variables and constraints without any syntactic substitutions. They occur when reifying expressions that involve division, `mod` and `rem`. For example, queries like:

```
| ?- X #= Y/0 #<=> B.

| ?- X #= Y mod 0 #<=> B.
```

incorrectly *failed* although they should succeed with `B = 0`. We reported this issue to the support team of SICStus, and it is corrected as of version 4.0.3.

3.5 ECLiPSe

ECLiPSe ([SS12]) is a another popular and widely used plattform for CLP. ECLiPSe ships with an IC (Interval Constraint) library, which is a hybrid integer/real interval arithmetic constraint solver. IC replaces the `fd`, `ria` and `range` libraries which are also included in the ECLiPSe distribution.

According to the ECLiPSe constraint library manual which describes the IC library, "integer variables and constraints ought to behave as expected until the values being manipulated become large enough that they approach the precision limit of a double precision floating point number (2^{51} or so). Beyond this, lack of precision may mean that the solver cannot be sure which integer is intended, in which case the solver starts behaving more like an interval solver than a traditional finite domain solver."

Indeed, using the latest version of ECLiPSe (6.1) at the time of this writing, we find for example that 2^{52} is correctly evaluated using the IC library on a 32-bit platform:

```
[eclipse 5]: X #= 2^52.

X = 4503599627370496
Yes (0.00s cpu)
```

However, when evaluating 2^{53}, the IC library resorts to intervals:

```
[eclipse 6]: X #= 2^53.

X = X{9007199254740991 .. 9007199254740994}


Delayed goals:
        X{9007199254740991 .. 9007199254740994} #=
          9007199254740991.0__9007199254740994.0
Yes (0.00s cpu)
```

This behaviour of the IC library means that it cannot be uniformly used for all integer calculations that arise in programs. In Section 4.5, we will see why this is desirable.

3.6 Gecode

Gecode is a toolkit for developing constraint-based systems and applications. Like ILOG, Choco (see respective homepages) and other CP systems, Gecode is not a *logic* programming system and therefore also not a CLP(FD) system. Nevertheless, it is interesting to briefly compare correctness aspects of a CP system to the CLP(FD) systems we discuss in this chapter. We choose Gecode as one of the most prominent and well-known representatives of CP systems.

3.6.1 Guarantee of mistakes

To the question "Does Gecode have bugs?", we find in the Gecode manual ([STL10]) the following answer:

> Yes, of course! But, Gecode is very thoroughly tested (tests cover almost 100%) and extensively used (several thousand users). If something does not work, we regret to inform you that this is most likely due to your program and not Gecode. Again, this does not mean that Gecode has no bugs. But it does mean that it might be worth searching for errors in your program first.

Given the complexity of Gecode, it is understandable that its authors have taken this attitude towards the inevitable mistakes in contains. However, as authors of constraint systems, can we not formulate and guarantee more interesting properties about our systems? We will see examples for formulating and verifying specific properties of our code in Section 5.6.

3.6.2 History of corrections

Gecode is notable for the meticulous ChangeLog that is included in the source distribution. The authors of Gecode are to be applauded for stating precisely what was corrected and improved for each release, and giving full credits to contributors and users who reported issues. Fig. 3.6 shows an excerpt of a ChangeLog entry that describes corrections in the finite domain integer component of Gecode. We include this excerpt to show that correctness considerations must also be applied in constraint systems that do not support logical variables.

```
1  Changes in Version 4.0.0 (2013-03-14)
2
3    - Bug fixes
4      - Fixed precede constraint with less than two values.
5        (minor, thanks to Roberto Castañeda Lozano)
6      - Fixed a bug where bounds consistent distinct reported
7        subsumption instead of failure in certain cases.
8        (major, thanks to Lin Yong)
9      - Fixed potential rounding issues in sqr and sqrt constraints.
10       (minor)
11     - Fixed copying of tuple sets in extensional constraints and
12       IntSets in sequence constraints (could lead to crashes when
13       using parallel search). (major, thanks to Manuel Baclet)
14     - Added missing propagation for nary min/max constraint.
15       (minor, thanks to Jean-Noël Monette)
16     - Make extensional constraints work with empty tuple sets.
17       (minor, thanks to Peter Nightingale)
18     - Fix count (global cardinality constraint) for multiple
19       occurrences of the same value in the cover array.
20       (minor, thanks to Peter Nightingale)
```

Figure 3.6: Examples of corrections in Gecode

3.7 Summary of properties

B-Prolog ([Zho12]) is an example of another widely used CLP(FD) system. It is notable for using *action rules* ([Zho06]) in its implementation.

However, instead of attempting to cover or even mention all CLP(FD) systems that are currently available, we now summarize some important *properties* of available CLP(FD) systems as follows:

- Reasoning is limited to comparatively small integers. When these limits are exceeded, one of the following happens:

  (1) *errors* are thrown (example: SICStus Prolog)

  (2) the system fails *silently* (example: GNU Prolog)

  (3) the behaviour is hard to predict (example: ECLiPSe)

  Of these options, only (1) is declaratively sound. However, to find all solutions, it would be even better if these limits did not exist at all.

- The CLP(FD) systems we discuss in this chapter are inherently *not monotonic* or fail to satisfy elementary algebraic properties such as commutativity of conjunction due to their defaulty representation (see Section 4.2) of arithmetic expressions: Variables that occur in expressions are implicitly constrained to integers, although declaratively, they stand for compound expressions as well. We will discuss this problem and how it can be avoided in the following chapter.

- No guarantees are provided regarding termination. In the following chapters, we will see why it is desirable that constraint propagation *always* terminates in a CLP(FD) system.

- If guarantees are given, they typically state the efficiency of global propagators in $\mathcal{O}$-notation, or the type of consistency that is reached after filtering. However, there are other important guarantees that should be stated explicitly in manuals. Examples of valid questions that should be answered in manuals are:

  1. Can labeling yield redundant solutions?
  2. Does labeling always terminate?
  3. Is labeling always complete?

  We will see why such guarantees are desirable, and also formulate guarantees for individual propagators in the following chapters.

4 A new CLP(FD) system

4.1 Introduction

To overcome several of the limitations that we explained in the previous chapter, we implemented a new CLP(FD) system. When we started to implement the system in 2007, our goals were:

- No artificial limits on the size of domains and arising numbers.
- Always terminating propagation.
- The system shall be implemented in Prolog to keep it concise and easy to study, analyse and improve.

Ultimately, and as we will show in this chapter, these goals all serve the purpose of ensuring correctness above all else. We present the system in the following sections, giving special attention to correctness considerations. We again emphasize our conscious decision to include portions of the actual Prolog source code, since only the code that is actually executed matters when considering correctness issues. Even if there were a different language to precisely describe what our solver does, that description would in our opinion likely be at least as verbose as the actual Prolog implementation that we present, and the advantage of the Prolog code is that it is directly executable. We hope that our readers enjoy studying the code, learn from it, and find all remaining mistakes.

4.2 Defaulty representations and monotonicity

Before describing any particular feature or approach of our system, we explain the notion of *defaulty* representations of data and why we avoid them in our solver. Consider the following *type* definition, describing a Prolog clause body:

```
is_body(true).
is_body((A,B)) :- is_body(A), is_body(B).
is_body(G)     :- is_goal(G).
```

Informally, this means:

- the atom `true` is a valid clause body
- a term of the form `(A,B)`, sometimes called "*and-list*", is a clause body if both `A` and `B` are clause bodies
- a term `G` is a clause body if it is a *goal*. We omit the definition of `is_goal/1` since it is not relevant for explaining the main problem with this representation of clause bodies.

According to this definition, the term `(true,(true,true))`, which can also be written as `(true,true,true)` is a valid clause body. And indeed we see from the following interaction with SWI-Prolog that it is the case:

```
?- is_body((true,true,true)).
true .
```

However, from the blank that it issued after `true`, we also see that `is_body/1` is not deterministic in this case. The reason is that the third clause head (`is_body(G)`) is so general that it subsumes every term, and this has to be so because there is no more specific term that subsumes all possible goals in this representation of Prolog code. To obtain a deterministic predicate, it is tempting to introduce `!/0` to commit as early as possible, leaving the third clause as a *default* case that is tried when the ones before it cannot be applied:

```
is_body(true)  :- !.
is_body((A,B)) :- !, is_body(A), is_body(B).
is_body(G)     :- is_goal(G).
```

A representation whose handling requires or suggests such a *default* case, or – more generally – cannot be expressed via pattern matching, is aptly called *defaulty* and is problematic for the following reason: While the addition of `!/0` makes the above query deterministic

```
?- is_body((true,true,true)).
true.
```

the most general query now only yields a single solution:

```
?- is_body(Body).
Body = true.
```

This destroys *commutativity* of goals:

```
?- is_body(Body), Body = (true,true).
false.

?- Body = (true,true), is_body(Body).
Body = (true, true).
```

and renders the program *non-monotonic*, meaning that adding further constraints can yield new solutions:

```
?- is_body(Body), Body = (true,true).
false.

?- Body = (true,true), is_body(Body), Body = (true,true).
Body = (true, true).
```

We cannot change the built-in representation of Prolog clause bodies, but we can:

1. avoid defaulty representations for all internal data structures
2. transform defaulty representations to cleaner data structures before doing anything else with them, and – if necessary – transform them back to defaulty structures only after nothing else needs to be done with them.

Defaulty representations *do* have an advantage, which is mainly *convenience* for users when typing them. Suppose for example that we settle on the following representation of Prolog clause bodies and at the same time represent the program itself according to this specification:

```
is_body(true).
is_body((A,B)) :- g(is_body(A)), g(is_body(B)).
is_body(g(G))  :- g(is_goal(G)).
```

Informally, this means:

- the atom `true` is a valid clause body
- `(A,B)` is a clause body if `A` and `B` are clause bodies
- a term `g(G)` is a clause body if `G` is a goal.

This representation is *not* defaulty since goals are now uniquely distinguished by a common functor `g/1`. Note, however, that this representation requires that users consistently wrap goals into such a structure, as the example itself already shows in the bodies of the last two clauses. While the declarative advantages of this representation are evident, the program itself can hardly be called more readable than the initial version. In practice, therefore, we will often have to allow defaulty data structures for the sake of convenience, and convert them to cleaner representations before doing anything else with them.

A Prolog *list* is an example of a data structure that is *not* defaulty, since the two possible cases are readily distinguished by their functors:

```
is_list([]).
is_list([_|Rest]) :- is_list(Rest).
```

A list is not a defaulty data structure even when its *elements* use a defaulty representation.

Throughout the implementation of our system, we use *lists* to represent conjunctions of goals, since lists can be conveniently described with Prolog's definite clause grammars (DCGs) and many built-in predicates are readily available for reasoning about lists. When actual Prolog code needs to be emitted, we use the predicate `list_goal/2` (Fig. 4.1) to convert such lists to executable Prolog goals.

```
1  list_goal([], true).
2  list_goal([G|Gs], Goal) :- foldl(list_goal_, Gs, G, Goal).
3
4  list_goal_(G, G0, (G0,G)).
```

Figure 4.1: Converting a list of goals to an executable Prolog goal

4.3 A monotonic CLP(FD) system

We discussed the notion of defaultyness and its relation to monotonicity for the following reason: All CLP(FD) systems that we describe in this thesis use a defaulty representation for finite domain expressions and are also *not* monotonic. We have already seen an example of GNU Prolog's non-monotonicity in Section 3.3.1. However, there is an even deeper declarative shortcoming that all mentioned systems have in common: When a logical variable is encountered in a finite domain expression, it is implicitly constrained to *integers*, while declaratively, it stands for other expressions as well. Consider for example the following interaction with GNU Prolog:

```
| ?- X #= 3, X = (1+2).

no
```

and contrast it with the result of exchanging the two goals, which makes the query succeed, violating commutativity of goals and monotonicity:

```
| ?- X = (1+2), X #= 3.

X = 1+2

yes
```

Indeed, the query succeeds for any finite domain expression `X` that evaluates to 3, for example `X = (5-2)` or `X = (3*2-3)`, but only if the unification is placed before the finite domain constraint. Similar declarative problems are readily found in other constraint systems as well. Clearly, there is no way to satisfy both of the following requirements at the same time:

(1) the query `X #= 3` must yield the fixed solution `X = 3`

(2) the query `X #= 3` must yield as solutions all finite domain expressions `X` that evaluate to 3.

Requirement (1) is necessitated by the way users interact with the constraint system and expect concrete integers as solutions, and requirement (2) is necessary for obtaining commutativity of goals and monotonicity.

In our system, users can avoid this dilemma by explicitly marking finite domain *variables* in CLP(FD) expressions with the functor `?/1`: When `?(X)` appears in a finite domain expression, `X` is constrained to integers and cannot be any compound expression. For better readability, users can define `?` as a postfix operator with the declaration

```
:- op(5, xf, ?).
```

so that `?(X)` can also be written as `X?`. Using this syntax with our system, which we will do throughout the following sections, the above queries yield:

```
?- X? #= 3, X = (1+2).
false.

?- X = (1+2), X? #= 3.
ERROR: Type error: ‘integer’ expected, found ‘1+2’
```

Note that this *type error* can be replaced by *silent failure* while preserving declarative equivalence, but in general *instantiation errors* cannot. When the `?/1`-syntax is used consistently in queries, our CLP(FD) system is monotonic: Posting further constraints cannot yield new solutions. For backwards compatibility with other Prolog systems, we also support plain variables in constraints, which are still implicitly constrained to integers. However, our system also provides the flag `clpfd_monotonic` to enforce the `?/1` syntax. When this flag is set to `true`, then a query like

```
?- X #= 3.
```

yields an *instantiation error* because too little is known about `X` in this case: Declaratively, `X` stands for every finite domain expression that evaluates to 3, and there is no way to express that with residual goals when we want to obtain the single solution `X = 3` at the same time. When `X` is wrapped in `?/1`, then the query succeeds as expected with the unique solution `X = 3`. The wrapper `?/1` can be omitted for variables that are already constrained to integers, for example:

```
?- X? #> 0, abs(X) #= 3.
X = 3.
```

4.4 System architecture

We now describe the architecture of our CLP(FD) system, by which we mean the interaction of its most important components. These components are in turn explained in more detail in the following sections.

We start by defining what we want from a CLP(FD) system. Speaking in most general terms, we expect:

(a) to be able to provide a set of *terms*, namely constraints

(b) to obtain again a set of terms, namely residual goals, as a result.

Between these two steps, the CLP(FD) system should make an effort to compute results that are in some sense appropriate or satisfy a property like bounds consistency. Notice that unconditional solutions, which are reported as a set of syntactic unifications, can be regarded as a special case of residual goals.

To satisfy these requirements, our constraint solver:

(1) provides a set of predicates that implement constraints and search

(2) performs constraint propagation on variables that are involved in constraints

(3) transforms remaining constraints into residual goals so that they can be reported.

In these steps, the contributions that distinguish our system from others are, in no particular order:

- Constraints are *monotonic* if the flag `clpfd_monotonic` is set to `true`.
- New domain-specific languages are used in the internal implementation of several important constraints that our system provides. These languages describe the semantics of constraint reification, compactified arithmetic, propagator selection, and parsing of expressions in arithmetic constraints and are explained throughout the following sections.
- Our system reasons over arbitrarily large integers, which yields new application opportunities that we present.
- Constraint propagation *always* terminates in our system. As we will show, this is an important property in several scenarios.

In typical CSPs, a constraint solver spends most of its computation time in step (2) above: This is where propagators are applied until some form of consistency is reached or it becomes clear that the problem has no solution.

In our system, we use *attributed variables* ([Hol92]) to store the constraints each variable is involved in. More precisely, each CLP(FD) variable gets an attribute of the form:

```
clpfd_attr(Left, Right, Spread, Dom, Ps)
```

The purpose of `Left`, `Right` and `Spread` is explained in Section 4.6. `Dom` is the variable's domain, represented as explained in Section 4.10. `Ps` stores the propagators that need to be invoked when the variable is instantiated or its domain changes. `Ps` is a term of the form:

```
fd_props(Gs, Bs, Os)
```

where `Gs`, `Bs` and `Os` are lists of propagators that need to be invoked in different situations:

- `Gs` are propagators that must be invoked (only) when the variable becomes *ground*, i.e., instantiated to a concrete integer. An example is the propagator for disequality (`X? #\= Y?`), which can only perform any filtering when either of its arguments is instantiated and need not be invoked in other situations.
- `Bs` are propagators that must be invoked (only) when one of the domain *boundaries* of the variable changes. An example is the propagator for addition (`X? #= Y? + Z?`), which ensures bounds consistency and need not be invoked in other situations.
- `Os` are propagators that must be invoked when the domain of the variable changes in any way. An example is the `all_distinct/1` constraint which we explain in Section 4.13.2.

As can be deduced from this description, all propagators that can affect a variable are invoked when the variable becomes instantiated to a concrete integer, since this means that its domain is restricted to a single element.

The propagators that need to be invoked are stored in each of these lists in the form:

```
propagator(P, State)
```

`P` denotes the propagator that needs to be invoked and its arguments, for example `pplus(X, Y, Z)`. We show the entire source code of two propagators in Section 4.11. `State` is a free variable that can be used to change the state of the propagator via attributes. This can be used to avoid redundant invocations of the same propagator, or to disable future invocations of the propagator for this variable.

In our CLP(FD) system, there is a predicate `run_propagator/2` which must be called like:

```
run_propagator(P, State)
```

when a specific propagator `P` needs to be invoked. Each propagator corresponds to a specific clause of this predicate, from which other predicates can be invoked as well. Users can also provide their own clauses to implement custom filtering algorithms.

As we explain in Section 4.6, propagation always terminates in our system. However, for any fixed integer N, it is easy to construct cases where `run_propagator/2` is invoked N times even with a fixed set of constraints. For example, the query

```
?- X? #> Y?, Y #> X, X in 0..1000.
false.
```

leads to more than 500 invocations of `run_propagator/2`, in which the domains of both `X` and `Y` are reduced until no more elements remain.

To limit the amount of local stack space that such propagation chains require, `run_propagator/2` does not usually call itself recursively either directly or indirectly when further constraints need to be invoked. Instead, we maintain two global queues of propagators that still need to be invoked. We use SWI-Prolog's global backtrackable variables to store both queues in a single term of the form:

```
fast_slow(Ps1, Ps2)
```

Both `Ps1` and `Ps2` are either the atom `[]` or queues of the form `Ps-Tail` where further elements can be appended in $\mathcal{O}(1)$ time by instantiating `Tail`. `Ps` is a list of propagators that need to be invoked. Fig. 4.2 shows how the queues are created and accessed. As can be seen from this code, when a propagator is fetched via `pop_queue/1`, the propagators that are stored in the first argument of the global queue representation have priority over those that are stored in the second. As the functor already indicates, this is because the propagators that are stored in the first argument are typically faster. The more time consuming propagators of global constraints are only run after the faster propagators have already filtered as many domain elements as possible. It is important to reset each of the two queues to `[]` when no more elements remain (line 21), so that the elements that have been removed from the queue can be properly garbage collected. The predicate `do_queue/0` (not shown) triggers all queued propagators, until fixpoint.

We have consciously kept our system's architecture as simple as possible while trying to get acceptable performance in typical use cases. It is clear that one propagator queue is needed to perform constraint propagation in $\mathcal{O}(1)$ local stack space. An additional, second propagator queue seems to be the bare minimum that is necessary for meaningful performance improvements. Yet, this apparently simple design already provides ample room for mistakes, as we will see in Section 5.5.

```
1  make_queue :- nb_setval('$clpfd_queue', fast_slow([], [])).
2
3  push_queue(E, Which) :-
4          nb_getval('$clpfd_queue', Qs),
5          arg(Which, Qs, Q),
6          (   Q == [] ->
7              setarg(Which, Qs, [E|T]-T)
8          ;   Q = H-[E|T],
9              setarg(Which, Qs, H-T)
10         ).
11
12 pop_queue(E) :-
13         nb_getval('$clpfd_queue', Qs),
14         (   pop_queue(E, Qs, 1) ->  true
15         ;   pop_queue(E, Qs, 2)
16         ).
17
18 pop_queue(E, Qs, Which) :-
19         arg(Which, Qs, [E|NH]-T),
20         (   var(NH) ->
21             setarg(Which, Qs, [])
22         ;   setarg(Which, Qs, NH-T)
23         ).
```

Figure 4.2: Accessing the propagator queues

4.5 Reasoning over arbitrarily large integers

As already mentioned, our CLP(FD) systems supports reasoning over arbitrarily large integers. For compatibility with SICStus Prolog, we support the atoms `inf` and `sup` in domain expressions. However, these atoms now denote the actual infinities instead of masking underlying smaller numbers. As we show in this section, supporting arbitrarily large integers yields new and interesting application opportunities for constraint solvers.

One can hardly speak of *finite* domain constraint solvers any more when domains can clearly be infinite as well. Still, given that there are always technological limits for the representation of any particular integer in a CLP(FD) system, calling them CLP($\mathcal{Z}$) seems at least equally deceiving. We therefore continue to call the systems we discuss in this thesis – including ours – CLP(**FD**) systems, since this is how they are known by convention although some of them can also represent and reason about infinite domains.

4.5.1 Large integers in conventional CLP(FD) tasks

In Section 2.2, we have already seen an example where reasoning over comparatively large integers is necessary to solve a problem over rational numbers with integer arithmetic. Well-known examples of large integers that arise in conventional CLP(FD) tasks are still comparatively rare, since most CLP(FD) systems can currently reason only over quite small integers and have therefore so far not been applicable to such problems. However, large integers arise not only in contrived examples, but in interesting practical applications as well.

For instance, a Prolog library called `julian` is an interesting practical

example where large integers arise in the context of calculations with dates and times that can be specified up to nanosecond resolution. Internally, this library uses our CLP(FD) system to efficiently evaluate queries that can be very general. Michael Hendricks has generously made this library freely available as a package for SWI-Prolog. You can install it with SWI-Prolog 6.3 or later via:

```
?- pack_install(julian).
```

When this library and our CLP(FD) system are loaded, we can for example ask: "During Dwight D. Eisenhower's presidency (1953–1961), when did the 4th of July fall on Sunday?":

```
?- form_time([dow(sunday), Year-07-04]),
   Year in 1953..1961.
```

and obtain the system's answer:

```
Year = 1954.
```

As another example, asking for the representation of *today* via:

```
?- form_time(today, T).
```

yields an integer that determines the current day, and a finite domain variable whose domain contains every nanosecond of this day:

```
T = datetime(56599, _G299),
_G299 in 0..86399999999999.
```

`library(julian)` is a great application of traditional CLP(FD) reasoning over comparatively large and naturally arising domains, and an example where our system's support for arbitrarily large integers is very useful.

4.5.2 Uniform integer arithmetic

Our constraint system's ability to reason over arbitrarily large integers allows you to use finite domain constraints instead of conventional built-in arithmetic with `is/2`, `>/2` etc. This makes integer arithmetic uniformly available via finite domain constraints.

Uniform integer arithmetic via finite domain constraints has obvious didactic advantages when teaching Prolog, since a smaller set of predicates needs to be explained and learned. At the same time, using finite domain constraints instead of low-level arithmetic can make programs significantly more general with acceptable overhead.

For example, consider the definition of `n_factorial/2` shown in Fig. 4.3. It uses finite domain constraints uniformly, and the predicate can therefore be used in all directions.

```
1  n_factorial(0, 1).
2  n_factorial(N, F) :-
3          N? #> 0,
4          N1? #= N - 1,
5          F? #= N * F1?,
6          n_factorial(N1, F1).
```

Figure 4.3: Relating a non-negative integer `N` to its factorial `F`

As with the conventional definition, we can ask for a number's factorial:

```
?- n_factorial(38, F).
F = 523022617466601111760007224100074291200000000 ;
false.
```

Moreover, we can now also ask in the other direction:

```
?- n_factorial(N, 3).
false.

?- n_factorial(N, 265252859812191058636308480000000).
N = 30.
```

We can also leave both arguments unspecified and still obtain solutions:

```
?- n_factorial(N, F).
N = 0, F = 1 ;
N = 1, F = 1 ;
N = 2, F = 2 ;
N = 3, F = 6 .
```

A version of `n_factorial/2` that is deterministic when its first argument is instantiated is shown in Fig. 4.4. It uses `zcompare/3`, which is a CLP(FD) version of the built-in `compare/3` predicate. `zcompare(R, X, Y)` is true if `X` is in relation `R` to `Y`, where `R` is either `<`, `=` or `>`.

```
1  n_factorial(N, F) :-
2          zcompare(R, N, 0),
3          n_factorial_(R, N, F).
4
5  n_factorial_(=, _, 1).
6  n_factorial_(>, N, F) :-
7          F? #= F0? * N,
8          N1? #= N - 1,
9          n_factorial(N1, F0).
```

Figure 4.4: Relating `N` to its factorial `F`, deterministic when `N` is instantiated

To transparently bring the performance of CLP(FD) constraints closer to that of conventional arithmetic predicates when the constraints are used in

modes that can also be handled by built-in arithmetic, our system uses `goal_expansion/2` to rewrite constraints at compilation time. Code that dynamically checks whether built-in arithmetic predicates can be used directly is transparently inserted.

CLP(FD) constraints can thus be used instead of conventional integer arithmetic with acceptable performance overhead in most programs.

4.6 Ensuring terminating propagation

In CLP(FD) systems that support arbitrarily large numbers, nonterminating propagation chains can arise. For example, the following queries lead to nonterminating propagation unless we take measures against it:

```
?- X? #> Y?, Y #> X, X #> 0.

?- X? #> X? * X? .

?- X? #> abs(X?).
```

Even if we were to allow unbounded propagation, we could not solve all tasks that can be expressed in our system. This is because our system can represent arbitrary Diophantine equations, which are known to be undecidable in general.

It is very desirable that constraint propagation always terminates, for the following reasons:

- when propagation cannot tell whether a set of constraints is consistent, the system should show residual goals instead of looping, so that stronger reasoners can be applied
- it significantly simplifies black-box testing (see Chapter 5), since constraints can be posted in any order, without risking nontermination
- it is a necessary condition for guaranteeing that `labeling/2` always terminates (see Section 4.14)

There are three ways in which nonterminating propagation can arise: Either the lower boundary of some domain increases without bound, or the upper boundary decreases without bound, or the distance between the smallest and largest integer in a domain representation (see Section 4.10.1) increases without bound. These three cases are shown in Fig. 4.5.

In our system, we guarantee terminating propagation by allowing each of these changes to occur at most *once* for each domain that is still infinite. The terms `Left`, `Right` and `Spread` (see Section 4.4) are used to keep track of these three changes for each domain. We reset the history of these

changes when users post new constraints, to give each posted constraint an opportunity for propagation. So far, this simple restriction has worked acceptably well in most practical use cases we are aware of. Users who absolutely need it and are aware of the possible consequences can set the Prolog flag `clpfd_propagation` to `full` for unrestricted propagation.

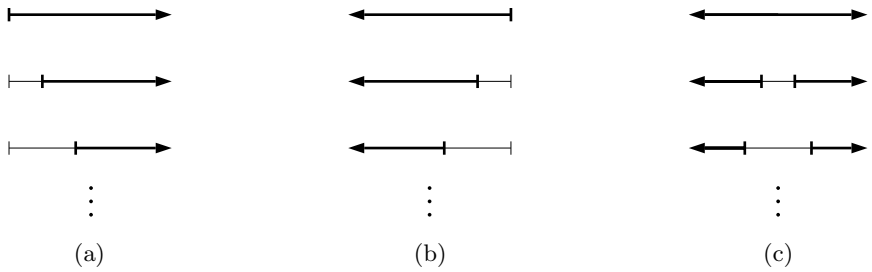


Figure 4.5: The three possible shapes of nonterminating domain changes.

4.7 Source code organisation

At the time of this writing, `library(clpfd)` comprises about 6,400 lines of Prolog source code, including documentation and comments. The approximate number of lines of codes (LOC) that are dedicated to its most important components are shown in Table 4.1. Most of these components are explained in the following sections.

system component	LOC
global constraints	1,883
propagators for arithmetic constraints	951
reification	622
reasoning about domains	600
labeling and optimisation	331
queue handling	295
compactified arithmetic	185
projecting attributes to residual goals	169
selecting propagators for constraints	164
parsing arithmetic expressions	76

Table 4.1: Amount of source code for important system components

4.8 Domain-specific languages

Our system is entirely written in Prolog. However, we found that some of its parts are best described by dedicated domain-specific languages (DSLs), which we translate to executable Prolog code at compile-time. DSLs are

typically devised with the goal of increased expressiveness and ease of use compared to general-purpose programming languages in their domains of application ([MHS05]). Examples of DSLs include *lex* and *yacc* ([JL87b]) for lexical analysis and parsing, regular expressions for pattern matching, HTML for document mark-up, VHDL for electronic hardware descriptions and many other well-known instances.

DSLs are also known as "*little languages*" ([Ben86]), where "little" primarily refers to the typically limited intended or main practical application scope of the language. For example, PostScript is a "little language" for page descriptions in this terminology.

Indexicals (see Section 3.3.2) are among the most prominent and well-known examples of DSLs in CLP(FD) systems.

The usefulness of deriving large portions of code automatically from shorter descriptions also motivates the use of *variable views*, a DSL to automatically derive *perfect* propagator variants, in the implementation of Gecode ([ST08], [ST13]).

Action rules (see Section 3.7) and Constraint Handling Rules ([Frü98]) are both Turing-complete languages that are very well-suited for implementing constraint propagators and even entire constraint systems like the finite domain constraint solver of B-Prolog.

These examples of DSLs are mainly used for the description and generation of constraint *propagation* code in practice. In the following sections, we contribute to these uses of DSLs by presenting new DSLs that we use in new and different contexts. They allow you to concisely express compactified arithmetic, *selection* of propagators (which is a task that is distinct and independent of their actual underlying implementation) and constraint reification with desirable declarative properties.

4.9 Compactified arithmetic

The first example of a DSL that we devised when implementing our constraint solver is the simplest to explain and understand: It describes the properties of *compactified* integer arithmetic, which means arithmetic on the topologically compact set $\mathbb{Z} \cup \{-\infty, +\infty\}$. As already mentioned in Section 4.5, we support the atoms `inf` and `sup` (denoting negative and positive infinity, respectively) in domain expressions. Internal calculations for domain boundaries must be able to correctly handle these atoms. When a domain boundary is the number N, it is internally represented as the compound term `n(N)`.

As an example of a domain calculation that arises in a solver, consider shifting a domain boundary `B` by an integer offset `O`, where `S` shall be the shifted result. This shifting occurs for example in the propagator for addition, when one of the summands or the sum is already instantiated. If all domain boundaries were integers, we could obtain it via:

```
S is B + O
```

However, `B` might be infinite, and therefore the built-in `is/2` predicate cannot be used directly. We therefore introduced a predicate `cis/2` (compactified `is/2`) that behaves like `is/2` and correctly handles infinities in addition to numbers (with wrapper `n/1`), allowing us to write:

```
S cis B + n(O)
```

In addition to `cis/2`, we define `cis_geq/2`, `cis_gt/2`, `cis_leq/2` and `cis_lt/2` as infix operators that we use to evaluate and compare expressions of compactified arithmetic.

For `cis/2`, we adopted the convention that a variable that is *not* wrapped in `n/1` is either one of the two infinities or a number (wrapped in `n/1`) at run-time, but never a compound expression. This convention lets us expand `cis/2` calls to auxiliary predicates at compile-time, reducing the overhead of inspecting the expression at run-time.

Compactified arithmetic expressions are partially evaluated at compile-time by using the `goal_expansion/2` mechanism of SWI-Prolog as shown in Fig. 4.6. The first clause of `goal_expansion/2` relates the right-hand side of a `cis/2` call to its left-hand side by producing a conjunction of goals that replaces the call. The other clauses of `goal_expansion/2` transform the different comparison operators to a few elementary cases. In contrast to `cis/2`, these operators cannot handle nested expressions, since there was so far no need support this in the implementation of our system.

We use the following `cis/2` call to illustrate the expansion mechanism:

```
NXU cis min((abs(n(Z))+n(1))*abs(n(Y))-n(1), XU)
```

This goal actually occurs in our solver during the calculation of upper boundaries of division. It is replaced by the following goals at compile-time. Variables that start with an underscore are introduced during the goal expansion phase and do not occur anywhere else:

```
cis_abs(n(Z), _G369),
cis_plus(_G369, n(1), _G375),
cis_abs(n(Y), _G380),
cis_times(_G375, _G380, _G384),
cis_minus(_G384, n(1), _G390),
cis_min(_G390, XU, NXU))
```

To complete our implementation of compactified arithmetic, we show the Prolog definitions for arithmetic operations with infinities in Fig. 4.7 and Fig. 4.8. Due to the previous expansion phase, these predicates only

```
goal_expansion(A cis B, Expansion) :-
        phrase(cis_goals(B, A), Goals),
        list_goal(Goals, Expansion).
goal_expansion(A cis_lt B, B cis_gt A).
goal_expansion(A cis_leq B, B cis_geq A).
goal_expansion(A cis_geq B, cis_leq_numeric(B, N)) :-
        nonvar(A), A = n(N).
goal_expansion(A cis_geq B, cis_geq_numeric(A, N)) :-
        nonvar(B), B = n(N).
goal_expansion(A cis_gt B, cis_lt_numeric(B, N))   :-
        nonvar(A), A = n(N).
goal_expansion(A cis_gt B, cis_gt_numeric(A, N))   :-
        nonvar(B), B = n(N).

cis_goals(V, V)          --> { var(V) }, !.
cis_goals(n(N), n(N))    --> [].
cis_goals(inf, inf)      --> [].
cis_goals(sup, sup)      --> [].
cis_goals(sign(A0), R)   -->
        cis_goals(A0, A), [cis_sign(A, R)].
cis_goals(abs(A0), R)    -->
        cis_goals(A0, A), [cis_abs(A, R)].
cis_goals(-A0, R)        -->
        cis_goals(A0, A), [cis_uminus(A, R)].
cis_goals(A0+B0, R)      -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_plus(A, B, R)].
cis_goals(A0-B0, R)      -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_minus(A, B, R)].
cis_goals(min(A0,B0), R) -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_min(A, B, R)].
cis_goals(max(A0,B0), R) -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_max(A, B, R)].
cis_goals(A0*B0, R)      -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_times(A, B, R)].
cis_goals(div(A0,B0), R) -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_div(A, B, R)].
cis_goals(A0//B0, R)     -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_slash(A, B, R)].
cis_goals(A0^B0, R)      -->
        cis_goals(A0, A), cis_goals(B0, B), [cis_exp(A, B, R)].
```

Figure 4.6: Goal expansion of compactified arithmetic

need to cover three elementary cases: the two infinities and numbers, but no compound expressions. Note that *indeterminate forms* like $+\infty + (-\infty)$ are pragmatically defined by these predicates as needed for the specific use cases of our constraint solver (in this concrete case: `sup+inf = sup`), and may need different interpretations in other applications. As can be seen in these definitions, Prolog's built-in arithmetic with `is/2` is internally used when possible, i.e., when no infinities are involved. This is where we transparently benefit from the underlying Prolog system's arbitrary precision integer arithmetic, which is available in most modern Prolog systems.

Of the DSLs presented in this chapter, the language of compactified arithmetic is maybe least applicable to other finite domain constraint systems, since only a few of them need to support expressions with symbolic infinities. Still, we consider it a valuable tool for documenting, in directly executable form, the error-prone boundary expressions for constraints like multiplication and floored division, which are much less obvious than those of addition especially when infinities are involved.

```
1  cis_gt(sup, B0) :- B0 \== sup.
2  cis_gt(n(N), B) :- cis_lt_numeric(B, N).
3
4  cis_lt_numeric(inf, _).
5  cis_lt_numeric(n(B), A) :- B < A.
6
7  cis_gt_numeric(sup, _).
8  cis_gt_numeric(n(B), A) :- B > A.
9
10 cis_geq(inf, inf).
11 cis_geq(sup, _).
12 cis_geq(n(N), B) :- cis_leq_numeric(B, N).
13
14 cis_leq_numeric(inf, _).
15 cis_leq_numeric(n(B), A) :- B =< A.
16
17 cis_geq_numeric(sup, _).
18 cis_geq_numeric(n(B), A) :- B >= A.
19
20 cis_min(inf, _, inf).
21 cis_min(sup, B, B).
22 cis_min(n(N), B, Min) :- cis_min_(B, N, Min).
23
24 cis_min_(inf, _, inf).
25 cis_min_(sup, N, n(N)).
26 cis_min_(n(B), A, n(M)) :- M is min(A,B).
27
28 cis_max(sup, _, sup).
29 cis_max(inf, B, B).
30 cis_max(n(N), B, Max) :- cis_max_(B, N, Max).
31
32 cis_max_(inf, N, n(N)).
33 cis_max_(sup, _, sup).
34 cis_max_(n(B), A, n(M)) :- M is max(A,B).
35
36 cis_plus(inf, _, inf).
37 cis_plus(sup, _, sup).
38 cis_plus(n(A), B, Plus) :- cis_plus_(B, A, Plus).
39
40 cis_plus_(sup, _, sup).
41 cis_plus_(inf, _, inf).
42 cis_plus_(n(B), A, n(S)) :- S is A + B.
43
44 cis_minus(inf, _, inf).
45 cis_minus(sup, _, sup).
46 cis_minus(n(A), B, M) :- cis_minus_(B, A, M).
47
48 cis_minus_(inf, _, sup).
49 cis_minus_(sup, _, inf).
50 cis_minus_(n(B), A, n(M)) :- M is A - B.
51
52 cis_uminus(inf, sup).
53 cis_uminus(sup, inf).
54 cis_uminus(n(A), n(B)) :- B is -A.
```

Figure 4.7: Arithmetic operations with infinities (part 1/2)

```
1  cis_abs(inf, sup).
2  cis_abs(sup, sup).
3  cis_abs(n(A), n(B)) :- B is abs(A).
4
5  cis_times(inf, B, P) :-
6          (   B cis_lt n(0) -> P = sup
7          ;   B cis_gt n(0) -> P = inf
8          ;   P = n(0)
9          ).
10 cis_times(sup, B, P) :-
11         (   B cis_gt n(0) -> P = sup
12         ;   B cis_lt n(0) -> P = inf
13         ;   P = n(0)
14         ).
15 cis_times(n(N), B, P) :- cis_times_(B, N, P).
16
17 cis_times_(inf, A, P)     :- cis_times(inf, n(A), P).
18 cis_times_(sup, A, P)     :- cis_times(sup, n(A), P).
19 cis_times_(n(B), A, n(P)) :- P is A * B.
20
21 cis_exp(inf, n(Y), R) :-
22         (   even(Y) -> R = sup
23         ;   R = inf
24         ).
25 cis_exp(sup, _, sup).
26 cis_exp(n(N), Y, R) :- cis_exp_(Y, N, R).
27
28 cis_exp_(n(Y), N, n(R)) :- R is N^Y.
29 cis_exp_(sup, _, sup).
30 cis_exp_(inf, _, inf).
31
32
33 cis_sign(sup, n(1)).
34 cis_sign(inf, n(-1)).
35 cis_sign(n(N), n(S)) :- S is sign(N).
36
37 cis_div(sup, Y, Z)  :- ( Y cis_geq n(0) -> Z = sup ; Z = inf ).
38 cis_div(inf, Y, Z)  :- ( Y cis_geq n(0) -> Z = inf ; Z = sup ).
39 cis_div(n(X), Y, Z) :- cis_div_(Y, X, Z).
40
41 cis_div_(sup, _, n(0)).
42 cis_div_(inf, _, n(0)).
43 cis_div_(n(Y), X, Z) :-
44         (   Y =:= 0 -> (  X >= 0 -> Z = sup ; Z = inf )
45         ;   Z0 is X // Y, Z = n(Z0)
46         ).
47
48 cis_slash(sup, _, sup).
49 cis_slash(inf, _, inf).
50 cis_slash(n(N), B, S) :- cis_slash_(B, N, S).
51
52 cis_slash_(sup, _, n(0)).
53 cis_slash_(inf, _, n(0)).
54 cis_slash_(n(B), A, n(S)) :- S is A // B.
```

Figure 4.8: Arithmetic operations with infinities (part 2/2)

4.10 Domains

In this section, we explain how domains are internally represented in our solver, and which predicates are available for reasoning about domains.

4.10.1 Domain representation

In our constraint solver over integers, a domain is a finite set of disjoint integer intervals. Internally, domains are represented as interval trees. Each node of an interval tree in our solver is one of:

- The atom `empty`, representing the empty set.
- A term `from_to(F, T)`, where `F` and `T` are domain boundaries, denoting all integers I such that `F cis_leq n(I)` and `n(I) cis_leq T`. This definition works for finite as well as infinite boundaries. See Section 4.9 for the representation of domain boundaries with compactified arithmetic (`n(N)`, `inf` and `sup`).
- A term `split(N, Left, Right)`, where `N` is an integer, and `Left` and `Right` are domains (again represented as interval trees) whose elements are all less than and greater than `N`, respectively. The represented domain is the union of `Left` and `Right`. The integer `N` is a *hole*, i.e., an integer that is not an element of the domain.

Fig. 4.9 shows a Prolog "type" definition to clarify our domain representation. The type definition can be used for internal consistency checks in the constraint solver. It also serves as our first example of how the compactified arithmetic operators are used.

Our choice to represent domains as interval trees remains to some extent arbitrary and motivated for example by trying to implement global constraints like `tuples_in/2` efficiently. This global constraint repeatedly checks whether a domain contains a specified integer, which is implemented by `domain_contains/2` in our system (see Section 4.10.2). When an interval tree is (sufficiently) balanced, `domain_contains/2` is performed in $\mathcal{O}(\log(n))$ time, where n denotes the number of intervals of the domain.

Lists of intervals, which are used for example in SICStus Prolog, are another natural Prolog representation of domains. One advantage of lists is that all required domain operations can be performed in $\mathcal{O}(n)$ time, where n is again the number of intervals.

```
1  is_domain(empty).
2  is_domain(from_to(From,To)) :-
3          is_boundary(From), is_boundary(To),
4          From cis_leq To.
5  is_domain(split(S, Left, Right)) :-
6          integer(S),
7          is_domain(Left), is_domain(Right),
8          all_less_than(Left, S),
9          all_greater_than(Right, S).
10
11 is_boundary(n(N)) :- integer(N).
12 is_boundary(inf).
13 is_boundary(sup).
14
15 all_less_than(empty, _).
16 all_less_than(from_to(From,To), S) :-
17         From cis_lt n(S), To cis_lt n(S).
18 all_less_than(split(S0,Left,Right), S) :-
19         S0 < S,
20         all_less_than(Left, S),
21         all_less_than(Right, S).
22
23 all_greater_than(empty, _).
24 all_greater_than(from_to(From,To), S) :-
25         From cis_gt n(S), To cis_gt n(S).
26 all_greater_than(split(S0,Left,Right), S) :-
27         S0 > S,
28         all_greater_than(Left, S),
29         all_greater_than(Right, S).
```

Figure 4.9: Type definition of our domain representation

4.10.2 Domain properties

Fig. 4.10 shows several predicates that give us useful information about domains: First of all, `default_domain/1` states the default domain for all finite domain variables. It is not advisable to change the default domain to a finite one, since that can lead to the loss of valid solutions without raising an error. As the names imply, `domain_infimum/2` and `domain_supremum/2` relate a domain to its infimum and supremum, `domain_num_elements/3` gives us the number of elements of a finite domain, and `domain_contains/2` is true iff the domain contains the given integer. Notice that the integer *must* be wrapped in `n/1` when it is used in a `cis/2` expression.

We show these definitions as examples for working with our domain representation. We do not show the Prolog definitions of *all* predicates that we use to test domain properties in the following sections; the remaining definitions are reasonably straight-forward, and interested readers can study them in the latest source code of our library.

```
1  default_domain(from_to(inf,sup)).
2
3  domain_infimum(from_to(I, _), I).
4  domain_infimum(split(_, Left, _), I) :- domain_infimum(Left, I).
5
6  domain_supremum(from_to(_, S), S).
7  domain_supremum(split(_, _, Right), S) :- domain_supremum(Right, S).
8
9  domain_num_elements(empty, n(0)).
10 domain_num_elements(from_to(From,To), Num) :- Num cis To - From + n(1).
11 domain_num_elements(split(_, Left, Right), Num) :-
12         domain_num_elements(Left, NL),
13         domain_num_elements(Right, NR),
14         Num cis NL + NR.
15
16 domain_contains(from_to(From,To), I) :-
17         From cis_leq n(I), n(I) cis_leq To.
18 domain_contains(split(S, Left, Right), I) :-
19         (   I < S -> domain_contains(Left, I)
20         ;   I > S -> domain_contains(Right, I)
21         ).
```

Figure 4.10: Examples of domain properties

4.10.3 Relations between domains

In addition to the simple definitions of domain properties, there are of course also predicates that define relations between two or more domains. They are used for set-theoretic operations on domains like removing elements and constructing unions and intersections. Again, these predicates essentially need to consider the three possible cases (`empty`, `from_to/2`, `split/3`) of an interval node and manipulate it accordingly.

We now present the full source code of two domain operations. The first example is one of the simplest domain operations: shifting a domain by an integer offset, shown in Fig. 4.11. Notice that as in `domain_contains/2`, the

second argument of the predicate is a Prolog *integer*, and that it again must be wrapped in `n/1` when it is used in a `cis/2` expression.

```
1  domain_shift(empty, _, empty).
2  domain_shift(from_to(From0,To0), O, from_to(From,To)) :-
3          From cis From0 + n(O), To cis To0 + n(O).
4  domain_shift(split(S0, Left0, Right0), O, split(S, Left, Right)) :-
5          S is S0 + O,
6          domain_shift(Left0, O, Left),
7          domain_shift(Right0, O, Right).
```

Figure 4.11: Shifting a domain by an integer offset

The second example, shown in Fig. 4.12, relates two domains to their *intersection*. Notice that the intersection cannot be `empty`, since this would mean that some variable has run out of domain elements. Moreover, empty leaves are eliminated from the interval tree already when the intersection is being constructed.

Note that an interval tree representation of a domain is not uniquely determined. For example, the following terms represent the same domain:

```
split(0, from_to(n(-5),n(-3)), from_to(n(1),n(2)))

split(-2, from_to(n(-5),n(-3)), from_to(n(1),n(2)))
```

However, our implementation of `domains_intersection/3` preserves the following important property, where `==/2` denotes Prolog term equivalence:

If $\texttt{T}_1$ and $\texttt{T}_2$ are interval trees in our representation, and $\texttt{T}_1$ contains all integers that are contained in $\texttt{T}_2$, and $\texttt{domains\_intersection}(\texttt{T}_1, \texttt{T}_2, \texttt{I})$ holds, then $\texttt{I} == \texttt{T}_1$.

This property lets us use `==/2` to detect whether a domain has changed, provided that we adopt the following convention:

When a domain `D1` must be assigned to a variable `X` whose domain is currently `D0`, then we first call `domains_intersection(D0, D1, DN)`, where `DN` is a fresh variable, and then assign the domain `DN` to `X`.

As we will see in the next section, propagators adhere to this convention.

Other relations between domains include for example `domain_expand/3` and `domain_contract/3`, which are used to scale intervals by a constant multiple, `domains_union/3`, `domain_negate/2` etc. We again refer interested readers to the actual source code of our library to study these definitions.

```
1  domains_intersection(D1, D2, Intersection) :-
2          domains_intersection_(D1, D2, Intersection),
3          Intersection \== empty.
4
5  domains_intersection_(empty, _, empty).
6  domains_intersection_(from_to(L0,U0), D2, Dom) :-
7          narrow(D2, L0, U0, Dom).
8  domains_intersection_(split(S,Left0,Right0), D2, Dom) :-
9          domains_intersection_(Left0, D2, Left1),
10         domains_intersection_(Right0, D2, Right1),
11         (   Left1 == empty -> Dom = Right1
12         ;   Right1 == empty -> Dom = Left1
13         ;   Dom = split(S, Left1, Right1)
14         ).
15
16 narrow(empty, _, _, empty).
17 narrow(from_to(L0,U0), From0, To0, Dom) :-
18         From1 cis max(From0,L0), To1 cis min(To0,U0),
19         (   From1 cis_gt To1 -> Dom = empty
20         ;   Dom = from_to(From1,To1)
21         ).
22 narrow(split(S, Left0, Right0), From0, To0, Dom) :-
23         (   To0 cis_lt n(S) -> narrow(Left0, From0, To0, Dom)
24         ;   From0 cis_gt n(S) -> narrow(Right0, From0, To0, Dom)
25         ;   narrow(Left0, From0, To0, Left1),
26             narrow(Right0, From0, To0, Right1),
27             (   Left1 == empty -> Dom = Right1
28             ;   Right1 == empty -> Dom = Left1
29             ;   Dom = split(S, Left1, Right1)
30             )
31         ).
```

Figure 4.12: Intersection of two domains

4.11 Arithmetic constraints

We now give a more formal definition of arithmetic constraints that are implemented in our system. An arithmetic *expression* is one of the Prolog terms described in Table 4.2. The definition is inductive: $expr_1$ and $expr_2$ again denote arithmetic expressions.

expression	meaning
logical variable	unknown integer (see Section 4.3)
`?(Var)`	`Var` is a finite domain variable or integer
integer	given number
`-`*expr*	unary minus
$expr_1$ `+` $expr_2$	addition
$expr_1$ `*` $expr_2$	multiplication
$expr_1$ `-` $expr_2$	subtraction
$expr_1$ `^` $expr_2$	exponentiation
`min`($expr_1$,$expr_2$)	minimum
`max`($expr_1$,$expr_2$)	maximum
$expr_1$ `mod` $expr_2$	modulo induced by floored division
$expr_1$ `rem` $expr_2$	modulo induced by truncated division
`abs`($expr_1$)	absolute value
$expr_1$ `/` $expr_2$	truncated integer division

Table 4.2: Arithmetic expressions

Arithmetic constraints are relations between arithmetic expressions. Table 4.3 lists arithmetic constraints that are available in all systems that we mention in this thesis.

constraint	meaning
$expr_1$ `#>=` $expr_2$	$expr_1$ is greater than or equal to $expr_2$
$expr_1$ `#=<` $expr_2$	$expr_1$ is less than or equal to $expr_2$
$expr_1$ `#=` $expr_2$	$expr_1$ equals $expr_2$
$expr_1$ `#\=` $expr_2$	$expr_1$ is not equal to $expr_2$
$expr_1$ `#>` $expr_2$	$expr_1$ is strictly greater than $expr_2$
$expr_1$ `#<` $expr_2$	$expr_1$ is strictly less than $expr_2$

Table 4.3: Arithmetic constraints

In our system, we generally aim for *bounds consistency* in the implementation of all arithmetic constraints, at least for the most common use cases. For example, *addition* and *multiplication* are always bounds consistent, and *division* is bounds consistent (at least) if the divisor is instantiated.

4.11.1 Parsing arithmetic expressions

Our system provides infix operators so that arithmetic expressions and constraints can be naturally written and processed as Prolog terms. When parsing a compound arithmetic expression, we introduce auxiliary variables for subexpressions. For example, the arithmetic expressions in:

```
X? #= Y? * Z? + D?
```

are decomposed as follows, where `T` is a fresh variable:

```
X? #= T? + D?,
T? #= Y? * Z?
```

This decomposition allows us to limit the implementation of arithmetic propagators to a few elementary cases. Hence, each arithmetic expression shown in Table 4.2 and each arithmetic constraint shown in Table 4.3 corresponds to a propagator that only needs to consider finite domain variables and integers as its arguments, but no compound expressions. The names of these propagators are built by prepending a `p` to the arithmetic operation, for example: `pplus`, `ptimes`, `pmin` etc.

We have devised a DSL for describing this decomposition of arithmetic expressions into elementary cases. The language is a list of *decomposition rules* of the form `M => As`, where `M` is a matcher and `As` is a list of actions that are performed when `M` matches the expression that is being decomposed. At compilation time, Prolog clauses are generated from this DSL.

More formally, a *matcher* `M` is one of the terms `m(P)`, `g(G)` or `?(X)`. `P` is a *pattern* that involves an arithmetic expression and its arguments, and `G` is a Prolog goal. A rule is applicable for a given expression `E` if:

- the matcher is `m(P)` and `E` is syntactically subsumed by `P`. Each variable in `P` stands for the decomposition of the corresponding subexpression of `E` according to the same decomposition rules. This is where the rules are applied recursively to each subexpression.
- the matcher is `g(G)` and `G` succeeds
- the matcher is `?(X)` and `E` has the form `?(X)`. Note that the subexpression of `E` that corresponds to `X` is *not* decomposed but taken exactly as it appears. This is because, as explained in Section 4.3, `?/1` is used to indicate a finite domain variable or integer, and a type error must be raised for other terms.

In a decomposition rule `M => As`, each element in the list `As` is one of the two actions described in Table 4.4. When a rule is applicable, its actions are performed in the order they occur in `As`, and no further rules are tried.

`g(G)`	Call the Prolog goal `G`.
`p(P)`	Post a constraint `P`. This is a shorthand notation for a specific sequence of goals that attach a constraint to all involved variables and trigger it.

Table 4.4: Valid actions in a list `As` of a rule `M => As`

Fig. 4.13 shows the complete declarative description of how arithmetic expressions are decomposed, using our DSL. Given expression `E`, the result is the finite domain variable or integer `R`, using the decomposition rules.

```
1  :- op(800, xfx, =>).
2
3  parse_clpfd(E, R,
4       [g(cyclic_term(E)) => [g(domain_error(clpfd_expression, E))],
5        g(var(E))         => [g(non_monotonic(E)),
6                              g(constrain_to_integer(E)), g(E = R)],
7        ?(E)              => [g(must_be_fd_integer(E)), g(R = E)],
8        g(integer(E))     => [g(R = E)],
9        m(-A)             => [p(ptimes(-1,A,R))],
10       m(A+B)            => [p(pplus(A, B, R))],
11       g(power_var_num(E, V, N)) => [p(pexp(V, N, R))],
12       m(A*B)            => [p(ptimes(A, B, R))],
13       m(A-B)            => [p(pplus(R,B,A))],
14       m(A^B)            => [p(pexp(A, B, R))],
15       m(min(A,B))       => [g(A #>= ?(R)), g(B #>= R), p(pmin(A,B,R))],
16       m(max(A,B))       => [g(A #=< ?(R)), g(B #=< R), p(pmax(A,B,R))],
17       m(A mod B)        => [g(B #\= 0), p(pmod(A, B, R))],
18       m(A rem B)        => [g(B #\= 0), p(prem(A, B, R))],
19       m(abs(A))         => [g(?(R) #>= 0), p(pabs(A, R))],
20       m(A/B)            => [g(B #\= 0), p(pdiv(A, B, R))],
21       g(true)           => [g(domain_error(clpfd_expression, E))]
22      ]).
```

Figure 4.13: Parsing arithmetic expressions

As can be seen in this code, a *domain error* is raised when `E` is a *cyclic term*. The predicate `non_monotonic(X)` raises an *instantiation error* if `X` is an unconstrained variable and the flag `clpfd_monotonic` is set to `true` (see Section 4.3). `constrain_to_integer/1` does what its name implies: It constrains a variable to integers. `must_be_fd_integer(X)` succeeds if its argument is a variable or an integer, and otherwise raises an exception. `power_var_num(E, V, N)` is true when `E` can be written as `V^N`. When none of the previous rules matches, a *domain error* is used to indicate that the given expression is not an arithmetic expression according to Table 4.2.

Parsing arithmetic expressions is comparatively straight-forward, and one may wonder whether implementing a DSL is worth the effort when using plain Prolog is almost equally convenient. Indeed we initially used Prolog for this task. However, only after using our DSL did we for example notice that *subtraction* can be expressed as *addition* (line 13). We found a different CLP(FD) system that needlessly uses one addition and one multiplication. In our experience, such improvements are easier to notice with our DSL.

4.11.2 Selecting propagators for constraints

Some propagators are already chosen when arithmetic expressions are decomposed as explained in the previous section. It now remains to choose suitable propagators for arithmetic *constraints*. First, notice that the arithmetic constraints shown in Table 4.3 can be reduced at least to the following elementary cases: `#>=`, `#=` and `#\=`. The other arithmetic constraints can be expressed with them. For example, the constraint

```
X? #< Y?
```

can be written as:

```
Y? #>= T?,
X? + 1 #= T?
```

One could even further reduce the set of supported elementary constraints. However, this is not advisable for performance reasons: Introducing auxiliary variables and constraints can lead to intermediate propagation steps that may not even be necessary. For example, consider the constraint:

```
X? + Y? #\= Z?
```

According to the decomposition rules shown in the previous section, we can decompose the arithmetic expression on the left-hand side and transform this constraint into:

```
X? + Y? #= T?,
T? #\= Z?
```

The drawback of this decomposition is that significant information has been lost: In the original disequality constraint (`#\=`), it is clear that the propagator can only perform any filtering and therefore only needs to be invoked when either `X + Y` or `Z` are concrete integers. In the shown decomposition, `X` and `Y` are involved in an equality constraint that performs constraint propagation for *addition* every time a domain boundary changes, even if the variables are not yet fully instantiated.

We therefore implement more propagators than are strictly needed, giving special priority to patterns that frequently occur in practical applications. It is then left to detect when such specialized propagators can actually be applied to a given constraint expression. This is the task of selecting suitable propagators for specified constraints.

Manually selecting fitting propagators for given constraints is quite error-prone, and one has to be careful not to accidentally unify variables that occur in expressions with patterns one looks for. To simplify this task and make it less error-prone, we devised a DSL in the form of a simple committed-choice

language that consists of *matching rules*. As in the previous section, each rule is again a term of form `M => As`, where `M` is a matcher and `As` is a list of actions that are performed when `M` matches the constraint that is being posted.

The syntax and meaning of terms are now slightly different: A *matcher* `M` is a term of the form `m_c(P, C)`. `P` is a *pattern* that involves an arithmetic *constraint* and its arguments, and `G` is a Prolog goal. The basic building-blocks of a pattern are explained in Table 4.5. These building-blocks can be nested inside arbitrary symbolic expressions. A rule is applicable for a given constraint if the constraint is subsumed by `P` and its building-blocks, and additionally `G` succeeds. A matcher `m_c(P, true)`, can be more compactly written as `m(P)`.

`any(X)`	Matches any term, unifying `X` with it.
`var(X)`	Matches a variable or integer, unifying `X` with it.
`integer(X)`	Matches an integer, unifying `X` with it.

Table 4.5: Basic building-blocks of a pattern

In a rule `M => As`, each element in the list `As` is one of the actions described in Table 4.6. When a rule is applicable, its actions are performed in the order they occur in `As`, and no further rules are tried.

`g(G)`	Call the Prolog goal `G`.
`d(X, Y)`	Decompose the arithmetic expression `X` according to the decomposition rules of the previous section. The result is `Y`.
`p(P)`	Post a constraint propagator `P`.
`r(X, Y)`	Rematch the rule's constraint, using arguments `X` and `Y`. Equivalent to `g(call(F,X,Y))`, where `F` is the functor of the rule's pattern.

Table 4.6: Valid actions in a list `As` of a rule `M => As`

Figure 4.14 shows some of the matching rules that we use in our constraint system. We omit several actions (indicated by `[...]`), since they are not particularly instructive. It is more interesting to see which patterns of constraints are actually handled specially in our system. Yet, the figure is only an excerpt. For example, in our actual system, nested additions are also detected via our DSL, and handled by a dedicated propagator which includes specialized reasoning for linear equations. `symmetric/1` is used to state which of the constraints are symmetric. For these constraints, the patterns are also applied symmetrically.

At compilation time, actual Prolog code is again generated from this DSL. To detect whether a pattern can be applied, the generated clauses frequently use a sequence like `nonvar(T), T = ...` to avoid accidentally unifying a

variable with a term we want to detect. Such mistakes are easy to make in practice when writing matching code manually, and are then often made in only some of several clauses. By using a DSL to generate matching code mechanically in all cases, such errors are much easier to detect, because they likely affect several or even all clauses. Once corrected, all of them are correct at once.

Such a declarative description has also other advantages: First, it allows automated subsumption checks to detect whether specialized propagators are accidentally overshadowed by other rules. This is also a mistake that we found easy to make and hard to detect when manually selecting propagators. Second, when DSLs similar to the one we propose here are also used in other constraint systems, it is easier to compare supported specialized propagators, and to support common ones more uniformly across systems. Third, improvements to the expansion phase of the DSL benefits potentially many propagators at once.

We found that the languages features we introduced above for matchers and actions allow us to select a large variety of intended specialized propagators in practice, and believe that other constraint systems may benefit from this or similar syntax as well. To the best of our knowledge, our system is the first CLP(FD) solver that uses such a DSL to select propagators for constraints.

```
1  symmetric(#=).
2  symmetric(#\=).
3
4  matches([
5           m(any(X) - any(Y) #>= integer(C))     => [...],
6           m(integer(X) #>= any(Z) + integer(A)) => [...],
7           m(abs(any(X)-any(Y)) #>= integer(I))  => [...],
8           m(abs(any(X)) #>= integer(I))         => [...],
9           m(integer(I) #>= abs(any(X)))         => [...],
10          m(any(X) #>= any(Y))                  =>
11              [d(X, RX), d(Y, RY), g(geq(RX, RY))],
12
13          m(var(X) #= var(Y))        => [...],
14          m(var(X) #= var(Y)+var(Z)) => [p(pplus(Y,Z,X))],
15          m(var(X) #= var(Y)-var(Z)) => [p(pplus(X,Z,Y))],
16          m(var(X) #= var(Y)*var(Z)) => [p(ptimes(Y,Z,X))],
17          m(var(X) #= -var(Z))       => [p(ptimes(-1, Z, X))],
18          m(var(X) #= any(Y))        => [d(Y,X)],
19          m(any(X) #= any(Y))        => [d(X, RX), d(Y, RX)],
20
21          m(var(X) #\= integer(Y))             => [...],
22          m(var(X) #\= var(Y))                 => [...],
23          m(var(X) #\= var(Y) + var(Z))        => [...],
24          m(var(X) #\= var(Y) - var(Z))        => [...],
25          m(var(X) #\= var(Y)*var(Z))          => [...],
26          m(integer(X) #\= abs(any(Y)-any(Z))) => [...],
27          m(any(X) #\= any(Y) + any(Z))        => [...],
28          m(any(X) #\= any(Y) - any(Z))        => [...],
29          m(any(X) #\= any(Y))                 =>
30              [d(X, RX), d(Y, RY), g(neq(RX, RY))]
31         ]).
```

Figure 4.14: Selecting propagators for arithmetic constraints (excerpt)

4.11.3 Limitations of indexicals

It is sometimes desirable to dynamically take the effects of variable aliasing into account in propagators. However, this cannot be easily expressed with indexicals. As a consequence, we have for example in GNU Prolog:

```
| ?- X + X #= 4.

X = 2

yes
```

which also works when different but aliased variables are used:

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

X = 2
Y = 2

yes
```

When we exchange the goals, we obtain a different result:

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

X = _#22(0..4)
Y = _#41(0..4)

yes
```

In SICStus Prolog, we obtain:

```
 | ?- X + X #= 4.
 X = 2 ?
```

However, even when two different variables are aliased before the constraint is posted, we obtain a different result:

```
| ?- X = Y, X + Y #= 4.
Y = X,
X in inf..sup ?
```

In order to avoid such discrepancies in our system's answers, we have chosen not to use indexicals in our implementation. Instead, we implement all propagators in plain Prolog, which gives us the necessary constructs to

handle variable aliasing and similar special cases that cannot be expressed with indexicals.

Notice also from Fig. 3.1 that indexicals become increasingly more complicated for nonlinear constraints like `xy_eq_z`. Moreover, in the case of GNU Prolog, reasoning only needs to be applied to non-negative integers. As we will see in the definition of multiplication, the boundary expressions are much more complicated when negative integers can arise as well. The corresponding indexical will also increase in complexity.

4.11.4 Important internal predicates

In the following sections, some predicates are used which are internally defined in our solver. Instead of providing the Prolog specification of these predicates, we describe them informally in Table 4.7. The implementation of these predicates is reasonably straight-forward, and interested readers can study their definitions in the actual source code of our system.

goal	meaning
`do_queue`	As mentioned in Section 4.4, `do_queue` triggers all queued propagators, until fixpoint. This predicate is typically not invoked directly in propagators. It is called for example in `attr_unify_hook/2`, an internal predicate that is called after unifying attributed variables.
`fd_get(X, XD, XPs)`	Equivalent to `fd_get(X,_,_,XD,XPs)`.
`fd_get(X,XL,XU,XD,XPs)`	True if `X` is a finite domain variable with domain `XD`, lower boundary `XL`, upper boundary `XU`, and `Ps` are the propagators that need to be invoked when the domain of `X` changes. Fails if `X` is an integer.
`fd_put(X, XD, Ps)`	Assigns the domain `XD` and propagators `Ps` to the finite domain variable `X`. Raises an error when `X` is an integer. When `XD` differs from the previous domain of `X` and constraint propagation is admissible (see Section 4.6), it adds the propagators that need to be invoked to the propagation queues (see Section 4.4). When `XD` consists of a single element `I`, it posts `X = I`.
`kill(State)`	Prevents further invocations of the propagator whose state is `State`.

Table 4.7: Important internal predicates

4.11.5 Addition

We now present the complete source code of the propagator that implements *addition* of arithmetic expressions in our system. Other arithmetic constraints are implemented analogously. Addition is a simple and frequently used constraint that has a comparatively straight-forward implementation because propagation proceeds in similar ways for each of the arguments.

Like all arithmetic propagators, addition only needs to handle the case of variables or integers as its arguments, since nested expressions are decomposed to elementary cases by applying our DSL. Moreover, when arithmetic propagators are triggered, these variables are already constrained to integers. Therefore, the `?/1` syntax need not be used when these variables occur in constraints that are posted within arithmetic propagators.

As explained in Section 4.4, the propagator for addition is implemented as a clause of `run_propagator/2`. Since addition is a relation between three finite domain expressions, it is represented by a propagator with three arguments: `pplus(X, Y, Z)`, which means that the sum of `X` and `Y` is `Z`. This term is inserted into the list of propagators that each of these variables is involved in. `pplus/3` is a propagator that implements bounds consistency and is only invoked when a domain boundary of any of the involved variables changes. `State` is a variable that is shared among all three arguments, to avoid redundant invocations of the same propagator, and also to deactivate the propagator for all variables at once when it is no longer needed.

Fig. 4.15 shows the complete source code of the propagator that implements addition in our system. We have opted for a monolithic implementation that performs filtering for all three variables in the same clause. It is possible to split the filtering logic into several distinct cases that may depend for example on whether the lower or upper domain boundaries of any variable have changed. However, this would require us to manage more propagators and to consider all possible interactions between them to ensure that propagation still works as intended in all cases. We have therefore chosen not to do this. It would be interesting to derive these distinct cases automatically from a more declarative description.

The code applies filtering as far as it can, depending on which of the arguments are already instantiated. For example, if `nonvar(X)` succeeds because `X` is instantiated, and `X` is `0`, then the propagator is deactivated and `Y = Z` is posted. If both `X` and `Y` are instantiated, the propagator is deactivated and `Z` is unified with the arithmetic result of `X+Y`. The propagator is deactivated and further filtering is delegated to *multiplication* when `X` and `Y` are aliased.

When `fd_put/3` is called, further constraints may be triggered, and we have to be careful not to accidentally invoke `fd_put/3` when its first argument is instantiated. We will say more about these properties in Section 5.6.

```
run_propagator(pplus(X,Y,Z), MState) :-
        (   nonvar(X) ->
            (   X =:= 0 -> kill(MState), Y = Z
            ;   Y == Z -> kill(MState), X =:= 0
            ;   nonvar(Y) -> kill(MState), Z is X + Y
            ;   nonvar(Z) -> kill(MState), Y is Z - X
            ;   fd_get(Z, ZD, ZPs),
                fd_get(Y, YD, _),
                domain_shift(YD, X, Shifted_YD),
                domains_intersection(ZD, Shifted_YD, ZD1),
                fd_put(Z, ZD1, ZPs),
                (   fd_get(Y, YD1, YPs) ->
                    O is -X,
                    domain_shift(ZD1, O, YD2),
                    domains_intersection(YD1, YD2, YD3),
                    fd_put(Y, YD3, YPs)
                ;   true
                )
            )
        ;   nonvar(Y) -> run_propagator(pplus(Y,X,Z), MState)
        ;   nonvar(Z) ->
            (   X == Y -> kill(MState), even(Z), X is Z // 2
            ;   fd_get(X, XD, _),
                fd_get(Y, YD, YPs),
                domain_negate(XD, XDN),
                domain_shift(XDN, Z, YD1),
                domains_intersection(YD, YD1, YD2),
                fd_put(Y, YD2, YPs),
                (   fd_get(X, XD1, XPs) ->
                    domain_negate(YD2, YD2N),
                    domain_shift(YD2N, Z, XD2),
                    domains_intersection(XD1, XD2, XD3),
                    fd_put(X, XD3, XPs)
                ;   true
                )
            )
        ;   (   X == Y -> kill(MState), 2*X #= Z
            ;   X == Z -> kill(MState), Y = 0
            ;   Y == Z -> kill(MState), X = 0
            ;   fd_get(X, XD, XL, XU, XPs),
                fd_get(Y, _, YL, YU, _),
                fd_get(Z, _, ZL, ZU, _),
                NXL cis max(XL, ZL-YU),
                NXU cis min(XU, ZU-YL),
                update_bounds(X, XD, XPs, XL, XU, NXL, NXU),
                (   fd_get(Y, YD2, YL2, YU2, YPs2) ->
                    NYL cis max(YL2, ZL-NXU),
                    NYU cis min(YU2, ZU-NXL),
                    update_bounds(Y, YD2, YPs2, YL2, YU2, NYL, NYU)
                ;   NYL = n(Y), NYU = n(Y)
                ),
                (   fd_get(Z, ZD2, ZL2, ZU2, ZPs2) ->
                    NZL cis max(ZL2,NXL+NYL),
                    NZU cis min(ZU2,NXU+NYU),
                    update_bounds(Z, ZD2, ZPs2, ZL2, ZU2, NZL, NZU)
                ;   true
                )
            )
        ).

update_bounds(X, XD, XPs, XL, XU, NXL, NXU) :-
        (   NXL == XL, NXU == XU -> true
        ;   domains_intersection(XD, from_to(NXL, NXU), NXD),
            fd_put(X, NXD, XPs)
        ).
```

Figure 4.15: Propagator for addition, where `X + Y #= Z`

4.11.6 Multiplication

As the second example for an arithmetic propagator, we present the entire source code for *multiplication* in our system. The source is similar to *addition* in that it consists of a case distinction that applies filtering for all variables in a single clause, shown in Fig. 4.16. However, the code is significantly more involved and requires several auxiliary predicates which we also present.

Ensuring bounds consistency for *multiplication* requires us to consider more cases than one may initially expect. While computing the domain boundaries of the *product* of a multiplication is very simple (see Fig. 4.17), it is much harder to correctly adjust the boundaries of each of the two *factors*.

Several rules for propagating multiplication are given in [AZ07]. As is well-noted in the paper, these rules *fail* to enforce bounds consistency for the constraint $x \cdot y = z$ when $D(x)$ and $D(y)$ are both of the form $[l..h]$ with $l < 0$ and $h > 0$ while z can assume either only positive numbers, or only negative numbers. The solution that is proposed in the paper, and which we also employ in our propagator, is to temporarily split the domains in a positive interval and a negative interval. Bounds consistency is then achieved by applying the rules to the resulting subproblems, and updating the domain of each variable with the union of the domains that are computed in these subproblems. This decomposition is performed in the predicate `min_max_factor/8`, shown in Fig. 4.18. For the constraint $x \cdot y = z$, `min_max_factor/8` is given, in the following order: $l(z)$, $u(z)$, $l(x)$, $u(x)$, $l(y)$, $u(y)$ and computes the new values of $l(y)$ and $u(y)$ in its last two arguments, where l and u denote the lower and upper boundary of its argument's domain, respectively.

An example where the decomposition into subproblems is performed internally:

```
?- X in -2..1, Y in -2..1, X*Y #= 2.
X in -2.. -1,
X? * Y? #= 2,
Y in -2.. -1.
```

For all other cases, the decomposition into subproblems is not used. Instead, the predicates `min_factor/5` and `max_factor/5`, both shown in Fig. 4.19, are used to compute the new $l(y)$ and $u(y)$ in a comparatively straight-forward way. Note though that these predicates still contain quite elaborate and error-prone conditions that distinguish the different cases. In [SS05], similar rules are defined that do not require a decomposition into subproblems. However, these rules cannot be used directly in our solver because they may lead to float overflows when dividing large arbitrary precision integers. We therefore show our own implementation, which also ensures bounds consistency, in directly executable form in these figures.

```
run_propagator(ptimes(X,Y,Z), MState) :-
    (   nonvar(X) ->
        (   nonvar(Y) -> kill(MState), Z is X * Y
        ;   X =:= 0 -> kill(MState), Z = 0
        ;   X =:= 1 -> kill(MState), Z = Y
        ;   nonvar(Z) -> kill(MState), 0 =:= Z mod X, Y is Z // X
        ;   (   Y == Z -> kill(MState), Y = 0
            ;   fd_get(Y, YD, _),
                fd_get(Z, ZD, ZPs),
                domain_expand(YD, X, Scaled_YD),
                domains_intersection(ZD, Scaled_YD, ZD1),
                fd_put(Z, ZD1, ZPs),
                (   fd_get(Y, YDom2, YPs2) ->
                    domain_contract(ZD1, X, Contract),
                    domains_intersection(YDom2, Contract, NYDom),
                    fd_put(Y, NYDom, YPs2)
                ;   kill(MState), Z is X * Y
                )
            )
        )
    ;   nonvar(Y) -> run_propagator(ptimes(Y,X,Z), MState)
    ;   nonvar(Z) ->
        (   X == Y ->
            kill(MState),
            integer_kth_root(Z, 2, R),
            NR is -R,
            X in NR \/ R
        ;   fd_get(X, XD, XL, XU, XPs),
            fd_get(Y, _, YL, YU, _),
            min_max_factor(n(Z), n(Z), YL, YU, XL, XU, NXL, NXU),
            update_bounds(X, XD, XPs, XL, XU, NXL, NXU),
            (   fd_get(Y, YD2, YL2, YU2, YPs2) ->
                min_max_factor(n(Z),n(Z),NXL,NXU,YL2,YU2,NYL,NYU),
                update_bounds(Y, YD2, YPs2, YL2, YU2, NYL, NYU)
            ;   (   Y =\= 0 ->
                    0 =:= Z mod Y, kill(MState), X is Z // Y
                ;   kill(MState), Z = 0
                )
            )
        ),
        (   Z =\= 0 -> neq_num(X, 0), neq_num(Y, 0)
        ;   true
        )
    ;   (   X == Y -> kill(MState), X^2 #= Z
        ;   fd_get(X, XD, XL, XU, XPs),
            fd_get(Y, _, YL, YU, _),
            fd_get(Z, _, ZL, ZU, _),
            min_max_factor(ZL, ZU, YL, YU, XL, XU, NXL, NXU),
            update_bounds(X, XD, XPs, XL, XU, NXL, NXU),
            (   fd_get(Y, YD2, YL2, YU2, YPs2) ->
                min_max_factor(ZL, ZU, NXL, NXU, YL2, YU2, NYL, NYU),
                update_bounds(Y, YD2, YPs2, YL2, YU2, NYL, NYU)
            ;   NYL = n(Y), NYU = n(Y)
            ),
            (   fd_get(Z, ZD2, ZL2, ZU2, ZPs2) ->
                min_product(NXL, NXU, NYL, NYU, NZL),
                max_product(NXL, NXU, NYL, NYU, NZU),
                (   NZL cis_leq ZL2, NZU cis_geq ZU2 -> ZD3 = ZD2
                ;   domains_intersection(ZD2, from_to(NZL,NZU), ZD3),
                    fd_put(Z, ZD3, ZPs2)
                ),
                (   domain_contains(ZD3, 0) ->  true
                ;   neq_num(X, 0), neq_num(Y, 0)
                )
            ;   true
            )
        )
    ).
```

Figure 4.16: Propagator for multiplication, where `X*Y #= Z`

```
min_product(L1, U1, L2, U2, Min) :-
        Min cis min(min(L1*L2,L1*U2),min(U1*L2,U1*U2)).

max_product(L1, U1, L2, U2, Max) :-
        Max cis max(max(L1*L2,L1*U2),max(U1*L2,U1*U2)).
```

Figure 4.17: Domain boundary calculations for a multiplication's product

```
min_max_factor(L1, U1, L2, U2, L3, U3, Min, Max) :-
        (   U1 cis_lt n(0),
            L2 cis_lt n(0), U2 cis_gt n(0),
            L3 cis_lt n(0), U3 cis_gt n(0) ->
            maplist(in_(L1,U1), [Z1,Z2]),
            in_(L2, n(-1), X1), in_(n(1), U3, Y1),
            (   X1*Y1 #= Z1 ->
                (   fd_get(Y1, _, Inf1, Sup1, _) -> true
                ;   Inf1 = n(Y1), Sup1 = n(Y1)
                )
            ;   Inf1 = inf, Sup1 = n(-1)
            ),
            in_(n(1), U2, X2), in_(L3, n(-1), Y2),
            (   X2*Y2 #= Z2 ->
                (   fd_get(Y2, _, Inf2, Sup2, _) -> true
                ;   Inf2 = n(Y2), Sup2 = n(Y2)
                )
            ;   Inf2 = n(1), Sup2 = sup
            ),
            Min cis max(min(Inf1,Inf2), L3),
            Max cis min(max(Sup1,Sup2), U3)
        ;   L1 cis_gt n(0),
            L2 cis_lt n(0), U2 cis_gt n(0),
            L3 cis_lt n(0), U3 cis_gt n(0) ->
            maplist(in_(L1,U1), [Z1,Z2]),
            in_(L2, n(-1), X1), in_(L3, n(-1), Y1),
            (   X1*Y1 #= Z1 ->
                (   fd_get(Y1, _, Inf1, Sup1, _) -> true
                ;   Inf1 = n(Y1), Sup1 = n(Y1)
                )
            ;   Inf1 = n(1), Sup1 = sup
            ),
            in_(n(1), U2, X2), in_(n(1), U3, Y2),
            (   X2*Y2 #= Z2 ->
                (   fd_get(Y2, _, Inf2, Sup2, _) -> true
                ;   Inf2 = n(Y2), Sup2 = n(Y2)
                )
            ;   Inf2 = inf, Sup2 = n(-1)
            ),
            Min cis max(min(Inf1,Inf2), L3),
            Max cis min(max(Sup1,Sup2), U3)
        ;   min_factor(L1, U1, L2, U2, Min0),
            Min cis max(L3,Min0),
            max_factor(L1, U1, L2, U2, Max0),
            Max cis min(U3,Max0)
        ).

in_(L, U, X) :-
        fd_get(X, XD, XPs),
        domains_intersection(XD, from_to(L,U), NXD),
        fd_put(X, NXD, XPs).
```

Figure 4.18: Decomposing a multiplication into subproblems when necessary

```
1  min_factor(L1, U1, L2, U2, Min) :-
2          (   L1 cis_geq n(0), L2 cis_gt n(0), finite(U2) ->
3              Min cis div(L1+U2-n(1),U2)
4          ;   L1 cis_gt n(0), U2 cis_lt n(0) -> Min cis div(U1,U2)
5          ;   L1 cis_gt n(0), L2 cis_geq n(0) -> Min = n(1)
6          ;   L1 cis_gt n(0) -> Min cis -U1
7          ;   U1 cis_lt n(0), U2 cis_leq n(0) ->
8              (   finite(L2) -> Min cis div(U1+L2+n(1),L2)
9              ;   Min = n(1)
10             )
11         ;   U1 cis_lt n(0), L2 cis_geq n(0) -> Min cis div(L1,L2)
12         ;   U1 cis_lt n(0) -> Min = L1
13         ;   L2 cis_leq n(0), U2 cis_geq n(0) -> Min = inf
14         ;   Min cis min(min(div(L1,L2),div(L1,U2)),
15                         min(div(U1,L2),div(U1,U2)))
16         ).
17
18 max_factor(L1, U1, L2, U2, Max) :-
19         (   L1 cis_geq n(0), L2 cis_geq n(0) -> Max cis div(U1,L2)
20         ;   L1 cis_gt n(0), U2 cis_leq n(0) ->
21             (   finite(L2) -> Max cis div(L1-L2-n(1),L2)
22             ;   Max = n(-1)
23             )
24         ;   L1 cis_gt n(0) -> Max = U1
25         ;   U1 cis_lt n(0), U2 cis_lt n(0) -> Max cis div(L1,U2)
26         ;   U1 cis_lt n(0), L2 cis_geq n(0) ->
27             (   finite(U2) -> Max cis div(U1-U2+n(1),U2)
28             ;   Max = n(-1)
29             )
30         ;   U1 cis_lt n(0) -> Max cis -L1
31         ;   L2 cis_leq n(0), U2 cis_geq n(0) -> Max = sup
32         ;   Max cis max(max(div(L1,L2),div(L1,U2)),
33                         max(div(U1,L2),div(U1,U2)))
34         ).
35
36 finite(n(_)).
```

Figure 4.19: Domain boundary calculations for factors of a multiplication

4.12 Reification

All arithmetic constraints that are shown in Table 4.3, as well as the constraint `in/2`, can be *reified*, which means reflecting their truth values into Boolean values represented by the integers 0 and 1 denoting *false* and *true*, respectively. If P and Q denote reifiable constraints or Boolean variables, then they can be subjected to the constraints shown in Table 4.8, which are themselves also reifiable.

constraint	meaning
`#\` P	true iff P is false
P `#\/` Q	true iff either P or Q
P `#/\` Q	true iff both P and Q
P `#==>` Q	true iff P implies Q
P `#<==` Q	true iff Q implies P
P `#<==>` Q	true iff P and Q are equivalent

Table 4.8: Constraint reification

When implementing constraint reification, it is tempting to proceed as follows: For concreteness, consider reified equality (`#=/2`) of two arithmetic expressions E_1 and E_2. We could introduce two finite domain variables, say $\texttt{T}_1$ and $\texttt{T}_2$, and post the constraints $\texttt{T}_1$ `#=` E_1 and $\texttt{T}_2$ `#=` E_2, thus using the constraint solver itself to decompose the (possibly compound) expressions E_1 and E_2, and reducing reified equality of two *expressions* to equality of two finite domain *variables* (or integers), which is easier to implement. Unfortunately, this strategy yields wrong results in general. Consider for example the constraint:

```
(X? / 0 #= Y? / 0) #<==> B?
```

It is clear that the relation `X? / 0 #= Y? / 0` cannot hold, since a divisor can never be 0. A valid, declaratively equivalent answer to the above constraint is thus for example (note that `X` and `Y` must be constrained to integers for the relation to hold):

```
B = 0, X in inf..sup, Y in inf..sup
```

However, if we decompose the equality `X? / 0 #= Y? / 0` into two auxiliary constraints $\texttt{T}_1$ `#= X? / 0` and $\texttt{T}_2$ `#= Y? / 0` and post them, then (with strong enough propagation of division) both auxiliary constraints fail, and thus the whole query *incorrectly* fails. While devising a DSL for reification, we found one commercial Prolog system and one freely available system that indeed incorrectly failed in this case. After we reported the issue, the problem was immediately fixed.

To reflect the intended relational semantics, it is thus necessary to implement *definedness* correctly when reifying constraints. See also [FS09], where our constraint system, in contrast to others that were tested, correctly handles all reification test cases, which we attribute in part to the DSL presented in this section. Once any subexpression of a relation becomes undefined, the relation cannot hold and its associated truth value must be 0. Undefinedness can occur when $Y = 0$ in the expressions X/Y, X mod Y, and X rem Y. Parsing an arithmetic expression that occurs as an argument of a constraint that is being reified is thus at least a ternary relation, involving the expression itself, its arithmetic result, and its Boolean definedness.

There is a fourth desirable component in addition to those just mentioned: It is useful to keep track of *auxiliary variables* that are introduced when decomposing subexpressions of a constraint that is being reified. The reason for this is that the truth value of a reified constraint may turn out to be irrelevant, for instance the implication `0 #==>` C holds for both possible truth values of the constraint C, thus auxiliary variables that were introduced to hold the results of subexpressions while parsing C can be eliminated. However, we need to be careful: A constraint propagator may *alias* user-specified variables with auxiliary variables. For example, in `abs(X?) #= T?, X #>= 0`, a constraint system may deduce `X = T`. Thus, if `T` was previously introduced as an auxiliary variable, and `X` was user-specified, then `X` must still retain its status as a constrained variable.

These considerations motivate the following DSL for parsing arithmetic expressions in reified constraints, which we believe can be useful in other constraint systems as well: A *reification rule* is a term `M => As`. A *matcher* `M` has the same syntax and semantics as a matcher in decomposition rules (see Section 4.11.1). The list `As` consists of actions that are described in Table 4.9. The predicate `parse_reified/4`, shown in Figure 4.20, contains our full declarative specification for parsing arithmetic expressions in reified constraints, relating an arithmetic expression `E` to its result `R`, Boolean definedness `D`, and auxiliary variables according to the given rules.

The rules are applied in the order specified, committing to the first rule whose head matches. This specification is translated to Prolog code at compile-time and used in other predicates.

The deletion of auxiliary variables and constraints when they are no longer necessary is useful when introducing constraint programming to beginners, and often also for efficiency reasons. As an example, consider the query and its result:

```
?- X? #= 3 #\/ Y? #= 4 #<==> B?, Y = 4.
Y = 4,
B = 1,
X in inf..sup.
```

`g(G)`	Call the Prolog goal `G`.
`d(D)`	`D` is 1 if and only if all subexpressions of `E` are defined.
`p(P)`	Add the constraint propagator `P` to the constraint store.
`a(A)`	`A` is an auxiliary variable that was introduced while parsing the given compound expression `E`.
`a(X,A)`	`A` is an auxiliary variable, unless `A==X`.
`a(X,Y,A)`	`A` is an auxiliary variable, unless `A==X` or `A==Y`.
`skeleton(X,Y,D,Z,P)`	A "skeleton" propagator is posted. When `Y` cannot become 0 any more, it adds the propagator `P(X,Y,Z)` to the constraint store and binds `D=1`. When `Y` is 0, it binds `D=0`. When `D=1` (i.e., the constraint must hold), it posts `Y#\=0`.

Table 4.9: Valid actions in a list `As` of a reification rule `M => As`

Other constraint systems, such as SICStus, still retain and show an additional arithmetic constraint on the variable `X` in the case above although it is no longer semantically relevant. Removal of irrelevant constraints can also significantly improve performance on some benchmarks. As an example of a benchmark that uses reification extensively, we took a solution to the so-called "Nonogram"-puzzle that was generously posted to `comp.lang.prolog` by Bart Demoen on Jan. 22nd 2009. By dynamically removing constraints that are no longer semantically relevant, both run-time and inference count decrease by more than 30% in this case.

The code that is generated from this DSL is a DCG nonterminal called `parse_reified_clpfd(E, R, D)`. It relates an arithmetic expression `E` to its result `R` and its Boolean definedness `D`, and describes a list of auxiliary variables and propagators that are introduced when decomposing `E`. The propagators that implement reified constraints (like `reified_and` and `reified_or`, whose implementation we omit) use this list to disable auxiliary entities when they are no longer relevant. Fig. 4.21 shows how the nonterminal is used during constraint reification. `reify(E, B, Ps)` relates a reifiable expression `E` to its Boolean truth value `B` and a list of auxiliary propagators and variables `Ps` that were introduced during the decomposition of `E` and its arithmetic expressions.

To the best of our knowledge, our constraint system is the first one to describe the full declarative semantics of reification in such brevity. While indexicals can be used to describe reification of individual atomic constraints, they cannot express when auxiliary constraints and variables that were introduced when decomposing nested expressions are no longer needed, in contrast to the DSL we propose in this section.

```
parse_reified(E, R, D,
     [g(cyclic_term(E)) => [g(domain_error(clpfd_expression, E))],
      g(var(E))     => [g(non_monotonic(E)),
                        g(constrain_to_integer(E)), g(R = E), g(D=1)],
      g(integer(E)) => [g(R=E), g(D=1)],
      ?(E)          => [g(must_be_fd_integer(E)), g(R=E), g(D=1)],
      m(A+B)        => [d(D), p(pplus(A,B,R)), a(A,B,R)],
      m(A*B)        => [d(D), p(ptimes(A,B,R)), a(A,B,R)],
      m(A-B)        => [d(D), p(pplus(R,B,A)), a(A,B,R)],
      m(-A)         => [d(D), p(ptimes(-1,A,R)), a(R)],
      m(max(A,B))   => [d(D), p(pgeq(R, A)), p(pgeq(R, B)),
                        p(pmax(A,B,R)), a(A,B,R)],
      m(min(A,B))   => [d(D), p(pgeq(A, R)), p(pgeq(B, R)),
                        p(pmin(A,B,R)), a(A,B,R)],
      m(abs(A))     => [g(?(R)#>=0), d(D), p(pabs(A, R)), a(A,R)],
      m(A/B)        => [skeleton(A,B,D,R,pdiv)],
      m(A mod B)    => [skeleton(A,B,D,R,pmod)],
      m(A rem B)    => [skeleton(A,B,D,R,prem)],
      m(A^B)        => [d(D), p(pexp(A,B,R)), a(A,B,R)],
      g(true)       => [g(domain_error(clpfd_expression, E))]]
    ).
```

Figure 4.20: Parsing arithmetic expressions in reified constraints

```
reify(E, B) :- reify(E, B, _).

reify(Expr, B, Ps) :-
        (   acyclic_term(Expr), reifiable(Expr) ->
            phrase(reify(Expr, B), Ps)
        ;   domain_error(clpfd_reifiable_expression, Expr)
        ).

reify(E, B) --> { B in 0..1 }, reify_(E, B).

reify_(E, B) --> { var(E), !, E = B }.
reify_(E, B) --> { integer(E), E = B }.
reify_(?(B), B) --> [].
reify_(V in Drep, B) -->
        { drep_to_domain(Drep, Dom) },
        propagator_init_trigger(reified_in(V,Dom,B)),
        a(B).
reify_(#\ Q, B) -->
        reify(Q, QR),
        propagator_init_trigger(reified_not(QR,B)),
        a(B).
reify_(L #>= R, B)   --> arithmetic(L, R, B, reified_geq).
reify_(L #= R, B)    --> arithmetic(L, R, B, reified_eq).
reify_(L #\= R, B)   --> arithmetic(L, R, B, reified_neq).
reify_(L #> R, B)    --> reify_(L #>= (R+1), B).
reify_(L #=< R, B)   --> reify_(R #>= L, B).
reify_(L #< R, B)    --> reify_(R #>= (L+1), B).
reify_(L #==> R, B)  --> reify_((#\ L) #\/ R, B).
reify_(L #<== R, B)  --> reify_(R #==> L, B).
reify_(L #<==> R, B) --> reify_((L #==> R) #/\ (R #==> L), B).
reify_(L #/\ R, B)   --> boolean(L, R, B, reified_and).
reify_(L #\/ R, B)   --> boolean(L, R, B, reified_or).

arithmetic(L, R, B, Functor) -->
        { phrase((parse_reified_clpfd(L, LR, LD),
                  parse_reified_clpfd(R, RR, RD)), Ps),
          Prop =.. [Functor,LD,LR,RD,RR,Ps,B] },
        list(Ps),
        propagator_init_trigger([LD,LR,RD,RR,B], Prop),
        a(B).

boolean(L, R, B, Functor) -->
        { reify(L, LR, Ps1), reify(R, RR, Ps2),
          Prop =.. [Functor,LR,Ps1,RR,Ps2,B] },
        list(Ps1), list(Ps2),
        propagator_init_trigger([LR,RR,B], Prop),
        a(LR, RR, B).

a(X,Y,B) -->
        (   { nonvar(X) } -> a(Y, B)
        ;   { nonvar(Y) } -> a(X, B)
        ;   [a(X,Y,B)]
        ).

a(X, B) -->
        (   { var(X) } -> [a(X, B)]
        ;   a(B)
        ).

a(B) -->
        (   { var(B) } -> [a(B)]
        ;   []
        ).

list([])     --> [].
list([L|Ls]) --> [L], list(Ls).
```

Figure 4.21: Implementation of constraint reification. The nonterminal `propagator_init_trigger(Vs, P)` attaches the propagator `P` to each variable in `Vs` and triggers it. If `Vs` is omitted, all variables of `P` are used.

4.13 Global constraints

Constraints that define relations between many variables at once are often called *global constraints*. There are at least two fundamental techniques for implementing global constraints in a CLP(FD) system:

(a) We can *decompose* a global constraint into more elementary constraints and post them individually.

(b) We can implement a dedicated propagator that applies global reasoning on all involved variables.

In this section, we present two important global constraints that illustrate both techniques:

- `cumulative/2` describes a set of tasks with specific properties. It is implemented by expressing the constraint with a conjunction of simpler constraints.
- `all_distinct/1` describes a list of finite domain variables that are pairwise distinct. It is implemented with a dedicated filtering algorithm that is much stronger than pairwise disequalities.

Further global constraints that are available in our system include:

- `automaton/8` is true if a list of finite domain variables is accepted by a finite automaton that can also be used for counting transitions. We show the implementation of a special case of this constraint in Section 6.6.
- `circuit/1` is true if a list `Vs` of finite domain variables induces a *Hamiltonian circuit*. The k-th element of `Vs` denotes the successor of node k.
- `element/3` is similar to `nth1/3` with the difference that it performs deterministic filtering instead of backtracking.
- `global_cardinality/2` relates a list of finite domain variables to the number of occurrences of specific elements given as `Key-Number` pairs.
- `lex_chain/1` is true if a list of lists of finite domain variables is lexicographically non-decreasing.
- `tuples_in/2` is true if each tuple is an element of a given relation, like the `fd_relation/2` constraint of GNU Prolog that we used in Section 3.3.3.

These constraints are all implemented in one of the two ways described above, or with a combination of both methods.

Decomposing a constraint into more elementary constraints is attractive due to its simplicity. However, it typically incurs significant overhead: It means that more constraints need to be managed at run-time, and global reasoning cannot be applied because some information is usually lost during the decomposition.

On the other hand, writing dedicated propagators for global constraints involves more implementation work, and the risk of introducing mistakes is higher with complex propagators.

4.13.1 `cumulative/2`

The `cumulative/2` constraint is used as follows:

```
cumulative(Tasks, Options)
```

`Tasks` is a list of tasks of the form `task(`$\mathtt{S_i}$`,`$\mathtt{D_i}$`,`$\mathtt{E_i}$`,`$\mathtt{C_i}$`,`$\mathtt{T_i}$`)`. $\mathtt{S_i}$ denotes the start time, $\mathtt{D_i}$ the positive duration, $\mathtt{E_i}$ the end time, $\mathtt{C_i}$ the non-negative resource consumption, and $\mathtt{T_i}$ the task identifier of task i. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds if at any time during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit, which is 1 by default.

`Options` is a list of options. Currently, the only supported option is:

- `limit(L)`
  The integer `L` is the global resource limit.

Several strong filtering algorithms for `cumulative/2` appear in the literature ([MH08], [Vil09], [SW10]). At the time of this writing, we have not yet implemented any of these variants. Instead, we express the constraint as a conjunction of more elementary constraints. This formulation was kindly suggested to us by Neng-Fa Zhou, the author of B-Prolog. The main idea is to introduce a Boolean variable $\mathtt{B_{ij}}$ for each task i and time-slot j, which is 1 iff task i is active during time j. The constraint $\sum_i \mathtt{B_{ij}} \cdot \mathtt{C_i} \leq \mathtt{L}$ must hold for each time-slot j, where i ranges over all tasks.

Fig. 4.22 shows the complete definition of `cumulative/2`. Constraint reification is used in line 37 to establish the connection between a task's activity at time `T` and a Boolean variable `B`. A task's contribution to the cumulative resource consumption at time `T` is determined by `contribution_at/4` and depends on $\mathtt{B_{ij}}$ and the task's resource consumption $\mathtt{C_i}$. Line 20 relates the amount of consumed resources at time `T0` to the global resource limit `L`.

Fig. 4.23 shows an example of a schedule for 8 tasks of various durations, each consuming one unit of a resource that is limited to 3. There is no way to schedule these tasks in less than 14 time-slots under this constraint.

```
cumulative(Tasks, Options) :-
        must_be(list(list), [Tasks,Options]),
        (   memberchk(limit(L), Options) -> must_be(integer, L)
        ;   L = 1
        ),
        (   Tasks = [] -> true
        ;   maplist(task_bs, Tasks, Bss),
            maplist(arg(1), Tasks, Starts),
            maplist(fd_inf, Starts, MinStarts),
            maplist(arg(3), Tasks, Ends),
            maplist(fd_sup, Ends, MaxEnds),
            min_list(MinStarts, Start),
            max_list(MaxEnds, End),
            resource_limit(Start, End, Tasks, Bss, L)
        ).

resource_limit(T, T, _, _, _) :- !.
resource_limit(T0, T, Tasks, Bss, L) :-
        maplist(contribution_at(T0), Tasks, Bss, Cs),
        sum(Cs, #=<, L),
        T1 is T0 + 1,
        resource_limit(T1, T, Tasks, Bss, L).

task_bs(Task, InfStart-Bs) :-
        Task = task(Start,D,End,_,_Id),
        ?(D) #> 0,
        ?(End) #= ?(Start) + ?(D),
        maplist(finite_domain, [End,Start,D]),
        fd_inf(Start, InfStart),
        fd_sup(End, SupEnd),
        L is SupEnd - InfStart,
        length(Bs, L),
        task_running(Bs, Start, End, InfStart).

task_running([], _, _, _).
task_running([B|Bs], Start, End, T) :-
        ((T #>= Start) #/\ (T #< End)) #<==> ?(B),
        T1 is T + 1,
        task_running(Bs, Start, End, T1).

contribution_at(T, Task, Offset-Bs, Contribution) :-
        Task = task(Start,_,End,C,_),
        ?(C) #>= 0,
        fd_inf(Start, InfStart),
        fd_sup(End, SupEnd),
        (   T < InfStart -> Contribution = 0
        ;   T >= SupEnd -> Contribution = 0
        ;   Index is T - Offset,
            nth0(Index, Bs, B),
            ?(Contribution) #= B*C
        ).
```

Figure 4.22: Complete definition of `cumulative/2`

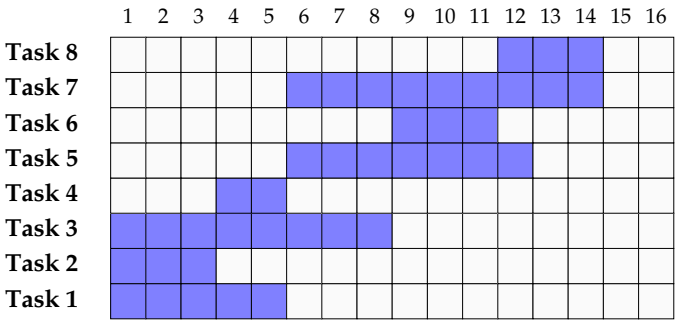


Figure 4.23: A schedule for 8 tasks of various durations, `L` = 3

4.13.2 `all_distinct/1`

The `all_distinct/1` constraint is true iff all elements of a list of finite domain variables are pairwise distinct. We have implemented the algorithm described by Jean-Charles Régin in [Rég94] for ensuring domain consistency of `all_distinct/1`.

The filtering algorithm works on the so-called *value graph* of the variables that are constrained to be pairwise distinct and whose domains are finite. The value graph is a bipartite graph with:

- one node n_{x_i} for each finite domain variable x_i
- one node n_j for each integer $j \in \bigcup D(x_i)$
- an edge between n_{x_i} and n_j iff $j \in D(x_i)$.

Each assignment of variables to values that satisfies `all_distinct/1` corresponds to a *matching* that covers all nodes n_{x_i} in the value graph and is hence a *maximum matching*. If there is no matching that covers all nodes n_{x_i}, the constraint cannot be satisfied. Moreover, using a property due to Berge, edges that belong to *no* maximum matching can be detected and removed from the value graph after reasoning over the strongly connected components (see Section 4.13.3) of a related graph. For example, the query

```
?- [X,Y] ins 1..2, Z in 1..3, all_distinct([X,Y,Z]).
```

yields the value graphs shown in Fig. 4.24 (a) and Fig. 4.24 (b) before and after `all_distinct/1` propagation, respectively. This is clearly stronger than posting pairwise disequalities and also takes more computation time.

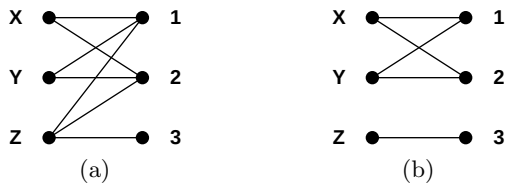


Figure 4.24: Value graph (a) before and (b) after `all_distinct/1`

Fig. 4.25 shows how we compute a matching that covers all nodes n_{x_i}, using the Edmonds-Karp algorithm. We use attributed variables to represent the value graph, with one variable for each node. Initially, each node n_j must have the attribute `free` attached, to mark it as a free node on the right-hand side. The edges of each node are stored as lists in the attribute `edges`. Each edge is a term of the form `flow_to(F, To)` or `flow_from(F, From)`, where the variable `F` is used to store the edge's associated *flow* in an attribute called `flow` (which is either 0 or 1), and `To` and `From` are adjacent nodes. The `parent` attribute is used to reconstruct an augmenting path after a free node was found during breadth-first search.

```
maximum_matching([]).
maximum_matching([FL|FLs]) :-
        augmenting_path_to([[FL]], Levels, To),
        phrase(augmenting_path(FL, To), Path),
        maplist(maplist(clear_parent), Levels),
        del_attr(To, free),
        adjust_alternate_1(Path),
        maximum_matching(FLs).

clear_parent(V) :- del_attr(V, parent).

reachables([]) --> [].
reachables([V|Vs]) -->
        { get_attr(V, edges, Es) },
        reachables_(Es, V),
        reachables(Vs).

reachables_([], _) --> [].
reachables_([E|Es], V) -->
        edge_reachable(E, V),
        reachables_(Es, V).

edge_reachable(flow_to(F,To), V) -->
        (   { get_attr(F, flow, 0),
              \+ get_attr(To, parent, _) } ->
            { put_attr(To, parent, V-F) },
            [To]
        ;   []
        ).
edge_reachable(flow_from(F,From), V) -->
        (   { get_attr(F, flow, 1),
              \+ get_attr(From, parent, _) } ->
            { put_attr(From, parent, V-F) },
            [From]
        ;   []
        ).

augmenting_path_to(Levels0, Levels, Right) :-
        Levels0 = [Vs|_],
        Levels1 = [Tos|Levels0],
        phrase(reachables(Vs), Tos),
        Tos = [_|_],
        (   member(Right, Tos), get_attr(Right, free, true) ->
            Levels = Levels1
        ;   augmenting_path_to(Levels1, Levels, Right)
        ).

augmenting_path(S, V) -->
        (   { V == S } -> []
        ;   { get_attr(V, parent, V1-Augment) },
            [Augment],
            augmenting_path(S, V1)
        ).

adjust_alternate_1([A|Arcs]) :-
        put_attr(A, flow, 1),
        adjust_alternate_0(Arcs).

adjust_alternate_0([]).
adjust_alternate_0([A|Arcs]) :-
        put_attr(A, flow, 0),
        adjust_alternate_1(Arcs).
```

Figure 4.25: Computing a matching that covers all variables

4.13.3 Tarjan's strongly connected components algorithm

Many of the global constraints that are implemented in our system require us to find the so-called *strongly connected components* (SCCs) of a graph. The strongly connected components of a directed graph are its maximal strongly connected subgraphs, which are graphs where there is a path from each vertex to every other vertex. There is a well-known algorithm for determining SCCs, described by Robert E. Tarjan in [Tar72].

Tarjan's algorithm for computing SCCs requires a global *index* and a global *stack*. To simulate a global variable in Prolog, we could pass its state to each predicate via additional arguments, representing its states before and after modification. Additional arguments typically make the code harder to follow, especially for predicates that do not even access these arguments directly, but only thread them through to other predicates that they call. We therefore use DCG notation to implicitly thread the global state through predicates that do not need to access it directly. Predicates that need to read or modify the state can use the convenient DCG nonterminals `state//1` and `state//2`, respectively. These DCG rules are shown in Fig. 4.26. They use semicontext notation to refer to the state. The use of the nonterminal `state(S)` can be read as "the current state is `S`". The use of `state(S0, S)` can be read as "the current state is `S0`, and henceforth it is `S`".

```
state(S), [S] --> [S].
state(S0, S), [S] --> [S0].
```

Figure 4.26: Accessing and modifying the state in DCGs

Our Prolog implementation of Tarjan's algorithm is shown in Fig. 4.27. We represent the global state as a term of the form

```
s(Index, Stack, Succ)
```

where `Index` is the current index (an integer), `Stack` is the list of vertices in the stack, and `Succ` is a binary predicate that relates a vertex v to a list of successors, i.e., vertices that are reachable from v via a directed edge. To use this algorithm, the predicate `scc/2` is called with a list of vertices and a predicate name for finding successors. Each vertex must be represented by a logical variable, and the algorithm uses variable attributes to store information about each vertex. Notably, it uses the attribute `in_stack` to determine in $\mathcal{O}(1)$ time whether a vertex is currently in the stack. After the algorithm terminates, vertices that belong to the same SCC have the same integer value stored in their `lowlink` attribute, starting with 0.

Our implementation is linear in the number of vertices and edges and is thus an asymptotically optimal implementation of Tarjan's algorithm.

```
1  scc(Vs, Succ) :- phrase(scc(Vs), [s(0,[],Succ)], _).
2
3  scc([])     --> [].
4  scc([V|Vs]) -->
5          (   vindex_defined(V) -> scc(Vs)
6          ;   scc_(V), scc(Vs)
7          ).
8
9  scc_(V) -->
10         vindex_is_index(V),
11         vlowlink_is_index(V),
12         index_plus_one,
13         s_push(V),
14         successors(V, Tos),
15         each_edge(Tos, V),
16         (   { get_attr(V, index, VI),
17               get_attr(V, lowlink, VI) } -> pop_stack_to(V, VI)
18         ;   []
19         ).
20
21 vindex_defined(V) --> { get_attr(V, index, _) }.
22
23 vindex_is_index(V) -->
24         state(s(Index,_,_)),
25         { put_attr(V, index, Index) }.
26
27 vlowlink_is_index(V) -->
28         state(s(Index,_,_)),
29         { put_attr(V, lowlink, Index) }.
30
31 index_plus_one -->
32         state(s(I,Stack,Succ), s(I1,Stack,Succ)),
33         { I1 is I+1 }.
34
35 s_push(V)  -->
36         state(s(I,Stack,Succ), s(I,[V|Stack],Succ)),
37         { put_attr(V, in_stack, true) }.
38
39 vlowlink_min_lowlink(V, VP) -->
40         { get_attr(V, lowlink, VL),
41           get_attr(VP, lowlink, VPL),
42           VL1 is min(VL, VPL),
43           put_attr(V, lowlink, VL1) }.
44
45 successors(V, Tos) --> state(s(_,_,Succ)), { call(Succ, V, Tos) }.
46
47 pop_stack_to(V, N) -->
48         state(s(I,[First|Stack],Succ), s(I,Stack,Succ)),
49         { del_attr(First, in_stack) },
50         (   { First == V } -> []
51         ;   { put_attr(First, lowlink, N) },
52             pop_stack_to(V, N)
53         ).
54
55 each_edge([], _) --> [].
56 each_edge([VP|VPs], V) -->
57         (   vindex_defined(VP) ->
58             (   v_in_stack(VP) ->
59                 vlowlink_min_lowlink(V, VP)
60             ;   []
61             )
62         ;   scc_(VP),
63             vlowlink_min_lowlink(V, VP)
64         ),
65         each_edge(VPs, V).
66
67 v_in_stack(V) --> { get_attr(V, in_stack, true) }.
```

Figure 4.27: Tarjan's strongly connected components algorithm in Prolog

4.14 Properties of labeling/2

As already mentioned in Section 2.5, labeling means systematically trying out values for variables. Our system provides a predicate called `labeling/2` that is used as `labeling(Options, Vs)` and labels the list of finite domain variables `Vs` according to `Options`. This predicate is largely compatible with that of SICStus Prolog (see Section 2.6), but there are two important differences. In our system, `labeling/2`:

- *always* terminates
- is *always* complete.

Both properties are independent of any specified options. We discuss the advantages of these properties in the following two sections.

The code for `labeling/2` is reasonably straight-forward, and interested readers can study its definition in the source code of our library.

4.14.1 labeling/2 always terminates

A CLP(FD) program can often be decomposed into two distinct parts:

(1) the posting of all relevant constraints

(2) the search for concrete solutions via `labeling/2`.

It is good practice ([Neu97]) to separate these parts by using a dedicated predicate for part (1).

When such a program is executed, it is very common that part (1) takes very little computation time, and part (2) takes a very long time, for example several weeks or even years in the case of unresolved interesting theoretical or practical problems. It is thus very desirable to know for certain at every point in time that a constraint solver is still searching for concrete solutions, and is not entangled in an infinite propagation chain. In other words, it is desirable to guarantee the following property: If part (1) terminates, which can often be easily observed, then the whole program terminates.

In our CLP(FD) system, this property holds because `labeling/2` *always* terminates. To guarantee this property, it is necessary that constraint *propagation* always terminates, since labeling triggers constraint propagation. For example, in CLP(FD) systems that do not guarantee terminating propagation, the equivalent of the following query correctly produces a conditional solution for `B = 0`, and then leads to nontermination on backtracking:

```
?- (X? #> abs(X?)) #<==> B?, labeling([], [B]).
```

As in SICStus Prolog, `labeling/2` raises an *instantiation error* when the domain of any variable in `Vs` is infinite at the time of the call.

4.14.2 `labeling/2` is always complete

In several CLP(FD) systems, there are *optimisation* options for `labeling/2`. For example, in SICStus Prolog, `minimize(E)` and `maximize(E)` can be used as labeling options to minimize or maximize the value of the arithmetic expression `E`.

To the best of our knowledge, all systems that provide such options *commit* to the first optimal solution that they find. This makes `labeling/2` incomplete in general.

In our system, `labeling/2` is *always* complete. When the optimisation options `min(E)` or `max(E)` as used, then the solutions are given in increasing and decreasing order with respect to the value of `E`, respectively.

The incomplete behaviour of other systems can be easily emulated with our system by simply committing to the first solution. It is much harder to obtain a complete predicate from an incomplete one. The importance of completeness is not only theoretical: In many cases, there are several different optimal solutions, and users may prefer some of them according to other criteria. Therefore, they may want to see all solutions for specific optima or within given margins.

Fig. 4.28 shows the Prolog code we use for optimising the value of expressions given in `Whats` as explained above, using the variables `Vs` and options `Options` for `labeling/2`. When the time limit is exceeded, we report the best solution found so far.

```
optimise(Vars, Options, Whats) :-
        Whats = [What|WhatsRest],
        Extremum = extremum(none),
        (   catch(store_extremum(Vars, Options, What, Extremum),
                  time_limit_exceeded,
                  false)
        ;   Extremum = extremum(n(Val)),
            arg(1, What, Expr),
            append(WhatsRest, Options, Options1),
            (   Expr #= Val,
                labeling(Options1, Vars)
            ;   Expr #\= Val,
                optimise(Vars, Options, Whats)
            )
        ).

store_extremum(Vars, Options, What, Extremum) :-
        catch((labeling(Options, Vars), throw(w(What))), w(What1), true),
        functor(What, Direction, _),
        maplist(arg(1), [What,What1], [Expr,Expr1]),
        optimise(Direction, Options, Vars, Expr1, Expr, Extremum).

optimise(Direction, Options, Vars, Expr0, Expr, Extremum) :-
        must_be(ground, Expr0),
        nb_setarg(1, Extremum, n(Expr0)),
        catch((tighten(Direction, Expr, Expr0),
               labeling(Options, Vars),
               throw(v(Expr))), v(Expr1), true),
        optimise(Direction, Options, Vars, Expr1, Expr, Extremum).

tighten(min, E, V) :- E #< V.
tighten(max, E, V) :- E #> V.
```

Figure 4.28: Optimisation

4.15 Performance

Neng-Fa Zhou, the author of B-Prolog, has kindly integrated our constraint solver in his benchmarks[1], which we reproduce in Table 4.10. The results show that our solver is on average two orders of magnitude slower on these benchmarks than the fastest system (B-Prolog itself), and about 30 times slower than the constraint solver of SICStus Prolog.

In part, this may certainly be attributed to the fact that SWI-Prolog itself (i.e., the system without the finite domain constraint solver) is already more than 4 times slower than B-Prolog (and more than 3 times than SICStus) on average on benchmarks that are deemed to be in some sense representative of a Prolog system's performance, and which are also available on the website. Our library is written in Prolog and is thus heavily influenced by the speed of the underlying Prolog system itself. We found that both sets of benchmarks are already faster with YAP ([CRD12]) by more than a factor of 2 on average.

On the other hand, if large integers are needed, our CLP(FD) solver is the only option of all tested systems and can not be compared to any others at all in this case.

While the comparatively slow speed of our constraint solver will certainly rule out its usage in many industrial settings, it is already being used at several universities in France, Germany, Italy, Austria and other countries for teaching and research purposes and has so far shown more than acceptable performance for these use cases.

benchmark	B-Prolog	ECLiPSe	GNU Prolog	SICStus	SWI
`alpha`	1	8.80	1.21	3.78	92
`bridge`	1	3.02	0.85	6.06	174
`cars`	1	4.68	0.93	1.97	125
`color`	1	8.68	0.97	2.97	114
`eq10`	1	4.38	3.03	3.94	128
`eq20`	1	4.27	1.73	2.74	72
`magic3`	1	6.54	1.09	2.83	134
`magic4`	1	7.52	1.56	3.87	137
`olympic`	1	9.43	1.61	2.76	134
`queens1`	1	13.19	1.18	12.61	264
`sendmoney`	1	6.20	2.62	5.55	156
`sudoku81`	1	7.45	1.35	4.52	103
`zebra`	1	5.96	1.66	7.04	95
mean	1	6.93	1.52	4.66	133

Table 4.10: Performance on different benchmarks

[1] available from *http://www.probp.com/performance.htm*

5 Testing a CLP(FD) system

5.1 Introduction

Authors of CLP(FD) systems should ask themselves: How can we make sure that our CLP(FD) system yields the results that we intended? In this section, we present two basic methods that we used to test our constraint system:

(1) tests of behaviour

(2) analysis of source code.

Prolog is well-suited for both approaches: First, built-in backtracking and Prolog's interactive toplevel let us concisely express and rapidly run collections of test cases. We consider this one of Prolog's most intriguing features, and – more generally – find that a programming language's practical usefulness is strongly related to its expressiveness and ease of use in the area of testing programs written in that language. Second, Prolog is a *homoiconic* language (meaning that Prolog source code is naturally represented as Prolog terms), and its homoiconicity and reflective abilities let us conveniently inspect and analyse the source code of individual propagators with the host language itself. Contrast this with a constraint solver written in a programming language that is not homoiconic, and the difficulty of then using the same language for analysing the solver's source code.

Method (1) is also known as *black-box* testing. Since our system guarantees monotonicity and terminating propagation, black-box testing can be applied more extensively than for other CLP(FD) systems.

Method (2) is related to *white-box* testing. To the best of our knowledge, we are the first to apply *abstract interpretation* to the source code of propagators to establish further guarantees about our system.

5.2 Properties of logical variables

Among the concepts that distinguish Prolog from many other programming languages is its support for *logical variables*. A logical variable can actually be *uninstantiated*. When testing or analysing a Prolog predicate, we therefore not only have to take into account concrete possible values of its variables, but also think about cases in which the predicate is used with any or all of its arguments uninstantiated, as is the case for most finite domain constraints and their propagators. This language feature by itself can make Prolog programs much more general than users of other languages may be used to. It also significantly increases the number of possible cases we have to take into account when testing Prolog predicates compared to, say, functions in other languages, where we at least know that each variable at all times stands for *some* concrete value and will never be *uninstantiated*.

Moreover, two variables may both be uninstantiated, but *aliased* by constraints that unify them. So in addition to taking into account uninstantiated variables, we also have to consider the possibility of syntactically distinct variables that are semantically the same.

These issues provide a rich source of potential mistakes when writing propagators, since human authors typically cannot think of all possible cases at the same time and may forget to handle cases that arise in rare contrived cases as well as in practical applications.

5.3 Systematic test cases for a CLP(FD) system

It is very hard to verify the *semantics* of individual propagators. For example: Does the propagator for *multiplication*, which is shown in Fig. 4.16, really establish bounds consistency, or have we maybe forgotten to handle a case in Fig. 4.19? Verifying whether a CLP(FD) system actually establishes bounds consistency in a given example essentially requires us to emulate the filtering that the system is supposed to do. Basically, this would require us to write or use another CLP(FD) system to test a given system, which would raise the same questions.

However, we can test other guaranteed properties of a CLP(FD) system without reasoning about the semantics of individual propagators. For example, to the best of our knowledge, our CLP(FD) system is the first widely available system that guarantees *monotonicity*, and hence this property can be tested via black-box tests: For two monotonic constraints `A` and `B`, the goals `(A, B)` and `(B, A)` must yield semantically equivalent results. In particular, it must never be the case that one of them succeeds unconditionally, and the other one fails. It is for example admissible that one of them succeeds, and the other one raises an exception. Notice that *unification* is also monotonic, and hence we can use unification for syntactic substitutions of terms. This makes formulating black-box tests easier than in systems that do not allow such dynamic substitutions (see also Section 4.3). Other systems guarantee this property only to a limited extent, and we can therefore test them only in limited ways with this approach. Still, as we will see, the general method is useful throughout all available CLP(FD) systems.

In general, the black-box tests we use for testing our system only terminate when they find a mistake. Otherwise, they generate increasingly complex expressions and compare the results of goals that must be semantically equivalent. This testing methodology may seem unusual at first. However, it is well-justified since – as we have seen in Section 3.3.3 – mistakes can surface also after weeks of computation time.

When posting various constraints in different ways, it is very useful to know that constraint propagation *always* terminates in our system (see Section 4.6). Hence, the system will never be entangled in an infinite propagation chain, regardless of the order we use for posting constraints.

Fig. 5.1 shows an example of systematic black-box tests that we use to test our system. These tests are applicable to other systems as well since they do not generate large integers and only require very limited support for dynamic syntactic substitutions. The predicate `run/0` systematically generates reifiable constraints `C` of increasing depth. Finite domain variables that appear in arithmetic expressions are constrained to integers between -3 and 3 in this example. Each generated constraint `C` and its variables `Vs` are then used in three declaratively equivalent ways:

- `C` is posted, then `Vs` are labeled
- `?(X) #==> C` and `X #= 1` are posted, then `Vs` are labeled
- `\# \# C` is posted, then `Vs` are labeled

The three lists of corresponding solutions are then compared, and an error is raised if they are not the same. In line 35, we ensure that the implication `0 #==> C` succeeds, which must always hold.

```
1  d_c(D, A #> B)         :- d_exp(D, A), d_exp(D, B).
2  d_c(D, A #< B)         :- d_exp(D, A), d_exp(D, B).
3  d_c(D, A #= B)         :- d_exp(D, A), d_exp(D, B).
4  d_c(D, A #=< B)        :- d_exp(D, A), d_exp(D, B).
5  d_c(D, A #>= B)        :- d_exp(D, A), d_exp(D, B).
6  d_c([_|D], A #/\ B)    :- d_c(D, A), d_c(D, B).
7  d_c([_|D], A #\/ B)    :- d_c(D, A), d_c(D, B).
8  d_c([_|D], A #==> B)   :- d_c(D, A), d_c(D, B).
9  d_c([_|D], A #<==> B)  :- d_c(D, A), d_c(D, B).
10
11 d_exp(_, X)            :- X in -3..3.
12 d_exp(_, N)            :- N in -3..3, indomain(N).
13 d_exp(_, X)            :- X in -3..3, between(-3, 3, N), X #\= N.
14 d_exp([_|D], A+B)      :- d_exp(D, A), d_exp(D, B).
15 d_exp([_|D], A-B)      :- d_exp(D, A), d_exp(D, B).
16 d_exp([_|D], A*B)      :- d_exp(D, A), d_exp(D, B).
17 d_exp([_|D], A/B)      :- d_exp(D, A), d_exp(D, B).
18 d_exp([_|D], abs(A))   :- d_exp(D, A).
19 d_exp([_|D], min(A,B)) :- d_exp(D, A), d_exp(D, B).
20 d_exp([_|D], max(A,B)) :- d_exp(D, A), d_exp(D, B).
21 d_exp([_|D], A mod B)  :- d_exp(D, A), d_exp(D, B).
22 d_exp([_|D], A rem B)  :- d_exp(D, A), d_exp(D, B).
23
24 run :-
25         length(D, L),
26         portray_clause(L),
27         d_c(D, C),
28         term_variables(C, Vs),
29         findall(Vs, (C,label(Vs)), Sols1),
30         findall(Vs, (?(X)#==>C,X#=1,label(Vs)), Sols2),
31         findall(Vs, (#\ #\ C, label(Vs)), Sols3),
32         (   Sols1 == Sols2, Sols1 == Sols3 -> true
33         ;   throw(C)
34         ),
35         (   0 #==> C, label(Vs) -> true
36         ;   throw(0#==>C)
37         ),
38         false.
```

Figure 5.1: Testing reification of arithmetic constraints

5.4 Advantages of black-box testing

Black-box testing is attractive because it is easy to formulate test cases based on algebraic properties, and Prolog's built-in backtracking and all-solutions predicates make it convenient to generate increasingly complex cases and to compare their results. When expressions are generated exhaustively, one can even derive some guarantees about the system's behaviour after it successfully passes each level of black-box tests.

Black-box tests can also be applied to other CLP(FD) systems, and can be used to test propagators that are comparatively difficult to analyse in other ways. For instance, we cannot tell from the outside whether `all_distinct/1` internally computes a maximum matching and the SCCs of a graph. However, desirable properties of the constraint's *observable* behaviour are easy to test. For example, when `Vs` is a list of variables with finite domains, we require that `(all_distinct(Vs), label(Vs))` and `(label(Vs), all_distinct(Vs))` yield exactly the same solutions. Fig. 5.2 shows how this property can be systematically tested for increasingly longer lists `Vs` and systematic variations of domains. This property may seem self-evident at first glance. However, it makes sense to test global constraints in this way: When all arguments are ground, as they are after labeling, the propagator is reduced to a simple checker and usually works correctly. Hence it can be used as a reference. It is the actual filtering, which may involve complex computations on a CSP's value graph, that is more likely to contain programming mistakes. Similar tests can be applied to other propagators as well.

```
1  neqs([], _)     --> [].
2  neqs([N|Ns], X) --> neqs(Ns, X), ( [] ; [X #\= N] ).
3
4  disequalities([], _)      --> [].
5  disequalities([V|Vs], Ns) --> disequalities(Vs, Ns), neqs(Ns, V).
6
7  run :-
8          length(Vs, LVs), portray_clause(LVs),
9          numlist(1, LVs, Ns), phrase(disequalities(Vs,Ns), Cs),
10         Vs ins 1..LVs, maplist(call, Cs),
11         findall(Vs, (label(Vs),all_distinct(Vs)), L1),
12         findall(Vs, (all_distinct(Vs),label(Vs)), L2),
13         (   L1 == L2 -> true
14         ;   throw((maplist(call, [Vs ins 1..LVs|Cs])))
15         ),
16         false.
```

Figure 5.2: Testing `all_distinct/1`

5.5 Limitations of black-box testing

The interaction between propagators can be quite complex in a constraint system, and black-box tests are unlikely to find all mistakes since they can only ever cover a finite subset of an infinite search space. For example, recall

from Section 4.4 that we use two kinds of queues for scheduling the invocation of propagators in our system: one queue for fast propagators, and one for slower propagators. For some constraints that require complex filtering algorithms, it is advantageous to use two propagators: one simple and fast propagator that is inserted into the former queue, and one more complex and slower propagator, scheduled in the latter queue. We actually use this strategy in the implementation of the `global_cardinality/2` constraint in our system (see Section 4.13): `gcc_check/1` is a propagator that performs rudimentary and fast filtering, and `gcc_global/1` is a complex and slower propagator that reasons about a specific graph. Importantly, `gcc_global/1` relies on `gcc_check/1` to remove all variables that have become instantiated from internal lists, so that `gcc_global/1` only needs to reason about variables. Consider now how these propagators interact with the `tuples_in/2` constraint in the following query:

```
?- tuples_in([[A,C,B]], [[3,1,3],[4,2,4]]),
   global_cardinality([A,B,D], [3-1,4-2]),
   A = 4.
```

The unification `A = 4` causes `gcc_check/1` and `gcc_global/1` to be queued in the fast and slow queue, respectively. The propagator corresponding to `tuples_in/2` is also queued in the fast queue. Then, constraint propagation starts. First, `gcc_check/1` is invoked and cannot perform any filtering. Next, the propagator corresponding to `tuples_in/2` is invoked and detects that the only fitting binding is `[A,C,B] = [4,2,4]`. It thus simultaneously instantiates `C` and `B` (to 2 and 4, respectively). Critically, `tuples_in/2` is thus capable of instantiating two or more variables at once *while* propagators are scheduled in the queues. Instantiation of `C` causes a `do_queue` (see Section 4.11.4) in `attr_unify_hook/2`. Notice that `gcc_check/1` is *not* (yet) queued again because `C` does *not* participate in the `global_cardinality/2` constraint. However, `do_queue` next invokes the `gcc_global/1` propagator that is still in the queue. Since `gcc_check/1` has not yet had a chance to run due to the unification mechanism's first handling the constraints `C` is involved in, `gcc_global/1` must take into account that some of the variables it reasons about have become instantiated but `gcc_check/1` was not invoked.

Therefore, `gcc_check/1` must always be called also in `gcc_global/1`. However, a single call of `gcc_check/1` is not enough, as can be shown with an analogous test case. It is also necessary to re-invoke the fixpoint computation via `do_queue` to really remove all remaining integers from the list of variables.

Black-box tests cannot be used to guarantee such properties, since the absence of a counterexample in a finite subset of an infinite search space does not imply that none at all exists. It is therefore desirable to formulate relevant properties of our code and verify them in other ways, as we will do in the next section.

5.6 Automated analysis of individual propagators

We now analyse the *source code* of propagators to give stronger guarantees. The source code is finite and can be analysed in its entirety. However, source code can also be quite complex and use many different language features, which also makes its analysis complex.

Our goal in this section is to present a self-contained example of an analyser that ensures the following properties of the propagators for *addition* and *multiplication* (shown in Fig. 4.15 and Fig. 4.16, respectively):

(1) Domain operations (`domain_shift/3`, `domain_expand/3` etc.) are always used with the correct argument types. For example, a goal like `domain_contains(D, n(0))` must not occur in a propagator because the second argument is not an integer. Note how easy it is to accidentally write such a goal instead of the intended `domain_contains(D, 0)` when writing a propagator, because the wrapper `n/1` *must* be used for numbers that occur in a `cis/2` expression. See Section 4.9 for more information about compactified arithmetic with `cis/2`, and Section 4.10.3 for predicates over domains. In the following, we assume that these predicates are correctly implemented.

(2) Explicit unifications (using `=/2`) always unify terms of the same kind. By this we mean for example that a finite domain variable may only be unified with another finite domain variable or integer. Note how easy it is to accidentally write a goal like `Z = n(Y)` instead of the intended `Z = Y` when writing a propagator because the `n/1` wrapper *must* be used in `cis/2` expressions. Consider also that the propagator code for *multiplication* (Fig. 4.16) contains unifications that involve `Y` as well as `n(Y)`, which further increases the likelihood of such mistakes.

A unification that succeeds syntactically may of course still fail semantically due to other constraints that are not part of the propagator code itself. For example, the unification `X = 0` that appears in the code of a propagator fails if $0 \notin D(X)$.

(3) The last argument of `domain_shift/3`, `domain_expand/3` and other relations between domains is always a variable *at run-time* before the goal is called. As a necessary but not sufficient condition, it must of course also be a Prolog variable in the source code of the propagator.

It is easy to accidentally reuse a source-level variable when computing new domains. For example, the following must *not* occur in a propagator, because `YD1` is accidentally reused:

```
(   fd_get(Y, YD1, YPs) ->
    0 is -X,
    domain_shift(ZD1, 0, YD1)
```

After this is corrected by replacing `YD1` with the fresh variable `YD2`, this snippet actually occurs in the propagator for *addition*. As a consequence, accidentally reusing `YD1` is a mistake that can easily occur at that place.

(4) Similarly, the first argument of `fd_put/3` is always a variable *at run-time* when the predicate is called. This must be so because if the built-in predicate `put_attr/3` is accidentally called with an instantiated first argument, a so-called *uninstantiation error* is raised because an uninstantiated variable is expected for attaching attributes. Note that a previously successful call of `fd_get/3` does not in general guarantee that a variable is still uninstantiated, because constraint propagation may be invoked due to goals between the two calls.

Properties (3) and (4) are very similar, but there is an important difference between them: Property (3) can be reduced to a simple static analysis of the propagator by ensuring that the last argument of the mentioned relations is always a *fresh* variable in the source code. As a consequence, it is always a variable at run-time. Property (4) requires a more elaborate analysis, because the first argument of `fd_put/3` is typically a variable that is involved in several constraints and may, as mentioned, become instantiated because certain goals can trigger further propagators.

Properties (1) and (2) can also be guaranteed by *type systems* for Prolog, while properties (3) and (4) require reasoning about *instantiations* and would require a *mode system* that can detect unexpected instantiations. Note that any constrained variable that a propagator reasons about *may or may not* become instantiated by `fd_put/3` and subsequent constraint propagation. Even the powerful assertion language of Ciao ([HBC+12]) cannot express this non-local effect. In addition, and as we will now demonstrate, all of the above properties can be ensured by providing enough information about the predicates that are directly used in the propagator code, whereas type and mode systems typically require more information or analysis of indirectly used predicates as well and thus may make it harder to establish relevant properties of selected propagators in isolation.

At first glance, writing a program that checks all the above properties may seem a daunting task. After all, the control flow inside propagators can get quite complicated due to several nested conditions, and all branches of the computation need to be checked while keeping track of which conditions are satisfied in each branch. Luckily though, we need not do this explicitly. Instead, we use Prolog's built-in backtracking and unification to walk through the source code while implicitly keeping track of the conditions and bindings that hold in each branch.

We use *abstract interpretation* ([CC92]) on propagators to check all properties. This means that instead of running the code with concrete instantia-

tions of variables, we use abstract elements to denote the *kind* of terms that each variable stands for. We use the following atoms as abstract elements:

- `b` represents a domain boundary
- `d` represents a domain
- `i` represents an integer

When unifications are to be performed, we can use these abstract elements to describe many possible concrete terms at once.

As already mentioned, the source code of each propagator is naturally represented as a Prolog term. Variables that occur in the propagator code are also variables in the term representation and can hence be used for unifications with abstract elements. All we have to do is to declaratively describe what each language construct means in terms of this abstract domain of types. Fig. 5.4 gives the complete source code that describes each language element that occurs in the propagators of *addition* and *multiplication* (see Fig. 4.15 and Fig. 4.16). A DCG is used to transparently thread an implicit argument through the code: It is necessary to keep track of all arguments of the propagator that is being processed, because each of these arguments may be instantiated to a concrete integer every time further propagators are triggered. This can happen for example in `fd_put/3` and `update_bounds/7`. If a language construct is encountered that cannot be handled or any of the properties (1)–(4) is violated, the program raises an *exception*.

Fig. 5.5 shows necessary additional definitions. Notably, `var_or_int/1` is a nondeterministic predicate, which succeeds if its argument is a variable, and binds it to `i` on backtracking, corresponding to the fact that a variable *may or may not* become instantiated during constraint propagation.

The built-in `clause/2` predicate allows us to obtain the Prolog term representation of a specific clause. We now use this predicate to analyse the clause body of the `pplus` propagator, which implements *addition*. The clause body of this propagator can be obtained as `Body` via:

```
clause(clpfd:run_propagator(pplus(X, Y,Z), _), Body)
```

Since the shown analyser can only handle the *unexpanded* source code, we must set the Prolog flag `clpfd_goal_expansion` to `false` before loading the library. Otherwise, the finite domain constraints that are used in propagators are expanded via `goal_expansion/2`, and the analyser has to be extended to handle further language elements like `integer/1` and `var/1`.

When the propagator is invoked during actual constraint propagation, each of its arguments is initially either a variable or integer. We therefore use `var_or_int/1` to simulate each possibility. With the following query, we try to find a *counterexample* to correctness, i.e., a case where any of the

properties (1)–(4) is violated and the program therefore raises an exception. Forced backtracking via `false` leads us through each possible branch of the propagator:

```
?- Vars = [X,Y,Z],
   P =.. [pplus|Vars],
   clause(clpfd:run_propagator(P, _), Body),
   maplist(var_or_int, Vars),
   phrase(walk(Body), [Vars], _),
   false.
```

yielding:

```
false.
```

Since this query *fails*, we can conclude that, if the analyser is correct and the variables are not aliased, properties (1)–(4) hold for the propagator that implements *addition*. An analogous query can be used to establish these properties for `ptimes` and also for the case of any aliasing between the three variables for both propagators. With suitable additional definitions, we have verified these properties for all arithmetic propagators that are implemented in our system.

Fig. 5.3 shows an intermediate state that arises when abstractly interpreting the propagator for *addition*. In this concrete case, the first argument is instantiated to an abstract integer, and several variables are already instantiated to abstract domains, which are indicated by the atom `d`.

```
1  run_propagator(pplus(i, Y, Z), MState) :-
2          (   nonvar(i)
3          ->  (   i =:= 0 -> kill(MState), Y = Z
4              ;   Y == Z -> kill(MState), i =:= 0
5              ;   nonvar(Y) -> kill(MState), Z is i + Y
6              ;   nonvar(Z) -> kill(MState), Y is Z - i
7              ;   fd_get(Z, d, ZPs),
8                  fd_get(Y, d, _),
9                  domain_shift(d, i, d),
10                 domains_intersection(d, d, d),
11                 fd_put(Z, d, ZPs),
12                 (   fd_get(Y, d, YPs) ->
13                     i is -i,
14                     domain_shift(d, i, d),
15                     domains_intersection(d, d, d),
16                     fd_put(Y, d, YPs)
17                 ;   true
18                 )
19                 ...
```

Figure 5.3: Abstract interpretation of the `pplus` propagator

To the best of our knowledge, this is the first time that *abstract interpretation* is applied to individual propagators of a CLP(FD) system to verify the properties we have formulated above. Abstract interpretation is easily applicable for our system since it is written in Prolog.

```
walk((If -> Then ; Else)) --> !,
        (   walk(If), walk(Then)
        ;   walk(\+ If), walk(Else)
        ).
walk(Either ; Or)         --> !, ( walk(Either) ; walk(Or) ).
walk(\+ (Either ; Or))    --> !, walk(\+ Either), walk(\+ Or).
walk(X = Y)               --> !,
        { (   ( var(X) ; var(Y) )-> X = Y
          ;   X == i -> i(Y)
          ;   throw(unif-(X=Y))
          ) }.
walk(Rel)              -->
        { functor(Rel, F, 2), memberchk(F, [==,=:=,=\=,>=,<,>]) }, !.
walk(\+ Rel)           -->
        { functor(Rel, F, 2), memberchk(F, [==,=:=,=\=,>=,<,>]) }, !.
walk( _ in _), [Vars] --> !, [Vars], { maplist(var_or_int, Vars) }.
walk(i is _)           --> !.
walk(_ #= _)           --> !.
walk((A,B))            --> !, walk(A), walk(B).
walk(\+ (A,B))         --> !, ( walk(\+ A) ; walk(A), walk(\+ B) ).
walk(fd_get(X, XD, _)) --> !, { fresh(d, XD), var(X) }.
walk(\+ fd_get(i, _, _)) --> !.
walk(fd_get(X, XD, XL, XU, _)) --> !,
     walk(fd_get(X, XD, _)), { maplist(fresh(b), [XL,XU]) }.
walk(\+ fd_get(i, _, _, _, _)) --> !.
walk(fd_put(X, _, _))          --> !, fd_put(X).
walk(nonvar(i))                --> !.
walk(\+ nonvar(X))             --> !, { var(X) }.
walk(kill(_))                  --> !.
walk(true)                     --> !.
walk(run_propagator(_, _))     --> !.
walk(even(I))                  --> !, { i(I) }.
walk(domain_shift(D,I,D1))     --> !, { d(D), i(I), fresh(d, D1) }.
walk(domain_expand(D,I,D1))    --> !, { d(D), i(I), fresh(d, D1) }.
walk(domain_negate(D, D1))     --> !, { d(D), fresh(d, D1) }.
walk(domain_contract(D,I,D1))  --> !, { d(D), i(I), fresh(d, D1) }.
walk(domain_contains(D,I))     --> !, { d(D), i(I) }.
walk(\+ domain_contains(_,_))  --> !.
walk(domains_intersection(D1,D2,D)) --> !,
        { d(D1), d(D2), fresh(d, D) }.
walk(Cis) -->
        { Cis =.. [F,A,B,C],
          memberchk(F, [cis_minus,cis_plus,cis_times,
                        cis_max,cis_min]), !,
          b(A), b(B), fresh(b, C) }.
walk(cis_geq(A,B))             --> !, { b(A), b(B) }.
walk(\+ cis_geq(_,_))          --> !.
walk(update_bounds(X,_,_,XL,XU,NXL,NXU)) --> !,
        { maplist(b, [XL,XU,NXL,NXU]) },
        fd_put(X).
walk(integer_kth_root(I, K, R)) --> !, { i(I), i(K), fresh(i, R) }.
walk(neq_num(_, N)) --> !, { i(N) }.
walk(min_max_factor(L1,U1,L2,U2,L3,U3,Min,Max)) --> !,
        { maplist(b, [L1,U1,L2,U2,L3,U3]),
          maplist(fresh(b), [Min,Max]) }.
walk(min_product(XL,XU,YL,YU,Min)) --> !,
        { maplist(b, [XL,XU,YL,YU]), fresh(b, Min) }.
walk(max_product(XL,XU,YL,YU,Max)) --> !,
        { maplist(b, [XL,XU,YL,YU]), fresh(b, Max) }.
walk(Code) --> [Vars], { throw(unknown-Code-Vars) }.
```

Figure 5.4: Abstract interpretation of a propagator's clause body

```
1  fd_put(X), [Vars] --> [Vars],
2          { must_be(var, X),
3            maplist(var_or_int, Vars) }.
4
5  var_or_int(X) :- var(X).
6  var_or_int(i).
7
8  b(B) :-
9          must_be(nonvar, B),
10         (   B == b -> true
11         ;   B = n(I) -> i(I)
12         ;   B == inf -> true
13         ;   B == sup -> true
14         ;   throw(nonboundary-B)
15         ).
16
17 d(D) :-
18         must_be(nonvar, D),
19         (   D == d -> true
20         ;   D = from_to(B1,B2) -> b(B1), b(B2)
21         ;   D = split(S,X,Y) -> i(S), d(X), d(Y)
22         ;   throw(nondomain-D)
23         ).
24
25 fresh(What, N) :-
26         (   What == b, nonvar(N), N = n(X) -> must_be(var, X), X = i
27         ;   must_be(var, N), N = What
28         ).
29
30 i(I) :- ( I == i -> true ; integer(I) -> true ; throw(nonint-I) ).
```

Figure 5.5: Additional definitions for analysing propagator code

6 Application: Rotating workforce scheduling

The measure and purpose of every constraint system is its ability to solve problems of practical relevance. While the main subject of this thesis are *correctness considerations* in CLP(FD) systems, we feel it would be incomplete without presenting a concrete use case of our solver. As one of many possible examples, we present a new software application for *rotating workforce scheduling* that we are implementing at our institute.

As we have seen in Section 4.15, we cannot directly compete in the area of performance with our CLP(FD) system. However, as we outline in the following sections, we benefit from the use and implementation of our system in the following ways, which likely apply to other situations as well:

- First, we use our system for prototyping. Since our system is freely available and supports many important constraints that are also available in other CLP(FD) systems, it can be readily used to experiment with different constraint formulations and to get a preliminary impression about which formulations are promising. The very convenient development tools that are available in SWI-Prolog, such as its graphical tracer, `make/0` facility and execution profiler, make this phase especially productive. Once a suitable model is found, we can run the formulation in other and faster Prolog systems with little effort.

- Second, our CLP(FD) system is implemented entirely in Prolog. This allows us to port the formulation of several constraints to systems that do not support them yet. We show an example of this in Section 6.6, where we use our formulation of the `automaton/3` constraint to make it available in B-Prolog and GNU Prolog with negligible effort.

- Third, as we saw in Section 4.4, the architecture of our CLP(FD) system is so simple that mistakes are less likely than in other systems, which need to take into account the subtle interactions of several mechanisms that improve performance (see for example Section 3.3.3). Moreover, due to the properties we guarantee in our system and since it is written in Prolog, we can test it more extensively (see Chapter 5) and can be more confident about its correctness. Hence, we use our own CLP(FD) system as a reference to check answers of other systems.

- Fourth, and as we will demonstrate, our system's strong filtering for several global constraints allows us to even solve some problems more efficiently than other CLP(FD) systems, which have chosen not to implement such strong filtering.

This combination of available CLP(FD) systems yields very competitive results, which we have published as: Markus Triska and Nysret Musliu,

A Constraint Programming Application for Rotating Workforce Scheduling, EA/AIE 2011, Studies in Computational Intelligence 363 (2011), pp. 83–88, on which this chapter is based.

6.1 Introduction

Computerized workforce scheduling has interested researchers for more than 30 years. To solve rotating workforce scheduling problems, different approaches have been used in the literature, including exhaustive enumeration ([NHS73], [But78]), constraint (logic) programming, genetic algorithms ([MM04]) and local search methods.

In this chapter, we describe *CP-Rota*, a new constraint application for rotating workforce scheduling that is currently being developed at our institute to solve real-life problems from industry. It is intended to complement *FCS*, a previously developed application that is currently commercially used in many companies in Europe. *CP-Rota* builds upon, contributes to and improves previous constraint programming approaches to rotating workforce scheduling in the following ways:

- *CP-Rota* is written in portable Prolog and will eventually be released under a permissive licence to benefit both researchers and practitioners. Much of its code is already available on request at the time of publication.
- *CP-Rota* implements new allocation strategies (available as options for users to choose) that we discovered and discuss in this chapter, which yield significantly improved performance on some real-life instances.
- Our benchmarks on real-life instances underline the potential of constraint programming in rotating workforce scheduling, also and especially due to using different language implementations where they excel.

6.2 Related work

Many different approaches for solving rotating workforce instances are documented in the literature. Balakrishnan and Wong [BW90] solved a problem of rotating workforce scheduling by modeling it as a network flow problem. Laporte [Lap99] considered developing the rotating workforce schedules by hand and showed how the constraints can be relaxed to get acceptable schedules. Musliu et al. [MGS02] proposed and implemented a method for the generation of rotating workforce schedules, which is based on pruning the search space by involving the decision maker during the generation of partial solutions. The algorithms have been included in a commercial product called First Class Scheduler (FCS) [GMS01], which is used by many companies in Europe. In [Mus06], Musliu applied a min-conflicts heuristic in

combination with tabu search. Although this yields good performance on many instances, the resulting search method is incomplete and its results are therefore not directly comparable with FCS. This paper also introduced 20 real-life problems collected from different areas in industry and the literature. [2]

The use of constraint logic programming for rotating workforce scheduling was first shown by Chan in [WGC01]. Laporte and Pesant [LP04] have also proposed a constraint programming algorithm for solving rotating workforce scheduling problems, implemented in ILOG and requiring custom extensions.

6.3 The rotating workforce scheduling problem

With *CP-Rota*, we focus on a specific variant of a general workforce scheduling problem, which we formally define in this section. The following definition is from [MGS02] and proved to be able to satisfactorily handle a broad range of real-life scheduling instances in commercial settings. A rotating workforce scheduling *instance* as discussed in the present section consists of:

- n: Number of employees.
- A: Set of m shifts (activities) : a_1, a_2, ..., a_m.
- w: Length of the schedule. A typical value is $w = 7$, to assign one shift type for each day of the week to each employee. The total length of a planning period is $n \times w$ due to the schedule's cyclicity as discussed below.
- R: Temporal requirements matrix, an $m \times w$-matrix where each element $R_{i,j}$ shows the required number of employees that need to be assigned shift type i during day j. The number o_j of day-off "shifts" for a specific day j is implicit in the requirements and can be computed as $o_j = n - \sum_{i=1}^{n} R_{i,j}$.
- Sequences of shifts not permitted to be assigned to employees. For example, one such sequence might be ND (Night Day): after working in the night shift, it is not allowed to work the next day in the day shift. A typical rotating workforce instance forbids several shift sequences, often due to legal reasons and safety concerns.
- MIN_s and MAX_s: Each element of these vectors shows, respectively, the required minimal and permitted maximal length of periods of consecutive shifts of the same type.

[2] These examples with three additional other examples previously proposed in the literature are available from *http://www.dbai.tuwien.ac.at/staff/musliu/benchmarks/*

- MIN_w and MAX_w: Minimal and maximal length of blocks of consecutive work shifts. A work block is a sequence consisting only of working shifts (without days-off in between). This constraint limits the number of consecutive days on which the employees can work without having a day off.

The task in rotating workforce scheduling is to construct a *cyclic schedule*, which we represent as an $n \times w$ matrix $S_{n,w} \in A \cup \{\text{day-off}\}$. Each element $S_{i,j}$ denotes the shift that employee i is assigned during day j, or whether the employee has time off. In a cyclic schedule, the schedule for one employee consists of a sequence of all rows of the matrix S.

The task is called *rotating* or *cyclic* scheduling because the last element of each row is adjacent to the first element of the next row, and the last element of the matrix is adjacent to its first element. Intuitively, this means that employee i ($i < n$) assumes the place (and thus the schedule) of employee $i+1$ after each week, and employee n assumes the place of employee 1. This cyclicity must be taken into account for the last three constraints above.

Figure 6.1 shows an example of a rotating workforce instance as specified by our customers and a possible solution.

```
#Length of the schedule
7

#Number of Employees
9

##Number of Shifts
3

# Temporal Requirements Matrix
2 2 2 2 2 2 2
2 2 2 3 3 3 2
2 2 2 2 2 2 2

#ShiftName, Start, Length, Min/Max-Length
D  360 480 2 7
A  840 480 2 6
N  1320 480 2 4

# Min/Max-Length of days-off blocks
2 4

# Min/Max-Length of work blocks
4 7

# Number of not allowed shift sequences of
# length 2 and 3, respectively
3 0

# Not allowed shift sequences
N D
N A
A D
```

(a)

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Empl. 1	-	-	-	-	D	D	D
Empl. 2	D	A	A	A	-	-	-
Empl. 3	-	D	D	D	A	A	A
Empl. 4	A	-	-	A	A	A	A
Empl. 5	N	N	N	-	-	D	D
Empl. 6	D	N	N	N	N	-	-
Empl. 7	A	A	A	N	N	N	-
Empl. 8	-	D	D	A	A	A	N
Empl. 9	N	-	-	D	D	N	N

(b)

Figure 6.1: (a) A rotating workforce instance and (b) a valid schedule

In the literature and in practice, different variants of workforce scheduling problems also appear. Such problems include for example nurse scheduling, where soft constraints like individual preferences of employees are of high importance.

Note that in [MGS02], finding as many non-isomorphic cyclic schedules as possible that satisfy all constraints, and are optimal in terms of weekends without scheduled work shifts (weekends off), are required. In the present section, we consider the generation of a single schedule that satisfies all the hard constraints given in the problem definition. Fulfilling all these constraints is usually sufficient in practice. The same constraints that we use in this section are used in the commercial software *FCS* for generating rotating workforce schedules. This system has been used since 2000 in practice for many companies in Europe and the scheduling variant we discuss in this section proved to be sufficient for a broad range of uses.

6.4 A new system for rotating workforce scheduling

FCS has been in commercial use since 2000 in several companies and proved to be a good solution for many applications in practice. However, the following reasons have motivated us to investigate other approaches as well:

- The code base of *FCS* has gotten quite large and hard to maintain. This makes user modifications difficult and error-prone.
- *FCS* is implemented in *Visual Basic* and thus depends on essentially a single supported language implementation, which is in addition also not freely available. This complicates the sharing of code with other researchers and practitioners for joint development and turns every mistake in the language implementation into a potentially show-stopping problem.
- From [Mus06], it is known that local search approaches – although incomplete and thus not applicable in some use cases – can significantly outperform *FCS* on some instances. We thus aimed to improve the running times of *FCS* to more closely match competing approaches while retaining the completeness of the search.

When we started to work on a different approach for the above reasons, we initially looked into constraint programming in the hope to significantly reduce the size of the code base. The promise of constraint programming was to just state the necessary requirements with high-level constraints and to then use built-in enumeration methods to search for solutions.

In addition, many available constraint logic platforms already support the same types of constraints and other built-in predicates. There was thus hope that we could free the new system from its dependence on a single language implementation. We thus implemented our application using the portable constraint programming model described in Section 6.5. The resulting new system performs competitively (see Section 6.9) and even outperforms *FCS* on several instances, making constraint logic programming a promising approach for this problem. We call the new system *CP-Rota*.

6.5 A CLP(FD) model for rotating workforce scheduling

We chose Prolog (with finite domain extension) as a suitable implementation language for *CP-Rota*. Having the goal of portability in mind to free *CP-Rota* from a single language implementation, we tried to use commonly available constraints wherever possible.

Our initial development environment was SWI-Prolog, which we chose due to its convenient libraries, tools and workflow, and also because it is freely available. We then ported the model to GNU Prolog due to its much better performance, and because it is also freely available. This required only comparatively simple changes, with the exception of the `automaton/3` constraint that we used, and whose implementation in GNU Prolog we describe in Section 6.6.

When experimenting with custom allocation strategies (Section 6.8), GNU Prolog's lack of garbage collection hindered testing with larger instances, and we therefore ported the model also to B-Prolog, which is also a very efficient Prolog implementation and available free of charge for personal use. In all these systems, we model the rotating workforce problem as follows:

- The schedule is represented as a list of lists, and each element is a finite domain variable that denotes the shift type scheduled for this position.
- The temporal requirements are enforced via `global_cardinality/2` constraints on the columns of the schedule. In GNU Prolog, the built-in `fd_exactly/3` constraint is used instead.
- The minimal/maximal-length constraints on consecutive shifts of the same type are enforced via `automaton/3`.
- Reified constraints are used to map shifts of all types to either "work" or "day-off", and a second `automaton/3` constraint is used on these reified variables to limit the number of consecutive work and day-off shifts.
- Reified constraints are also used to express forbidden patterns. For example, if "0 4 3" is forbidden, the constraint is:

  $$X_k \texttt{ \#= } 0 \texttt{ \#}\wedge X_{k+1} \texttt{ \#= } 4 \texttt{ \#==> } X_{k+2} \texttt{ \#}\backslash\texttt{= } 3$$

  for all variables X_k, also taking into account the schedule's cyclicity. In B-Prolog, `notin/2` (negated extensional) constraints are used for better performance.

It only took a few days to implement this basic model (700 LOC, including a 50 LOC parser for instance specification files and 50 LOC for

visualisations) and to get it to run on all of the above Prolog implementations. Only built-in constraints are used in all systems. In contrast, the development of *FCS* took several months.

6.6 The automaton/3 constraint

The `automaton/3` constraint (sometimes also called `regular`), first introduced in [Pes04], constrains a list of finite domain variables to be a member of a regular language described by a finite automaton. It is a built-in constraint in (among others) SICStus Prolog and described in its manual. For GNU Prolog, we used the code shown in Figure 6.2. We simply ported our implementation from SWI-Prolog to express the constraint in terms of the `fd_relation/2` constraint, a built-in constraint in GNU Prolog. We omit the definition of the auxiliary predicates `maplist/3` and `include/3`, which are defined as in SWI-Prolog. We used the same implementation in B-Prolog with minor modifications.

```
automaton(Sigs, Ns, As) :-
        memberchk(source(Source), Ns),
        include(sink, Ns, Sinks0), maplist(arg(1), Sinks0, Sinks),
        phrase((arcs_relation(As, Relation),
                nodes_nums(Sinks, SinkNums0),
                node_num(Source, Start)), [[]-0], _),
        phrase(transitions(Sigs, Start, End), Tuples),
        maplist(fd_relation(Relation), Tuples),
        fd_domain(End, SinkNums0).

transitions([], S, S) --> [].
transitions([Sig|Sigs], S0, S) --> [[S0,Sig,S1]],
        transitions(Sigs, S1, S).

nodes_nums([], []) --> [].
nodes_nums([Node|Nodes], [Num|Nums]) --> node_num(Node, Num),
        nodes_nums(Nodes, Nums).

arcs_relation([], []) --> [].
arcs_relation([arc(S0,L,S1)|As], [[From,L,To]|Rs]) -->
        node_num(S0, From), node_num(S1, To),
        arcs_relation(As, Rs).

node_num(Node, Num), [Nodes-C] --> [Nodes0-C0],
        { (   member(N-I, Nodes0), N == Node -> Num = I, C = C0,
              Nodes = Nodes0
          ;   Num = C0, C is C0 + 1, Nodes = [Node-C0|Nodes0]
          ) }.

sink(sink(_)).
```

Figure 6.2: An implementation of `automaton/3` for GNU Prolog

6.7 Visualising the search

Analogous to Section 2.7, we show the PostScript code (Fig. 6.3) that we used to visualise the constraint solving process, tailored for the specific case of rotating workforce scheduling. Fig. 6.4 shows an example of its usage and

the resulting picture. In this case, we simply emit PostScript instructions every time a concrete shift is assigned to a specific position in the schedule. On backtracking, the position is cleared. Fig. 6.5 shows the Prolog for emitting PostScript instructions.

Again, observing the constraint solving process in real-time can give valuable hints about which other strategies might be worth trying.

```
/init { /NRows exch def /NCols exch def 5 5 translate
    600 NCols NRows max div dup scale 0 setlinewidth
    0 1 NCols { dup 0 moveto NRows lineto stroke } for
    0 1 NRows { dup 0 exch moveto NCols exch lineto stroke } for
    /Palatino-Roman .75 selectfont -1 0 translate } bind def
/s { gsave NRows exch sub translate .5 .25 moveto names exch get
    dup stringwidth pop -2 div 0 rmoveto show grestore } bind def
/c { NRows exch sub 1 1 4 copy 1 setgray rectfill
    0 setgray rectstroke } bind def
```

Figure 6.3: Rudimentary PostScript definitions for visualising the search

```
7 3 init
/names [(-) (D) (A) (N) ] def
0 1 1 s    1 1 c
0 2 1 s    1 5 1 s
2 5 2 s    2 6 2 s    3 7 3 s
```

(a)

	-			D		
				A	A	
						N

(b)

Figure 6.4: (a) PostScript instructions and (b) the resulting picture

```
animate_rows([], _).
animate_rows([Row|Rows], NumRow0) :-
        animate_cols(Row, NumRow0, 1),
        NumRow1 #= NumRow0 + 1,
        animate_rows(Rows, NumRow1).

animate_cols([], _, _).
animate_cols([V|Vs], NumRow, NumCol0) :-
        freeze(V, show_shift(NumRow,NumCol0,V)),
        NumCol1 #= NumCol0 + 1,
        animate_cols(Vs, NumRow, NumCol1).

show_shift(Row, Col, V) :- format("~w ~w ~w s\n", [V,Col,Row]).
show_shift(Row, Col, _) :- format("~w ~w c\n", [Col,Row]), false.
```

Figure 6.5: Emitting PostScript instructions for each assigned shift type

6.8 Labeling and allocation strategies

The default strategy in *CP-Rota* is to first label the (reified) work/"day-off" Boolean variables. Then, all original variables of the schedule are labeled with the "first-fail" option. We call this strategy S_1. When S_1 did not yield a solution within 1000 seconds, we used Strategy S_2, which is to label all schedule variables from left to right, trying their values from lowest to

highest. If this does not yield a solution within 1000 seconds, S_3 is used: Reified constraints are used to compute, for each column, the number of still missing shifts of each type. Processing the columns in order, we then choose the shift type that misses the *least* number of elements in that column, and assign it to a feasible variable with *smallest* domain. When S_3 also fails to find a solution within 1000 seconds, S_4 is used: It is similar to S_3, except the columns are not processed from left to right, but in descending order of their number of still missing shifts of any type.

6.9 Comparison with the commercial system FCS

To the best of our knowledge, *FCS* is a state-of-the-art commercial system for generating rotating workforce schedules. To make the performance of *CP-Rota* directly comparable to the numbers that are published in [Mus06] and other papers which refer to these numbers, we tested all instances on a Pentium 4, 1.8 GHZ, 512 MB RAM.

Table 6.1 compares the performance of *CP-Rota* with that of *FCS* on 20 real-life instances that appear in [Mus06]. We used the latest versions of GNU Prolog and SWI-Prolog when we performed these benchmarks in January 2011 for our publication. Our experiments show that these timings are also valid for more recent versions. Except where stated otherwise, timing results are from GNU Prolog. Note that while GNU Prolog is typically much faster than SWI-Prolog, instances 9 and 17 are solved faster with SWI-Prolog due to its much stronger filtering for the `global_cardinality/2` constraint. We could not port this constraint to GNU Prolog so far because our implementation of this constraint uses SWI-Prolog's attributed variables for its filtering algorithm. Once attributed variables become available in GNU Prolog, it is trivial to port this constraint to GNU Prolog as well. This will likely further improve these running times. As already mentioned, we used our own system to check answers that are emitted by GNU Prolog.

The table shows that *CP-Rota* performs competitively and complements *FCS* so that 3 more instances can now be solved. On 7 instances, *CP-Rota* outperforms *FCS* already with its default strategy S_1, the converse holds for 6 instances. Faster times are shown in bold. Note that every time one of the systems is faster, it outperforms the other by several factors.

We cannot compare our results with the solver developed by Pesant and Laporte [LP04] which was used for solving similar rotating workforce scheduling problems. According to private communication with the authors, their solver can as of yet only be used for a limited number of the problems we solve in this section due to their slightly different formulation.

It would be interesting to perform a more detailed comparison with upcoming SMT formulations ([EM13]) for this task, and to perform and publish future benchmarks for all available systems on more recent hardware. The large relative differences will likely be reproducible with any hardware.

Ex.	n	*FCS* (time in sec)	*CP-Rota* (sec)	Strategy
1	9	0.9	**0.02**	S_1
2	9	0.4	**0.02**	S_1
3	17	1.9	**0.24**	S_1
4	13	1.7	**0.03**	S_1
5	11	3.5	**0.98**	S_1
6	7	2	**0.02**	S_1
7	29	16.1	**0.07**	S_2
8	16	**124**	964	S_1
9	47	>1000s	**19**	SWI, S_4
10	27	**9.5**	>1000s	–
11	30	**367**	>1000s	–
12	20	>1000s	>1000	–
13	24	>1000s	**114**	S_1
14	13	**0.54**	940	S_1
15	64	>1000s	>1000s	–
16	29	**2.44**	216	S_1
17	33	>1000s	**18**	SWI, S_3
18	53	**2.57**	>1000s	–
19	120	>1000s	>1000s	–
20	163	>1000s	>1000s	–

Table 6.1: Comparison between *FCS* and *CP-Rota*

7 Conclusion and future work

We have presented common limitations and mistakes of several widely used constraint systems to raise awareness of these issues among authors and users of these systems. These limitations may often be due to conscious decision, eventually sacrificing correctness for performance.

Among authors of common constraint systems, there is a tendency to compete in the area of performance instead of correctness. With the material presented in this thesis, we hope to shift the balance towards giving more consideration to questions that should in our opinion enjoy high priority:

- What are the limitations of our systems, and should we strive to remove them?
- What do we guarantee in our systems?
- How can we more extensively test our systems?

In particular, we encourage authors of constraint systems to guarantee algebraic properties like commutativity and monotonicity, since they simplify reasoning about programs and are hence useful for declarative debugging and automated tests.

As we have shown in this thesis, allowing arbitrarily large integers in finite domain constraints is also very useful and yields new application opportunities for constraint solvers. In particular, uniform integer arithmetic via finite domain constraints allows us to omit the explanation of lower-level arithmetic predicates in introductory Prolog courses and leads to more general and easier to understand programs.

Guaranteeing terminating propagation is important for black-box tests, to prevent the entanglement in an infinite propagation chain when constraints are posted. As we have seen in Section 4.14.1, it is also necessary for guaranteeing that labeling always terminates.

Based on these considerations, we have presented a new CLP(FD) system that gives several strong guarantees:

- It reasons over arbitrarily large integers.
- Constraint propagation *always* terminates.
- The system is *monotonic* if the flag `clpfd_monotonic` is `true`.

To the best of our knowledge, ours is the first widely available CLP(FD) system that gives any of these guarantees.

With new domain-specific languages for reification, compactified arithmetic, parsing arithmetic expressions and propagator selection, we have declaratively expressed parts of our system whose implementation would

otherwise be difficult and error-prone. Large portions of our system are generated from these specifications. If there is any mistake in the expansion phase, it is likely to affect several parts of our system at once and is thus easier to find. These languages may be useful in other constraint systems as well and may improve their correctness and efficiency.

Systematic test cases that we have presented are useful for testing other constraint systems as well, and automated analysis of individual propagators has helped to verify several important properties.

Applying our system to rotating workforce scheduling instances further underlines the usefulness of constraint programming for this task. Since our system is entirely implemented in Prolog, we could easily port its relevant parts to other Prolog systems and benefit from their higher performance, while using our system as a reference that is more likely correct due to its simple design and due to the testing methodologies we described in Chapter 5. In addition, we could even solve some instances faster than with other implementations due to the strong filtering for several global constraints that are implemented in our system. These filtering algorithms can be easily ported to other systems as well as soon as they provide an interface to attributed variables.

Before we give an outlook of what users can expect from our constraint system in the future, we briefly look back on the development history of our system, which started in 2007, in the hope that authors of other systems can learn from the mistakes we have already made. First of all: Yes, also the CLP(FD) system we present in this thesis had several mistakes in early versions. Most of them could have been avoided if we had developed the methods we describe in Chapter 5 earlier, since each of these mistakes either led to the violation of an algebraic property like monotonicity or commutativity, or was caused by violating one of the properties we formulated and verified for individual propagators. Unfortunately, we had for several years not noticed how easily *abstract interpretation* can be applied to individual propagators to establish several important properties.

Also, we had been delaying the implementation of several global constraints like `all_distinct/1` because they seemed not amenable to readable and efficient Prolog implementations. Two important observations allowed us to efficiently implement these filtering algorithms in our system: First, as explained in Section 4.13.2, *graphs* can be represented via attributed variables, which makes many important operations on them efficiently computable. Second, DCGs can be used to implicitly thread rarely used arguments through several predicates (see Section 4.13.3). This makes the code more readable and allows for easy and virtually verbatim transcriptions of algorithms that are specified imperatively, as they often are in the literature. Black-box tests as described in Section 5.3 significantly helped us to find mistakes in the implementations of these algorithms during development.

In February 2010, Salvador Fandiño García found the most recently reported mistake in our CLP(FD) system. The mistake was that *aliased* variables were not handled correctly in the `global_cardinality/2` constraint. Since then, no mistake was reported by users, while at the same time – judging from searches on the Internet – the usage of our system among academic users and hobbyists alike has been increasing. Of course, we have further improved our system since then and implemented for example stronger filtering for several arithmetic constraints after feedback from users.

In future work, we will generalise and extend the ideas that we have presented throughout this thesis in the following ways:

1. We will devise new DSLs and use them to describe remaining parts of our system in more declarative ways. This concerns in particular the implementation of individual propagators for arithmetic operations like *addition* and *multiplication*. The language we envision for this task will not describe the *actions* a propagator has to perform to filter inconsistent domain elements. Instead, we will axiomatically describe the arithmetic *properties* of these operations. Appropriate filtering actions for each argument shall then be automatically derived from these descriptions, and the propagator code we show for example in Fig. 4.15 and Fig. 4.16 shall be *generated*. Generating more code automatically will make any mistakes of the generation phase less isolated and thus easier to find. The language of *set theory* is one declarative way to express these operations and will serve as a useful starting point for this approach.

2. We will formulate and establish – for example via *abstract interpretation* – further invariants that hold in our system. Going beyond the analysis of individual propagators (Section 5.6), we will verify properties regarding the *interaction* of different propagators. We have seen an example of such an invariant in Section 5.5: `gcc_check/1` must always be called before `gcc_global/1`.

3. We will make our system faster while retaining its declarative properties and portability to other Prolog systems. In several benchmarks, our system currently does a lot of work that turns out to be unnecessary. For example, propagators are sometimes scheduled and then deactivated while they are still in the queue, and hence the overhead for managing propagators can even outweigh the time spent on propagation itself. Deactivating unnecessary propagators is similar to removing auxiliary variables and constraints in reified expressions (see Section 4.12) and can be described in an analogous way.

In addition, we will implement and test more global constraints and apply our system on further tasks of practical relevance.

8 Bibliography

References

[AZ07] K. R. Apt and P. Zoeteweij. An analysis of arithmetic constraints on integer intervals. *Constraints*, 12(4):429–468, 2007.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic*, 1999.

[Ben86] J. Bentley. Little languages. *Communications of the ACM*, 29(8):711–21, 1986.

[But78] B. Butler. Computerized manpower scheduling. Master's thesis, University of Alberta, Canada, 1978.

[BW90] Nagraj Balakrishnan and Richard T. Wong. A network model for the rotating workforce scheduling problem. In *Networks*, volume 20, pages 25–42, 1990.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CD96a] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *J. Log. Program.*, 27(3):185–226, 1996.

[CD96b] C. H. Colbourn and J. H. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

[CHLS06] Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *Australian Conference on Artificial Intelligence*, pages 49–58, 2006.

[CM12] Mats Carlsson and Per Mildner. SICStus Prolog - the first 25 years. *TPLP*, 12(1-2):35–66, 2012.

[Col99] Charles J. Colbourn. A Steiner 2-design with an automorphism fixing exactly $r+2$ points. *J. of Comb. Designs*, 7:375–380, 1999.

[CRD12] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *TPLP*, 12(1-2):5–34, 2012.

[DAC12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *TPLP*, 12(1-2):253–282, 2012.

[DC01] D. Diaz and Ph. Codognet. Design and implementation of the GNU Prolog System. *J. of Funct. and Logic Prog.*, 2001(6), 2001.

[DL02] M. Ducassé and L. Langevine. Automated analysis of CLP(FD) program execution traces. In *CP'02*, volume 2401 of *LNCS*, 2002.

[EM13] Christoph Erkinger and Nysret Musliu. Rotating workforce scheduling as satisfiability modulo theories. *under review*, 2013.

[Ert90] M. Anton Ertl. Coroutining und Constraints in der Logik-Programmierung. Diplomarbeit, Technische Universität Wien, Austria, 1990. In German.

[Frü98] Thom W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

[FS09] Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In Ian P. Gent, editor, *CP 2009*, volume 5732 of *LNCS*, pages 367–382. Springer, 2009.

[FSC04] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: A generic graphical user interface for constraint logic programming. *Constraints*, 9(4):241–262, 2004.

[Gal85] Hervé Gallaire. Logic programming: Further developments. In *SLP*, pages 88–96, 1985.

[GMS01] Johannes Gärtner, Nysret Musliu, and Wolfgang Slany. Rota: A research project on algorithms for workforce scheduling and shift design optimisation. In *Artificial Intelligence Communications*, volume 14(2), pages 83–92, 2001.

[HBC+12] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.

[HE80] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hol92] Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *PLILP*, volume 631, pages 260–268. Springer-Verlag, 1992. LNCS 631.

[HSD98] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *J. Log. Program.*, 37(1-3):139–164, 1998.

[JL87a] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[JL87b] S. C. Johnson and M. E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155–2176, 1987.

[Lap99] G. Laporte. The art and science of designing rotating schedules. In *J. of the Op. Res. Society*, volume 50, pages 1011–1017, 1999.

[LP04] G. Laporte and G. Pesant. A general multi-shift scheduling system. In *J. of the Op. Res. Society*, volume 55/11, pages 1208–1217, 2004.

[MGS02] Nysret Musliu, Johannes Gärtner, and Wolfgang Slany. Efficient generation of rotating workforce schedules. In *Discrete Applied Mathematics*, volume 118(1-2), pages 85–98, 2002.

[MH08] L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS J. on Computing*, 20(1):143–153, 2008.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.

[MM04] Michael Mörz and Nysret Musliu. Genetic algorithm for rotating workforce scheduling. In *Proceedings of second IEEE Int. Conf. on Comp. Cybernetics, Vienna, Austria*, pages 121–126, 2004.

[MTC12] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. *CoRR*, abs/1201.0749, 2012.

[Mus06] Nysret Musliu. Heuristic methods for automatic rotating workforce scheduling. In *International Journal of Computational Intelligence Research*, volume 2, pages 309–326, 2006.

[Neu97] U. Neumerkel. Teaching Prolog and CLP (tutorial). *ICLP*, 1997.

[NHS73] J. McEwen N. Heller and W. Stenzel. Computerized scheduling of police manpower. In *St. L. Police Dep., St. Louis, MO*, 1973.

[NRS97] Ulrich Neumerkel, Christoph Rettig, and Christian Schallhart. Visualizing solutions with viewers. In *Workshop on Logic Programming Environments*, pages 43–50, 1997.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP*, pages 482–495, 2004.

[PG83] Paul Pritchard and David Gries. The seven-eleven problem. Technical Report TR83-574, Cornell University, Computer Science Department, September 1983.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[SS05] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space? *ACM Trans. Program. Lang. Syst.*, 27(3):388–425, May 2005.

[SS12] Joachim Schimpf and Kish Shen. Ecl$^{\text{i}}$ps$^{\text{e}}$ - from LP to CLP. *TPLP*, 12(1-2):127–156, 2012.

[ST08] Christian Schulte and Guido Tack. Perfect derived propagators. In *CP*, pages 571–575, 2008.

[ST13] Christian Schulte and Guido Tack. View-based propagator derivation. *Constraints*, 18(1):75–107, 2013.

[STL10] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with gecode, 2010.

[Sut63] I. E. Sutherland. SKETCHPAD: A Man-Machine Graphical Communications System. Technical Report 296, MIT, 1963.

[SW10] Andreas Schutt and Armin Wolf. A new $\mathcal{O}(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In *CP*, pages 445–459, 2010.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1 (2)*, pages 146–160, 1972.

[Vil09] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In *CP*, pages 802–816, 2009.

[Wal75] D. L. Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[WGC01] P. Weil G. Chan. Cyclical staff scheduling using constraint logic programming. In *PATAT 2000, LNCS*, volume 2079, pages 159–175, 2001.

[Zho06] Neng-Fa Zhou. Programming finite-domain constraint propagators in Action Rules. *TPLP*, 6(5):483–507, 2006.

[Zho12] Neng-Fa Zhou. The language features and architecture of B-Prolog. *TPLP*, 12(1-2):189–218, 2012.

9 About the author

Contact

E-mail	triska@dbai.tuwien.ac.at

Education

2002 – 2006	Software Engineering (Vienna University of Technology), Bachelor of Science
2006 – 2008	Computational Intelligence (Vienna University of Technology), Master of Science
since 2008	doctoral studies (Vienna University of Technology)

Work

internships	Siemens, Allianz, Generali
2007 – 2008	Vienna University of Technology, teaching assistant
since 2008	Federal Ministry of Finance, IT project management

Homepage

http://www.logic.at/prolog/

Index