Cognitive Technologies

Managing Editors: D.M. Gabbay J. Siekmann

Springer-Verlag Berlin Heidelberg GmbH

Thom Frühwirth Slim Abdennadher

Essentials of Constraint Programming

With 27 Figures

Springer

Authors

Thom Frühwirth
Universität Ulm, Fakultät für Informatik
Albert-Einstein-Allee 11, 89069 Ulm, Germany
Thom.Fruehwirth@informatik.uni-ulm.de

Slim Abdennadher
Universität München, Institut für Informatik
Oettingenstraße 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de

Managing Editors

Prof. Dov M. Gabbay
Augustus De Morgan Professor of Logic
Department of Computer Science, King's College London
Strand, London WC2R 2LS, UK

Prof. Dr. Jörg Siekmann
Forschungsbereich Deduktions- und Multiagentensysteme, DFKI
Stuhlsatzenweg 3, Geb. 43, 66123 Saarbrücken, Germany

Library of Congress Cataloging-in-Publication Data

Frühwirth, Thom, 1962–
Essentials of constraint programming / Thom Frühwirth, Slim Abdennadher.
p. cm. – (Cognitive technologies)
Includes bibliographical references and index.
ISBN 978-3-642-08712-7 ISBN 978-3-662-05138-2 (eBook)
DOI 10.1007/978-3-662-05138-2

1. Constraint programming (Computer science). I. Abdennadher, Slim, 1967–
II. Title. III. Series.
QA76.612.F78 2003 005.1'1–dc21

ACM Computing Classification (1998):
D.1.3, D.1.6, D.3.1–3, F.3.2, F.4.1, G.2.1, I.2.3, I.2.8

ISSN 1611-2482
ISBN 978-3-642-08712-7

http://www.springer.de

Originally published by Springer-Verlag Berlin Heidelberg New York in 2003
Softcover reprint of the hardcover 1st edition 2003

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera-ready by authors
Printed on acid-free paper SPIN: 10770657 45/3142 GF– 543210

Preface

The use of constraints had its scientific and commercial breakthrough in the 1990s. Programming with constraints makes it possible to model and specify problems with uncertain, incomplete information and to solve combinatorial problems, as they are abundant in industry and commerce, such as scheduling, planning, transportation, resource allocation, layout, design, and analysis.

This book is a short, concise, and complete presentation of constraint programming and reasoning, covering theoretical foundations, algorithms, implementations, examples, and applications. It is based on more than a decade of experience in teaching and research about this subject.

This book is intended primarily for graduate students, researchers, and practitioners in diverse areas of computer science and related fields, including programming languages, computational logic, symbolic computation, and artificial intelligence. The book is complemented by a web-page with teaching material, software, links, and more.

We take the reader on a step-by-step journey through the world of constraint-based programming and constraint reasoning. Feel free to join in...

Acknowledgements

Thom thanks his wife Andrea and his daughter Anna – for everything. He dedicates his contribution to the book to the memory of his mother, Grete.

Slim thanks his wife Nabila and his daughters Shirine and Amira for their ongoing support and patience.

We thank the students in the constraint programming and reasoning courses at the Ludwig Maximilian University of Munich and the University of Pisa for their motivating interest, and in particular Alexandra Bosshammer, Michael Brade, Carlo Sartiani, Luiz C. P. Albini, and Miriam Baglioni for proof reading parts of the book.

Munich, October 2002 *Thom Frühwirth and Slim Abdennadher*

Contents

1. Introduction

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Eugene C. Freuder, Inaugural issue of the *Constraints Journal*, 1997

The idea of constraint-based programming is to solve problems by simply stating constraints (conditions, properties) which must be satisfied by a solution of the problem. For example, consider a bicycle number lock. We forgot the first digit, but remember some constraints about it: The digit was an odd number, greater than 1, and not a prime number. Combining the pieces of partial information expressed by these constraints (digit, greater than 1, odd, not prime) we are able to derive that the digit we are looking for is "9".

Constraints can be considered as pieces of partial information. Constraints describe properties of unknown objects and relationships between them. Constraints are formalized as distinguished, predefined predicates in first-order predicate logic. The objects are modeled as variables.

Constraints allow for a finite representation and efficient processing of possibly infinite relations. For example, each of the two arithmetic constraints $X+Y=7$ and $X-Y=3$ admits infinitely many solutions over the integers. Taken together, these two constraints can be simplified into the solution $X=5$ and $Y=2$.

From the mid-1980's, *constraint logic programming* combined the advantages of *logic programming* and *constraint solving*. Constraint-based programming languages enjoy elegant theoretical properties, conceptual simplicity, and practical success.

In *logic programming languages*, problem-solving knowledge is stated in a declarative way by rules that define relations. A solution is searched for by applying the rules to a given problem. A fixed strategy called *resolution* is used.

In *constraint solving*, efficient special-purpose algorithms are employed to solve sub-problems expressed by constraints.

As it runs, a *constraint program* successively generates constraints. As a special program, the *constraint solver* stores, combines, and simplifies the constraints until a solution is found. The partial solutions can be used to influence the run of the program.

T. Frühwirth et al., *Essentials of Constraint Programming*

The advantages of constraint logic programming are: declarative problem modeling on a solid mathematical basis, propagation of the effects of decisions using efficient algorithms, and search for optimal solutions.

The use of constraint programming supports the complete software development process. Because of its conceptual simplicity and efficiency *executable specifications*, *rapid prototyping*, and *ease of maintainance* are achievable.

Already since the beginning of the 1990's, constraint-based programming has been commercially successful. In 1996, the world wide revenue generated by constraint technology was estimated to be on the order of 100 million dollars. The technology has proven its merits in a variety of application areas, including decision support systems for scheduling, timetabling, and resource allocation.

For example, the system Daysy performs short-term personnel planning for Lufthansa after disturbances in air traffic (delays, etc.), such that changes in the schedule and costs are minimized. Nokia uses constraints for the automatic configuration of software for mobile phones. The car manufacturer Renault has been employing the technology for short-term production planning since 1995.

Overview of the Book

This book is intended as a concise and uniform overview of the fundamentals of constraint programming: languages, constraints, algorithms, and applications.

The first part of the book discusses classes of constraint programming languages. The second part introduces types of constraints and algorithms to solve them. Both parts include examples. The third part describes three exemplary applications in some detail. In the appendix, we briefly give syntax and semantics of first-order predicate logic which constitutes the formal basis of this book.

In the first part of the book, we introduce the basic ideas behind the classes of (concurrent) constraint logic programming languages in a uniform abstract framework.

In Chap. 4, we introduce logic programming. We define syntax, operational semantics in a calculus, and declarative semantics in first-order logic. We give soundness and completeness results that explain the formal connection between operational and declarative semantics. With Prolog we briefly introduce the best known representative and classic of logic programming languages.

Step by step we extend this class of programming languages in the following chapters. We will keep the structure of presentation and emphasize the commonalities and explain the differences.

In Chap. 5, we extend logic programming by constraints, leading to constraint logic programming. In Chap. 6, constraints present themselves as a

formalism for communication and synchronization of concurrent processes. In Chap. 7, we introduce a concurrent programming language for writing constraint solvers and constraint programs called Constraint Handling Rules.

In the second part of the book, we explain what a constraint solver does and what it should do. We define the notion of constraint system and explain the principles behind constraint-solving algorithms such as variable elimination and local-consistency techniques. We introduce common constraint systems such as Boolean constraints for circuit design, terms for program analysis, linear polynomial equations for financial applications, finite domains for scheduling, and interval constraints for solving arbitrary arithmetic expressions.

Constraint Handling Rules will come in handy to specify and implement the corresponding constraint-solving algorithms at a high level of abstraction. We will analyze termination, confluence, and worst case time complexity of the algorithms. For each constraint system, we give an example of its application.

In the third part of the book, we reach the commercial practice of constraint programming: We briefly describe the market for this technology, the involved software companies, applications areas, and sample concrete projects. Then we present in more detail three applications: from timetabling to internet-based rent advice and optimal placement of senders for wireless communication.

References to related literature and a detailed index conclude the book.

Since this book concentrates on the essentials of constraint programming and reasoning, it does not address the following topics: temporal and spatial constraints, dynamic (undoable) constraints, soft (prioritized) constraints, constraint-based optimization techniques, low-level implementation techniques, programming methodology, non-logic programming languages (functional, object oriented, imperative) with constraints, and databases with constraints.

A final remark: The web pages of this book at
`http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/pisa`
contain teaching aids like slides and exercises, as well as links to programming languages, tutorials, software, further references, and more.

Part I

Constraint Programming

2. Algorithm = Logic + Control

With this statement Robert Kowalski [35] stressed the difference between *what (logic)* and *how (control)* in implementing an algorithm in a computer program. A program consists of a logical component, which specifies the knowledge of the problem, and a control component, which determines how this knowledge is used in order to solve a problem.

In conventional programming adhering to the *procedural* programming style, a problem is solved by executing a sequence of program instructions. There is no separation between logic and control of a program.

The central idea of *declarative* programming, and in particular logic-based programming, is that the programmer only states (the logic of) the problem without having to worry about the control. A user of a program is not interested in how the solution to the problem is found (*procedural aspect*), but what the solution is (*declarative aspect*).

In *logic-based programming*, the problem and the program are both stated as specific logical formulae. Problem solving in this context means under which conditions the formula, which represents the problem, follows from the knowledge and the assumptions which are expressed in the program. Since the sequence of statements that should be executed to solve a problem is not given explicitly, declarative programming must include a predefined, general algorithm for control.

For example, consider sorting a list in ascending order. Procedurally, one can proceed as follows: go through the list and compare neighboring elements in the list. If the first element is greater than the second element, swap these two elements. Repeat the swaps for the complete list until the list does not change anymore.

From the declarative point of view, we know that S is the sorting of list L if S is a permutation of L and if the elements of S are arranged in ascending order. In the same way we can define permutations and orders. This knowledge forms our assumptions. The formula for stating the problem reads as follows: X is the sorted list L. For which value of X does this formula follow from the assumptions?

In logic-based programming, problems are stated logically as formulae, and these formulae can be regarded as *specifications*. Thus, logic-based programming languages are well suited to *rapid prototyping*, i.e., the fast devel-

T. Frühwirth et al., *Essentials of Constraint Programming*

opment of prototypes on the basis of incomplete, evolving specifications. The declarativity of logic-based programming also favorably affects debugging, testing, and the analysis of programs.

3. Preliminaries of Syntax and Semantics

In general, the term *logic-based programming languages* refers to computer languages that make (a subset of) a logic executable. There are languages based on non-classical logics like higher order logic and modal or temporal logic, but, like in this book, *logic programming* is mostly used as a synonym for computing in classical first-order predicate logic. A brief review of first-order logic and its notation as it forms the basis of this book can be found in Appendix A.

In this book, we will consider several classes of *constraint logic programming languages*. We will present them in a uniform and abstract way. We will give their *abstract syntax* and two kinds of semantics.

The syntax definitions will use *extended Backus-Naur form* (EBNF) notation. In EBNF, a production rule is of the form

$$\mathit{Name}\colon\ \ G, H ::= A \mid B,\ \ \mathit{Condition}$$

where capital letters stand for syntactical entities. The rule defines the symbols G and H (possibly with indices, e.g., G_1, H_1 or primed, e.g., G', H') that have a name *Name*. G and H can be of the form A or of the form B, provided the condition *Condition* holds.

A suitable notion of *state transition systems* that are able to encode calculi will define the *operational semantics (procedural semantics)* of each class of languages. The *declarative semantics* refers to the logical reading of a program as a set of formulae of first-order predicate logic.

We will use small and simple examples, including one running example, to illustrate the main computational aspects of the programming languages.

Finally, we present results relating the operational and declarative semantics of logic-based programming languages. On one hand, everything that is derivable, should also logically follow from the program (*soundness*), on the other hand, everything that follows should also be derivable (*completeness*).

In general, the operational semantics of a logic-based programming language is based on variations of the *resolution method* (Sect. A.2.5) that is concretized as a (logical) calculus.

The execution of a program is seen as a sequence of state transitions. A *state transition system* describes how we can proceed from one state of derivation to the next, and how the derivation starts and ends.

T. Frühwirth et al., *Essentials of Constraint Programming*

Definition 3.0.1. *A* state transition system *is a tuple* $(S, \mapsto)$*, where* S *is a set of states and where* $\mapsto$ *is a binary relation over states, called* transition relation.
There are two distinguished subsets of S*, the* initial *and the* final states.
A sequence of states and transitions $S_1 \mapsto S_2 \mapsto \ldots \mapsto S_n$ *is called a* derivation (computation) *of* S_n *starting from* S*.* $S \mapsto^* S'$ *denotes the reflexive-transitive closure of* $\mapsto$*.*

Reduction is used as a synonym for transition. The transition relation $\mapsto$ is usually defined by *transition rules* of the form

$$\begin{array}{ll} \text{If} & \textit{Condition} \\ \text{then} & S \mapsto S'. \end{array}$$

The notation means that a *(state) transition (reduction, derivation step)* from a state S to a state S' is possible if the condition *Condition* holds.

It is straightforward to concretize a calculus into a state transition system.

Example 3.0.1. For the *resolution calculus* defined in Appendix A, a state is a set of clauses. An initial state consists of the clauses representing the premises and the negated conclusion, while each clause containing the empty clause is a final state. The transition operation for the resolution calculus goes from a clause set S to the clause set $S \cup \{F\}$, where F is a resolvent or factor of members of S.

In a state transition system there are states that we would like to consider equivalent for the purpose of computation. Instead of modeling these equivalences with additional transition rules, they are abstracted away using a *congruence*. In short, the idea is that congruent states are considered the same, they are not worth distinguishing. Formally, a congruence has to be an equivalence relation.

Definition 3.0.2. *An* equivalence relation $\equiv$ *is a reflexive, symmetric, and transitive relation:*

(Reflexivity) $A \equiv A$
(Symmetry) *If* $A \equiv B$ *then* $B \equiv A$
(Transitivity) *If* $A \equiv B$ *and* $B \equiv C$ *then* $A \equiv C$

We are now ready to define the class of transition systems that we will use to define the operational semantics of constraint programming languages.

Definition 3.0.3. *A* (logical) calculus *is a triple* $(\Sigma, \equiv, T)$*, where:*

- Σ *is a signature for a first-order logic language.*
- $\equiv$ *is a congruence on states.*
- $T = (S, \mapsto)$ *is a transition system where the states* S *represent logical expressions over the signature* Σ*.*

The actual *congruence* (Fig. 3.1) that we will use for all classes of constraint languages expresses that the order of atoms in a conjunction does not matter. Also, $\top$ in a conjunction is redundant and $\bot$ makes all other conjunctions redundant.

Commutativity:	$G_1 \wedge G_2$	$\equiv$	$G_2 \wedge G_1$
Associativity:	$G_1 \wedge (G_2 \wedge G_3)$	$\equiv$	$(G_1 \wedge G_2) \wedge G_3$
Identity:	$G \wedge \top$	$\equiv$	G
Absorption:	$G \wedge \bot$	$\equiv$	$\bot$

Fig. 3.1. Congruence

We are now ready to consider the first class of constraint logic programming languages. It does not yet contain an explicit notion of constraints but forms the basis for the following classes of constraint languages.

4. Logic Programming

The first and most popular logic programming (LP) language is *Prolog*. It is still the basis of most constraint programming languages. Prolog's origins can be traced back to early work in automated theorem proving and planning (Fig. 4.1). Prolog was developed in the early 1970's, first independently and then jointly by Alain Colmerauer (Marseille) and Robert Kowalski (Edinburgh). Then David Warren (London) defined the *Warren Abstract Machine* (WAM) that lead to an efficient implementation of Prolog. The ideas behind the WAM strongly influenced the implementation of more recent languages like Java.

1964	J.A. Robinson, Resolution calculus
1972	A. Colmerauer, U. Marseille, and R. Kowalski, IC London, Prolog
1975	D.H.D. Warren, IC London, Efficient Prolog, WAM compiler
1982-1994	Fifth-Generation Computing Project, Japan; EC Esprit-Projects
1986-90's	Borlands Turbo-Prolog, Pascal+Prolog hybrid
1982	A. Colmerauer, Prolog II, U. Marseille, equality constraints
1984	Eclipse Prolog, ECRC Munich, later IC-PARC London
1985	Sicstus Prolog, Swedish Insitute of Computer Science (SICS)
1996	ISO Prolog standard

Fig. 4.1. History of logic programming

The high time for Prolog was from 1982 to 1994, when a concurrent version of the language was chosen as the basis of the ambitious Japanese fifth-generation computing project that tried to bring artificial intelligence applications into everyday life. Today, Prolog is mainly used commercially in expert systems. It is often the language of choice in implementing prototypes for research in artificial intelligence and computational logic, symbolic computation, and programming languages.

In the following, we will introduce the concepts behind Prolog and similar logic-based languages.

T. Frühwirth et al., *Essentials of Constraint Programming*

4.1 LP Calculus

We start by giving the abstract syntax of this class of languages and then move on to the operational semantics.

4.1.1 Syntax

The LP syntax is given in the following definitions and summarized in Fig. 4.2. The *signature* of the *LP logical calculus* contains arbitrary function and predicate symbols (Appendix A).

Definition 4.1.1. *A* goal *is either* $\top$ *(top) or* $\bot$ *(bottom), or an atom or a conjunction of goals.*

$\top$ is also called *empty goal.*

Definition 4.1.2. *A* (Horn) clause *is of the form* $A \leftarrow G$*, where* A *is an atom and* G *is a goal. We call* A *the* head *and* G *the* body *of the clause. Clauses of the form* $G \leftarrow \top$ *are called* facts*, all others are called* rules.
A (logic) program *is a finite set of Horn clauses.*

A predicate symbol is *defined* in a program if it occurs in the head of a clause.

Atom:	A, B	::=	$p(t_1, \ldots, t_n),\ n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid A \mid G \wedge H$
Clause:	K	::=	$A \leftarrow G$
Program:	P	::=	$K_1 \ldots K_m,\ m \geq 0$

Fig. 4.2. LP syntax

In the *LP logical calculus*, states are pairs consisting of a goal and a substitution. Intuitively, one starts from a goal that represents the given problem and, by applying the transition rules of the LP calculus, one tries to end with a substitution that represents the solution to the problem. However, not every final state represents a solution. We distinguish between successful and failed final states, and only the successful states are of interest.

Definition 4.1.3. *A* state *is a pair of the form* $<G, \theta>$*, where* G *is a goal and* θ *is a substitution. An* initial state *is a state of the form* $<G, \epsilon>$*, where* ϵ *is the identity substitution.*
A state is called successful final state *if it is of the form* $<\top, \theta>$*. It is called* failed final state *if it is of the form* $<\bot, \epsilon>$*.*

The notions of success and failure are lifted to derivations and goals.

Definition 4.1.4. *A derivation is* successful *if its final state is successful. A derivation is* failed *if its final state is failed. A derivation is* infinite *if it does not have a final state. A goal G is* successful *if it has a successful derivation starting with $<G, \epsilon>$. A goal G is* finitely failed *if it has only failed derivations starting with $<G, \epsilon>$.*

States have a logical reading.

Definition 4.1.5. *Let $<H, \theta>$ be a state occurring in a derivation starting with $<G, \epsilon>$. The formula $\exists \bar{x} H\theta$ is the* logical reading *of $<H, \theta>$, where $\bar{x}$ stands for the variables which occur in $H\theta$ but not in G.*

What we are really interested in is the substitution that occurs in a successful final state.

Definition 4.1.6. *A substitution θ is the* (computed) answer of a goal G *if there exists a successful derivation $<G, \epsilon> \mapsto^* <\top, \theta>$.*

In that context, G is also called *initial goal* or *query*.

4.1.2 Operational Semantics

Given a logic program P, we define the transition relation $\mapsto$ by introducing two reduction rules (Fig. 4.3). The main one is the transition **Unfold**. Only if it is not possible, the second transition **Failure** is used. **Unfold** basically corresponds to the resolution rule of the resolution calculus (Sect. A.2.5).

Unfold	
If	$(B \leftarrow H)$ is a fresh variant of a clause in P
and	β is the most general unifier of B and $A\theta$
then	$<A \wedge G, \theta> \mapsto <H \wedge G, \theta\beta>$
Failure	
If	there is no clause $(B \leftarrow H)$ in P
with	a unifier of B and $A\theta$
then	$<A \wedge G, \theta> \mapsto <\bot, \epsilon>$

Fig. 4.3. LP transition rules

An **Unfold** transition starting from a state $<G, \theta>$ is possible if, for some fresh variant of a clause $B \leftarrow H$ in the given program P and some atom A in the goal G, the head B and $A\theta$ are unifiable with the substitution β. In the resulting state, A is replaced by H and the substitutions θ and β are composed. A *fresh variant* of a clause is a renaming of this clause with variables that do not previously occur in P. The **Failure** transition applies when no clause can be found for the atom A for which **Unfold** is possible.

Non-determinism

Given a state $<A \wedge G, \theta>$, there are two degrees of *non-determinism* in the calculus when we want to reduce it to another state:

- Any atom in the conjunction $A \wedge G$ can be chosen as the atom A according to the congruence defined on states.
- Any clause $(B \leftarrow H)$ in P for which B and $A\theta$ are unifiable can be chosen.

Treating non-determinism we would like to make sure that all possible successful final states can be computed (completeness). While clause selection determines the computed answer of a derivation, the selection of the atom only affects the length of the derivation (infinitely in the worst case). Therefore, clause selection exhibits a *don't-know non-determinism*, while atom selection exhibits a *don't-care non-determinism*.

Both kinds of non-determinism are implicit in the LP calculus. Don't-care non-determinism for atom selection is expressed by the congruence. Don't-know non-determinism for clause selection can be made explicit in a refined version of **Unfold** (Fig. 4.4), where $\dot{\vee}$ denotes choice between states. This

UnfoldSplit	
If	$(B_1 \leftarrow H_1), \ldots, (B_n \leftarrow H_n)$ are fresh variants of all those clauses in P for which B_i $(1 \leq i \leq n)$ is unifiable with $A\theta$
and	β_i is the most general unifier of B_i and $A\theta$ $(1 \leq i \leq n)$
then	$<A \wedge G, \theta> \mapsto <H_1 \wedge G, \theta\beta_1> \dot{\vee} \ \ldots \ \dot{\vee} <H_n \wedge G, \theta\beta_n>$

Fig. 4.4. LP **Unfold** transition rule with case splitting

modified transition applies to trees of states rather than sequences of states. The root of the tree is the initial state. Nodes represent states. Every node has children according to an application of **UnfoldSplit**. Leaves are final states. We call such a tree a *search tree*, since we have to search for successful computations using the states resulting from an application of **UnfoldSplit**.

An implementation of the LP calculus has to fix a *selection strategy* for clauses and atoms that embodies the implicit control component. According to the type of non-determinism, one chooses an atom and then makes sure to try derivations with all possible clauses.

For conceptual simplicity and efficiency, usually *selection strategy* is fixed and based on the textual order of clauses and atoms in a program.

In a goal, the atoms are selected from *left to right*. For the left most atom, clauses are tried in textual order. If the selected clause leads to failure (finitely failed final state), the selection and its consequences are undone and the next clause is tried.

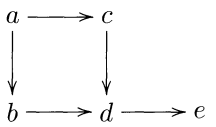
More precisely, *(chronological) backtracking (backtrack search)* is employed, which means that always the last clause selection is undone that

still admits another choice for selecting a clause. The advantage of this strategy is that undoing the last clause choice requires only a minimal change, while most of the computation (everything before the clause choice) can be kept.

This selection strategy corresponds to a left-to-right, depth-first exploration of the search tree that is associated with a computation as defined by the transition **UnfoldSplit**. It is also called *SLD resolution*. This strategy can be efficiently implemented using a stack-based approach, since, at any point in the computation, it suffices to store the current computation path instead of the overall search tree.

While simple and time and space efficient, this strategy has the disadvantage that it can get trapped in infinite derivations. A strategy like breadth-first search would avoid this, but is far too inefficient to be implemented as the basis of control for logic programming.

Example 4.1.1. To illustrate the LP calculus, we want to examine accessibility in a directed graph with the help of a logic program.



The edges of the graph are given by facts defining the predicate `egde/2`.

```
edge(a,b) ← ⊤          (e1)
edge(a,c) ← ⊤          (e2)
edge(b,d) ← ⊤          (e3)
edge(c,d) ← ⊤          (e4)
edge(d,e) ← ⊤          (e5)
```

The direct and indirect paths between the nodes of the graph are given by rules defining the predicate `path/2`.

```
path(Start,End) ←                          (p1)
     edge(Start,End)

path(Start,End) ←                          (p2)
     edge(Start,Node) ∧ path(Node,End)
```

The rule *p1* says that there is a path from node `Start` to node `End` if there is an edge from `Start` to `End`. The recursive rule *p2* says that there is a path from `Start` to `End` if there is an edge from `Start` to `Node` and a path from `Node` to `End`.

The goal `path(b,Y)` computes all nodes `Y` accessible from node `b`. If we select the first rule of the predicate `path` *p1* and a suitable edge, then the following derivation is possible:

$$
\begin{array}{ll}
 & \langle \texttt{path(b,Y)}, \varepsilon \rangle \\
\mapsto_{\textbf{Unfold}\ (p1)} & \langle \texttt{edge(S,E)}, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{Y}\} \rangle \\
\mapsto_{\textbf{Unfold}\ (e3)} & \langle \top, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{d}, \texttt{Y} \leftarrow \texttt{d}\} \rangle
\end{array}
$$

Note that for readability, we only show the substitutions that involve variables of the query and we rename variables in order to avoid the introduction of new variables.

A computed answer for the above program and the goal `path(b,Y)` is $\theta = \{\texttt{Y} \leftarrow \texttt{d}\}$, which means that node `d` is accessible from node `b`. The search tree for this goal, in Fig. 4.5 shows this successful derivation and also paths of other derivations.

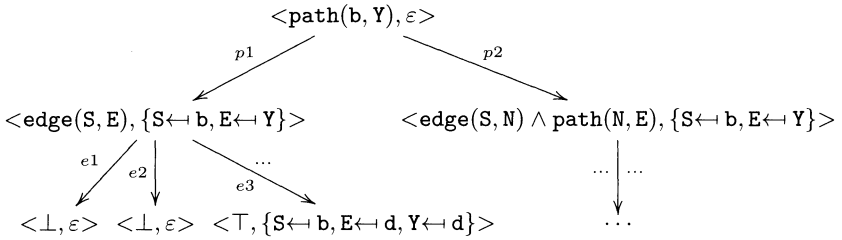


Fig. 4.5. Partial search tree for the goal `path(b,Y)`

If the second rule *p2* for `path` is selected instead, we have the derivation:

$$
\begin{array}{ll}
 & \langle \texttt{path(b,Y)}, \varepsilon \rangle \\
\mapsto_{\textbf{Unfold}\ (p2)} & \langle \texttt{edge(S,N)} \wedge \texttt{path(N,E)}, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{Y}\} \rangle \\
\mapsto_{\textbf{Unfold}\ (e3)} & \langle \texttt{path(N,E)}, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{Y}, \texttt{N} \leftarrow \texttt{d}\} \rangle \\
\mapsto_{\textbf{Unfold}\ (p1)} & \langle \texttt{edge(N,E)}, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{Y}, \texttt{N} \leftarrow \texttt{d}\} \rangle \\
\mapsto_{\textbf{Unfold}\ (e5)} & \langle \top, \{\texttt{S} \leftarrow \texttt{b}, \texttt{E} \leftarrow \texttt{e}, \texttt{N} \leftarrow \texttt{d}, \texttt{Y} \leftarrow \texttt{e}\} \rangle
\end{array}
$$

Hence, a second computed answer is $\sigma = \{\texttt{Y} \leftarrow \texttt{e}\}$.

Consider now the question whether there exists a path between the nodes `f` and `g`, i.e., the goal `path(f,g)`. With the first rule we get:

$$
\begin{array}{ll}
 & \langle \texttt{path(f,g)}, \varepsilon \rangle \\
\mapsto_{\textbf{Unfold}\ (p1)} & \langle \texttt{edge(S,E)}, \{\texttt{S} \leftarrow \texttt{f}, \texttt{E} \leftarrow \texttt{g}\} \rangle \\
\mapsto_{\textbf{Failure}} & \langle \bot, \varepsilon \rangle
\end{array}
$$

Selecting the second rule *p2* for `path` also leads to a failed derivation, provided the goal `edge(S,N)` is selected for reduction in the second derivation step. If, however, the goal `path(N,E)` is selected and the second rule *p2* for `path` is selected, we have the derivation:

$$
\begin{array}{ll}
 & \langle \texttt{path(f,g)}, \varepsilon \rangle \\
\mapsto_{\textbf{Unfold}\ (P2)} & \langle \texttt{edge(S,N)} \wedge \texttt{path(N,E)}, \{\texttt{S} \leftarrow \texttt{f}, \texttt{E} \leftarrow \texttt{g}\} \rangle \\
\mapsto_{\textbf{Unfold}\ (p2)} & \langle \texttt{edge(S,N)} \wedge \texttt{edge(N,N1)} \wedge \texttt{path(N1,E)}, \{\texttt{S} \leftarrow \texttt{f}, \texttt{E} \leftarrow \texttt{g}\} \rangle
\end{array}
$$

Repeated selection of the new goal for `path` results in an infinite derivation.

4.2 Declarative Semantics

The declarative semantics gives a *logical reading (meaning)* to a program as formulae.

A Horn clause $(A \leftarrow G)$ can be understood as an implication $(G \rightarrow A)$ which corresponds to a definite clause. The logical reading of a program P is then the universal closure of the conjunction of the clauses of P, denoted by $P^{\rightarrow}$.

However, with this simple declarative semantics only positive information can be derived. One would expect that failure of a goal is reflected in the declarative semantics in that the negation of the goal follows from the logical reading of the program.

Example 4.2.1. Let P be the logic program from example 4.1.1 and let G be the goal `path(f,g)`. This goal does not have a successful derivation. On the other hand, in the logical reading of the program, nothing can be concluded:

$$P^{\rightarrow} \not\models \texttt{path(f,g)} \quad \text{and} \quad P^{\rightarrow} \not\models \neg\texttt{path(f,g)}$$

However, if we "complete" $P^{\rightarrow}$ in such a way that it not only contains the necessary conditions (implications), but also the corresponding sufficient conditions (implications in the other direction), then $\neg$`path(f,g)` is a logical consequence from such a logical reading of the program.

The improved logical reading of a logic program P is given by its *Clark's completion* [12].

Definition 4.2.1. *Let P be a logic program.* Clark's completion *of P is the set of formulae $P^{\leftrightarrow} \cup CET$.*

We have to define the completion $P^{\leftrightarrow}$ and the theory CET (Clark's Equality Theory).

Definition 4.2.2. *The* completion *of P, denoted by $P^{\leftrightarrow}$, is defined as follows: For each predicate p in P with arity n, defined by the clauses*

$$\begin{array}{ll} p(\bar{t}_1) & \leftarrow G_1 \\ \vdots & \vdots \\ p(\bar{t}_m) & \leftarrow G_m, \end{array}$$

we add to $P^{\leftrightarrow}$ the formula

$$\forall \bar{x}\ (p(\bar{x}) \leftrightarrow \exists \bar{y}_1\ (\bar{t}_1 \doteq \bar{x} \wedge G_1) \quad \vee$$
$$\vdots \quad \vee$$
$$\exists \bar{y}_m\ (\bar{t}_m \doteq \bar{x} \wedge G_m)),$$

where $\bar{x}$ stands for a sequence of n pairwise distinct fresh variables which do not occur in the clauses, the $\bar{t}_i$ stand for a sequence of n terms, and the $\bar{y}_i$ for the sequence of the variables occurring in G_i and t_i.

For each predicate symbol p, which is mentioned in P but not defined, the formula

$$\forall \bar{x}\ \neg p(\bar{x})$$

is added to $P^{\leftrightarrow}$.

Completion uses syntactic equality $\doteq$. It is a predefined binary predicate symbol and must be defined, too.

Definition 4.2.3. *Let Σ be a signature with infinitely many function symbols, including at least one constant. Then* CET *is the set of the universal closure of the formulae:*

Reflexivity:
$(\top \rightarrow x \doteq x)$
Symmetry:
$(x \doteq y \rightarrow y \doteq x)$
Transitivity:
$(x \doteq y \wedge y \doteq z \rightarrow x \doteq z)$
Compatibility:
$(x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \rightarrow f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n))$
Decomposition:
$(f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n) \rightarrow x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n)$
Contradiction (Clash):
$(f(x_1, \ldots, x_n) \doteq g(y_1, \ldots, y_m) \rightarrow \bot)$ *if $f \neq g$ or $n \neq m$*
Acyclicity:
$(x \doteq t \rightarrow \bot)$ *if t is function term and x appears in t*

The first three axioms say that $\doteq$ is an equivalence relation, the next three axioms deal with function symbols, namely, the construction and deconstruction of function terms. The last two axioms define when an equation is unsatisfiable: If two function terms with different function symbols are equated or if one side of the equation is properly contained in the other side of the equation. These formulae are actually *formula schemes*, because we need one instance of the formulae for each function symbol of the signature.

4.3 Soundness and Completeness

We present results relating the operational and declarative semantics of LP. On one hand, everything that is derivable should also logically follow from

the program (*soundness*), on the other hand, everything that follows should also be derivable (*completeness*).

In the theorems, we distinguish between successful and failed derivations [8].

Theorem 4.3.1 (Soundness and Completeness of successful derivations). *Let P be a logic program, G be a goal, and θ be a substitution.*

- **Soundness:** *If θ is a computed answer of G, then $P^{\leftrightarrow} \cup CET \models \forall G\theta$.*
- **Completeness:** *If $P^{\leftrightarrow} \cup CET \models \forall G\theta$, then a computed answer σ of G exists, such that $\theta = \sigma\beta$.*

The relationship $\theta = \sigma\beta$ means that the computed answer can be more general. This is because an answer need not instantiate all variables and any instance of it also follows from the logical reading of the program.

The results on failed derivations require that we avoid trivial cases of non-termination.

Definition 4.3.1. *A derivation is* fair *if it either fails or if each atom appearing in a derivation is selected after finitely many reductions.*

In other words, given a state, there is no atom that is ignored forever, i.e., never selected.

Theorem 4.3.2 (Soundness and Completeness of failed derivations). *Let P be a logic program and G be a goal. Any fair derivation starting with $<G, \epsilon>$ fails finitely if and only if $P^{\leftrightarrow} \cup CET \models \neg\exists G$.*

Such a result on failed derivations would not exist without Clark's completion of a program. Note that SLD resolution does not admit fair derivations in any case, since always the left most atom of a state is chosen for **Unfold**.

5. Constraint Logic Programming

Constraint logic programming (CLP) was developed in the mid-1980's as a natural combination of two declarative paradigms: constraint solving and logic programming (Fig. 5.1). This makes constraint logic programs more expressive, flexible, and in general more efficient than logic programs.

1963	I. Sutherland, Sketchpad, graphic system for geometric drawing
1970	U. Montanari, Pisa, Constraint networks
1970	R.E. Fikes, REF-ARF, language for integer linear equations
1972	A. Colmerauer, U. Marseille, and R. Kowalski, IC London, Prolog
1977	A.K. Mackworth, Constraint networks algorithms
1978	J.-L. Lauriere, Alice, language for combinatorial problems
1979	A. Borning, Thinglab, interactive graphics
1980	G.L. Steele, Constraints, first constraint-based language, in LISP
1982	A. Colmerauer, Prolog II, U. Marseille, equality constraints
1984	Eclipse Prolog, ECRC Munich, later IC-PARC London
1985	Sicstus Prolog, Swedish Insitute of Computer Science (SICS)
1987	H. Ait-Kaci, U. Austin, Life, equality constraints
1987	J. Jaffar and J.L. Lassez, CLP(X) - Scheme, Monash U. Melbourne
1987	J. Jaffar, CLP($\Re$), Monash U. Melbourne, linear polynomials
1988	P. v. Hentenryck, CHIP, ECRC Munich, finite domains, Booleans
1988	P. Voda, Trilogy, Vancouver, integer arithmetics
1988	W. Older, BNR-Prolog, Bell-Northern Research Ottawa, intervals
1988	A. Aiba, CAL, ICOT Tokyo, non-linear equation systems
1988	W. Leler, Bertrand, term rewriting for defining constraints
1988	A. Colmerauer, Prolog III, U. Marseille, list constraints and more

Fig. 5.1. Early history of constraint-based programming

Constraint satisfaction problems (CSP) over finite domains were already investigated in the 1970's within the context of artificial intelligence. In general, a constraint problem consists of a set of variables and constraints. Constraints are predicates which express properties of variables, as well as relations between variables. A *solution* of a constraint *problem* is a *valuation* of the variables with values such that all constraints are satisfied.

At the end of the 1970's constraints started to be integrated in tools and programming languages. At the same time, efforts were made to make logic programming (LP) more declarative (with a flexible selection strategy),

T. Frühwirth et al., *Essentials of Constraint Programming*

faster (with improved search), and more general (with extended equality). A flexible selection strategy would mean that handling of certain atoms can be delayed, thus getting away from the fixed left-to-right order. Improved search would mean to detect failing derivations earlier.

Extending equality would mean to consider interpreted function symbols. For example, the term 1+2 should be equivalent to 3 by interpreting + as addition. Consequently, syntactic equality is generalized to an equation, which must be solved.

These equations can be understood as constraints. These constraints are handled by a special algorithm implemented in a *constraint solver*. They are delayed until enough information in the form of other constraints is available in order to simplify and solve them. If constraints become inconsistent, the current derivation fails. For example, the constraint problem $X-Y=3 \wedge X+Y=7$ will lead to the solution $X=5 \wedge Y=2$. The constraint goal $X<Y \wedge Y<X$ fails without the need to know values for the variables.

Obviously, the easiest way for constraint solving is to generate a possible solution, i.e., enumerate all possible values for the variables, and then to test if the constraints are satisfied. This is the so-called *generate-and-test* methodology used in LP. Unfortunately, this methodology is impractical in most cases. The problem is that this method only uses the constraints in a passive manner, to test the result of applying values, rather than using them to infer values that form a solution. In the so-called *constraint-and-generate* methodology, first the constraints are applied to reduce the search space (i.e., the number of possible solutions) and then (if needed) a solution is generated. For example the solution of the following problem $X \in \{1,2\} \wedge Y \in \{1,2\} \wedge Z \in \{1,2\} \wedge X=Y \wedge X \neq Z \wedge Y>Z$ is found after seven choices (i.e., after testing seven possible solutions) using the generate-and-test method, contrasting with the constraint-and-generate methodology where the solution can be found without making any choice, i.e., the constraint $Y>Z$ determines the values of Y and Z to be 2 and 1, respectively. Now, the constraint $X=Y$ propagates the information that $X=2$ and the constraint $X \neq Z$ remains satisfied.

In 1982, Colmerauers LP language Prolog II already extended unification by treating infinite, cyclic terms (rational trees). In the second half of the 1980's the first CLP languages, CLP($\Re$), CHIP and Prolog III were developed. In 1987, Jaffar and Lassez introduced the CLP(X) scheme that forms the basis for describing languages independent of the constraint system they use.

CLP($\Re$) offered a clean, declarative solution for treating arithmetic expressions in LP languages by introducing equations between linear arithmetic expressions over floating point numbers. In Prolog III, not only rational trees but also linear equations existed - but over rational numbers. In CHIP, constraints over finite domains were integrated into an LP language, as well as constraints from Boolean algebra.

CLP languages combine the advantages of LP languages (declarative, for arbitrary predicates, non-deterministic) with those of constraint solvers (declarative, efficient for special predicates, deterministic). The combination of search with solving constraints is particularly useful. Combinatorial problems can be tackled, which usually have exponential complexity, i.e., with increasing size they become practically unsolvable.

5.1 CLP Calculus

The CLP calculus is a generalization of the LP calculus. It replaces the unification for treating syntactic equality by the more general *constraint solving* for handling constraints.

5.1.1 Syntax

The only difference between CLP syntax and LP syntax is the fact that constraints are introduced, which can occur in goals and therefore in the body of a clause.

In the signature of the CLP calculus, the set of predicate symbols is augmented with *constraint symbols*. We introduce two constraint symbols *true* and *false* that have the same meaning as $\top$ and $\bot$, respectively. Furthermore, we assume that the constraints include $\doteq$. The set of constraint symbols is defined by a consistent first-order constraint theory (CT) and handled by a predefined, given constraint solver. In particular, CT defines $\doteq$ as the syntactic equality over Herbrand terms by including CET. Constraints are discussed in detail in Part II of this book.

Definition 5.1.1. *An* atom *is an expression of the form* $p(t_1, \ldots, t_n)$*, where* p *is an* n*-ary predicate symbol and* $t_1, \ldots, t_n$ *are terms.*
An atomic constraint *is an expression of the form* $c(t_1, \ldots, t_n)$*, where* c *is an* n*-ary constraint symbol and* $t_1, \ldots, t_n$ *are terms. A* constraint *is either an atomic constraint or a conjunction of constraints.*
A goal *is either* $\top$ *(top) or* $\bot$ *(bottom) or an atom or an atomic constraint or a conjunction of goals.*

Clauses for CLP are defined in the same way as for LP, except that their bodies now can contain constraints.

Definition 5.1.2. *A* (CL) clause *is of the form* $A \leftarrow G$*, where* A *is an atom and* G *is a goal.*
A CL program *is a finite set of CL clauses.*

These syntax definitions are summarized in Fig. 5.2.

In the following, we present the operational semantics of CLP as a state transition system. We first define states. As in LP, they have two components.

Atom:	A, B	::=	$p(t_1, \ldots, t_n),\ n \geq 0$
Constraint:	C, D	::=	$c(t_1, \ldots, t_n) \mid C \wedge D,\ n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid A \mid C \mid G \wedge H$
CL Clause:	K	::=	$A \leftarrow G$
CL Program:	P	::=	$K_1 \ldots K_m,\ m \geq 0$

Fig. 5.2. CLP syntax

The first component contains the constraints and the atoms that remain to be solved, and the other contains the constraints accumulated and simplified so far (whereas in LP we had substitutions).

Definition 5.1.3. *A* state *is a pair* $<G, C>$*, where* G *is a goal and* C *is a constraint.* G *is the* goal (store) *and* C *is the* (constraint) store.
An initial state *is a state of the form* $<G, \text{true}>$.
A successful final state *is a state of the form* $<\top, C>$ *and* C *is different from* false. *A state is a* failed final state *if it is of the form* $<G, \text{false}>$.

Note that $<\top, \mathit{false}>$ is a failed final state.

From now on only essential definitions for the calculus are given explicitly. All other definitions are to be taken from the figures or from a preceding calculus. In particular, the definitions for successful and failed derivations and goals from the LP calculus still apply.

5.1.2 Operational Semantics

The transition rules of the CLP calculus (Fig. 5.3) are chosen in order to stress the commonalities with the LP calculus. The transition rules **Unfold** and **Failure** are generalizations of those from LP. Constraints will be handled by the additional transition rule **Solve**.

The **Unfold** transition behaves as follows: If the equation between head of the clause and selected atom is consistent together with the current constraint store, then the atom is replaced by the body of the clause and the equality is added to the constraint store. Otherwise the derivation fails by applying the transition rule **Failure**.

The transition **Unfold** of the LP calculus computes the most general unifier between the head B of clause and the atom A in the context of substitution θ and adds the unifier to θ. In the CLP calculus, we impose an equality constraint between the B and A in the context of the constraint store C and add the equality constraint to the store C.

Note that strictly speaking, $B \doteq A$ is incorrect, since equality cannot be applied to atoms. However, we use $B \doteq A$ to denote the pairwise equating of the arguments of B and A. If B is of the form $p(t_1, \ldots, t_n)$ and if A is of the form $p(s_1, \ldots, s_n)$, then $B \doteq A$ stands for $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$.

The **Solve** transition behaves as follows. If the selected goal is a constraint, then it will be removed from the goal store and added to the con-

Unfold	
If	$(B \leftarrow H)$ is a fresh variant of a clause in P
and	$CT \models \exists\, ((B \doteq A) \wedge C)$
then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge C>$
Failure	
If	there is no clause $(B \leftarrow H)$ in P
with	$CT \models \exists\, ((B \doteq A) \wedge C)$
then	$<A \wedge G, C> \mapsto <\bot, \mathit{false}>$
Solve	
If	$CT \models \forall\, ((C \wedge D_1) \leftrightarrow D_2)$
then	$<C \wedge G, D_1> \mapsto <G, D_2>$

Fig. 5.3. CLP transition rules

straint store. Hereby the constraint store is simplified. The form of simplification depends on the constraint system and its constraint solver. It is tried to simplify inconsistent constraints to *false*. Thus, a failed final state can also be reached using the transition rule **Solve**. Note that by definition, a constraint need not be atomic, it can also be a conjunction of constraints.

Comparison with LP. While in LP we just accumulate and compose substitutions during a computation, in CLP we accumulate and simplify constraints. Like substitutions, constraints are never removed from the constraint store, therefore the information in the constraint store increases monotonically during derivations.

As in the LP calculus, we have two degrees of non-determinism in the calculus (selecting the goal and selecting the clause). Search trees are defined in the same way as in LP. Most implementations of the CLP languages also use depth-first search with chronological backtracking (SLD resolution).

CLP is an extension of LP. Constraint logic programs not containing constraints are logic programs. A derivation in LP can be expressed as CLP derivations, when substitutions are expressed by equality constraints. Substitutions of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ are written as equations $x_1 \doteq t_1 \wedge \ldots \wedge x_n \doteq t_n$.

CLP also generalizes the form of answers. An answer in LP languages is a substitution, while in CLP languages it is a constraint. Constraints as answers are useful, because they can summarize several (even infinitely many) LP answers into one (intensional) answer. For example, the goal $X+Y \geq 3 \wedge X+Y \leq 3$ is simplified into the intensional answer $X+Y \doteq 3$, where the variables do not have a value.

The notion of the logical reading of a state is similar to the one in LP, with the difference that constraints replace substitutions.

Definition 5.1.4. *Let* $<H, C>$ *be a state which occurs in a derivation starting with* $<G, \text{true}>$. *The formula* $\exists \bar{x}(H \wedge C)$ *is the* logical reading *of the state, where* $\bar{x}$ *stands for the variables which occur in* H *or* C *but not in* G.

In contrast to LP, an answer in CLP can be defined as the logical reading of a final state.

Definition 5.1.5. *An* answer (constraint) *of a goal* G *is the logical reading of a final state of a derivation starting with* $<G, \text{true}>$.

We are now ready to consider a simple example that will guide us through the various classes of constraint programming languages.

Example 5.1.1. Consider the following constraint logic program which implements the predicate `min/3`. `min(X,Y,Z)` means that `Z` is the minimum of `X` and `Y`:

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z} \quad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \wedge \texttt{Y}\doteq\texttt{Z} \quad (c2)$$

where $\leq$ and $\doteq$ are constraints with the usual meaning as total order and syntactic equality.

For the initial state $<\texttt{min}(1,2,\texttt{C}), \texttt{true}>$, we have the following derivation:

$$\begin{array}{ll} & <\texttt{min}(1,2,\texttt{C}),\ \texttt{true}> \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z},\ \ 1\doteq\texttt{X} \wedge 2\doteq\texttt{Y} \wedge \texttt{C}\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{C}\doteq 1\rangle \end{array}$$

First, the clause $(c1)$ is applied to unfold the initial state. Then, in the **Solve** transition a constraint solver simplifies the constraint store after adding the constraints from the goal store. The derivation is successful and the answer constraint is $\texttt{C}\doteq 1$. For readability, the answer constraint has been simplified by removing variables that do not occur in the initial goal.

Using the second clause $(c2)$ on the same initial state leads to an inconsistent constraint store, namely $2\leq 1 \wedge 2\doteq\texttt{C}$. Thus this derivation fails. The search tree (Fig. 5.4) shows that a depth-first traversal of such tree would encounter the successful derivation and then the failed one.

The initial goal `min(A,2,1)` has the derivation:

$$\begin{array}{ll} & <\texttt{min}(\texttt{A},2,1),\ \texttt{true}> \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\doteq\texttt{Z},\ \ \texttt{A}\doteq\texttt{X} \wedge 2\doteq\texttt{Y} \wedge 1\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{A}\doteq 1\rangle \end{array}$$

Using the first clause, the goal `min(A,2,2)` has the answer $\texttt{A}\doteq 2$. The second clause yields the same answer.

The goal `min(A,2,3)` fails. In Prolog, these transitions would lead to an error message, since no value for `A` is known yet and the comparison is only possible between known values.

For the initial goal `min(A,A,B)` two derivations are possible. Using the first clause, we have the following derivation:

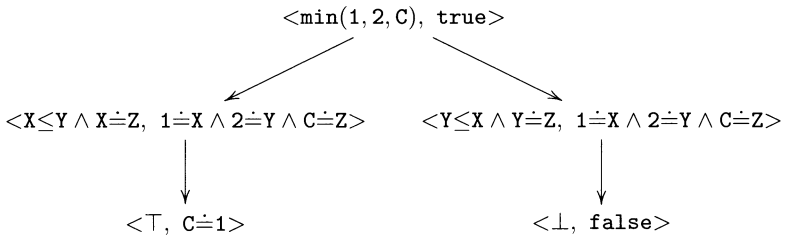


Fig. 5.4. Search tree for the goal `min(1,2,C)`

$$
\begin{array}{ll}
 & \texttt{<min(A, A, B), true>} \\
\mapsto_{\textbf{Unfold}\ (r1)} & \langle \texttt{X}\leq\texttt{Y} \ \wedge\ \texttt{X}\doteq\texttt{Z},\quad \texttt{A}\doteq\texttt{X} \ \wedge\ \texttt{A}\doteq\texttt{Y} \ \wedge\ \texttt{B}\doteq\texttt{Z} \rangle \\
\mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{A}\doteq\texttt{B} \rangle
\end{array}
$$

The intensional answer tells us that the minimum of two equal numbers is this number itself. Choosing the second clause ($c2$) leads to the same answer.

The general goal `min(A,B,C)` $\wedge$ `A`$\leq$`B` has the answer `A`$\doteq$`C` $\wedge$ `A`$\leq$`B` if clause ($c1$) is selected, but if the second clause ($c2$) is selected, the answer is `A`$\doteq$`C` $\wedge$ `A`$\doteq$`B`, this answer happens to be more specific.

5.2 Declarative Semantics

Constraint logic programs can be completed like logic programs [32]. In addition, however, the meaning of the constraints occurring in the program must be specified. The declarative semantics of a constraint logic program P is therefore the union of $P^{\leftrightarrow}$ with a *constraint theory* CT, while in LP we only extended $P^{\leftrightarrow}$ by the special theory CET.

5.3 Soundness and Completeness

The correspondence between operational and declarative semantics is also quite close in CLP. The theorems of LP and their proofs can be transferred to CLP: The Herbrand universe is replaced by an arbitrary constraint domain and CET by an appropriate constraint theory [32].

Theorem 5.3.1 (Soundness and Completeness of successful derivations). *Let P be a CL program and G be a goal.*

- **Soundness:** *If G has a successful derivation with answer constraint C, then $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$.*
- **Completeness:** *If $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$ and C is satisfiable in CT, then there are successful derivations for the goal G with answer constraints $C_1, \ldots, C_n$ such that $CT \models \forall(C \rightarrow (C_1 \vee \ldots \vee C_n))$.*

The theorem shows that in contrast to LP several answers must be considered and combined to achieve completeness.

Example 5.3.1. Let P be the following program:

$$\texttt{p(X,Y)} \leftarrow \texttt{X}\leq\texttt{Y}$$
$$\texttt{p(X,Y)} \leftarrow \texttt{X}\geq\texttt{Y}$$

The goal `p(X,Y)` has two successful derivations with answer constraints $\texttt{X}\leq\texttt{Y}$ and $\texttt{X}\geq\texttt{Y}$, respectively. It holds that $P^{\leftrightarrow} \cup CT \models \forall(\texttt{true} \rightarrow \texttt{p}(\texttt{X},\texttt{Y}))$.

According to the completeness result $CT \models \forall(\texttt{true} \rightarrow \texttt{X}\leq\texttt{Y} \vee \texttt{X}\geq\texttt{Y})$ must hold. This is the case, but each answer on its own is not sufficient to show this: $CT \not\models \forall(\texttt{true} \rightarrow \texttt{X}\leq\texttt{Y})$ and $CT \not\models \forall(\mathit{true} \rightarrow \texttt{X}\geq\texttt{Y})$.

If, however, the constraint system is *independent* (Chap. 8), the disjunction can be reduced to a disjunct. For example, syntactic equality as used in LP has the independence property.

The results for failed CLP derivations are as in LP.

Theorem 5.3.2 (Soundness and Completeness of failed derivations). *Let P be a CL program and G be a goal. $P^{\leftrightarrow} \cup CT \models \neg\exists G$ if and only if each fair derivation starting with $<G, \text{true}>$ fails finitely.*

6. Concurrent Constraint Logic Programming

At the end of the 1980's, concurrent constraint logic programming (CCLP) integrated ideas from concurrent LP [52] and CLP (Fig. 6.1).

Maher [37] proposed the ALPS class of languages. The ambitious Japanese Fifth-Generation Computing Project relied on a concurrent logic language. Saraswat [49] developed the ideas of the time further by introducing the *ask-and-tell* metaphor and by introducing the CC (concurrent constraints) language framework. Smolka proposed a concurrent programming model called OPM that subsumes functional and object-oriented programming [54].

1981	K. Clark and S. Gregory, Relational Language for Parallel Prog.
1982-94	Japanese Fifth-Generation Computing Project, KL1
1983	E. Shapiro, Concurrent Prolog, FCP (Flat Concurrent Prolog)
1983	K. Clark and S. Gregory, Parlog (Parallel Prolog)
1985	K. Ueda, GHC (Guarded Horn Clauses)
1987	M. Maher, ALPS language class
1989	V. Saraswat, CC language framework (Concurrent constraints)
1990	S. Haridi, AKL (Andorra Kernel Language)
1991	M. Hermengildo, CIAO (Parallel Multi-Paradigm Prolog Extension)
1992	G. Smolka, OZ (integrates functions, objects, and constraints)

Fig. 6.1. Early history of concurrent constraint logic programming

Processes are the main notion in concurrent programming. *Processes* (agents) are programs that are executed concurrently and that can interact with each other. Processes can either execute local actions or *communicate* and *synchronize* by sending and receiving messages. The communicating processes build a process network which can change dynamically. For concurrency it does not matter if the processes are executed physically in parallel or if they are interleaved.

In CCLP, concurrently executing processes communicate via a common constraint store. The processes are defined by predicates. Constraints take the role of (partial) messages and variables take the role of communication channels. Usually, communication is asynchronous. Running processes are goals that place and check constraints on shared variables.

T. Frühwirth et al., *Essentials of Constraint Programming*

This communication mechanism is based on *ask-and-tell* of constraints into a common constraint store. Tell refers to imposing a constraint as we know it from CLP. Ask is an inquiry whether a constraint already holds. Ask is realized by an *entailment* test. This test checks whether a constraint is implied by the current constraint store. Ask and Tell can be seen as generalizations of read and write from values to constraints. Ask corresponds to a *consumer*, Tell corresponds to a *producer* of constraints.

Processes are building blocks of *distributed systems*, where data and computations are physically distributed over a network of computers.

Processes can intentionally be non-terminating. Consider an operating system which usually should keep on running or a monitoring and control program which continuously processes incoming measurements and periodically returns intermediate results.

All this means for a process that in general decisions and resulting actions cannot be undone anymore. Search as in CLP languages (*don't-know non-determinism*) cannot be permitted any longer in this context. Instead, *don't-care non-determinism* is employed in the choice of clauses. This is also referred to as *committed choice*: If there are several applicable clauses, just one arbitrary clause of them will be chosen, alternative clauses will not be taken into account anymore. While the avoidance of search leads to a certain loss in expressiveness, it means a gain in efficiency.

In the context of concurrent distributed computation, failure should also be avoided. Failure of a goal atom (i.e., a single process) always entails the failure of the entire computation (i.e., all participating processes). In applications such as operating or monitoring systems this would be fatal.

In [49] Saraswat presents a general language framework called CC that permits both *don't-care* and *don't-know* non-determinism (search). Implemented CCLP languages are AKL, CIAO and Mozart (former OZ). They admit programmable search which is encapsulated into one process.

6.1 CCLP Calculus

We restrict our attention to the committed-choice CCLP languages. As we will see in the following chapter on CHR, an extension with search is relatively straightforward.

6.1.1 Syntax

The only difference between CCLP syntax and CLP syntax is that clauses are extended by *guards* (Fig. 6.2). Guards are preconditions on the applicability of a clause for unfolding.

Definition 6.1.1. *A* (CCL) clause *is of the form* $A \leftarrow C \mid G$, *where the* head A *is an atom,* C *is a constraint called* guard, *and the* body G *is a goal. A* CCL program *is a finite set of CCL clauses.*

A guard *true* is often omitted together with the commit symbol |.

Atom:	A, B	::=	$p(t_1, \ldots, t_n), \quad n \geq 0$
Constraint:	C, D	::=	$c(t_1, \ldots, t_n) \mid C \wedge D, n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid A \mid C \mid G \wedge H$
CCL Clause:	K	::=	$A \leftarrow C \mid G$
CCL Program:	P	::=	$K_1 \ldots K_m, \ m \geq 0$

Fig. 6.2. CCLP syntax

6.1.2 Operational Semantics

We present the operational semantics of CCLP as a state transition system. The definitions of states, derivations, and goals are the same as in the CLP calculus, with one important exception that is due to different transition rules. There is a new kind of final state.

Definition 6.1.2. *A state* $<G, C>$ *with* G *different from* $\top$ *and* C *different from* false *is called* deadlocked *if no more transitions are possible.*

Deadlocked states are the consequence of having no **Failure** transition anymore. Since the derivation in a deadlocked state in some sense could not finish successfully, deadlocks should be avoided and they are usually attributed to programming errors.

Given a CCLP program P, we define the transition relation $\mapsto$ by introducing two transition rules (cf. Fig. 6.3).

Since the CCLP calculus has to rely on committed choice of clauses (instead of search as in CLP), we have to be much more cautious with the application of a clause because we cannot undo it later. While in the CLP calculus a consistency test is performed for the application of the **Unfold** transition rule, an entailment test has to be performed in the CCLP calculus. The *entailment test* checks the implication of a constraint D by a store C, i.e., $CT \models \forall\, (C \rightarrow D)$.

In the transition rule **Unfold** for CCLP, entailment is checked for the guard of the CCL clause.

A clause with head B and guard D is *applicable* to A in the context of constraints C, when the condition holds that $CT \models \forall(C \rightarrow \exists\bar{x}((B \doteq A) \wedge D)$. Any of the applicable clauses can be applied, but it is a committed choice, it cannot be undone. If a clause $(B \leftarrow D \mid H)$ is applied to A, the transition removes A from the goal store, adds the body H to the goal store and also adds the equation $B \doteq A$ and the guard D to the constraint store.

We now discuss in more detail the *clause applicability condition* $CT \models \forall(C \rightarrow \exists\bar{x}((B \doteq A) \wedge D)$. The equation $(B \doteq A)$ is a notational shorthand for equating the arguments of the head B and the atom A.

Unfold	
If	$(B \leftarrow D \mid H)$ is a fresh variant of a clause in P with variables $\bar{x}$
and	$CT \models \forall\, (C \rightarrow \exists \bar{x}((B \doteq A) \wedge D))$
then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge D \wedge C>$
Solve	
If	$CT \models \forall\, ((C \wedge D_1) \leftrightarrow D_2)$
then	$<C \wedge G, D_1> \mapsto <G, D_2>$

Fig. 6.3. CCLP transition rules

Operationally, the clause applicability condition can be checked as follows. Given the constraints of C, try to solve the constraints $(B \doteq A \wedge D)$ without further constraining (touching) any variable in A and C. This means that we first check that A *matches* B and then check the guard D under this matching. Thus, *matching* means that A is an instance of B, i.e., it is only allowed to instantiate variables of B but not variables of A. This is achieved by equating B and A by existentially quantifying all variables from the clause, $\bar{x}$. Matching is also called *one-sided unification.*

The **Solve** transition is as in CLP. Note that as in CLP, the **Solve** transition leads to failure if the resulting store becomes inconsistent.

The following example illustrates the committed-choice behavior of the transition **Unfold** in the CCLP calculus.

Example 6.1.1. The following CCL program implements flipping of a coin.

```
flip(Side) ← Side≐head
flip(Side) ← Side≐tail
```

Depending on the selected clause, the result is different. The output of coin flipping is not predictable. In particular, a goal `flip(head)` can either be failed or successful.

With the operational semantics given above, it is not possible to express that two atoms can be unfolded in parallel. It only allows the executions to be interleaved. Parallelism can be made explicit by the following transition rule (Fig. 6.4).

Parallel	
If	$<G_1, C_1> \mapsto <H_1, D_1>$
and	$<G_2, C_2> \mapsto <H_2, D_2>$
then	$<G_1 \wedge G_2, C_1 \wedge C_2> \mapsto_{Parallel} <H_1 \wedge H_2, D_1 \wedge D_2>$

Fig. 6.4. CCLP with explicit parallelism

We now consider our running example of computing the minimum again.

Example 6.1.2. We define `min` with the following CCL program:

$$\texttt{min(X,Y,Z)} \leftarrow \texttt{X}\leq\texttt{Y} \mid \texttt{X}\doteq\texttt{Z} \qquad (c1)$$
$$\texttt{min(X,Y,Z)} \leftarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Y}\doteq\texttt{Z} \qquad (c2)$$

This CCLP implementation of `min` corresponds more to a usual program in a conventional programming language. For example, the first rule $c1$ can be read as if $\texttt{X}\leq\texttt{Y}$ then $\texttt{min(X,Y,Z)}$ can be simplified to $\texttt{X}\doteq\texttt{Z}$. We can consider the argument positions of the head atom that do not contain variables and the variables of the head that occur in the guard as *input parameters*, and all others as *output parameters*. So there is a notion of directness which is not presented in the formal model of LP and CLP, where we can also compute backwards.

However, clauses with guards are still more general than if-then-else or case statements in conventional programming. A guard entailment generalizes the notion of a test, as we will see in the following sample goals. In this sense, also the notion of input and output is generalized. We do not have to know specific values for the input parameters, it is enough to know certain constraints about them (those that are expected by the guards). In the case of `min`, we have to know the order between `X` and `Y`.

Let us consider some derivations now. To the goal `min(1,2,C)` the first clause $c1$ is applicable, since the entailment test is fulfilled:

$$CT \models \forall(\mathit{true} \rightarrow \exists\, \texttt{X},\texttt{Y},\texttt{Z}((1\doteq\texttt{X} \wedge 2\doteq\texttt{Y} \wedge \texttt{C}\doteq\texttt{Z}) \wedge \texttt{X} \leq \texttt{Y}))$$

This leads to the following derivation:

$$\begin{array}{ll} & \langle \texttt{min}(1,2,\texttt{C}), \mathit{true}\rangle \\ \mapsto_{\textbf{Unfold}\ (c1)} & \langle \texttt{X}\doteq\texttt{Z},\ 1\doteq\texttt{X}\wedge 2\doteq\texttt{Y}\wedge\texttt{C}\doteq\texttt{Z}\rangle \\ \mapsto_{\textbf{Solve}} & \langle \top,\ \texttt{C}\doteq 1\rangle \end{array}$$

This derivation is the only possible one in the CCLP calculus, while in the CLP a failure was the result of selecting the second clause.

In contrast to the CLP calculus, some derivations in CCLP will lead in deadlocked states now: On the initial state $\langle \texttt{min}(\texttt{A},2,1), \mathit{true}\rangle$ no transition rule can be applied since there is no information about the relationship between `A` and 2. This is also the case for the goals `min(A,2,2)` and `min(A,2,3)`. In the CLP calculus these goals could be completely solved, but it was necessary to try both CL clauses.

For the goal `min(A,A,B)` both CCL clauses fulfill the entailment test. Only one of the clauses will be applied. In this case, independent of the selected clause, the answer constraint is $\texttt{A}\doteq\texttt{B}$.

The goal $\texttt{min}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{A}\leq\texttt{B}$ leads to the answer $\texttt{A}\leq\texttt{B} \wedge \texttt{A}\doteq\texttt{C}$ by selecting the first clause. Similarly, the goal $\texttt{min}(\texttt{A},\texttt{B},\texttt{C}) \wedge \texttt{A}\geq\texttt{B}$ with the second clause leads to the answer $\texttt{A}\geq\texttt{B} \wedge \texttt{B}\doteq\texttt{C}$.

The next example involves an infinite derivation based on a cyclic producer and consumer relationship between processes.

Example 6.1.3. We consider the classical *Hamming's Problem*, which is to compute an ordered ascending sequence of all numbers whose only prime factors are 2, 3 or 5. The sequence starts with the numbers $1 \diamond 2 \diamond 3 \diamond 4 \diamond 5 \diamond 6 \diamond 8 \diamond 9 \diamond 10 \diamond 12 \diamond 15 \diamond 16 \diamond 18 \diamond 20 \diamond 24 \diamond 25 \diamond \ldots$. Two neighboring numbers later in the sequence are 79164837199872 and 79254226206720.

The idea for solving this problem is based on the following observation: any element of the sequence can be obtained by multiplying a previous number of the sequence with 2, 3 or 5. The only exceptions is the initial number 1. Hence, once we have the number 1, we can compute all elements of the sequence.

We define a non-terminating process `hamming(S)` that will produce the numbers as elements of the infinite sequence `S`. Now we pretend that the (infinite!) sequence `S` is already known (even though we just know that it starts with 1). We implement three processes `mults`, which multiply the numbers of `S` with 2, 3, and 5, respectively, as soon as the numbers are known. Two `merge` processes combine the three resulting sequences into an ordered one without duplicates. However, this sequence is the desired sequence `S`.

```
hamming(S) ←
  mults(S,2,S2) ∧ mults(S,3,S3) ∧ mults(S,5,S5) ∧
  merge(S2,S3,S23) ∧ merge(S5,S23,S)

mults(X◇S,N,XSN) ← XSN≐X*N◇SN ∧ mults(S,N,SN)

merge(X◇In1,Y◇In2,XYOut) ← X≐Y |
    XYOut≐X◇Out ∧ merge(In1,In2,Out)
merge(X◇In1,Y◇In2,XYOut) ← X<Y |
    XYOut≐X◇Out ∧ merge(In1,Y◇In2,Out)
merge(X◇In1,Y◇In2,XYOut) ← X>Y |
    XYOut≐Y◇Out ∧ merge(X◇In1,In2,Out)
```

A process `mults(S,N,SN)` delays (suspends) until the first argument `S` is a sequence with a known first element. It then multiplies this number by `N` and puts the result as the first element of the output sequence `SN`. The recursion ensures that all elements of the input sequence are multiplied one after the other. Note that there is no base case because we assume that the sequence is infinite. In any case, the `mults` process will simply suspend if the sequence ends or if its remainder is not yet known.

A `merge` process compares the first elements of the two input sequences as soon as they are known and puts the smaller of them on the output sequence before recursively continuing. If the two elements are the same, only one of them is kept (duplicates are removed).

In this way, the `mults` and `merge` processes synchronizes itself and communicates via the shared sequence variables. The result is a concurrent-process network where the processes can be executed in parallel.

If we start by unfolding the goal `hamming(S)`, all `mults` and `merge` processes will suspend, because nothing about the input sequences of each process is known. But when we supply the additional constraint that `S` starts with the number `1`, i.e., $S \doteq 1 \diamond S1$, the process network will produce all Hamming numbers.

The goal `mults(1◇S,2,S2)` with the transition rules **Unfold** and **Solve** leads to $S2 \doteq 2 \diamond S2N \wedge$ `mults(S,2,S2N)`.

Overall, the three `mults` processes yield

$S2 \doteq 2 \diamond S2N \wedge S3 \doteq 3 \diamond S3N \wedge S5 \doteq 5 \diamond S5N \wedge$
`mults(S,2,S2N)` $\wedge$ `mults(S,3,S3N)` $\wedge$ `mults(S,5,S5N)` $\wedge$
`merge(S2,S3,S23)` $\wedge$ `merge(S5,S23,S)`.

Since the first numbers of the sequences `S2` and `S3` are known, the goal `merge(S2,S3,S23)` can unfold with the second clause for `merge`. Thus, the first number of the sequence `S23` is computed:

$S23 \doteq 2 \diamond S23N \wedge$ `merge(S2N,S3,S23N)`.

Now the goal `merge(S5,S23,S)` yields $S \doteq 2 \diamond SN \wedge$ `merge(S5,S23N,SN)`.

The determination of the first element of `S` causes the `mults` processes to be woken and the next numbers of the Hamming's sequence will be computed; ad infinitum.

CCLP with Atomic Tell

In the CCLP calculus presented above, the constraints occurring in the body of a clause are added without any consistency check. So there is still the danger of failure due to the **Solve** transition. This kind of tell operation is called *eventual tell*.

The CCLP calculus can be extended by *atomic tell*. In *atomic tell* operations, a process may place constraints into the store only if they are consistent together.

CCL Clause	K	::=	$A \leftarrow C : D \mid G$

Fig. 6.5. Extended CCLP syntax of clauses

The syntactic difference to eventual tell is that the constraint to be told atomically is part of the guard of a clause (Fig. 6.5). Hence, if the constraint store becomes inconsistent as a consequence of an atomic tell, the operation

can be undone and the clause will not be applied (Fig. 6.6). The formalization and the overall behavior is similar to the consistency test in the CLP calculus except that there is no search.

Unfold	
If	$(B \leftarrow D_1 : D_2 \mid H)$ is a fresh variant of a clause in P
and	$CT \models \forall\, (C \rightarrow \exists \bar{x}((B \doteq A) \wedge D_1))$
and	$CT \models \exists((B \doteq A) \wedge D_1 \wedge D_2 \wedge C)$
Then	$<A \wedge G, C> \mapsto <H \wedge G, (B \doteq A) \wedge D_1 \wedge D_2 \wedge C>$
Solve	
If	$CT \models (C \wedge D_1) \leftrightarrow D_2$
Then	$<C \wedge G, D_1> \mapsto <G, D_2>$

Fig. 6.6. CCLP transition rules extended with atomic tell

The extended CCLP transition rule **Unfold** can only be applied if adding the constraint D_2 keeps the constraint store C consistent.

With this extended calculus, it is possible to keep the constraint store consistent and therefore prevent failure by allowing constraints only in the guard, but not in the body anymore. However, it has been shown that such languages are strictly less expressive than languages with eventual tell [18].

Example 6.1.4. The `min` example can be implemented with atomic tell as follows:

```
min(X,Y,Z) ← X≤Y : X≐Z | true
min(X,Y,Z) ← Y≤X : Y≐Z | true
```

All goals of Example 6.1.2 lead to the same answer constraints. The difference between both calculi can be illustrated with erroneous goals like `min(1,2,3)`. This goal is failed using eventual tell as in Example 6.1.2. Here, with atomic tell, the goal is deadlocked, i.e., no transition rule can be applied. Therefore the goal does not cause failure (which would affect all processes in the network).

6.2 Declarative Semantics

The declarative semantics of CCL programs is analogous to that of CL programs. The symbols "|" and ":" are interpreted as conjunctions. Thus, a CCL clause of the form $A \leftarrow C : D \mid G$ corresponds to the clause $A \leftarrow C \wedge D \wedge G$. Clark's completion of a CCL program is defined as in CLP.

6.3 Soundness and Completeness

In principle, the soundness and completeness theorems of CLP (Sect. 5.3) would apply for CCLP as well, but the completeness theorem of CLP refers to several derivations. Since CCLP goals have only one derivation, there is a discrepancy between operational and declarative semantics. The following results are due to Maher [37].

The soundness theorem of CLP for successful derivations applies analogously for CCLP.

Theorem 6.3.1 (Soundness of successful derivations). *Let P be a CCL program and G be a goal. If G has a successful derivation with an answer constraint C, then $P^{\leftrightarrow} \cup CT \models \forall(C \rightarrow G)$.*

The soundness results also applies to deadlocked states.

A class of CCL programs, called *deterministic programs*, for which completeness results can be given has been identified by Maher. In a deterministic program, no two clauses for the same predicate have overlapping guards. Two guards overlap if their conjunction is consistent. This means that in a derivation, at most one clause can be chosen for a goal (and that any possible order of clause applications results in the same final state).

Example 6.3.1. We define `min` as deterministic CCL predicate:

```
min(X,Y,Z) ← X≤Y : X≐Z | true
min(X,Y,Z) ← Y<X : Y≐Z | true
```

The difference to previous versions is that the ask guard of the second clause is more strict. We can therefore expect that there are cases where a transition that was possible before is not possible anymore since the guard is stricter.

With one exception the goals of Example 6.1.2 lead to the same answer constraints. Even though, for the goal `min(A,A,B)`, only the first clause fulfills the transition condition, the answer constraint is `A≐B` as before. The state <`min(A, B, C), A≤B`> leads to `A≤B` $\wedge$ `A≐C`, as expected. However the state <`min(A, B, C), A≥B`> is now deadlocked.

The restriction to deterministic CCL programs and to goals having at least one fair derivation allows for the desired completeness result.

Theorem 6.3.2 (Completeness of successful derivations). *Let P be a deterministic CCL program and G be a goal with at least one fair derivation. If $P^{\leftrightarrow} \cup CT \models \forall\ (C \rightarrow G)$ and C is consistent in CT, then each successful derivation of G has an answer constraint C' such that*

$$CT \models \forall\ (C \rightarrow C').$$

This completeness theorem does not hold if G has no fair derivations. Note that a goal with only deadlocked derivations has no fair derivation at all.

Example 6.3.2. Let P be the CCL program for `min` from Example 6.1.2 and let G be the goal `min(A,B,C)`.

Then $\mathcal{P}^{\leftrightarrow}, CT \models \forall(\mathtt{A} \leq \mathtt{B} \wedge \mathtt{A} \doteq \mathtt{C} \rightarrow \mathtt{min}(\mathtt{A}, \mathtt{B}, \mathtt{C}))$ holds. However, the initial state $<G, \mathit{true}>$ is deadlocked. Thus, there is no fair derivation and no answer constraint. Therefore, the theorem is not applicable.

For the class of deterministic CCL programs, the following holds: If a goal has a finitely failed derivation, then any fair derivation will finitely fail.

Theorem 6.3.3 (Soundness and Completeness of failed derivations). *Let P be a deterministic CCL program and G be a goal with at least one fair derivation. Then the following statements are equivalent:*

- $P^{\leftrightarrow} \cup CT \models \neg\exists G$
- *G has a finitely failed derivation.*
- *Each fair derivation of G fails finitely.*

Part II

Constraint Systems

7. Constraint Handling Rules

One lesson learned from applications is that constraints are often heterogeneous and application specific. In the beginning of constraint programming, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard.

Several proposals have been made to allow more flexibility and customization of constraint solvers. The most far-reaching proposal is Constraint Handling Rules (CHR) [22]. CHR is a constraint language originally designed for writing constraint solvers. CHR is essentially a concurrent committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved.

CHR defines both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence, e.g., $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \Leftrightarrow \mathtt{X}=\mathtt{Y}$. Propagation adds new constraints, which are logically redundant but may cause further simplification, e.g., $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z} \Rightarrow \mathtt{X} \leq \mathtt{Z}$.

Besides defining the behavior of constraints, CHR, and also its extension CHR$^\vee$, can be and have been used

- for theorem proving and computational logic providing forward and backward chaining, (integrity) constraints, tabulation, abduction, and deduction,
- as general-purpose concurrent constraint language,
- as flexible production rule system with constraints.

Implementations of CHR are available in Sicstus and Eclipse Prolog, Java and Haskell, among others.

We introduce CHR in this section and will use CHR in the remainder of the book to specify and implement constraint solvers. Also, the applications we introduce in the last part of the book rely on CHR.

T. Frühwirth et al., *Essentials of Constraint Programming*

7.1 CHR Calculus

The CHR calculus can be seen as a generalization of the CCLP calculus.

7.1.1 Syntax

The CHR syntax is given in the following definitions and summarized in Fig. 7.1.

We use two disjoint sets of predicate symbols for two different kinds of constraints: *(built-in) constraint symbols* and *CHR constraint symbols (user-defined symbols)*. Constraints are defined as in CLP. Built-in constraints are handled by predefined given constraint solvers. CHR constraints are defined by a CHR program.

Definition 7.1.1. *A* simplification rule *is of the form* $E \Leftrightarrow C \mid G$. *A* propagation rule *is of the form* $E \Rightarrow C \mid G$, *where the* head E *is a CHR constraint, the* guard C *is a built-in constraint, and the* body G *is a goal.*
A goal *is either* $\top$ *or* $\bot$ *or a built-in constraint or a CHR constraint or a conjunction of goals.*
A CHR program *is a finite set of rules.*

A CHR constraint symbol is *defined* in a CHR program if it occurs in the head of a rule. A guard "*true*" can be omitted together with the commit symbol $\mid$.

Built-in Constraint:	C, D	::=	$c(t_1, \ldots, t_n) \mid C \wedge D, n \geq 0$
CHR Constraint:	E, F	::=	$e(t_1, \ldots, t_n) \mid E \wedge F, n \geq 0$
Goal:	G, H	::=	$\top \mid \bot \mid C \mid E \mid G \wedge H$
CHR Rule:	R	::=	$E \Leftrightarrow C \mid G \mid E \Rightarrow C \mid G$
CHR Program:	P	::=	$R_1 \ldots R_m, m \geq 0$

Fig. 7.1. CHR syntax

Example 7.1.1. Given the built-in constraints *true* and $\doteq$, we define a CHR constraint for the partial-order relation $\leq$:

$$\texttt{X}\leq\texttt{Y} \Leftrightarrow \texttt{X}\doteq\texttt{Y} \mid \texttt{true} \quad (r1)$$
$$\texttt{X}\leq\texttt{Y} \wedge \texttt{Y}\leq\texttt{X} \Leftrightarrow \texttt{X}\doteq\texttt{Y} \quad (r2)$$
$$\texttt{X}\leq\texttt{Y} \wedge \texttt{Y}\leq\texttt{Z} \Rightarrow \texttt{X}\leq\texttt{Z} \quad (r3)$$
$$\texttt{X}\leq\texttt{Y} \wedge \texttt{X}\leq\texttt{Y} \Leftrightarrow \texttt{X}\leq\texttt{Y} \quad (r4)$$

The CHR program specifies and implements reflexivity ($r1$), antisymmetry ($r2$), transitivity ($r3$), and idempotence ($r4$) in a straightforward way. The reflexivity rule ($r1$) states that $\texttt{X}\leq\texttt{Y}$ is logically true, provided it is the case that $\texttt{X}\doteq\texttt{Y}$. The antisymmetry rule ($r2$) means $\texttt{X}\leq\texttt{Y} \wedge \texttt{Y}\leq\texttt{X}$ is logically equivalent to $\texttt{X}\doteq\texttt{Y}$. The transitivity rule ($r3$) states that the conjunction of $\texttt{X}\leq\texttt{Y}$ and $\texttt{Y}\leq\texttt{Z}$ implies $\texttt{X}\leq\texttt{Z}$. The idempotence rule ($r4$) states that multiple occurrences of the same $\leq$ constraint are logically equivalent to one occurrence.

7.1.2 Operational Semantics

Like in CLP and CCLP, a state consists of two components: the *goal store* and the *constraint store.*

Definition 7.1.2. *A* state *is a pair* $<G, C>$*, where* G *is a goal and* C *is a (built-in) constraint.*
An initial state *is a state of the form* $<G, \text{true}>$.
A state is called successful final state *if it is of the form* $<E, C>$ *and no transition is applicable, and* C *is different from* false. *A state is called* failed final state *if it is of the form* $<G, \text{false}>$.

Note that unlike CLP and CCLP, a successful state can have a non-empty goal store, provided the goals are all CHR constraints and no transition is possible anymore. Indeed, the deadlocked states of CCLP are considered to be successful states in CHR, because they represent CHR constraints that could not be further simplified.

Simplify	
If	$(F \Leftrightarrow D \mid H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall\, (C \rightarrow \exists \bar{x}(F \doteq E \wedge D))$
then	$<E \wedge G, C> \mapsto <H \wedge G, (F \doteq E) \wedge D \wedge C>$
Propagate	
If	$(F \Rightarrow D \mid H)$ is a fresh variant of a rule in P with variables $\bar{x}$
and	$CT \models \forall\, (C \rightarrow \exists \bar{x}(F \doteq E \wedge D))$
then	$<E \wedge G, C> \mapsto <E \wedge H \wedge G, (F \doteq E) \wedge D \wedge C>$
Solve	
If	$CT \models (C \wedge D_1) \leftrightarrow D_2$
then	$<C \wedge G, D_1> \mapsto <G, D_2>$

Fig. 7.2. CHR transition rules

Given a CHR program P, we define the transition relation $\mapsto$ by introducing three transition rules (cf. Fig. 7.2).

The **Simplify** transition of CHR is like the **Unfold** transition of CCLP, except that the head of a CHR rule is a conjunction of atoms instead of a single atom.

To **Simplify** CHR constraints E means to remove them from the state $<E \wedge G, C>$ and to add the body H of a fresh variant of a simplification rule $(F \Leftrightarrow D \mid H)$ to the goal store and the equation $F \doteq E$ and the guard C to the constraint store, provided E matches the head F and the guard D is implied by the built-in constraints C.

Note that the equation $F \doteq E$ in this context corresponds to pairwise matching of the conjuncts of F and E and that according to the congruence the conjuncts in the goal store and hence in E can be permuted (so that a proper matching is possible).

The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints E in the state. Trivial non-termination is avoided by not applying a rule a second time to the same constraints.

The **Solve** transition is as in CLP and CCLP.

Example 7.1.2. Recall the program for $\leq$ of example 7.1.1. Operationally, the rule ($r1$) removes occurrences of constraints that match $\mathtt{X} \leq \mathtt{X}$. The antisymmetry rule ($r2$) means that if we find $\mathtt{X} \leq \mathtt{Y}$ as well as $\mathtt{Y} \leq \mathtt{X}$ in the current store, we can replace them by the logically equivalent $\mathtt{X} \doteq \mathtt{Y}$. The transitivity rule ($r3$) propagates constraints. We add the logical consequence $\mathtt{X} \leq \mathtt{Z}$ as a redundant constraint. The idempotence rule ($r4$) absorbs multiple occurrences of the same constraint.

A derivation of the goal $\mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}$ proceeds as follows (CHR constraints which are considered in the current transition step are underlined):

$$
\begin{array}{ll}
 & <\underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{C} \leq \mathtt{A}} \wedge \mathtt{B} \leq \mathtt{C},\ \mathtt{true}> \\
\mapsto_{\textbf{Propagate}\ (r3)} & <\mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \underline{\mathtt{B} \leq \mathtt{C}} \wedge \underline{\mathtt{C} \leq \mathtt{B}},\ \mathtt{true}> \\
\mapsto_{\textbf{Simplify}\ (r2)} & <\underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{B} \leq \mathtt{A}},\ \mathtt{B} \doteq \mathtt{C}> \\
\mapsto_{\textbf{Simplify}\ (r2)} & <\underline{\mathtt{A} \doteq \mathtt{B}},\ \mathtt{B} \doteq \mathtt{C}> \\
\mapsto_{\textbf{Solve}\ (r2)} & <\top, \mathtt{A} \doteq \mathtt{B} \wedge \mathtt{B} \doteq \mathtt{C}>
\end{array}
$$

We now consider our running example of `min` again.

Example 7.1.3. Let $\leq$ and $<$ be built-in constraint symbols now. We now define `min` as CHR constraint, where `min(X,Y,Z)` means that `Z` is the minimum of `X` and `Y`:

$$
\begin{array}{ll}
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{X} \leq \mathtt{Y} \mid \mathtt{Z} \doteq \mathtt{X} & (r1) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Y} \leq \mathtt{X} \mid \mathtt{Z} \doteq \mathtt{Y} & (r2) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Z} < \mathtt{X} \mid \mathtt{Y} \doteq \mathtt{Z} & (r3) \\
\mathtt{min(X,Y,Z)} \Leftrightarrow \mathtt{Z} < \mathtt{Y} \mid \mathtt{X} \doteq \mathtt{Z} & (r4) \\
\mathtt{min(X,Y,Z)} \Rightarrow \mathtt{Z} \leq \mathtt{X} \wedge \mathtt{Z} \leq \mathtt{Y} & (r5)
\end{array}
$$

The first two rules ($r1$) and ($r2$) are as in the CCLP program and correspond to the usual definition of `min`.

In contrast to the CCLP definition of `min`, we also want to be able to compute backwards. In contrast to the CLP definition of `min`, where this was achieved by search using two clauses, in CHR this is achieved by committed-choice rules that explicitely express the cases where a simplification is possible. So the two rules ($r3$) and ($r4$) simplify `min` if the order between the result `Z` and one of the input variables is known.

The last rule ($r5$) propagates constraints. It states that `min(X,Y,Z)` unconditionally implies $\texttt{Z}\leq\texttt{X} \wedge \texttt{Z}\leq\texttt{Y}$. Operationally, the transition **Propagate** adds these logical consequences as redundant constraints while the `min` constraint is kept.

To the goal `min(1,2,M)` the first rule is applicable:

$$\begin{array}{ll} & \texttt{<min}(1,2,\texttt{M}),\texttt{true>} \\ \mapsto_{\textbf{Simplify}\ (r1)} & \texttt{<M}\doteq 1,\texttt{true>} \\ \mapsto_{\textbf{Solve}} & \texttt{<}\top,\texttt{M}\doteq 1\texttt{>} \end{array}$$

To the goal $\texttt{min(A,B,M)} \wedge \texttt{A}\leq\texttt{B}$ the first rule is applicable:

$$\begin{array}{ll} & \texttt{<min}(\texttt{A},\texttt{B},\texttt{M}) \wedge \texttt{A} \leq \texttt{B}, \mathit{true}\texttt{>} \\ \mapsto_{\textbf{Solve}} & \texttt{<min}(\texttt{A},\texttt{B},\texttt{M}),\texttt{A} \leq \texttt{B}\texttt{>} \\ \mapsto_{\textbf{Simplify}\ (r1)} & \texttt{<M}\doteq\texttt{A},\texttt{A} \leq \texttt{B}\texttt{>} \\ \mapsto_{\textbf{Solve}} & \texttt{<}\top,\texttt{M}\doteq\texttt{A} \wedge \texttt{A} \leq \texttt{B}\texttt{>} \end{array}$$

Redundancy from a propagation rule is useful, as the goal `min(A,2,2)` shows. To this goal only the propagation rule is applicable, but to the resulting state the second rule becomes applicable:

$$\begin{array}{ll} & \texttt{<min}(\texttt{A},2,2),\texttt{true>} \\ \mapsto_{\textbf{Propagate}\ (r5)} & \texttt{<min}(\texttt{A},2,2) \wedge 2 \leq \texttt{A} \wedge 2 \leq 2,\texttt{true>} \\ \mapsto_{\textbf{Solve}} & \texttt{<min}(\texttt{A},2,2),2 \leq \texttt{A}\texttt{>} \\ \mapsto_{\textbf{Simplify}\ (r2)} & \texttt{<}2\doteq 2,2 \leq \texttt{A}\texttt{>} \\ \mapsto_{\textbf{Solve}} & \texttt{<}\top,2 \leq \texttt{A}\texttt{>} \end{array}$$

In this way, we find out that for `min(A,2,2)` to hold, $2 \leq \texttt{A}$ must hold.

Another interesting derivation involving the propagation rule is the following one (we omit the initial state for brevity):

$$\begin{array}{ll} & \texttt{<min}(\texttt{A},\texttt{B},\texttt{M}),\texttt{A}\doteq\texttt{M}\texttt{>} \\ \mapsto_{\textbf{Propagate}\ (r5)} & \texttt{<min}(\texttt{A},\texttt{B},\texttt{M}) \wedge \texttt{M} \leq \texttt{A} \wedge \texttt{M} \leq \texttt{B},\texttt{A}\doteq\texttt{M}\texttt{>} \\ \mapsto_{\textbf{Solve}} & \texttt{<min}(\texttt{A},\texttt{B},\texttt{M}),\texttt{M} \leq \texttt{B} \wedge \texttt{A}\doteq\texttt{M}\texttt{>} \\ \mapsto_{\textbf{Simplify}\ (r1)} & \texttt{<}\top,\texttt{M} \leq \texttt{B} \wedge \texttt{A}\doteq\texttt{M}\texttt{>} \end{array}$$

Let us look at some more goals for `min` that lead to deadlocked states in CCLP and in CLP, the goals could only be completely solved by trying both clauses for `min` in CLP.

The goal `min(A,2,1)` leads to $\texttt{A}\doteq 1$ via rule ($r4$). In CLP, we had two answers for this goal, where one answer was a generalization of the other one.

The goal `min(A,2,3)` leads to failure via rule ($r5$). In CLP, we had failure after trying both clauses.

The goal `min(A,A,M)` leads to $\texttt{A}\doteq\texttt{M}$ via rule ($r1$). Alternatively, rule ($r2$) is applicable with the same result. However, since CHR is a committed-choice language, only one of the rules will be applied. In CLP, we had two identical answers for this goal.

7.2 Declarative Semantics

CHR can be given a declarative semantics (which can be seen as an extension of the semantics proposed for Guarded Rules [53]). It differs from the semantics of CLP and CCLP in that Clark's completion is not used. Rather, each CHR rule alone gives rise to a formula. This "stronge" declarative semantics comes from the fact that CHR is concerned with solving constraints (that always admit a logical reading) and not with defining arbitrary predicates. (If CHR is used as a general-purpose programming language, another declarative semantics based on linear logic can be more useful.)

The logical reading of a simplification rule of the form $E \Leftrightarrow C \mid G$ is a logical equivalence provided the guard holds:

$$\forall(C \rightarrow (E \leftrightarrow \exists \bar{y}\ G)),$$

where $\bar{y}$ are the variables that appear only in the body G.

The logical reading of a propagation rule of the form $E \Rightarrow C \mid G$ is an implication provided the guard holds:

$$\forall(C \rightarrow (E \rightarrow \exists \bar{y}\ G)).$$

As in CLP and CCLP, the logical reading of a CHR program P is the conjunction of the logical readings of its rules $\mathcal{P}$ united with a *constraint theory* CT that defines the built-in constraint symbols.

The logical reading of a state and the definition of answer constraints is the same as in CLP and CCLP.

7.3 Soundness and Completeness

We now relate the operational and declarative semantics of CHR. The proofs for the following theorems can be found in [2].

Definition 7.3.1. *A* computable constraint *of G is the logical reading of a state which appears in a derivation of G.*

The following results are based on the fact that the transition steps of CHR preserve the logical reading of states. Lemma 7.3.1 says that all states in a derivation are logically equivalent.

Lemma 7.3.1 (Equivalence of States). *Let P be a CHR program and G be a goal. Then, for all computable constraints C_1 and C_2 of G, the following holds: $\mathcal{P} \cup CT \models \forall\ (C_1 \leftrightarrow C_2)$.*

In the soundness and completeness results, we need not distinguish between successful and failed derivations.

Theorem 7.3.1 (Soundness). *Let P be a CHR program and G be a goal. If G has a derivation with answer constraint C, then $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$.*

Theorem 7.3.2 (Completeness). *Let P be a CHR program and G be a goal with at least one finite derivation. If $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$, then G has a derivation with answer constraint C' such that $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C')$.*

Theorem 7.3.2 is stronger than the completeness result for CLP languages as presented in Sect. 5. We can reduce the disjunction in the completeness theorem presented there to a single disjunct in the completeness theorem of CHR. This is possible, since the declarative semantics of CHR is stronger and consequently, according to Lemma 7.3.1, all computable constraints of a given goal are equivalent.

The completeness theorem does not hold if G has no finite derivation:

Example 7.3.1. Let P be the following CHR program:

$\mathtt{p} \Leftrightarrow \mathtt{p}$

Let the goal G be $\mathtt{p}$. It holds that $\mathcal{P} \cup CT \models \mathtt{p} \leftrightarrow \mathtt{p}$. However, G has only one infinite derivation.

From the soundness theorem we can derive the following statement about failed derivations.

Corollary 7.3.1. *Let P be a CHR program and G be a goal. If G has a finitely failed derivation, then $\mathcal{P} \cup CT \models \neg\exists G$.*

The converse of Corollary 7.3.1 does not hold in general.

Example 7.3.2. Let P be the following CHR program

$\mathtt{p} \Leftrightarrow \mathtt{q}$
$\mathtt{p} \Leftrightarrow \mathtt{false}$

$\mathcal{P} \cup CT \models \neg\mathtt{q}$, but $\mathtt{q}$ has no finitely failed derivation.

We will see that confluence will improve on this situation. Confluence generalizes the notion of determinism as introduced for CCLP programs.

7.4 Confluence

We have already shown in the previous section that for every CHR program, the result of a derivation of a given goal will always have the same meaning. However, it is not guaranteed that the result is syntactically the same. In particular, a solver may be able to detect inconsistency of constraints with one order of rule applications but not with another.

The *confluence* property of a program guarantees that any derivation for a goal results in the same final state no matter which of the applicable rules are applied.

Definition 7.4.1. *A CHR program is called* confluent *if for all states S, S_1, and S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$, then S_1 and S_2 are joinable. Two states S_1 and S_2 are called* joinable *if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are variants.*

To analyze confluence of a given CHR program, we cannot check joinability starting from any given state, because in general there are infinitely many such states. However, for terminating programs, one can restrict the joinability test to a finite number of "minimal" states.

Definition 7.4.2. *Let R_1 be a simplification rule and R_2 be a (not necessarily different) rule, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and C_i be the guard of rule R_i $(i = 1, 2)$, then a* critical ancestor state of R_1 and R_2 *is*

$$<H_1 \wedge A_1 \wedge H_2, \quad (A_1 \doteq A_2) \wedge C_1 \wedge C_2>,$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1 \doteq A_2) \wedge C_1 \wedge C_2)$.

Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2, then the tuple (S_1, S_2) is a critical pair *of R_1 and R_2. A critical pair (S_1, S_2) is* joinable *if S_1 and S_2 are joinable.*

The following theorem gives a decidable, sufficient, and necessary condition for confluence of a terminating CHR program [1]. A CHR program is called *terminating* if there are no infinite derivations.

Theorem 7.4.1 (Confluence of CHR). *A terminating CHR program is confluent if and only if all its critical pairs are joinable.*

Example 7.4.1. Recall the program for $\leq$ of Example 7.1.1. The following critical pair stems from the critical ancestor state $<\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}>$ of $(r2)$ and $(r3)$

$$(S_1, S_2) = (<\mathtt{X} \doteq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}>, \quad <\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \mathtt{X} \leq \mathtt{Z}, \mathtt{true}>)$$

is joinable. A derivation beginning with S_1 proceeds as follows:

$$\begin{array}{ll} & <\underline{\mathtt{X} \doteq \mathtt{Y}} \wedge \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}> \\ \mapsto_{\mathbf{Solve}} & <\mathtt{Y} \leq \mathtt{Z}, \mathtt{X} \doteq \mathtt{Y}> \end{array}$$

A derivation beginning with S_2 results in the same final state:

$$\begin{array}{ll} & <\underline{\mathtt{X} \leq \mathtt{Y}} \ \wedge \ \underline{\mathtt{Y} \leq \mathtt{X}} \ \wedge \ \mathtt{Y} \leq \mathtt{Z} \ \wedge \ \mathtt{X} \leq \mathtt{Z}, \mathtt{true}> \\ \mapsto_{\mathbf{Simplify}} \ (r2) & <\underline{\mathtt{X} \doteq \mathtt{Y}} \wedge \mathtt{Y} \leq \mathtt{Z} \wedge \ \mathtt{X} \leq \mathtt{Z}, \mathtt{true}> \\ \mapsto_{\mathbf{Solve}} & <\mathtt{Y} \leq \mathtt{Z} \wedge \ \mathtt{X} \leq \mathtt{Z}, \mathtt{X} \doteq \mathtt{Y}> \\ \mapsto_{\mathbf{Simplify}} \ (r4) & <\mathtt{Y} \leq \mathtt{Z}, \mathtt{X} \doteq \mathtt{Y}> \end{array}$$

The following theorems state that the completeness result of CHR can be improved for the class of confluent and terminating CHR programs [2].

Theorem 7.4.2. *Let P be a terminating and confluent CHR program and G be a goal, then the following are equivalent:*

- $\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow G)$.
- *G has a derivation with answer constraint C' such that*

  $$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C').$$

- *Every derivation of G has an answer constraint C' such that*

  $$\mathcal{P} \cup CT \models \forall\ (C \leftrightarrow C').$$

Corollary 7.4.1. *Let P be a terminating and confluent CHR program and G be a goal with at least one answer constraint consisting of only built-in constraints, then $\mathcal{P} \cup CT \models \neg\exists G$ if and only if G has a finitely failed derivation.*

7.5 CHR$^\vee$: Adding Disjunction

CHR was originally intended as a declarative language for rapid prototyping and efficient implementation of constraint solvers. As a concurrent committed-choice language, CHR lacks the don't-know non-determinism of CLP. However, a simple extension of CHR suffices to be able to subsume the expressive power of CLP: one allows disjunctions on the right-hand sides of CHR rules. We call the extended language "CHR$^\vee$". CHR$^\vee$ allows to write the entire application in a uniform language [5].

Similar in spirit to the **UnfoldSplit** rule in CLP, we introduce the following additional transition **Split** for CHR$^\vee$, so we can deal with disjunction $\vee$ in Fig. 7.3.

Split

$$<(H_1 \vee H_2) \wedge G, C> \mapsto <H_1 \wedge G, C> \mid <H_2 \wedge G, C>$$

Fig. 7.3. CHR$^\vee$ transition rule with case splitting

CHR$^\vee$ will be used in the remainder of this book.

8. Constraint Systems and Constraint Solvers

In this part of the book, we introduce the most common constraint systems. They are the result of taking a data type together with its operations and interpreting the resulting expressions as constraints. These constraint systems use the universal data types of numbers (integers or reals) to represent scalar data or terms to represent structured data.

8.1 Constraint Systems

A constraint system formally specifies the syntax and semantics of the constraints of interest. Constraints are considered as special predicates of first-order logic. A constraint system states which are the constraint symbols, how they are defined, and which constraint formulae are actually used in and useful for reasoning. The following definition is based on Höhfeld and Smolka [28] and Jaffar and Maher [32].

Definition 8.1.1. *A* constraint system *is a tuple* $(\Sigma, \mathcal{D}, \mathcal{CT}, \mathcal{C})$*, where*

- Σ *is a* signature *that contains the nullary constraint symbols* true *and* false *and the binary constraint symbol* $=$ *for equality.*
- $\mathcal{D}$ *is a* domain (universe) *together with an interpretation of the function and constraint symbols in* Σ*.*
- CT *is a* constraint theory *that is a non-empty and consistent theory over* Σ*.*
- $\mathcal{C}$ *are the* allowed constraints*, a set of formulae that contains the constraints* true*,* false*, and* $\doteq$*, and that is closed under existential quantification and conjunction.*

CT defines the semantics and $\mathcal{C}$ the syntax of the constraint system. The minimal requirements on allowed constraints come from their use in constraint programming languages. The constraints *true*, *false*, and $\doteq$ play a prominent role. In the calculi, we are mainly concerned with conjunctions of atomic constraints whose variables are (implicitly) existentially quantified. Closedness under existential quantification means that the names of variables do not matter, i.e., it implies closedness under variable renaming. In addition, the

T. Frühwirth et al., *Essentials of Constraint Programming*

atomic constraints may be syntactically restricted so that they can be solved efficiently.

A minimal and essential example of a constraint system is the one for syntactic equality. The domain $\mathcal{D}$ has been put first for didactic reasons.

Constraint System E

- The domain $\mathcal{D}$ is the Herbrand universe
- The signature Σ contains
  - countably infinitely many function symbols (including at least one constant)
  - the constraint symbols *true*, *false*, and $\doteq$
- The constraint theory CT is CET.
- The allowed constraints are given in EBNF-notation:

$$\mathcal{C} ::= \mathit{true} \mid \mathit{false} \mid s \doteq t \mid \exists \bar{x}\, \mathcal{C} \mid \mathcal{C} \wedge \mathcal{C}$$

where s and t are terms over the signature of E.

8.2 Properties of Constraint Systems

We now discuss some desirable properties of constraint systems. First of all, the constraint theory should be *decidable* for allowed constraints at least.

Satisfaction-completeness means that the constraint theory can deal with allowed constraints. The property allows the theory to determine the consistency (satisfiability) of all allowed constraints.

Definition 8.2.1. *Let $(\Sigma, \mathcal{D}, \mathcal{CT}, \mathcal{C})$ be a constraint system.*

- *A constraint theory CT is* complete *if for every constraint $C \in \mathcal{C}$, either $CT \models C$ or $CT \models \neg C$.*
- *CT is* satisfaction-complete *if for every constraint $C \in \mathcal{C}$, either $CT \models \exists C$ or $CT \models \neg\exists C$.*

Completeness in this context refers to arbitrary constraints and is usually too strong a condition. The constraint theory CET used in the constraint system E is complete and hence also satisfaction-complete.

A desirable property for the efficient treatment of negated constraints is the *independence of negated constraints*. In such a constraint system, a conjunction of positive and negated constraints is satisfiable if and only if each negated constraint itself is satisfiable together with the positive constraint.

Definition 8.2.2. *A constraint system $(\Sigma, \mathcal{D}, \mathcal{CT}, \mathcal{C})$ has the* independence of negated constraints property *if for all constraints $C, C_1, \ldots, C_n \in \mathcal{C}$,*

$$CT \models \exists(C \wedge \neg C_1 \wedge \ldots \wedge \neg C_n) \text{ iff } CT \models \exists(C \wedge \neg C_i) \text{ for all } i \in \{1, \ldots, n\}.$$

For satisfaction-complete constraint systems, independence of negated constraints is equivalent to the *strong-compactness property.* This property improved the completeness result for CLP languages (Chap. 5).

Definition 8.2.3. *A constraint system* $(\Sigma, \mathcal{D}, \mathcal{CT}, \mathcal{C})$ *has the* strong-compactness property *if for all constraints* $C, C_1, \ldots, C_n \in \mathcal{C}$,

$$CT \models \forall(C \to C_1 \vee \ldots \vee C_n) \text{ iff } CT \models \forall(C \to C_i) \text{ for some } i \in \{1, \ldots, n\}.$$

Independence of negated constraints holds for the following constraint systems with infinite domains:

- Infinite Boolean algebras with positive constraints
- Linear arithmetic equations over real or rational numbers
- Herbrand terms with infinitely many function symbols including at least one constant (the constraint system E)
- Feature trees with infinitely many sorts and features
- Restricted classes of set constraints (without empty sets)

We now look at some constraint systems that do not have this property.

Example 8.2.1. The independence of negated constraints does not hold in a constraint system that defines the order relation $<$ over integers. For example, we have that

- $CT \models \exists X, Y \; \neg(X{<}Y)$
- $CT \models \exists X, Y \; \neg(X \doteq Y)$
- $CT \models \exists X, Y \; \neg(Y{<}X)$

However, we also have

- $CT \not\models \exists X, Y \; (\neg(X{<}Y) \wedge \neg(X \doteq Y) \wedge \neg(Y{<}X))$

Independence of negated constraint does not hold for constraint systems with a finite set of values.

Example 8.2.2. The independence of negated constraints holds in the constraint system E. If we change the signature of E to be finite, then the property is lost. For example, consider a signature for E with only one constant and one unary function symbol:

- $CET \models \exists X, Y \; (X \doteq f(Y) \wedge \neg(Y \doteq a))$
- $CET \models \exists X, Y, Z \; (X \doteq f(Y) \wedge \neg(Y \doteq f(Z)))$.

However, we also have

- $CET \not\models \exists X, Y, Z \; (X \doteq f(Y) \wedge \neg(Y \doteq a) \wedge \neg(Y \doteq f(Z)))$

Independence of negated constraint does not hold for constraint systems with non-trivial complementary constraints. A constraint is trivial if it is logically equivalent to *true* or *false*. Two constraints C and D are complementary if $C \leftrightarrow \neg D$.

Example 8.2.3. Consider the constraint system E. If we add a constraint symbol $\neq$ to the signature and the formula $s{\neq}t \leftrightarrow \neg(s{\doteq}t)$ to the constraint theory, then the resulting constraint system does not have the independence-of-negated-constraints property:

- $CT \models \exists X\ (true \wedge \neg(X{\doteq}a))$,
- $CT \models \exists X\ (true \wedge \neg(X{\neq}a))$.

However, we also have

- $CT \not\models \exists X\ (true \wedge \neg(X{\doteq}a) \wedge \neg(X{\neq}a))$.

Sometimes, independence of negated constraints can be restored by expressing one of the complementary constraints as the negation of the other. For example, if $\neq$ is expressed by negating $\doteq$, independence holds. This is not the case if $\doteq$ is expressed by negating $\neq$.

8.3 Capabilities of Constraint Solvers

A *constraint solver* implements an algorithm for solving allowed constraints in accordance with the constraint theory. A constraint solver collects the constraints that arrive *incrementally* from one or more running programs. It puts them into the constraint store. It tests their satisfiability, simplifies and if possible solves them.

More precisely, a constraint solver should be able to perform the following *reasoning services* (in order of importance):

Satisfiability (Consistency) test

The solver returns *false* if C is inconsistent: $CT \models \neg\exists C$.

Example. $X{>}X$ is inconsistent, $X{>}Y$ is not.

In words, the solver implements a *decision procedure* for satisfiability of allowed constraints. Syntactic equality and linear polynomial equations admit a satisfaction-complete algorithm, an efficient Boolean constraint solver does not. Boolean satisfiability is NP-complete and thus has exponential worst case time complexity. This means there is no efficient algorithm to solve it.

Simplification

The solver tries to transform a given constraint C into a logically equivalent, but simpler constraint D: $CT \models \forall(C \leftrightarrow D)$.

Example. $X{\leq}2 \wedge X{\leq}4$ is simplified into $X{\leq}4$, $2{*}X{=}6$ into $X{=}3$.

The intuition is that a simpler constraint can be handled more efficiently when new constraints arrive. It may also improve the presentation of the answer constraint. However, what simpler exactly means depends on the

constraint system, and is often in the eye of the beholder. For example, we may prefer a formulation with the least number of variables, but this may not be the formulation with the least possible size (see variable projection/elimination below). Finding the most simple representation of a constraint can be substantially harder than solving it.

Determination

Detect that a variable X occurring in a constraint C can only take a unique value: $CT \models \forall(C \rightarrow X{=}v)$, where v is a value.

Example. $X{\leq}2 \wedge 2{\leq}X$ implies $X{=}2$, $X^2{=}X \wedge X < 1$ implies $X{=}0$.

This special case of simplification is important for representing answer constraints as solutions that give values to variables. Determination also supports a simple way of communication between different constraint solvers via shared variables by exchanging values for those variables.

Variable projection/elimination

Eliminate a variable X by projecting a constraint C onto all other variables: $CT \models \exists XC \leftrightarrow D$, where D does not contain X.

Example. Projection of $\exists Y(X{<}Y \wedge Y{<}Z)$ onto X and Z results in $X{<}Z$ over the reals and $X{+}1{<}Z$ over the integers. In the constraint system E, in the formula $\exists Y(X{=}f(Y))$, the variable Y cannot be eliminated.

Projection may keep the constraint store small and simplify the answer constraint by eliminating local variables. However, in some cases, the elimination of variables may yield a significant increase in the size and number for constraints. For example, elimination of Y in $\exists Y(X_1{<}Y \wedge \ldots \wedge X_m{<}Y \wedge Y_1{<}Z \wedge \ldots \wedge Y_n{<}Z)$ yields $n{*}m$ constraints of the form $X_i{<}Y_j$.

Entailment test

Check whether a constraint C implies D: $CT \models \forall(C \rightarrow D)$?

Example. $X{<}Y$ entails $X{\leq}Y$, but not vice versa.

Entailment is required for guard checks in concurrent constraint languages like CCLP and CHR.
An (incomplete) entailment test can be implemented as follows: if $C \wedge D$ simplifies to C, then $C \rightarrow D$.

All the reasoning services can be regarded and implemented as variations of simplification that maintains a *normal form* of the constraints.

The constraint solver is expected to implement these reasoning services efficiently, more precisely, the average time complexity should be a polynomial of low degree, typically not worse than cubic. To achieve this efficiency, one is conten with incomplete implementations of reasoning services that otherwise would take exponential time.

8.4 Properties of Constraint Solvers

We regard the constraint solver as a function *solve* that takes an allowed constraint and returns its simplified form. In particular, *solve* should be

Correct
If $solve(C) = D$, then $CT \models \forall(C \leftrightarrow D)$

Failure-preserving
If $solve(C) = false$, then $solve(C \wedge D) = false$

Satisfaction-complete
If $CT \models \neg\exists C$, then $solve(C) = false$

Satisfaction-completeness implies failure-preservation.

Idempotent
$solve(solve(C)) = solve(C)$

In words, the function *solve* computes a fixpoint. There is no gain in simplifying simplified constraints again.

Independence of variable naming
If C and D are variants of each other, then $solve(C) = solve(D)$

Surprisingly, this condition is not always satisfied. Algorithms that are based on variable elimination (e.g., for solving linear polynomial equations) often rely on an order of variables. This order may change if variables are renamed, and this may lead to different results.

Congruence respecting
Associative $solve((C \wedge D) \wedge E) = solve(C \wedge (D \wedge E))$
Identity $solve(C \wedge true) = solve(C)$
Commutative $solve(C \wedge D) = solve(D \wedge C)$

The constraint solver should respect the properties of conjunction as expressed by the congruence relation used in the operational semantics for CLP (Chap. 5). Associativity and identity hold easily. For commutativity, similar remarks as for independence of variable renaming apply.

Incremental
$solve(solve(C) \wedge D) = solve(C \wedge D)$

In words, simplifying C and then simplifying the result together with newly arrived constraints D should give the same result as simplifying $C \wedge D$. Ideally, the incremental computation $solve(solve(C) \wedge D)$ should not be more costly than $solve(C \wedge D)$.

Canonical
If $CT \models \forall(C \leftrightarrow D)$, then $solve(C) = solve(D)$

This is a very strong condition, since it implies all the previous ones and thus is seldom met.

If the constraint solver is implemented by a terminating and confluent CHR program, then it will be failure-preserving, idempotent, congruence respecting, and incremental.

8.5 Principles of Constraint-Solving Algorithms

There are two main approaches for constraint-solving algorithms, variable-elimination and local-consistency (local-propagation) techniques. Variable elimination is usually satisfaction-complete, while local-consistency techniques have to be interleaved with search to achieve completeness.

Variable Elimination

Typically, the allowed constraints are *equations*. Other constraints will be transformed into equations if possible. The transformation may introduce auxiliary variables and simple constraints on them. For example, we may replace $X>Y$ by $X=Y+Z \wedge Z>0$.

Given an equation $e_1=e_2$, we call e_1 *l.h.s. (left-hand side)* and e_2 *r.h.s. (right-hand side)* of the equation. A *normal form* for an equation is typically of the form $X=e$, where X is a variable and e is an expression of some specific syntactic form. For example, the linear polynomial $2X + 3Y$ is the normal form of the arithmetic expression $Y + 2(X + Y)$.

A *solved (normal) form* or *solution* of constraints is a logically equivalent formulation of the constraints that determines variables (gives values to variables) and that is, if possible, unique. A solution is usually a conjunction of syntactic equality constraints of the form $X=v$, where X is the only l.h.s. occurrence of the variable and v is a value.

For example, $X=Y \wedge X=Z$ is not in solved form, because X occurs twice on the l.h.s. of an equation. $X=Y \wedge Y=Z$ is in solved form, as is $X=Z \wedge Z=Y$. Hence, this solved form is not unique.

Variable-elimination algorithms compute the solved form by eliminating multiple occurrences of variables. We repeatedly choose an equation $X=e$ and replace all other occurrences of X by e. We simplify the resulting new expressions such that the normal form is maintained. A well-known variable-elimination algorithm is Gaussian elimination for solving linear polynomial equations (Chap. 11). For example, in $X=7-Y \wedge X=3+Y$, we can remove the second occurrence of X. This may result in $X=7-Y \wedge Y=2$. Removing Y finally leads to the solution $X=5 \wedge Y=2$.

To ensure termination of variable elimination, we may rely on an order of variables and expressions. Because the results of variable elimination may change depending on the chosen equation and on the order of variables, solvers based on variable elimination are usually not confluent. For example, $X=7-Y-2Z \wedge X=3+Y$ may lead to $X=3+Y \wedge Y=2-Z$ or $X=5-Z \wedge Y=2-Z$.

Local Consistency (Local Propagation)

In this method, small fixed-size sub-problems of the initial problem are considered repeatedly until a fixpoint is reached. The sub-problems are simplified and new implied (redundant) constraints are computed (propagated) from them. The constraints are added hoping that they cause simplification.

For example, we may consider sub-problems consisting of two constraints. Given $X>Y \wedge Y>Z \wedge Z>X$, the first two constraints imply $X>Z$. This constraint and the third initial constraint $Z>X$ imply *false*.

For any given problem, there is only a polynomial number of small sub-problems. So if we can deal with sub-problems in polynomial time, the overall algorithm will also be polynomial.

Classical consistency algorithms were first explored for *constraint networks* in artificial intelligence research in the late 1960's. The main algorithms are *arc consistency* (Chap. 12.1) and *path consistency*. Originally, the algorithms involved unary and binary constraints over finite sets of values only.

Local-consistency methods often require that expressions are in *flat normal form*, where variables are the only arguments of functions (i.e., functions are not allowed to be nested).

A term is *flat* if it is a variable or a function symbol applied to variables. Every term can be *flattened* by performing the opposite of variable elimination. Each non-variable sub-expression is replaced by a new variable that is equated with the sub-expression. For example, $X+X+Y>5$ is flattened into $W>F \wedge X+V=W \wedge X+Y=V \wedge F=5$. The flat normal form of $X^2>3Y$ is $L>R \wedge L=X^T \wedge T=2 \wedge R=Z+F \wedge F=4$.

The advantage of the flat normal form is the uniform treatment of the allowed constraints in the constraint solver. The disadvantage is the introduction of auxiliary variables. Consistency methods are sensitive to the representation of the constraints, but there is usually no efficient way to find an optimal representation.

Search

Local-consistency methods must be combined with search to achieve satisfaction-completeness, i.e., *global consistency*. Search brings back exponential complexity to combinatorial and other NP-complete constraint problems, because dependencies between choices are not and cannot be fully taken care of. Search is also called *branching* because it will introduce branches in the *search tree*. Search is a *case analysis*, that is *case splitting* by introducing *choices*.

Usually, search is interleaved with constraint solving. A search step is performed, it adds a new constraint, that is simplified together with the existing constraints. This process is repeated until a solution is found.

Search can be done by trying possible values for a variable $X{=}v_1 \vee \ldots \vee X{=}v_n$. Such a *search routine/procedure* is called a *labeling procedure* or *enumeration procedure.*

Often, a labeling procedure will use heuristics to choose the next variable and value for labeling. The chosen sequence of variables is called a *variable ordering.* For example, we may count the occurrences of variables in a constraint problem. Then we choose the variable that occurs most for labeling in the hope that this will cause most simplification. Choosing the most constrained variable first is called *first-fail principle*, since we may expect that labeling this variable will lead to failure quickly, thus *pruning* the search tree early.

Similarly, since the next value for labeling a variable must be chosen, there is also a *value ordering.* For an example, see the Boolean constraint solver in Chap. 9.

In the general case, a search routine replaces a constraint by a logically equivalent disjunction of constraints, where the disjuncts are pairwise unsatisfiable (or at least do not imply each other). For example, $X{\neq}Y$ can be expressed as $X{<}Y \vee X{>}Y$.

8.6 Preliminaries

In the next chapter, we introduce the common constraint systems. For each constraint system, we will give its allowed constraints, its constraint theory, an algorithm to implement it, properties of the algorithm, and an example of a typical application.

A variety of algorithms exists for constraint systems, mostly adapted from artificial intelligence and operations research. For didactic reasons, we will concentrate on the basic principles of these algorithms. We will use the CHR language extended with disjunction $\text{CHR}^{\vee}$ (Chap. 7) to specify and implement these algorithms. In this way, we can describe the algorithms in a concise and compact manner. This makes it easier to analyze their termination, confluence, and worst case time complexity. Due to space limitations, the analyses will be somewhat informal at times.

For convenience and uniformity, we will assume that a CHR rule is never applied a second time to the (syntactically) same conjunction of constraints. In CHR implementations, a similar behavior is achieved in an efficient way by the use of hybrid, so-called simpagation rules, relying on textual rule application order, compiler options, and/or additional constraints in the guards of rules.

The programs have been tested with the Sicstus Prolog implementation of CHR and should run in other CHR implementations with little modification. Implementation-specific details like compiler settings have been omitted for the sake of generality.

The programs will use concrete syntax of Prolog implementations of CHR$^\vee$. Instead of $\Leftrightarrow$ and $\Rightarrow$ for rules, the notation `<=>` and `==>` is used. Conjunction $\wedge$ is written as comma `','`. Disjunction $\vee$ is written as semicolon `';'`. Rules are terminated with a period `'.'`. One-line comments start with `'%'`. Lists have a special notation in Prolog, e.g., `[1,2,3,4]` is a list of four elements. The empty list is `[]`. The term `[X|L]` denotes the list whose first element is `X` and whose remainder (tail) is the list `L`.

The constraint symbol $=$ will be implemented either as `=` if it refers to built-in syntactical equality of Prolog, or as `eq` if it is a CHR constraint. The constraint symbols $<, \leq, >, \geq, \neq$ will be implemented either as `<, =<, >, >=, \=` if they refer to built-in arithmetic constraints of Prolog, or as `lt, le, gt, ge, ne` if they are CHR constraints. The Prolog built-in `X is E` computes the result of the arithmetic expression `E` and equates it with the variable `X`. The built-in prefix operator `not` negates its argument, a conjunction of built-in constraints.

Before we introduce specific constraint systems, we introduce the notions of constraint systems and constraint solvers together with their desirable properties and the principles behind constraint reasoning and solving algorithms.

9. Boolean Algebra B

We start with the Boolean constraint system that admits a simple algorithm to solve constraints [41].

Constraint System B

Domain
Truth values 0 and 1

Signature

- Function symbols.
  - Truth values 0 and 1
  - Unary connective $\neg$
  - Binary connectives $\sqcap, \sqcup, \oplus, \rightarrow, \leftrightarrow$
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbol $=$

Constraint theory
Instances of $\neg X{=}Z$ and $X \odot Y{=}Z$ according to the following truth table, where $\odot \in \{\sqcap, \sqcup, \oplus, \rightarrow, \leftrightarrow\}$.

X	Y	$\neg X$	$X \sqcap Y$	$X \sqcup Y$	$X \oplus Y$	$X \rightarrow Y$	$X \leftrightarrow Y$
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

Allowed atomic constraints

$$C ::= \mathit{true} \mid \mathit{false} \mid X = Y \mid \neg X = Y \mid X \odot Y = Z,$$

where X, Y, and Z are variables or truth values.

T. Frühwirth et al., *Essentials of Constraint Programming*

The domain consists of the truth values `0` for falsity, `1` for truth. The signature includes these constants and the usual logical connectives of *propositional logic* as function symbols. To avoid confusion with the connectives $\wedge$ and $\vee$ used for conjunctions and disjunctions of arbitrary constraints, the symbols $\sqcap$ and $\sqcup$ are used for the logical connectives inside Boolean constraints.

The constraint theory is given by a truth table. Boolean expressions are equal if they denote the same truth value. The theory is decidable and complete. Despite its simplicity, the problem of determining whether a Boolean constraint is satisfiable is NP-complete, i.e., the worst case running time of any algorithm solving this problem is exponential in the size of the problem.

The allowed atomic constraints are in *flat normal form*, each constraint contains at most one logical connective. Non-flat constraints can be flattened. For example, $(X \sqcap Y) \sqcup Z = \neg W$ can be flattened into $(U \sqcup Z = V) \wedge (X \sqcap Y = U) \wedge (\neg W = V)$.

As the allowed atomic constraints correspond to Boolean functions, we call the arguments X and Y of the allowed atomic constraints *inputs* and the last one, Z, *output*.

9.1 Local-Propagation Constraint Solver

In the Boolean constraint solver a local-consistency algorithm is used. It simplifies one atomic Boolean constraint at a time into one or more syntactic equalities (i.e., built-in constraints) whenever possible. The rules for $X \sqcap Y = Z$, which is represented in relational form as `and(X,Y,Z)`, are as follows. For the other connectives, they are analogous.

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z) <=> X=Y | Y=Z.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
```

For example, the first rule says that the constraint `and(X,Y,Z)`, when it is known that the input `X` is `0`, can be reduced to asserting that the output `Z` must be `0`. Hence, the constraint `and(X,Y,Z), X=0` will result in `X=0, Z=0`. Note that a rule for `Z=0` is missing, since this case admits no simplification.

The above rules are based on the idea that, given a value for one of the variables in a constraint, we try to detect values for other variables. This approach of determining variables is called *value propagation* and is similar in spirit to *constant propagation* as used in data flow analysis of programs. Value propagation is frequently used in constraint-based graphical user interfaces

to maintain invariants of the layout. Apt [6] has shown that value propagation for satisfiable Boolean constraints corresponds to performing hyper-arc consistency (Chap. 12) on the constraints.

Value propagation is also related to *unit propagation*, a special case of resolution as used in the Davis-Putnam-Loveland procedure [17]. Resolution uses a different representation of Boolean constraints, namely clauses. These *satisfiability (SAT) problems* are one of the most well-studied problems in computer science.

However, the Boolean solver goes beyond propagating values, since it also propagates equalities between variables. For example, `and(1,Y,Z), neg(Y,Z)` will reduce to `false`, and this cannot be achieved by value propagation alone.

It should be noted that rules such as these can be generated automatically from the constraint theory [7, 4].

Termination. The above rules obviously terminate, since a CHR constraint is always reduced to built-in constraints.

Confluence. The solver is also confluent. The critical pairs are easy to construct, since all the heads are identical. For example, the rules (`and(X,Y,Z) <=> Z=1 | X=1,Y=1`) and (`and(X,Y,Z) <=> X=Y | Y=Z`) lead to the *critical pair* (`X=1, Y=1, X=Y, Z=1`) and (`Y=Z, X=Y, Z=1`). Both states simplify to (`X=1, Y=1, Z=1`).

Complexity. We will give an informal derivation of the worst case time complexity. Let c be the number of atomic Boolean constraints in a query. In each derivation step, one rule is applied and it will remove one constraint. Hence, there can be at most c derivation steps.

In each derivation step, in the worst case, we check each of the at most c constraints in the current state against the given set of rules. Checking the applicability of one constraint against one rule can be done in quasi-constant time. Rule application is also possible in quasi-constant time. This assumes that the syntactical equality is handled in quasi-constant time using the classical *union-find algorithm* [16].

So the worst case time complexity of applying the above rules is slightly worse than $O(c^2)$ [23].

Search

The above solver is incomplete. (It must be, since it has polynomial complexity and solving Boolean constraints has exponential complexity.) For example, the solver cannot detect inconsistency of `and(X,Y,Z), and(X,Y,W), neg(Z,W)`.

For Boolean constraints, search can be done by trying the values `0` or `1` for a variable. The search routine for Boolean constraints can be implemented in CHR$^\vee$ by a labeling procedure `enum` that takes a list of variables as argument. The order of the variables in the list determines the variable order.

```
enum([]) <=> true.
enum([X|L]) <=> bool(X), enum(L).

  bool(X) <=> (X=0 ; X=1).
```

An efficient implementation has to make sure that constraint solving and search are interleaved. In Prolog, this can be achieved by putting the search part at the end of a query.

For example, consider the query `and(X,Y,Z), and(X,Y,W), neg(Z,W), enum([X,Y,Z,W])`. The derivation will reach `enum` without simplifying any constraints. `enum` will call `bool(X)`, which will try to impose the constraint `X=0`. This will cause the constraint solver to simplify the `and` constraints into `X=0, Z=0, W=0`, which will in turn cause `neg(Z,W)` to fail. Backtracking will undo `X=0` and its consequences, and `X=1` will be tried. This time we get `Y=Z, Y=W`, and hence `neg` will fail again. There are no more choices for `X`, so the query fails finitely and *false* is returned as a result.

To improve the labeling, we can introduce a *variable ordering*. We count the occurrences of variables in a given Boolean constraint problem. Then we choose the variable that occurs most for labeling in the hope that this will cause most simplification. This heuristic is an instance of the *first-fail principle*.

We can also introduce a *value ordering*. We count the cases in which the values `0` and `1` cause simplification. In particular, choosing `0` for the last argument of `and` does not cause any simplification. Based on the counts, we may decide to try one of the values first.

Other Approaches

We briefly discuss other approaches for solving Boolean constraints.

- *Generic Consistency Methods (Local Propagation)*
  Boolean constraints can be translated into a constraint problem over finite integer domains (Chap. 12) that are solved using consistency techniques. This avoids the need for special-purpose Boolean constraint propagation algorithms and increases expressiveness, since arithmetic functions are available. The resulting constraints are called *pseudo-Boolean*.
- *Theorem Proving*
  The famous *SAT problems* can be regarded as propositional Boolean constraint problems in clausal form. Already the 3-SAT problem (conjunction of clauses with at most three variables) is NP-complete.
  Boolean constraints can be transformed into clauses in linear time and vice versa. For example, $X \sqcap Y{=}Z$ (i.e., $X \sqcap Y \leftrightarrow Z$) is logically equivalent to $(X \sqcup \neg Z) \sqcap (Y \sqcup \neg Z) \sqcap (\neg X \sqcup \neg Y \sqcup Z)$.
  Variants of resolution can be employed to solve problems in clausal form. However, the currently most successful algorithms for SAT problems do

not fit the requirements for constraint solvers, since they are based on randomly flipping the truth value of a variable in attempted solutions.

- *Integer Programming*
  This technique from operations research uses linear programming methods to solve Boolean problems expressed as linear polynomial equations over the integers. A wide range of methods exist, but the algorithms are often not incremental, i.e., all constraints have to be known from the beginning. On the other hand, it is often possible to compute best solutions that maximize a given function (optimization) (Chap. 11). Another variable elimination method that has been used is the *Gröbner basis* method [11].
- *Boolean Unification*
  An extension of syntactic unification is used to solve Boolean equalities. Boolean unification computes a single, most general solution. The problem is that Boolean variable elimination requires the introduction of auxiliary variables, and the size of the final solution may be exponential in the size of the original problem. The Boolean expressions are encoded efficiently as *binary decision diagrams (BDD).*

9.2 Application: Circuit Analysis

Boolean constraints are mainly used to model digital circuits. They are applied to generate, specialize, simulate, and analyze (verify and test) the circuits.

We consider the *full-adder circuit.* It adds three single-digit binary numbers `I1,I2,I3`, where `I3` is called the *carry-in*, to produce a single number consisting of two digits `O1,O2`, where `O2` is called the *overflow* or *carry-out.* Several full-adders can be interconnected to implement a n-bit adder.

A circuit consist of *(logical) gates*, which correspond to allowed atomic constraints. The full-adder circuit can be implemented by the rule:

```
add(I1,I2,I3,O1,O2) <=>
          and(I1,I2,A1),
          xor(I1,I2,X1),
          and(X1,I3,A2),
          xor(X1,I3,O1),
           or(A1,A2,O2).
```

For example, the constraint `add(I1,I2,I3,O1,O2),I3=0,O2=1` will reduce to `I3=0,O2=1,I1=1,I2=1,O1=0`. The derivation proceeds as follows: because `I3=0`, the output `A2` of the `and` gate with input `I3` must be `0`. As `O2=1` and `A2=0`, the input `A1` of the `or` gate must be `1`. `A1` is the output of an `and` gate, so its inputs `I1` and `I2` must be both `1`. Hence, the output `X1` of the first `xor` gate must be `0`, and therefore also the output `O1` of the second `xor` gate must be `0`.

Fault Analysis

We want to find the fault in a given hardware circuit that does not behave according to its logical specification (as given by Boolean constraints). If there is a mismatch between the expected and observed input-output behavior, we say that the circuit is *faulty*.

For fault analysis, each gate G_i is associated with a Boolean variable F_i with the meaning:

- If G_i is faulty, then $F_i = 1$.
- If $F_i = 0$, then G_i is not faulty.

Note that a faulty gate can still be correct for some inputs. For the full-adder, this leads to the following logical specification. For simplicity, we do not transform the Boolean expressions into allowed constraints.

$$\begin{array}{ll}
\neg F_1 \rightarrow (I_1 \sqcap I_2 = A_1) & \text{And gate } G_1 \\
\neg F_2 \rightarrow (I_1 \oplus I_2 = X_1) & \text{Xor gate } G_2 \\
\neg F_3 \rightarrow (X_1 \sqcap I_3 = A_2) & \text{And gate } G_3 \\
\neg F_4 \rightarrow (X_1 \oplus I_3 = O_1) & \text{Xor gate } G_4 \\
\neg F_5 \rightarrow (A_1 \sqcup A_2 = O_2) & \text{Or gate } G_5
\end{array}$$

Under the hypothesis of minimal conflict we assume that at most one gate in a faulty circuit is faulty:

$$\neg(F_1 \sqcap F_2) \wedge \neg(F_1 \sqcap F_3) \wedge \neg(F_1 \sqcap F_4) \wedge \neg(F_1 \sqcap F_5) \wedge \neg(F_2 \sqcap F_3) \wedge$$
$$\neg(F_2 \sqcap F_4) \wedge \neg(F_2 \sqcap F_5) \wedge \neg(F_3 \sqcap F_4) \wedge \neg(F_3 \sqcap F_5) \wedge \neg(F_4 \sqcap F_5)$$

For example, given the inputs I_1=0, I_2=0 and I_3=1, we observe the output O_1=0 and O_2=1. According to the above specification, the Boolean constraints will simplify and compute the following values for the fault variables: F_1=0, F_2=1, F_3=0, F_4=0, F_5=0. This means that the `xor` gate G_2 is faulty.

The rule below is a possible implementation of this specification.

```
faultanalysis(X,Y,Z,O1,O2,F1,F2,F3,F4,F5) <=>
    and(X,Y,A1),   xor(A1,I1,NF1),  imp(NF1,F1),
    xor(X,Y,XO1),  xor(XO1,I2,NF2), imp(NF2,F2),
    and(I2,Z,A2),  xor(A2,I3,NF3),  imp(NF3,F3),
    xor(Z,I2,XO1), xor(XO1,O1,NF4), imp(NF4,F4),
    or(I1,I3,OR2), xor(OR2,O2,NF5), imp(NF5,F5),

    and(F1,F2,0), and(F1,F3,0), and(F1,F4,0), and(F1,F5,0),
    and(F2,F3,0), and(F2,F4,0), and(F2,F5,0),
    and(F3,F4,0), and(F3,F5,0),
    and(F4,F5,0),

    enum([F1,F2,F3,F4,F5]).
```

10. Rational Trees *RT*

We have already introduced the constraint system E dealing with *Herbrand terms (or: first-order terms, finite trees)* and the syntactic equality constraint. Here, we consider an important variation of E.

Constraint System *RT*

Domain
Herbrand universe

Signature

- Infinitely many function symbols.
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbol $\doteq$

Constraint theory
Reflexivity:
$\forall(true \rightarrow x \doteq x)$
Symmetry:
$\forall(x \doteq y \rightarrow y \doteq x)$
Transitivity:
$\forall(x \doteq y \wedge y \doteq z \rightarrow x \doteq z)$
Compatibility:
$\forall(x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \rightarrow f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n))$
Decomposition:
$\forall(f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n) \rightarrow x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n)$
Contradiction (Clash):
$\forall(f(x_1, \ldots, x_n) \doteq g(y_1, \ldots, y_m) \rightarrow false)$ if $f \neq g$ or $n \neq m$

Allowed atomic constraints

$$C ::= true \mid false \mid s \doteq t$$

where s and t are terms over the signature Σ.

T. Frühwirth et al., *Essentials of Constraint Programming*

In early Prolog implementations, the *occur-check* was omitted for efficiency reasons. The result was a unification algorithm that could go into an infinite loop. In Prolog II, an algorithm for properly handling the resulting infinite terms was introduced [14]. This class of infinite terms is called *rational trees.*

A *rational tree* is a (possibly infinite) tree which has a finite set of subtrees. For example, the infinite tree $f(f(f(\ldots)))$ only contains itself. It has a finite representation as a directed (possibly cyclic) graph or as an equality constraint, e.g., $X \doteq f(X)$.

The constraint system *RT* for rational trees is based on the one for finite trees, the constraint system *E*. In particular, the constraint theory is just *CET* with the acyclicity axiom (occur-check) omitted.

The domain is the Herbrand universe (Appendix A) of all terms that can be built out of the function symbols in the signature.

Like *CET*, the constraint theory is decidable and satisfaction-complete. However, it is not complete. For example, $\exists X, Y \ (X \doteq f(X) \wedge Y \doteq f(Y) \wedge \neg X \doteq Y)$ does not follow from the theory, nor does its negation. One more axiom concerning implied equalities is needed for a complete theory [38].

A conjunction of allowed constraints is *solved (in solved normal form)* if it is of the form

- *false* or
- $X_1 \doteq t_1 \wedge \ldots \wedge X_n \doteq t_n$ $(n \geq 0)$, where
  - $X_1, \ldots, X_n$ are pairwise distinct
  - X_i is different to t_j if $i \leq j$

Note that an empty conjunction (n=0) is equivalent to *true*. In words, if a variable occurs on the l.h.s of an equation, it does not occur as the l.h.s. or r.h.s. of any subsequent equation.

For example, the equation $f(X, b) \doteq f(a, Y)$, the equations $X \doteq t \wedge X \doteq s$ and $X \doteq Y \wedge Y \doteq X$ are all not in solved form, while $X \doteq Z \wedge Y \doteq Z \wedge Z \doteq t$ is solved. The solved form is not unique, e.g., $X \doteq Y$ and $Y \doteq X$ are logically equivalent but syntactically different solved forms, as are $X \doteq f(X)$ and $X \doteq f(f(X))$.

From the solved form we can read off the most general unifier of the given set of initial equations by interpreting each equation $X_i \doteq t_i$ as a substitution $X_i \mapsto t_i$. We can verify this by replacing, in a fair way, in the initial equations each occurrence of X_i by t_i until the l.h.s. and r.h.s. of each equation are syntactically identical. If infinite rational trees are involved, the resulting equations will still contain variables.

10.1 Variable Elimination Constraint Solver

The following algorithm to solve equations over rational trees is similar to the one in [14], but unlike this and most other algorithms for unification, it does

not rely on substitutions (that can cause exponential blow-up of the size of terms). Dealing with infinite terms, it is the most difficult solver of the book.

The implementation relies on a total order on terms, expressed by the built-in constraint `X@<Y`. In that order, terms of smaller size are smaller, and variables are smallest and totally ordered. The *size of a term* is the number of occurrences of function symbols in the term. The order s`@<`t must hold if

- s and t are both variables and s is ordered before t.
- s is a variable and t is a function term.
- s and t are both function terms and the size of s is less than the size of t.

Note that with the total order `@<`, the conditions for the solved normal form can be restated as X_i`@<`X_{i+1} and X_i`@<`t_i, since `@<` is transitive and implies $\neq$. The built-in constraint `X@=<Y` holds if `Y@<X` does not hold.

We need some more auxiliary built-in constraints if we want to be independent of the representation of terms in the implementation: `var(X)` tests if `X` is a variable, `nonvar(X)` tests if `X` is not a variable. `same_functor(T1,T2)` tests if `T1` and `T2` have the same function symbol and the same arity. `args2list(T1,L1)` holds if `L1` is the list of arguments of the term `T1`.

The auxiliary CHR constraint `same_args(L1,L2)` pairwise equates the elements of the two lists.

```
reflexivity   @ X eq X <=> var(X) | true.

orientation   @ T eq X <=> var(X),X@<T | X eq T.

decomposition @ T1 eq T2 <=> nonvar(T1),nonvar(T2) |
                same_functor(T1,T2),
                args2list(T1,L1),args2list(T2,L2),
                same_args(L1,L2).

confrontation @ X eq T1, X eq T2 <=> var(X),X@<T1,T1@=<T2 |
                X eq T1, T1 eq T2.

  same_args([],[]) <=> true.

  same_args([T1|L1],[T2|L2]) <=> T1 eq T2, same_args(L1,L2).
```

It is easy to see that the logical readings of the rules `reflexivity` and `orientation` are consequences of the corresponding axioms *Reflexivity* and *Symmetry* in the constraint theory. The rule `decomposition` implements the axioms *Compatibility*, *Decomposition*, and *Contradiction (Clash)*. When there is a clash, `same_functor` will fail. The rule `confrontation` is a consequence of *Transitivity* and *Symmetry*. The rule was chosen over transitivity for efficiency (it does not increase the number of equations). It performs a limited amount of variable elimination by only considering l.h.s. of equations.

If no more rule of the solver is applicable, the final conjunction of equations is in solved form. This can be proven by contradiction: if the equations were not solved, one of the rules would be applicable.

Example 10.1.1. We equate two terms. The constraints that are rewritten by a transition are underlined. For readability, we do not show the intermediate states involving the auxiliary CHR constraint `same_args`.

```
                        h(Y,f(a),g(X,a)) eq h(f(U),Y,g(h(Y),U)))
↦decomposition↦*        Y eq f(U), f(a) eq Y, g(X,a) eq g(h(Y),U)
↦orientation            Y eq f(U), Y eq f(a), g(X,a) eq g(h(Y),U)
↦decomposition↦*        Y eq f(U), Y eq f(a), X eq h(Y), a eq U
↦orientation            Y eq f(U), Y eq f(a), X eq h(Y), U eq a
↦confrontation          Y eq f(U), f(U) eq f(a), X eq h(Y), U eq a
↦decomposition↦*        Y eq f(U), U eq a, X eq h(Y), U eq a
↦confrontation          Y eq f(U), U eq a, X eq h(Y), a eq a
↦decomposition↦*        Y eq f(U), U eq a, X eq h(Y)
```

Example 10.1.2. Here is a simple example involving infinite rational trees.

```
                        X eq f(X), X eq f(f(X))
↦confrontation          X eq f(X), f(X) eq f(f(X))
↦decomposition↦*        X eq f(X), X eq f(X)
↦confrontation          X eq f(X), f(X) eq f(X)
↦decomposition↦*        X eq f(X), X eq X
↦reflexivity            X eq f(X)
```

i.e., the second constraint is redundant.

Termination. The solver terminates, the proof of [14] or the proof of [19] can be adapted. It is based on the following observations.

- The solver only produces equations between given terms or their subterms.
- The `reflexivity` rule removes an equation.
- The `orientation` is only applicable at most once to an equation, since its arguments can only be reversed at most once.
- The `decomposition` rule produces equations between the arguments of the initial equations, thus the new equations have arguments of smaller size.
- The `confrontation` rule does not change the first equation and replaces the second equation `X eq T2` by `T1 eq T2`. The guard of the rule ensures that `T1` is between `X` and `T2` in the order `@<`. This means that with repeated applications of the rule to the second equation, the current `T1` gets closer from below to `T2` but can never exceed it. Since there is only a finite number of equations and terms up to a given size in any given problem, the `confrontation` rule cannot be applied infinitely often.

Confluence. For a given equation, the guards of the rules exclude each other pairwise, so at most one type of rule is applicable to a given equation. However, in a conjunction of equations there may be several ways of applying the `confrontation` rule. Indeed, the solver is not confluent due to the `confrontation` rule. For example, depending on the order of rule applications, applying the confrontation rule to exhaustion to the equations `X eq Y1, X eq Y2, X eq Y3`, where variables are ordered according to their first occurrence, may yield `X eq Y1, Y1 eq Y2, Y2 eq Y3` or `X eq Y1, Y1 eq Y3, Y2 eq Y3`.

Complexity. The auxiliary built-in constraints can be implemented such that they take constant time. For the order constraints this can be achieved by computing the sizes of the terms and their subterms in a given problem once and storing them. For a single equation, the `reflexivity` and `orientation` rule need constant time. The `decomposition` rule can be repeatedly applied to the terms in an equation until it fails or one of the arguments is a variable. This complete decomposition will at worst take time linear in the number of function symbols and variables in the initial equation. The number of variables also gives an upper bound on the number of equations that can be produced by the rule. Applying the `confrontation` rule to exhaustion to two given equations will take time linear in the number of different variables in the given problem, because the rule replaces a variable by a larger variable until a function term is reached where each rule application can be made in constant time using indexing on variables.

Due to the intricate interaction between the `decomposition` rule and the `confrontation` rule, the complexity of the overall solver is worse than linear. It is an open problem if the worst case time complexity of the solver is polynomial in the number of function symbols and variables that occur in a given problem.

Classical Algorithms

Herbrand [27] gave an informal description of a unification algorithm, [47] rediscovered a similar algorithm when he introduced his resolution procedure for first-order logic. There are quasi-linear time algorithms for unification. They can be considered as extensions of the *union-find algorithm* [16] from constants to trees. For finite trees (Herbrand terms), see [39] and [45]. For rational trees, see [30].

10.2 Application: Program Analysis

Syntactic equality of terms is an essential constraint system for (constraint) logic programming, since terms are the universal data structure and equalities can be used to build, access, and take apart terms.

In program analysis, one represents and reasons about properties of programs. We will use an infinite rational tree to represent a recursive data type and to type check terms.

A *list* is defined recursively:

- the constant `nil` is a list.
- A binary list constructor `cons` applied to a term of type `Element` and to a list results in a list.

The type of lists can be defined by a rational tree:

```
List eq (nil or cons(Element,List)).
```

where `or` is a binary infix operator separating alternatives that is only used in type expressions. A term is of type `list` if we can map it onto this rational tree.

The following type checker defines a constraint `Term of Type` that holds if `Term` is of type `Type`.

```
Term of (Type1 or Type2) <=>
        (Term of Type1 ; Term of Type2).

Term of Type <=> nonvar(Term),nonvar(Type) |
        same_functor(Term,Type),
        args2list(Term,Args),args2list(Type,Types),
        check_args(Args,Types).

Term of Type, Type eq Type1 <=> var(Type) |
        Term of Type1, Type eq Type1.

    check_args([],[]) <=> true.

    check_args([Arg|Args],[Type|Types]) <=>
          Arg of Type, check_args(Args,Types).
```

Note the similarity with the rational tree solver.

We may use the constraint `list(List, Element)` to generate the type tree for lists:

```
list(List, Element) <=> List eq (nil or cons(Element,List)).
```

Then the constraint `list(NumberList,(0 or 1 or ...or 9))` defines a list over single-digit numbers. The constraint `cons(3,cons(0,nil)) of NumberList` performs a type check together with the `list` constraint. After simplifying `list` into an equation `eq`, the last rule of `of` will generate the new constraint (abbreviating `NumberList` do `NL`):

```
cons(3,cons(0,nil)) eq (nil or cons((0 or 1 or ... or 9),NL))
```

The first rule of `of` will first try

```
cons(3,cons(0,nil)) of nil
```

This will lead to failure with the second rule of `of`, on backtracking

```
cons(3,cons(0,nil)) of cons(0 or 1 or ... or 9,NL)
```

is tried. This leads to

```
check_args([3,cons(0,nil)],[0 or 1 or ... or 9,NL])
```

which leads to

```
3 of 0 or 1 or ... or 9, cons(0,nil) of NL
```

This will finally lead to success.

11. Linear Polynomial Equations $\Re$

One motivation for introducing constraints in the LP language Prolog was the non-declarative nature of the built-in predicates for arithmetic computations. Therefore, the first CLP languages included constraint solvers for linear polynomial equations and inequations over the real numbers (CLP($\Re$) [33]) or rational numbers (Prolog-III [15], CHIP [20]).

Constraint System $\Re$

Domain
The set $\Re$ of real numbers

Signature

- Function symbols.
  - The real numbers 0 and 1
  - Unary prefix operators $+$ and $-$
  - Binary infix operators $+$ and $*$
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$

Constraint theory
The linear existential fragment of Tarski's axiomatic theory of real closed fields for elementary geometry.

Allowed atomic constraints
Linear equations and inequations:

$$C ::= \mathit{true} \mid \mathit{false} \mid a_1 * X_1 + \ldots + a_n * X_n + b \odot 0,$$

where $n \geq 0$, $a_i, b \in \Re$, the coefficients $a_i \neq 0$, the variables $X_1, \ldots, X_n$ are totally ordered in strictly descending order, and $\odot \in \{=, <, \leq, >, \geq, \neq\}$. The l.h.s. of the equation is called *(linear) polynomial.*

Tarski's theory of real closed fields covers linear and non-linear polynomials [55]. It only refers to the real numbers 0 and 1, which are included

T. Frühwirth et al., *Essentials of Constraint Programming*

in the signature. The interpretation will map arithmetic expressions to the real numbers, which form the domain. The theory is complete and decidable, but intractable. However, the linear existential fragment is decidable in polynomial time.

11.1 Variable Elimination Constraint Solver

Typically, in constraint solvers, incremental variants of classical variable elimination algorithms [31] like Gaussian elimination for equations and Dantzig's Simplex algorithm for equations and inequations are implemented. Gaussian elimination has cubic complexity in the number of different variables in a problem. The Simplex algorithm has exponential worst case complexity but is polynomial on average.

For the implementation of these and similar algorithms, it does not matter if real or rational numbers are used. However, there are technicalities. In implementations, reals are represented by floating-point numbers. Therefore, rounding errors are unavoidable. A partial remedy is to avoid using variables for elimination that have a small coefficient. Rational numbers are precise, but their size can grow exponentially in the size of the problem (due to multiplication operations), thus they cause efficiency problems.

To illustrate the principle of *variable elimination*, we first consider conjunctions of equations only. It is *in solved form* if the left-most variable of each equation does not appear in any other equation. We compute the solved form by eliminating multiple occurrences of variables.

- Choose an equation $a_1 * X_1 + \ldots + a_n * X_n + b = 0$.
- Make its left-most variable explicit: $X_1 = -(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$.
- Replace all other occurrences of X_1 by $-(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$.
- Simplify the resulting equations into allowed constraints (this is always possible).
- Repeat until solved.

Actually, since constraints should be processed incrementally, we cannot eliminate a variable in *all* other equations at once, but rather consider the other equations one by one. Also, we do not make a variable explicit, but keep the original equation.

```
eliminate @ A1*X+P1 eq 0, PX eq 0 <=>
            find(A2*X,PX,P2) |
            normalize(A2*(-P1/A1)+P2,P3),
            A1*X+P1 eq 0, P3 eq 0.

empty @ B eq 0 <=> number(B) | zero(B).
```

The `eliminate` rule performs variable elimination. It takes any pair of equations with a common occurrence of a variable, `X`. In the first equation, the

variable appears left-most. This equation is used to eliminate the occurrence of the variable in the second equation. The first equation is left unchanged.

In the guard, the built-in constraint `find(A2*X,PX,P2)` tries to find the expression `A2*X` in the polynom `PX`, where `X` is the common variable. The polynom `P2` is `PX` with `A2*X` removed. The built-in constraint `normalize(E,P)` normalizes an arithmetic expression `E` into a linear polynomial `P`.

The `empty` rule says that if the polynomial contains no more variables, then the number `B` must be zero.

The solver is satisfaction-complete since it produces the solved form. (If a set of equations is not in solved form, then one of the rules of the solver is applicable.)

Example 11.1.1. The two equations

```
1*X+3*Y+5 eq 0, 3*X+2*Y+8 eq 0
```

match the `eliminate` rule, the variable `X` in the second equation is removed via

```
normalize(3*(-(3*Y+5)/1) + (2*Y+8), P3).
```

The resulting equations are

```
1*X+3*Y+5 eq 0, -7*Y+ -7= 0
```

This means that `Y` is `-1`. The `eliminate` rule is once again applicable, this time the order of the matching equations is reversed and `Y` is removed from the first equation via

```
normalize(3*(-(-7)/-7) + (1*X+5), P3)
```

The final result is:

```
1*X+2 eq 0,      -7*Y+ -7= 0
```

This means that `X` is `-2`.

Termination. The solver terminates. There is a finite number of variables for any given constraint problem and no new variables are introduced during derivation. The variables in each polynomial equation are ordered in strictly descending order. Hence, in the `eliminate` rule, the left-most, i.e., largest, variable of an equation is replaced by several strictly smaller ones.

Confluence. The solver is not confluent. Consider two equations with the same left-most variable. The rule `eliminate` can be applied in two different ways, resulting in different pairs of equations.

Complexity. Consider a problem with c equations and v different variables. c and v do not increase during derivation. So there can be at most cv occurrences of variables in a state of the derivation.

The complexity is determined by the cost of applying the `eliminate` rule. Each application of the rule removes a single occurrence of a variable from one equation. Hence, there are at most cv rule applications.

Every rule application introduces exactly one new constraint. If we want to apply the `eliminate` rule to it, we have to look for a partner constraint among the other $O(c)$ constraints. So there are $O(c)$ rule application attempts in the worst case.

The guard of the `eliminate` rule uses `find`, which can be implemented with a complexity linear in v. (If the rule is applied, `normalize` incurs a cost also linear to v.) So trying to apply the rule to a given constraint has complexity $O(cv)$.

Hence, the overall complexity is $O(c^2v^2)$, i.e., quadratic in the size of the problem.

Determined Variables

The solver can be extended by a rule to detect determined variables:

```
determine @ A*X+B eq 0 <=> number(B) | X is -B/A.
determined @ P eq 0 <=> find(A*X,P,P1),number(X) |
                  normalize(A*X+P1,P2), P2 eq 0.
```

The second rule is needed to deal with newly determined variables.

Inequations

We extend our solver to inequations. The idea is to use flattening to transform an inequation into an equation and an inequation on one variable: As in the Simplex algorithm, an inequation is replaced by an equation with the help of an additional variable, called a *slack variable*, that stands for the value of the polynom and captures the inequality with an additional constraint. For example, $P \geq 0$ is rewritten into $P = S \wedge S \geq 0$. In general, $P \odot 0$ is rewritten into $P = S \wedge S \odot 0$, where $\odot \in \{<, \leq, >, \geq, \neq\}$.

After normalizing the equation into an allowed constraint, one may apply the solver for equations and ignore the simple inequations until values for the slack variables are known (determination). However, this method alone is not satisfaction-complete anymore. A set of equations consisting only of slack variables (called *slack-only equations*) may be inconsistent even if different from *false* in solved form. For example, the conjunction of constraints $3 * S_1 + 4 * S_2 + 0 = 0 \wedge S_1 \geq 0 \wedge S_2 > 0$ is inconsistent.

To achieve satisfaction-completeness in the presence of slack-only equations, one can either introduce a more strict solved form (as done in CHIP),

or do more variable elimination to derive all implicit equalities (as done in CLP($\Re$)).

For the improved solved form used in CHIP, the slack variables in all equations have to be *re*ordered (and *re*solved) such that the coefficient of the left-most slack variable of an equation has a different sign than the constant. If this reordering is not possible, the equations are inconsistent. For example, $2 * S_1 + 3 * S_2 + 1 = 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0$ is inconsistent.

Optimization and Related Approaches

The Simplex algorithm can do more than solving equations. It can be used for *optimization*: it finds the solution of the equations that maximizes the value of a given *objective function*. This function is a linear polynom that we add to the problem. This problem is well studied in operations research, under the name of *linear programming* [51].

Another method for solving such optimization problems is the *barrier or interior-point method*, which was adapted from non-linear programming [34]. Both methods move from one solution to the next better one, until an optimum is found. These algorithms are therefore not directly suitable for implementation inside constraint solvers.

Symbolic arithmetic software packages like Mathematica, Maple and CPLEX offer even more solving power than these algorithms, but they are not tightly integrated in a programming language for solving conjunctions of allowed constraints incrementally and efficiently. However, such systems have been successfully loosely coupled with the CLP language Eclipse at IC-PARC and in the ILOG Optimization Suite.

11.2 Application: Finance

Mortgage

The calculation of a mortgage is one of the classic examples of CLP. The scenario is that one takes a loan and pays back a certain amount for a certain number of months at a certain interest rate. The mortgage calculation can be concisely expressed by a recursive rule in CHR$^\vee$:

```
% D: Amount of Loan, Debt, Principal
% T: Duration of loan in months
% I: Interest rate per month
% R: Rate of payments per month
% S: Balance of debt after T months

mortgage(D, T, I, R, S) <=>
       T eq 0,
```

```
D eq S
  ;
T gt 0,
T1 eq T - 1,
D1 eq D + D*I - R,
mortgage(D1, T1, I, R, S).
```

The base case is that we do not pay back any more, i.e., `T eq 0`. Then the current debt is the final balance, i.e., `D eq S`. Otherwise `T gt 0`, and we calculate the remaining debt `D1` in the next month (`T1 eq T-1`) taking into account the repayment `R` and the interest rate `I`.

The query `mortgage(100000,360,0.01,1025,S)` results in `S`$\doteq$`12625.90` (rounded). This demonstrates the effect of accumulation of interest: even though we have paid back 360 times 1025 over time, there is still a final debt of 12625.90.

A characteristic of constraint programming is the lack of a notion of inputs and outputs. With the same rule, we can also compute what initial loan we can pay back completely under the conditions above: the query `mortgage(D,360,0.01,1025,0)` results in `D`$\doteq$`99648.79`, only a slightly lower amount.

But how much longer would we have to pay for the original loan of 100000? The query `S le 0, mortgage(100000,T,0.01,1025,0)` is unsatisfiable. This is because the repayment does not exactly add up to the loan with the accumulated interest. The query

```
-1025 lt S, S le 0, mortgage(100000,T,0.01,1025,S)
```

results in `T`$\doteq$`374, S`$\doteq$`-807.96`, so the repayment in the final, 374th month is not the full rate, it is just $1025 - 807.96$.

We may also be interested in the general relationship between initial loan and monthly rate of repayment under our initial conditions: The query `mortgage(D,360,0.01,R,0)` results in `R eq 0.0102861198*D`, i.e., the monthly payment is about 1% of the loan.

However, if the interest rate `I` is left unknown, the equation `D1 eq D + D*I - R` will be non-linear after one recursion step, since `D1`, the new `D`, is not known. Proceeding with the recursion will thus not determine `D1` and `D`, the equation remains non-linear.

12. Finite Domains *FD*

In this constraint system, variables are constrained to take their value from a given, finite set. Choosing integers for values allows for arithmetic expressions as constraints. Constraint propagation proceeds by removing values from the sets of possible values that do not participate in any (partial) solution.

Constraint System *FD*

Domain
The set $\mathcal{Z}$ of integers

Signature

- Function symbols.
  - The integers 0 and 1
  - Lists
  - Binary infix operators $+$, and $..$ for intervals
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$, and *in* for domains

Constraint theory
Presburger's arithmetic extended by

- $X \leq Y \leftrightarrow \exists Z\, X + Z = Y$
- $X \ in \ n..m \leftrightarrow n \leq X \wedge X \leq m$
- $X \ in \ [k_1, \ldots, k_l] \leftrightarrow X = k_1 \vee \ldots \vee X = k_l$

Allowed atomic constraints
Linear equations and inequations:

$$C ::= true \mid false \mid X \ in \ n..m \mid X \ in \ [k_1, \ldots, k_l] \mid X \odot Y \mid X + Y = Z$$

where $n,\ m,\ k_1, \ldots, k_l (l \geq 0)$ are integers, $\odot \in \{=, <, >, \leq, \geq, \neq\}$ and $X,\ Y$ and Z are pairwise distinct variables.

T. Frühwirth et al., *Essentials of Constraint Programming*

Finite domains are one of the success stories of CLP. Many real-life combinatorial problems can be expressed in this constraint system, most prominently scheduling and planning applications. This constraint system appeared in one of the first CLP languages CHIP [20]. It was the result of a synthesis of LP and finite-domain constraint networks as explored in artificial intelligence research since the late 1960's. Other influential CLP languages based on finite domains are clp(FD) [13] and cc(FD) [58].

The theory underlying this constraint system is Presburger's arithmetic. It axiomatizes the linear fragment of arithmetic over natural numbers with + and =. It is complete and decidable. The theory only refers to the numbers 0 and 1, which are included in the signature. The interpretation will map arithmetic expressions to the integers, which form the domain. Linearity means that there is no multiplication between variables. Presburger's theory can be extended to accommodate the additional constraints and negative numbers.

The *domain constraint* X *in* D means that the variable X takes its value from the given finite domain D. More precisely, X *in* $[k_1, \ldots, k_l]$ denotes an *enumeration domain constraint*, where the possible values of X are explicitly enumerated. X *in* $n..m$ denotes an *interval domain constraint*, where the values of X must be in the given interval $n..m$ (bounds included). According to the constraint theory, a domain constraint with the empty domain, X *in* [] or X *in* $n..m$ $(n > m)$, is unsatisfiable.

The difference between an interval domain and an enumeration domain is in their algorithmic use. In the former, constraint simplification is performed only on the interval bounds, while in the latter each element in the enumeration is considered. For example, from X *in* $[1, 2, 3] \wedge X{\neq}2$ we can derive the tighter domain constraint X *in* $[1, 3]$, while from X *in* $1..3 \wedge X{\neq}2$ no constraint propagation is possible, since proper intervals cannot have "holes". Thus, enumeration domains allow more simplification (tighter domains). On the other hand, they are only tractable for sufficiently small enumerations.

The allowed atomic constraints are in *flat normal form* and integers are not allowed in the place of variables. A determined variable $(X{=}v)$ is expressed by a domain constraint X *in* $[v]$ or X *in* $v..v$.

Any linear polynomial equation can be expressed as a conjunction of allowed constraints. First, the coefficients of the polynomial are multiplied with a number such that they are all become integers. Then the multiplications are rewritten as sums, e.g., $3X$ becomes $X + X + X$. Finally, the resulting expression is flattened. For example, $X{+}X{+}Y{>}5$ is flattened into $W{>}F \wedge X{+}V{=}W \wedge X{+}Y{=}V \wedge F$ *in* $[5]$.

12.1 Arc Consistency

Arc consistency is a classical local-consistency algorithm from constraint networks in artificial intelligence that originally was restricted to enumeration

domain constraints and binary constraints. *Hyper-arc consistency* extends arc consistency from binary to arbitrary n-ary constraints.

We will employ forms of *arc consistency* enforcement to simplify finite-domain constraint problems. In an arc-consistent atomic constraint, every value of every domain takes part in a solution of the constraint. To achieve arc consistency, it suffices to find and remove those values that do not participate in any solution. Similar to propagating values in the case of Boolean constraints (Chap. 9), we propagate sets of possible values.

Let $X_1, \ldots, X_n$ be pairwise distinct variables. An atomic constraint $c(X_1, \ldots, X_n)$ is *(hyper-)arc consistent* with respect to a conjunction of enumeration domain constraints X_1 *in* $D_1 \wedge \ldots \wedge X_n$ *in* D_n, if for all $i \in \{1, \ldots, n\}$ and for all values v_i in D_i ($v_i \in \{k_{1i}, \ldots, k_{li}\}$) the constraint $\exists(X_1$ *in* $D_1 \wedge \ldots \wedge X_i{=}v_i \wedge \ldots \wedge X_n$ *in* $D_n \wedge c(X_1, \ldots, X_n))$ is satisfiable. A conjunction of constraints is arc consistent if each atomic constraint in it is arc consistent.

In other words, an atomic constraint is arc consistent if for each variable in the constraint and for each value in the domain of the variable, there exist values in the domains of the other variables such that the constraint is satisfied.

For example, X *in* $[1,2,3] \wedge X{\neq}2$ is not arc consistent, but X *in* $[1,3] \wedge X{\neq}2$ is. The constraint X *in* $[1,2,3] \wedge Y$ *in* $[1,2,3] \wedge X{<}Y$ is not arc consistent, but X *in* $[1,2] \wedge Y$ *in* $[2,3] \wedge X{<}Y$ is.

The definition of arc consistency requires that the variables in each atomic constraint are pairwise distinct. Multiple occurrences of the same variable must be renamed apart. For example, the constraint $X{\neq}X$ is represented by $X{\neq}Y \wedge X{=}Y$. Note that the unsatisfiable constraint X *in* $D \wedge Y$ *in* $D \wedge X{\neq}Y \wedge X{=}Y$ is arc consistent for all domains D with more than one value. Arc consistency does not imply satisfiability.

Because arc consistency is sensitive to flattening, propagation in the flat normal form can be weaker than otherwise. For example, the constraint X *in* $[1,2] \wedge Z$ *in* $[2,3,4] \wedge 2X{=}Z$ is not arc consistent, but its flattened normal form X *in* $[1,2] \wedge Y$ *in* $[1,2] \wedge Z$ *in* $[2,3,4] \wedge X{=}Y \wedge X{+}Y{=}Z$ is.

An atomic constraint can be made arc consistent by deleting those values from the domain of its variables that do not participate in any solution of the constraint. A conjunction of constraints can be made arc consistent by making each atomic constraint arc consistent. Obviously, this approach describes a local-consistency algorithm (Chap. 8), because we consider sub-problems of one atomic constraint together with the domain constraints of its variables.

The worst case time complexity of (hyper-)arc consistency is $O(cd^n)$ for arbitrary n-ary constraints [36, 43, 44], where c is the number of constraints and d is the size of the largest domain.

The finite-domain constraints of the programming language CHIP include *global constraints* that can take an arbitrary number of variables as argu-

ment and where domain propagation is performed on them simultaneously. For example, the constraint *alldifferent*$(X_1, \ldots, X_n)$ is logically equivalent to $\bigwedge X_i \neq X_j$ $(1 \leq i < j \leq n)$. However, arc consistency does not detect the unsatisfiability of $X_1 \neq X_2 \wedge X_1 \neq X_3 \wedge X_2 \neq X_3$ when the variables are constrained to the same domain of two values. On the contrary, the sophisticated algorithms that are employed with *alldifferent* achieve more propagation than arc consistency and detect unsatisfiability in this case [46].

For interval domains, a weaker but analogous form of arc consistency proves useful.

Let $X_1, \ldots, X_n$ be pairwise distinct variables. An atomic constraint $c(X_1, \ldots, X_n)$ is *bounds (or: box) consistent* with respect to a conjunction of interval domain constraints X_1 *in* $D_1 \wedge \ldots \wedge X_n$ *in* D_n, if for all $i \in \{1, \ldots, n\}$ and for all bounds v_i in D_i $(v_i \in \{n_i, m_i\})$ the constraint $\exists(X_1$ *in* $D_1 \wedge \ldots \wedge X_i = v_i \wedge \ldots \wedge X_n$ *in* $D_n \wedge c(X_1, \ldots, X_n))$ is satisfiable. A conjunction of constraints is bounds consistent if each atomic constraint in it is bounds consistent.

For example, X *in* $1..3 \wedge X \neq 2$ is bounds consistent. X *in* $1..3 \wedge Y$ *in* $1..3 \wedge X < Y$ is not bounds consistent, but X *in* $1..2 \wedge Y$ *in* $2..3 \wedge X < Y$ is. The constraint X *in* $D \wedge Y$ *in* $D \wedge X \neq Y \wedge X = Y$ is bounds consistent for all interval domains D with more than one value.

Analogously to arc consistency enforcement, constraints can be made bounds consistent by tightening their interval domains.

12.2 Local-Propagation Constraint Solver

For simplicity, we start with the bounds consistency algorithm for interval constraints [56, 10]. The implementation is based on interval arithmetic.

Interval Domains

In the solver, `in`, `le`, `eq`, and `add` are CHR constraints, the inequalities `<`, `>`, `=<`, `>=`, and `\=` are built-in arithmetic constraints, and `min`, `max`, `+`, and `-` are built-in arithmetic functions. Intervals of integers are closed under computations involving only these functions. The rules for bounds consistency affect the interval constraints only, the constraints `le`, `eq`, and `add` remain unaffected.

```
inconsistency @ X in A..B <=> A>B | false.
intersection  @ X in A..B, X in C..D <=>
                X in max(A,C)..min(B,D).
```

The `inconsistency` rule detects inconsistency due to an empty interval. The `intersection` rule intersects two intervals for the same variable.

Here are some sample rules for inequalities:

```
le @ X le Y, X in A..B, Y in C..D <=> B>D |
     X le Y, X in A..D, Y in C..D.
le @ X le Y, X in A..B, Y in C..D <=> C<A |
     X le Y, X in A..B, Y in A..D.

eq @ X eq Y, X in A..B, Y in C..D <=> A\=C |
     X eq Y, X in max(A,C)..B, Y in max(C,A)..D.
eq @ X eq Y, X in A..B, Y in C..D <=> B\=D |
     X eq Y, X in A..min(B,D), Y in C..min(D,B).

ne @ X ne Y, X in A..B, Y in C..D <=> A=C,C=D |
     X ne Y, X in (A+1)..B, Y in C..D.
...
```

`X le Y` means that `X` is less than or equal to `Y`. Hence, `X` cannot be larger than the upper bound `D` of `Y`. Therefore, if the upper bound `B` of `X` is larger than `D`, we can replace `B` by `D` without removing any solutions. Analogously, one can reason on the lower bounds to tighten the interval for `Y`. The `eq` constraint enforces the intersection of the intervals associated with its variables provided the bounds are not yet the same. The `ne` constraint can only cause a domain tightening if one of the intervals denote a unique value that happens to be the bound of the other intervals.

Example 12.2.1. Here is a sample derivation involving `le`:

```
      A in 2..3, B in 1..2, A le B
↦le   B in 1..2, A le B, A in 2..2
↦le   A le B, A in 2..2, B in 2..2.
```

Finally, we implement the ternary constraint for addition, where $X+Y=Z$ is represented in relational form as `add(X,Y,Z)`:

```
add @ add(X,Y,Z), X in A..B, Y in C..D, Z in E..F <=>
      not (A>=E-D,B=<F-C,C>=E-B,D=<F-A,E>=A+C,F=<B+D) |
      add(X,Y,Z),
      X in max(A,E-D)..min(B,F-C),
      Y in max(C,E-B)..min(D,F-A),
      Z in max(E,A+C)..min(F,B+D).
```

For addition, we use interval addition and subtraction to compute the interval of one variable from the intervals of the other two variables. Note that when an interval is subtracted, its bounds have to be interchanged. This is because $-(n..m) = (-m.. - n)$. These computed intervals are intersected with the existing intervals using `min` and `max`. The guard ensures that at least one interval becomes smaller whenever the rule is applied. (The built-in prefix operator `not` negates its argument, a conjunction of built-in constraints.)

Example 12.2.2. Here is an example derivation involving `add`:

```
A in 1..3, B in 2..4, C in 0..4, add(A,B,C)    ↦add
A in 1..3, B in 2..4, C in 0..4, add(A,B,C),
A in -1..2, B in 0..3, C in 3..7               ↦*intersection
add(A,B,C), A in 1..2, B in 2..3, C in 3..4
```

Termination. The rules `inconsistency` and `intersection` remove one interval constraint each. We assume that the remaining rules deal with non-empty intervals only. This can be enforced by additional guard constraints on the interval bounds which have been omitted from the code for readability. We can use the inequalities in the guards of the rules to show that in each rule, at least one interval in the body is strictly smaller than the corresponding interval in the head, while the other intervals remain unaffected.

Confluence. The solver is confluent provided the intervals are given.

Complexity. We assume that the arithmetic built-in constraints take constant time to compute. Given a variable, its associated domain constraint can be found in constant time using indexing on that variable. (There is only one such domain if the first two rules of the program are always applied first.)

Given a constraint problem, let $w = m{-}n{+}1$ be the maximum *width (size)* of an interval constraint X *in* $n..m$, v be the number of different variables and c be the number of constraints. c and v do not increase during derivation.

Since each rule application makes at least one interval smaller, the worst number of rule applications is $O(vw)$, it is not dependent on the number of constraints. Note that $O(v)$ may not exceed $O(c)$, since each allowed atomic constraint has at most three different variables.

Each rule application will generate a fixed number of new interval domain constraints. What is the cost of processing a new interval constraint? Its variable may appear in all c constraints, so there may be up to $O(c)$ rule tries (rule application attempts). Each rule try has constant cost.

So the worst case time complexity is $O(cvw)$.

Enumeration Domains

The rules for enumeration domains are similar to the ones for interval domains. Instead of interval arithmetic, we have to perform arithmetic operations on enumerations, i.e., sets of values, by performing the operations on each possible tuple of values.

In the exemplary rules below we assume that all domains are enumeration domains. We also assume that the arithmetic functions `max` and `min` are also applicable to lists of values. `filter_max` removes all values from a list that are larger than any value in another list.

```
inconsistency @ X in [] <=> false.
intersection  @ X in L1, X in L2 <=>
                intersection(L1,L2,L3), X in L3.
```

```
le @ X le Y, X in L1, Y in L2 <=> max(L1) > max(L2) |
     filter_max(L1,L2,L3),
     X le Y, X in L3, Y in L2.
...
```

Example 12.2.3. The query `X le Y, X in [4,6,7], Y in [3,7]` leads to `X le Y, X in [4,6,7], Y in [7]`. The query `X le Y, X in [2,3,4,5], Y in [1,2,3]` leads to `X le Y, X in [2,3], Y in [2,3]`. The query `X le Y, X in [2,3,4], Y in [0,1]` leads to `false`.

The built-in constraint `diff` holds if its argument are lists with different elements.

```
eq @ X eq Y, X in L1, Y in L2 <=> diff(L1,L2) |
     intersection(L1,L2,L3),
     X eq Y, X in L3, Y in L3.
```

The rule for addition of enumeration domains is as follows. The built-in constraints `all_substractions` and `all_additions` pairwise subtract and add their list arguments, respectively. The constraints return a list of all values computed, but without duplicates.

```
add @ add(X,Y,Z), X in L1, Y in L2, Z in L3 <=>
    all_substractions(L3,L2,L4),
    all_substractions(L3,L1,L5),
    all_additions(L1,L2,L6),
    not (L1=L4,L2=L5,L3=L6),
    |
    X in L4, Y in L5, Z in L6.
```

The arguments for termination, confluence, and complexity of this enumeration domain solver are similar to the interval domain solver. The complexity changes. Instead of the interval width w, we use the maximum size of an enumeration domain denoted by d. Because operations on arbitrarily large enumeration domains as performed by the built-in constraints may take up to $O(d^2)$, the overall complexity is thus $O(cvd^3)$.

Search

To achieve satisfaction-completeness, search must be employed. We implement the search routine analogous to the one for Boolean constraints (Chap. 9).

```
enum([]) <=> true.
enum([X|Xs]) <=> indomain(X), enum(Xs).
```

For enumeration domains, each value in the enumeration domain is tried. Note that $X{=}v$ is expressed as the allowed constraint `X in [V]`.

```
indomain(X), X in [V|L] <=> L=[_|_] |
             (X in [V] ; X in L, indomain(X)).
```

The guard ensures termination.

For interval domains, search is usually done by splitting intervals in two halves. This splitting can be repeated until the bounds of the interval are the same.

```
indomain(X), X in A..B <=> A<B |
              C is (A+B)//2,
             (X in A..C ; X in (C+1)..B), indomain(X).
```

The guard ensures termination.

Implementations

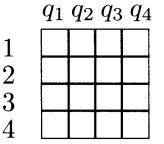
In practice, a hybrid, compact form of domains is used in implementations. The domain is a list of intervals, so an interval can have holes since it can be split if a value inside the interval needs to be removed. For small enumeration domains, bit vectors have also been used.

12.3 Applications: Puzzles and Scheduling

n-Queens Problem

The famous n-queens problem asks to place n queens $q_1, \ldots, q_n$ on an $n \times n$ chess board, such that they do not attack each other. This means that there are no two queens on the same row, column or diagonal. Since n queens are to be placed on a $n \times n$ chess board, each row and each column must have exactly one queen, and each diagonal at most one queen. We will use a variable q_i to denote the row position of the queen in the i-th column:

$q_1, \ldots, q_n \in \{1, \ldots, n\}$.



For any two queens, it holds that they are not on the same row and not on the same diagonal: $\forall\ i \neq j.\ q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|$.

The problem can be solved with a CHR program, where `N` is the size of the chess board and `Qs` is a list of `N` queen position variables:

```
solve(N,Qs) <=> make_domains(N,Qs), queens(Qs), enum(Qs).

queens([]) <=> true.
queens([Q|Qs]) <=> safe(Q,Qs,1), queens(Qs).
```

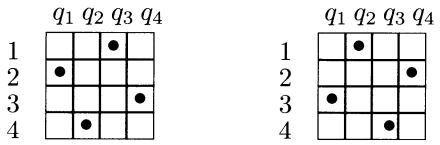
```
safe(X,[],N) <=> true.
safe(X,[Y|Qs],N) <=> no_attack(X,Y,N), safe(X,Qs,N+1).

no_attack(X,Y,N) <=> X ne Y, X+N ne Y, Y+N ne X.
```

The constraint `make_domains(N,Qs)` introduces an enumeration domain constraint with values from `1` to `N` for each queen in `Qs`. `queens` and `safe` are used to introduce a `no_attack` constraint between each ordered pair of queens. The parameter `N` is the value for $|i-j|$, the distance between the columns associated with the pair of queens. Finally, each `no_attack` constraint enforces the condition $\forall\, i{\neq}j.\ q_i{\neq}q_j \wedge |q_i - q_j| {\neq} |i-j|$ expressed as finite-domain constraints. (For simplicity, non-allowed constraints are used.)

A derivation starting from `solve(4,[Q1,Q2,Q3,Q4])` proceeds as follows. `make_domains` produces
`Q1 in [1,2,3,4],Q2 in [1,2,3,4],Q3 in [1,2,3,4],Q4 in [1,2,3,4]`. The `safe` predicate adds the `noattack` constraints, which in turn produce `ne` constraints. As they involve variables with no fixed value, no propagation occurs. Then, for labeling, `enum` is called.

The first variable to be labeled is `Q1`. Trying the first value in the initial domain, `1`, propagation reduces the domains of `Q2`, `Q3`, and `Q4`, resulting in `Q2 in [3,4], Q3 in [2,4], Q4 in [2,3]`.

Labeling with `enum` and constraint propagation continues until a solution is found: `[Q1,Q2,Q3,Q4] = [2,4,1,3]`. The other solution is `[Q1,Q2,Q3,Q4] = [3,1,4,2]`.



Scheduling

Scheduling is concerned with planning of the temporal order of *tasks (jobs)* in the presence of limited resources. A task may be a production step or lecture, the *resource* may be a machine, electrical energy, or lecture room. Typically, tasks compete for resources, because they are limited. The problem is to find a schedule with an optimal value for a given objective function (measuring time or use of other resources).

The classical *job shop scheduling problem* assumes that tasks have a fixed duration and cannot be interrupted. Resources are machines that can process at most one task at a time. The objective is to minimize the overall production time that is needed to accomplish all the tasks.

This problem can be expressed as a finite-domain constraint problem.

Each *task* T_i is associated with

- S_i: Starting time
- d_i: Duration (known)
- E_i: End time

This is modeled by the constraint

$$S_i + d_i = E_i$$

where the temporal variables S_i and E_i range between 0 and a maximum value.

There is a partial order between tasks which is expressed by *precedence constraints*. Task T_i must terminate before task T_j starts:

$$S_i + d_i \leq S_j$$

Again this constraint can be easily expressed as allowed constraint of *FD*.

Finally, a *capacity constraint* (in the simplest case) expresses that two tasks T_i and T_j cannot be processed at the same time

$$S_i + d_i \leq S_j \ \vee \ S_j + d_j \leq S_i$$

Since the disjunction should not be implemented by search (this would immediately lead to exponential complexity), the capacity constraint is often encoded by a special finite-domain constraint.

Additional constraints can model set-up times, release times, deadlines, as well as renewable resources, and non-availability or resources at certain times.

13. Non-linear Equations *I*

A simple way to tackle non-linear polynomial equations is to replace non-linear expressions by variables such that as many equations as possible become linear (as in CLP($\Re$) [33]). As with the introduction of slack variables, solving linear equations alone does not yield a complete method.

In constraint programming, complete methods like Gröbner Bases over complex numbers (CAL [48]) and Partial Cylindrical Algebraic Decomposition (RISC-CLP(Real) [29]) have been used. The Gröbner Bases [11] method can be seen as an extension of variable elimination to non-linear polynomials. It has much worse than double exponential time complexity.

Constraint System *I*

Domain
The set $\Re$ of real numbers

Signature

- Function symbols:
  - The real numbers 0 and 1
  - Arithmetic function symbols $+$, $*$, log, sin, exp, ... as well as '..' for intervals.
- Constraint symbols:
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$ as well as *in* for domains.

Constraint theory
An extension of the linear existential fragment of Tarski's axiomatic theory of real closed fields [55], including $X \; in \; n..m \leftrightarrow n \leq X \wedge X \leq m$

Allowed atomic constraints
Arithmetic equations and inequations:

$$C ::= true \mid false \mid X \; in \; n..m \mid X \odot Y \mid f(X_1, \ldots, X_l) = Z$$

where n and m are real numbers, $\odot \in \{=, <, \leq, >, \geq, \neq\}$, X, Y, Z and $X_1, \ldots, X_l$ ($l \geq 0$) are pairwise distinct variables and $f(X_1, \ldots, X_l)$ is flat term, i.e., a function symbol from the signature applied to variables.

T. Frühwirth et al., *Essentials of Constraint Programming*

Another approach is to use a local-consistency method based on interval arithmetic as pioneered in CLP(BNR) [10], and Numerica [57]. In this case, logarithmic and trigonometric functions can also be dealt with. This approach can be seen as a sophisticated extension of finite interval domains to the reals and to arbitrary arithmetic functions. While the time complexity is polynomial, the method can only approximate the values that form a solution, it is incomplete.

The constraint theory is an extension of the one for linear polynomials, $\Re$. The theory becomes undecidable once trigonometric functions are introduced. Their periodicity can express the integer number property. This means that the theory must include a model of Peano arithmetic, which is Presburger's arithmetic extended by multiplication. Gödel has shown that any consistent extension of Peano arithmetic is incomplete and thus there are undecidable formulae. However, these formulae are not expressible with the allowed constraints.

13.1 Local-Propagation Constraint Solver

The rules of the finite-interval domain solver (Chap. 12) can be modified to work for intervals of real numbers. However, unlike integer intervals, non-trivial real-number intervals admit infinitely many values. To avoid non-termination, intervals that are too small are not made smaller by the rules anymore. In other words, we limit the precision by choosing a certain granularity. However, there are only heuristics to determine the size of the smallest useful interval. Rounding errors in arithmetic computations with the floating-point representation are avoided by rounding the bounds of an interval outward.

The inclusion of multiplication, exponentiation, logarithmic and trigonometric functions introduces problems, because these functions are not monotonic anymore. This means that interval propagation is difficult to implement and not very effective.

Multiplication

We illustrate these problems with multiplication. The constraint `mult(X,Y,Z)` means $X*Y=Z$. Multiplication is not monotonic, e.g., $-2*-3$ is larger than $2*2$ but smaller than $3*3$, even though negative numbers are smaller than positive numbers. However, it suffices to consider multiplications between the interval bounds to account for this problem. We use propagation rules for the implementation.

```
mult_z @ mult(X,Y,Z), X in A..B, Y in C..D ==>
         M1 is A*C, M2 is A*D, M3 is B*C, M4 is B*D,
         Z in min(M1,M2,M3,M4)..max(M1,M2,M3,M4).
```

If we compute backwards, starting from the interval of `Z`, we have to avoid division by zero. In fact, if the interval of `X` (or `Y`) contains a zero, any value for the other input variable `Y` (or `X`) is possible. `X=0` cannot constrain `Y` since `0*Y=0` for any value of `Y`. The constraint `has_zero` checks if an interval contains a zero.

```
has_zero(A..B) <=> A=<0, 0=<B.

mult_y @ mult(X,Y,Z), X in A..B, Z in E..F ==>
         not has_zero(A..B) |
         M1 is E/A, M2 is E/B, M3 is F/A, M4 is F/B,
         Y in min(M1,M2,M3,M4)..max(M1,M2,M3,M4).
mult_x @ mult(Y,X,Z), X in A..B, Z in E..F ==> ...
```

Example 13.1.1. Interval propagation for the query `A in 0..0.3, B in 0..0.3, C in 0..0.3, A eq B, B eq C, mult(A,B,C)` results in intervals `A in 0.0..1.0e-07, B in 0.0..1.0e-07, C in 0.0..1.0e-07`, assuming the size of the smallest intervals is 1.0e-07. It cannot be discovered that the variables are determined, they can only take the value 0.

Solving `A in 2..2, B in 0..1, C in 0..1, B eq C, mult(A,B,C)` ($2B{=}B$) will half the intervals for `B` in each step, stopping if the interval is too small, but never reaching 0.

There is one special case to consider when intervals contain zeros. Since $X{=}0$ or $Y{=}0$ implies $Z{=}0$, we conversely have that $Z{\neq}0$ implies $X{\neq}0$ and $Y{\neq}0$. While we cannot remove a zero from inside an interval, we can sometimes remove the complete positive or negative sub-interval for X or Y. For example, `mult(X,Y,Z), X in -2..3, Y in -3..4, Z in 7..12` should simplify to `mult(X,Y,Z), X in 1.75..3, Y in 2.33, Z in 7..12`, since `-2*-3` is only `6`.

The propagation rules accomplish this behavior together with the previous rules:

```
mult_xyz @ mult(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
           has_zero(A..B), has_zero(C..D), not has_zero(E..F) |
           mult0(X,Y,Z).

  mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
        A*C<E | D>0, X in E/D..B.
  mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
        B*D<E | C<0, X in A..E/C.
  mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
        F<A*D | C<0, X in F/C..B.
  mult0(X,Y,Z), X in A..B, Y in C..D, Z in E..F ==>
        F<B*C | D>0, X in A..F/D.
```

Termination. As in the case of finite interval domains, the solver terminates, because intervals get smaller with each rule application. Intervals that are too small will not be considered by the rules.

Confluence. Due to stopping at small intervals and rounding, the solver is not confluent. Depending on the order of rule applications, the resulting intervals may be different due to accumulated roundings or because a rule application was not possible anymore because the interval was too small, while with another rule application the smaller interval was reached immediately.

Complexity. The complexity can be derived in the same way as for finite interval domains (Chap. 12). Let v be the number of different variables. We only have to adapt the definition of the width (size) w of an interval $n..m$. Let the width be w defined as the number of floating point numbers between n and m on the computer of interest. The worst case time complexity is $O(cvw)$.

Determined Variables

Due to stopping at small intervals and outward rounding, variables in the constraint system I will practically never be determined. Hence, the domains almost always admit infinitely many possible values.

Search

Search can be employed to make all intervals small. However, the solver remains incomplete, since the resulting interval will usually not determine its variable. However, in general, smaller intervals are more informative than larger ones.

For search, as with finite domain intervals, one divides intervals in halves until they are too small (*domain splitting*). An additional method is *probing*. It means to try out the interval bounds as values for the associated variable. Since unique values are not possible due to the limits of the floating-point representation, the smallest possible interval that encloses a bound is used. If a bound results in unsatisfiability, it can be removed from the interval. This is called *shaving*.

Improving and Extending Interval Propagation

The propagation behavior of the solver can be improved by changing the representation of the problem (to be non-flat) by removing or adding variables. However, there is no tractable method to determine the best representation of a problem.

Convergence acceleration refers to a method that watches the sequence of smaller and smaller intervals that are produced for a variable by propagation.

Based on this observation, an approximation function is computed to predict the future, shrunken interval.

Interval propagation alone, even with search, is too weak to solve interesting non-linear problems. One cannot always tell if there are no, one, or more solutions in a non-trivial interval. Even the bounds of an interval need not be a solution - due to outward rounding and irrational numbers that cannot be represented. On the positive side, intervals allow to approximate irrational numbers like π or $\sqrt{2}$ to arbitrary precision.

Advanced interval constraint solvers like Numerica combine interval propagation with variable elimination and interval extensions of Newton's approximation method to find the zeros of the functions expressed by the allowed constraints. Another approach is to approximate non-linear arithmetic expressions by linear ones, e.g., using Taylor expansions.

13.2 Applications

Non-linear constraints appear in the modeling, simulation, and analysis of physical, chemical, and molecular-biological processes and systems, and in hybrid systems. We will illustrate a simple case of interval consistency for non-linear polynomials with the Munich rent advisor (Chap. 16). Trigonometric functions are common in geometric reasoning for spatial databases and robot motion planning. Another application area is financial analysis.

Part III

Applications

14. Market Overview

This part of the book presents uses of constraint techniques in three classes of application: timetabling, optimal placement, and reasoning with imprecise and incomplete information. The three applications involved the authors of this book and were implemented in Prolog and CHR at the Ludwig-Maximilians-Universität of Munich (LMU) and at the European Computer Industry Research Center (ECRC) in Munich, Germany.

Before we introduce the applications, we give a brief overview of the commercial market of constraint programming as it presents itself at the time of writing this book. Up-to-date information can be found on the web-pages of this book.

Since the beginning of the 1990's, constraint-based programming has been commercially successful. CLP has proven its merits in a variety of application areas, including decision support systems for scheduling and resource allocation. The world wide revenue generated by constraint technology was estimated to be on the order of 100 million dollars in 1996 [59].

Constraint technology has matured. Constraint solvers and search techniques are offered not only in logic programming languages, but increasingly as software components for standard programming languages like C and Java.

The main commercial suppliers of constraint programming languages are

- SICS with Sicstus Prolog
- IC-PARC with Eclipse Prolog
- ILOG with ILOG Optimization Suite

as well as the smaller suppliers

- COSYTEC with CHIP
- PrologIA with Prolog IV
- Siemens with IF/Prolog

Sicstus and Eclipse Prolog are also the most used academic constraint languages. There are also many academic implementations like, e.g., Mozart (former OZ), which is object-oriented, and Screamer, which is Lisp-based. More details and links can be found on the web-pages of this book.

Constraint-based software can be used profitably for reasoning with incomplete but also complete information and for solving combinatorial problems in decision support systems (expert systems, intelligent agents).

T. Frühwirth et al., *Essentials of Constraint Programming*

The use of constraint programming supports the complete software development process, because executable specification and rapid prototyping are possible. In particular, the advantages are

- the declarative modeling of problems, which leads to robust, flexible, and maintainable software faster,
- the possibility of representing in a mathematical model incomplete and imprecise (inexact, uncertain, and fuzzy), as well as complete information,
- reasoning with such information and the automatic propagation of the effects as soon as new informations become known,
- the synergy between constraint solving and search and optimization techniques for solving hard combinatorial problems.

The flexibility of the constraint-based approach constitutes the main advantage over specialized tools which are likely to be harder to maintain, because they possibly cannot be adapted as easily to a change in the problem.

The main industrial application areas of constraint technology are on one hand design, synthesis, simulation, verification, and error diagnosis of electronic, electrical, mechanical and software components, as well as entire industrial processes and work flows. On the other hand, there are scheduling, planning, rostering, timetabling of personnel, finances, traffic, networks and other resources, as well as transport and placement optimization, configuration, and layout generation.

In research, constraints become a more and more mainstream formalism. Not only in areas of artificial intelligence, such as machine vision, natural language understanding, temporal and spatial reasoning, qualitative reasoning, computational logic, theorem proving, and intelligent agents, but also in diverse research fields such as computer graphics and user interfaces, program analysis, database systems, robotics, electrical engineering, circuit design, chemistry, linguistics, and molecular biology, to name a few.

Everywhere where resources have to be managed and their use to be optimized, constraint technology is a potential solution, therefore its use has been wide spread in industry from the very beginning. The following list mentions some early users of constraint-based tools:

- Transport facilities (British Airways, Lufthansa, Delta Airlines, Hong Kong International Terminals, SNCF)
- Automobile and aircraft industries (Ford, Renault, Peugeot, Citröen, Daimler, Michelin, Lockheed Martin, Airbus)
- Electronics (IBM, Alcatel, Philips, Hewlett-Packard, Whirlpool, Bosch, Motorola, Thomson, Siemens)
- Telecommunications (France Telecom, AT&T, Nokia)
- Finances (Citicorp, National Westminster Bank)
- Energy supplies (Shell)
- Food industries (Uncle Ben's)
- Commerce (El Corte Ingles)

- Chemistry (Rhone Poulenc)
- Military (Dassault)
- Public sector (NASA)

A random selection of early, pioneering concrete commercial applications from the first half of the 1990's are briefly mentioned here.

- The system Daysy performs short-term personnel planning for Lufthansa after disturbances in air traffic (delays, etc.), such that changes in the schedule and costs are minimized.
- Nokia uses IFProlog for the automatic configuration of software for mobile phones.
- Siemens uses the Circuit Verification Environment (CVE) developed in IF-Prolog with Boolean constraints for design and for verification of hardware (VLSI chips).
- As early as 1991 ICL developed with DecisionPower (a CHIP derivate) a placement application for Hong Kong International Terminals (one of the biggest container harbors in the world) for optimizing the placement of containers in warehouses between arrival and clearing.
- Renault has been using a version of CHIP in short-term production planning in its car manufacturing plants since 1995.
- Dassaults Application "Made" in CHIP decides where and how complex aircraft parts shall be cut out of a metal, so that as few waste and time losses as possible occur. Dassault operates several constraint-based applications.
- ILOG developed the tool "Sagitaire" for the French railways (SNCF) which plans timetables and tracks for over 1700 trains for the train station Paris du Nord.
- The LOCARIM system was developed by COSYTEC for France Telecom to propose a cabling of the telecommunication network of a building given its architectural plan.

A good survey paper on practical applications of CLP is [59].

15. Optimal Sender Placement for Wireless Communication

Mobile communications comes to company sites. Be it as digital cordless telecommunication (DECT) or as wireless local area networks (W-LAN). No cabling is required, but access points, i.e., small, local radio transmitters (senders) have to be installed. They should cover the installation site. Planning their locations is difficult, since the specifics of radio wave propagation have to be taken into account.

The industrial prototype POPULAR (Planning of Picocellular Radio) [25] was one of the first systems for *coverage planning*. It computes the minimal number of senders and their location, given a blue print of the building and information about the materials used for walls and ceilings. It does so by simulating the propagation of radio waves using ray tracing and subsequent constraint-based optimization of the number of senders that are needed to cover the whole building. Award-winning POPULAR was developed by the European Computer Industry Research Center (ECRC), Siemens Research and Development (ZFE), the Siemens Personal Networks Department (PN), and the Institute of Communication Networks at the Aachen University of Technology.

15.1 Approach

To solve the problem, one starts with computing the characteristics of the building using a grid of test points (Fig. 15.1). Each test point represents a possible receiver position. For each test point, the space where a sender could be put to cover the test point, the *radio cell*, is calculated using ray tracing. Ray tracing simulates the propagation of radio waves in the building, where they are absorbed and reflected by walls and floors. Alternatively, experimental measurements on site are possible.

The radio cell will usually be a rather odd-shaped object, since the coverage is not a smooth or differentiable function. A small change in the receivers location such as a move around the corner may cause a substantial change in the received power. However, if the test grid is sufficiently small (several test points per square meter), we can expect that if two neighboring test points are covered, the space in between - hence the whole building - can also be covered.

T. Frühwirth et al., *Essentials of Constraint Programming*

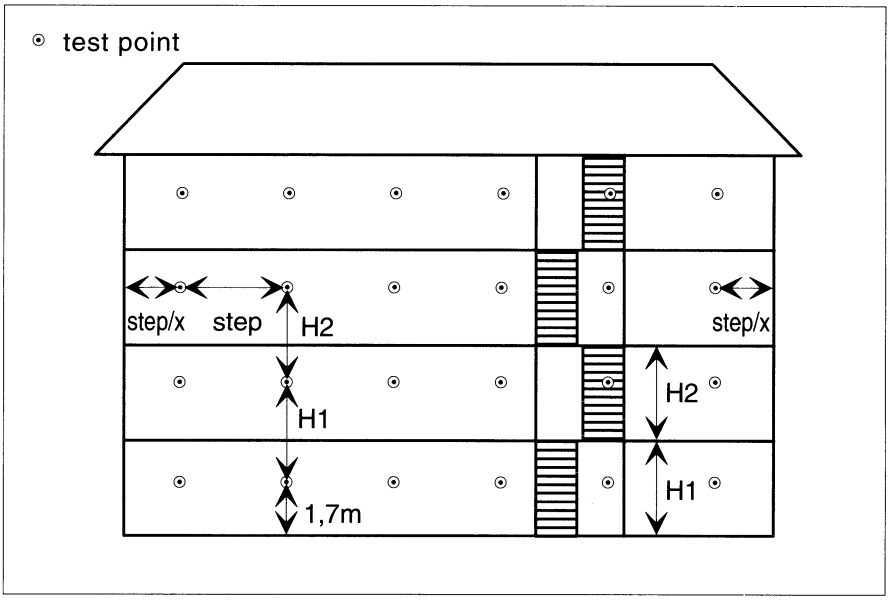


Fig. 15.1. Grid of test points in a building

Fig. 15.3 shows simple cases of radio wave propagation with one or two senders in an office building.

For each radio cell, a constraint is set up that there must be a location of a sender (geometrically speaking, a point) somewhere in that space. Then, one tries to find locations that are in as many radio cells at the same time as possible. Thus, the possible locations are constrained to be in the intersections of the radio cells covered. A sender at one of these locations will cover several test points at once. In this way, a first solution is computed. To minimize the number of senders, a branch-and-bound method is used. It consists in repeatedly searching for a solution with a smaller number of senders until the minimal number is found.

15.2 Solver

For simplicity of presentation, we restrict ourselves to two dimensions and we approximate the radio cell by a single rectangle. The 2-D coordinates are of the form `X#Y`. Rectangles are orthogonal to the coordinate system and are represented by a pair of their left upper and right lower corner coordinates, i.e., `Rectangle = r(A#B,C#D)`. A sender is characterized by its location. For each radio cell, a constraint `inside(Sender, Rectangle)` is imposed. It means that `Sender` is a point that must be inside `Rectangle`.

```
non_empty @ inside(S,r(A#B,C#D)) ==> A<C, B<D.
```

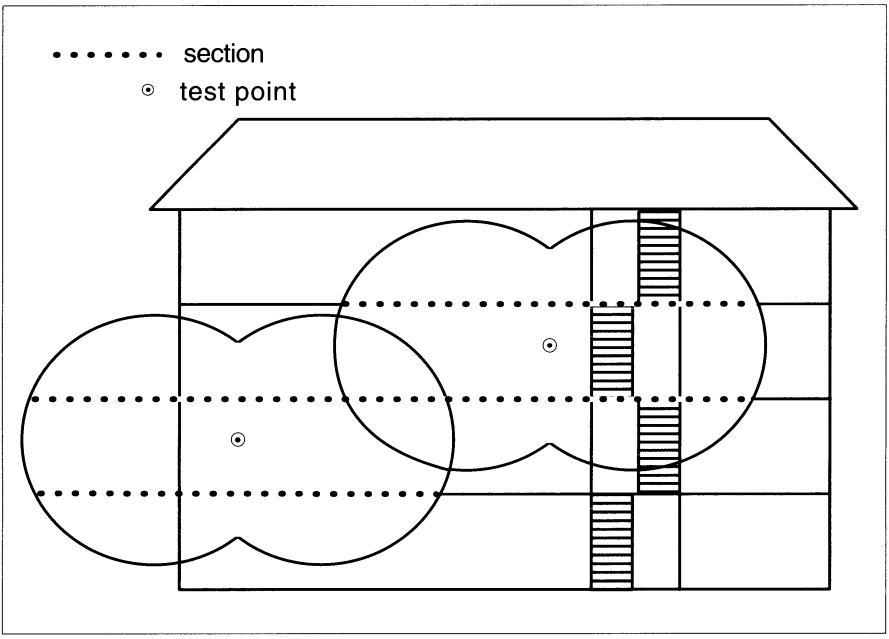


Fig. 15.2. Typical radio cells in a building

```
intersect @ inside(S,r(A1#B1,C1#D1)),inside(S,r(A2#B2,C2#D2))
        <=>
        A is max(A1,A2), B is max(B1,B2),
        C is min(C1,C2), D is min(D1,D2),
        inside(S,r(A#B,C#D)).
```

These rules can be seen as an extension of interval constraints to two dimensions. The first rule (named `non_empty`) says that the constraint `inside(S,r(A#B,C#D))` is satisfiable only if the rectangle has a non-empty area. The `intersect` rule handles the case when a sender's location `S` is constrained by two `inside` constraints to be in two rectangles at once. Then one can replace these two constraints by one new `inside` constraint whose rectangle is the intersection of the two initial rectangles.

To compute a solution, one tries to equate as many senders as possible. This can be accomplished by the following labeling procedure `label`:

```
label, inside(S1,R1), inside(S2,R2) ==> (S1=S2 ; true).
```

For every pair of senders `S1` and `S2`, the propagation rule tries to equate them using `S1=S2` in a disjunction with `true`. Equating senders causes the `intersect` rule to fire with the `inside` constraints associated with the senders. As a result of repeated labeling, a sender's location will be constrained more and more. Thus, the `intersect` rule will be applied again

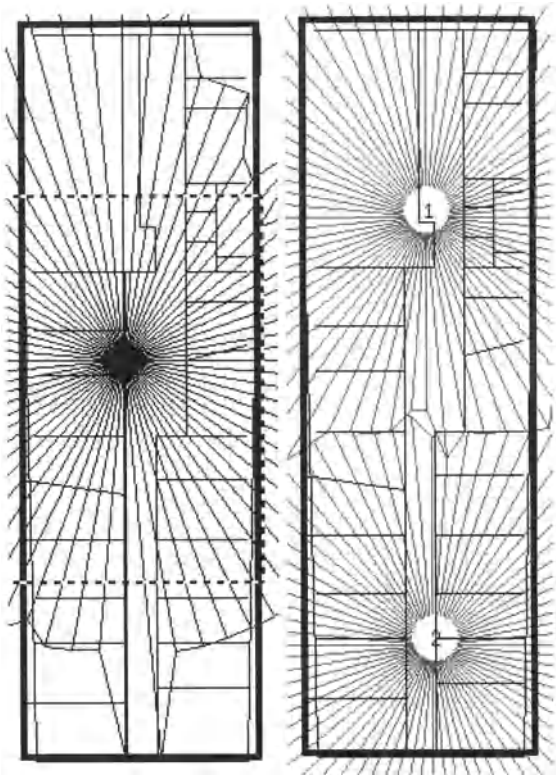


Fig. 15.3. Result of placing one or two senders

and again until the rectangle becomes very small. If the rectangle becomes empty, the `non-empty` rule causes failure and so initiates backtracking. This will undo one of the equations `S1=S2` by choosing the other disjunct `true`.

It is straightforward to extend this solver so that it works with unions of rectangles. Unions can describe the radio cell to any desired degree of precision. The union corresponds to a disjunctive constraint
`inside(S,R1)` $\vee \ldots \vee$ `inside(S,Rn)`
which is more compactly implemented as `inside(S,[R1,...,Rn])`.

```
intersect @ inside(S, L1), inside(S, L2) <=>
        intersect(L1, L2, L3),
        non_empty(L3),
        inside(S, L3).

intersect(L1, L2, L3) <=>
        setof(R, intersect1(L1,L2,R), L3).

intersect1(L1, L2, r(A#B,C#D)) <=>
```

```
member(r(A1#B1,C1#D1), L1),
member(r(A2#B2,C2#D2), L2),
A is max(A1,A2), B is max(B1,B2),
C is min(C1,C2), D is min(D1,D2),
A<C, B<D.          % non-empty
```

The built-in `setof` collects all non-empty results `R` of intersecting elements of the lists `L1` and `L2` in the list `L3`.

The above solver can be adapted quickly to work with other geometric objects than unions of rectangles by changing the definition of `intersect1/3`. The lifting to three dimensions just amounts to adding a third coordinate and code analogous to the one for the other dimensions. In practice, there are additional requirements: senders can only be installed on walls or ceilings, or along LAN cables. This can be expressed by constraining senders to be inside the allowed area, using additional `inside` constraints.

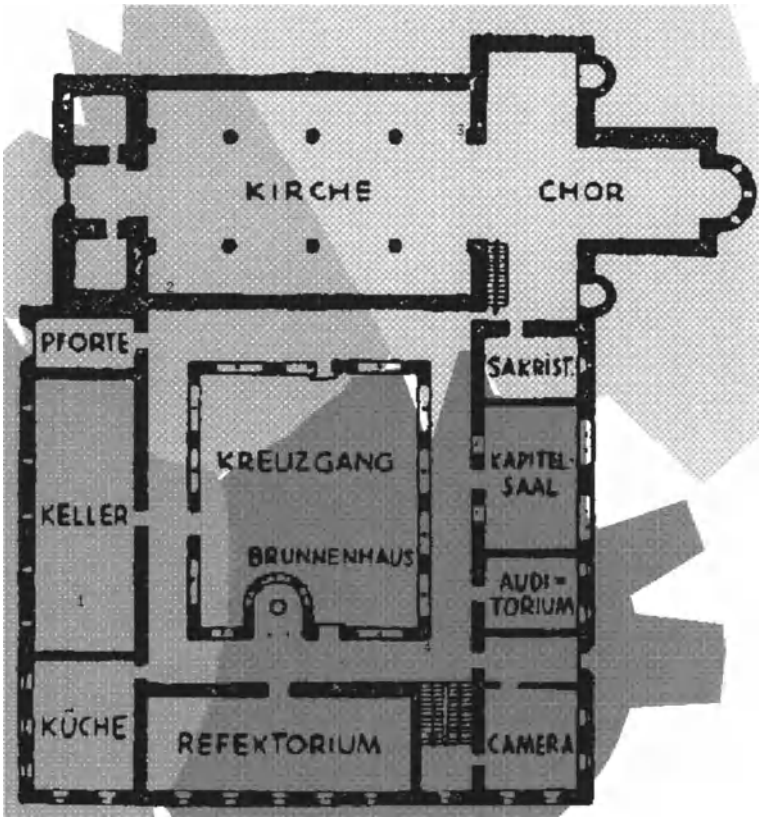


Fig. 15.4. Covering a medieval monastery

15.3 Evaluation

For a typical office building (that requires a handful of senders), an optimal placement is found by POPULAR within a few minutes. The overall quality of the placements produced is comparable to that of a human expert. Fig. 15.2 shows the result of covering a medieval monastery.

In 1994, the only other tool available with similar functionality was WISE [21], which is written in about 7500 lines of C++. For optimization WISE uses an adaptation of the Nelder-Mead direct search method that optimizes the percentage of the building covered. The CLP code for POPULAR is about 4000 lines of Prolog and CHR with more than half of it for graphics and user interface. POPULAR was written in roughly one man year. The big advantage of the CLP approach is flexibility, e.g., when changing the labeling heuristic or extending the solver.

16. The Munich Rent Advisor

The Munich Rent Advisor (MRA) [24], developed by the European Computer Industry Research Center (ECRC) and the Ludwig-Maximilians-Universität of Munich (LMU), is the electronic version of the "Mietspiegel"(MS) for Munich. MSs are published regularly by German cities. They are basically a written description of an expert system that allows to estimate the maximum fair rent for a flat. These estimates are legally binding.

The calculations are based on size, age, and location of the flat and a series of detailed questions about the flat and the house it is in. Some of these questions are difficult to answer. However, in order to be able to calculate the rent estimate by hand, all questions must be answered. Usually, the calculation is performed by hand in about half an hour by an expert from one of the renter's associations. The MRA brought the advising time down to a few minutes that the user needs to fill in the web form. Using constraints, the user of the MRA need not answer all questions. The user may not want to dislose information, does not care about the question, or know the (exact) answer. Tens of thousands have used the award-winning MRA service on the World-Wide Web (WWW).

16.1 Approach

The MS is derived from a statistical model compiled from sample data using statistical methods such as regression analysis. Due to the underlying statistical approach, there is the problem of inherent imprecision which is ignored in the printed version of the MS. Using constraints the MRA can account for the statistical imprecision.

The scheme for calculating the rent estimate is roughly as follows:

$$\begin{aligned} \mathit{Estimated\ Rent} = {} & \mathit{Size} * \mathit{Basic\ Rent\ per\ Square\ Meter} \\ & * (\mathit{Sum\ of\ Deviations\ as\ Percentage} + 100) * 0,01 \\ & * (\mathit{Imprecision\ Deviation\ Percentage} + 100) * 0,01 \\ & + \mathit{Fixed\ Costs} \end{aligned}$$

The calculation starts with the average rent per square meter taken from a table with about 200 entries. The deviations from the average rent are com-

T. Frühwirth et al., *Essentials of Constraint Programming*

puted from the answers regarding the size, location, features of the flat, as well as age and state of the house. There are six yes-no questions about features of the house concerning, e.g., number of floors, optical impression, lift, etc., and 13 yes-no questions about features of the flat concerning, e.g., central heating, separate shower, dish-washer, etc. The answers to these questions combined with the age of the house yield the deviations from the average rent. The overall deviation may be up to ±60%.

The MRA is available on the internet using a single form on the WWW (Fig. 16.1). To process the answers from the questionnaire and return its

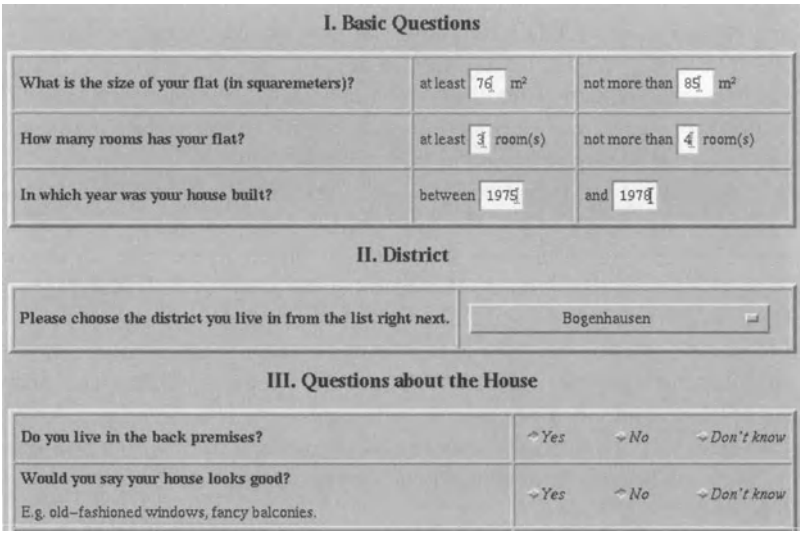


Fig. 16.1. Part of the form

result, a simple stable special-purpose web-server was written in Prolog. The MRA server only deals with the input posted from the form. A web page is assembled from the result of the calculation and sent back to the user (Fig. 16.2).

16.2 Solver

From a CLP point of view, the MRA application is rather atypical: it is not concerned with the NP-complete constraint-pruned search for a solution, but executing an existing calculation in the presence of partial information. The computation proceeds deterministically from constrained input variables (the user data) to constrained output variables (the rent estimate), since the original MS has already solved the problem. There is no need for full-fledged

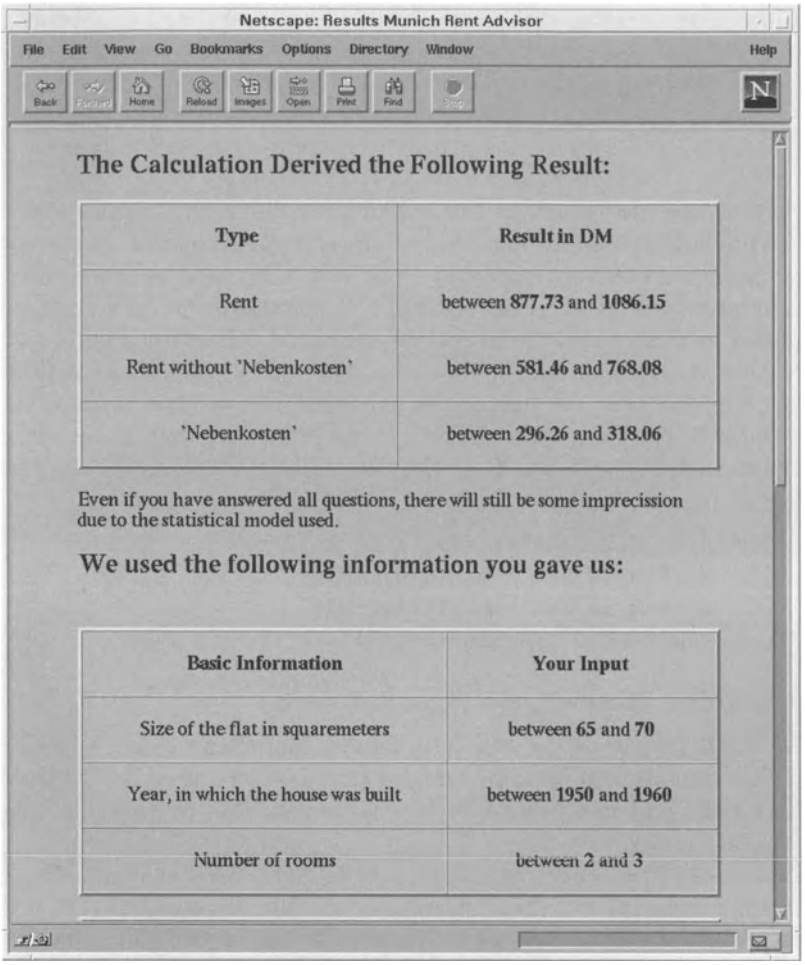


Fig. 16.2. Partial result of a sample query

constraint solving and labeling, only for constraint propagation in the forward direction: the answer one expects is the smallest interval covering all possible rents, not an enumeration of all possible rents by backtracking.

The approach was first to implement the tables, rules, and formulas of the MS in CLP as if the provided data were precise and completely known. Then constraints were added to capture the imprecision due to the statistical approach and incompleteness due to partial answers of the user. Finally, the formulas of the rent calculation were considered as constraints that refine the rent estimate by propagation from the input variables which are constrained by the partial answers.

In the MRA, dealing with imprecise numerical information involves non-linear arithmetic computations with intervals. An existing interval domain solver was modified, so that it can deal with equations of the form

$$c_0 + c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n = y \text{ and } c * x_1 * x_2 * \ldots * x_n = y,$$

where c_i and c are numbers and x_i and y are different variables and $n \geq 0$.

The implementation for linear equations is straightforward. In the solver, the equation $c_0 + c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n = y$ is represented by the constraint `sum(C0:C0+C1*X1+C2*X2+...+Cn*Xn+0=Y)`. The constant c_0 is replaced by the interval `C0:C0` and the summand `0` is introduced to end the summation. A constraint of the form `sum(Min:Max+Rest=Y)` means that the interval `Min:Max` plus the sum of the polynomial `Rest` gives an interval for the variable `Y`. The rules below define forward propagation: from the intervals associated with the variables `Xi` in the polynomial they compute an interval for `Y`:

```
sum(Min..Max+C*X+Rest=Y), X in A..B ==>
    NewMin is Min + min(C*A,C*B),
    NewMax is Max + max(C*A,C*B),
    sum(NewMin..NewMax+Rest=Y).

sum(Min..Max+0=Y) <=> Y in Min..Max.
```

The first rule reads: if there is a constraint `sum(Min..Max+C*X+Rest=Y)` where the variable `X` is between `A` and `B` (`X in A..B`), then `C*X` is between `min(C*A,C*B)` and `max(C*A,C*B)`. We can replace `C*X` by this interval and add it to `Min..Max`.

This results in the new constraint `sum(NewMin..NewMax+Rest=Y)`. After we have eliminated all variables this way, we are left with `sum(Min..Max+0=Y)`, which means `Y in Min..Max`, as is expressed by the second rule. Since we do not need backward propagation in the application, these two rules suffice.

The implementation for non-linear equations is analogous. The equation $c * x_1 * x_2 * \ldots * x_n = y$ is represented by `mult(C..C*X1*X2*...*Xn*1=Y)`.

```
mult(Min..Max*X*Rest=Y), X in A..B ==>
    NewMin is min(Min*A,Max*B,Max*A,Min*B),
    NewMax is max(Min*A,Max*B,Max*A,Min*B),
    mult(NewMin..NewMax*Rest=Y).

mult(Min..Max*1=Y) <=> Y in Min..Max.
```

16.3 Evaluation

To implement the MRA, it took a developer about 4 weeks to write the WWW user interface, only 2 weeks to write the calculation part and 1 week

to debug it. The high-level CLP approach means that the program can be easily maintained and modified. This is crucial, since every city and every new version comes with different tables and rules for the MS. One could have used standard interval domains to express the required constraints. However, it would have been quite difficult to tailor the amount and direction of constraint propagation to the needs of the application at hand.

17. University Course Timetabling

University course timetabling problems are combinatorial problems, which consist in scheduling a set of courses within a given number of rooms and time periods. Solving a real-world timetabling problem manually often requires a significant amount of time, sometimes several days or even weeks. Therefore, a lot of research has been invested in order to provide automated support for human timetablers. Contributions come from the fields of operations research (e.g., graph coloring, network flow techniques) and artificial intelligence (e.g., simulated annealing, tabu search, genetic algorithms, constraint satisfaction) [50].

In practice, there is no timetable that fulfills all the constraints. Thus, we have to distinguish two kinds of constraints. *Hard constraints* are conditions that must be always satisfied, *soft constraints* may be violated, but should be satisfied as far as possible.

Most existing constraint-based timetabling systems either do not support soft constraints [9] or use a branch-and-bound search instead of chronological backtracking [26]. Branch and bound starts out from a solution and requires the next solution to be better. Quality is measured by a suitable cost function that depends on the set of violated soft constraints. With this approach, however, soft constraints play no role in selecting variables and values, i.e., they do not guide search.

Another approach is to adopt techniques developed to propagate hard constraints; soft constraint propagation is intended to associate values with an estimate of how selecting a value will influence solution quality, i.e., which value is known (or expected to) violate soft constraints, or the other way round, which value is known (or expected to) satisfy soft constraints. By considering estimates in value selection, one hopes that the first solution will satisfy a lot of soft constraints. For example, [42] presents a commercial C++ library providing black-box constraint solvers and search methods for the nurse scheduling problem.

Inspired by this approach, a constraint-based timetabling system for the Ludwig-Maximilians-Universität of Munich, called IfIPlan[1], has been developed using a CHR solver which performs hard- and soft-constraint propagation [3].

[1] IfIPlan is an acronym for the German "Planer für das Institut für Informatik".

T. Frühwirth et al., *Essentials of Constraint Programming*

17.1 Approach

To model the university timetabling problem, only one variable for each course holding the period is needed, i.e., the starting time point, it has been scheduled for. Each domain of the variables consists of the whole week, the periods being numbered from 0 to 167, e.g., 9 denotes 9 a.m. on Monday, and so on. Requirements, wishes, and recommendations can be expressed with a small set of specialized constraints.

- *No-clash constraints* demand that a course must not clash with another one.
- *Preassignment constraints* and *availability constraints* are used to express teachers' preferences and that a course must (not) take place at a certain time.
- *Distribution constraints* make sure that there is at least one day (hour) between one course and another or that two courses are scheduled for different days.
- *Compactness constraints* make sure that one course will be scheduled directly after another.

With respect to soft constraints, we chose to distinguish three grades of preferences: weakly preferred, preferred and strongly preferred, which are translated to the integer weights 1, 3, and 9.

Since soft constraints may be violated, the values to be constrained must not be removed from the domain of the variable. Moreover, when we have to choose a value for the variable during search, we must be able to decide whether a certain value is a good choice or not. Therefore, each value must be associated with an assessment. We chose to represent a domain as a list of value-assessment pairs. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]`, then `X` may take one of values `3`, `4`, and `5`, whereas `4` is encouraged with assessment `1` and `5` is discouraged with assessment `-1`.

17.2 Solver

The solver is based on three types of constraints.

- `domain(X, D)` means that `X` must be assigned a value occurring in the list of value-assessment pairs `D`.
- `in(X, L, W)`: Its meaning depends on the weight `W`. If `W = inf`, i.e., if the constraint is hard, it means that `X` must be assigned a value occurring in the list `L`. If `W` is a number, i.e., if the constraint is soft, it means that the assessment for the values occurring in `L` should be increased by `W`.
- `notin(X, L, W)`, if hard, means that `X` must not be assigned any of the values occurring in the list `L`. If it is soft, it means that the assessment for the values occurring in `L` should be decreased by `W`.

Propagating a soft constraint is intended to modify the assessment of the values to be constrained. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]` and assume the existence of the constraint `in(X, [3], 2)` stating that `3` should be assigned to `X` with preference `2`. Then we have to increase the assessment for value `3` in the domain of `X` by adding `2` to the current assessment of `3`, obtaining the new domain `[(3, 2), (4, 1), (5, -1)]` for `X`. However, applying a hard constraint will still mean to remove values from the variable's domain. Consequently, an `in` constraint is processed by either pruning the domain or increasing the assessment for the given values.

```
fd_in_hard @ domain(X, D), in(X, L, W) <=> W = inf |
    domain_intersection(D, L, D1),
    domain(X, D1).

fd_in_soft @ domain(X, D), in(X, L, W) <=> W \= inf |
    increase_assessment(W, L, D, D1),
    domain(X, D1).
```

In case a hard `in` constraint has arrived, rule `fd_in_hard` looks for the corresponding `domain` constraint, which contains the current domain `D`, and replaces both by a new `domain` constraint, which contains the new domain `D1`. The domain `D1` results from intersecting `D` with the list of values `L`. Rule `fd_in_soft` works quite similar, except for `D1` results from `D` by increasing the assessments for the values occurring in `L`. Note that the guards exclude each other. Therefore, whichever constraint arrives, only one of the rules will be applicable. The rules for `notin` are similar.

```
fd_notin_hard @ domain(X, D), notin(X, L, W) <=> W = inf |
    domain_subtraction(D, L, D1),
    domain(X, D1).

fd_notin_soft @ domain(X, D), notin(X, L, W) <=> W \= inf |
    decrease_assessment(W, L, D, D1),
    domain(X, D1).
```

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a domain of a variable has been reduced to the empty list, the variable cannot be assigned a value without violating hard constraints. This case is dealt with by the following simplification rule.

```
fd_empty @ domain(_, []) <=> false.
```

With only one value left in a domain of a variable, we can assign the remaining value to the variable immediately.

```
fd_singleton @ domain(X, [(A, _)]) ==> X = A.
```

We use a propagation rule instead of a simplification rule because the `domain` constraint must not be removed. Without it the processing of `in` and `notin` constraints imposed on the domain of a variable would not be guaranteed and thus an inconsistency might be overlooked.

Treatment of Global Constraints. Up to now we only dealt with the simple constraints `domain, in` and `notin`. Now we exemplify how to express global (n-ary) application-level constraints in terms of `in` and `notin` constraints.

`no_clash(W, Xs)` means that, depending on the weight `W`, the variables from `Xs` must or should be assigned distinct values. It is translated to `notin` constraints. This translation is data-driven: whenever one of the variables from `Xs` is assigned a value, this value is discouraged or forbidden for the other variables by the following rule.

```
fd_no_clash @ no_clash(W, Xs) <=>
    Xs \= [_],
    select_ground_var(Xs, X, XsRest)
    |
    post_notin_constraints(W, X, XsRest),
    no_clash(W, XsRest).
```

The guard first makes sure that `Xs` contains at least two elements. Then it selects a ground variable `X` from `Xs`, remembering the other variables in `XsRest`. With no ground variable in `Xs`, the Prolog predicate `select_ground_var` fails. If the guard holds, `no_clash(W, Xs)` is replaced by

- `notin` constraints produced by the predicate `post_notin_constraints`, one for each member of `XsRest`, discouraging or forbidding the value `X`, and
- a `no_clash` constraint stating that the variables in `XsRest` should or must be assigned distinct values.

Note that the predicate `post_notin_constraints` fails in case `XsRest` contains the value `X`.

A singleton list of variables means that there is nothing more to do. This case is handled by the following rule.

```
fd_no_clash_singleton @ no_clash(_, [_]) <=> true.
```

The translation of the other application-level constraints either follows this scheme or is a one-to-one translation.

Interaction of the `no_clash` Rules and the Rules for Primitive Constraints. In the following, we present two derivations to show how the CHR rules interact with each other. In the first derivation, we deal only with hard constraints.

```
domain(X, [(1,0), (2,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(inf, [X,Y]), in(X, [1], inf)    ↦fd_in_hard
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(inf, [X,Y])    ↦fd_singleton
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(inf, [X,Y]), X=1    ↦fd_no_clash
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]), X=1
notin(Y, [1], inf), no_clash(inf, [Y])    ↦fd_no_clash_singleton
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]), X=1
notin(Y, [1], inf)    ↦fd_notin_hard
domain(X, [(1,0)]), X=1, domain(Y, [(2,0)])    ↦fd_singleton
domain(X, [(1,0)]), X=1, domain(Y, [(2,0)]), Y=2
```

In the second derivation, we want to show how the rules treat soft `no_clash` constraints.

```
domain(X, [(1,0), (2,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(1, [X,Y]), in(X, [1], inf)    ↦fd_in_hard
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(1, [X,Y])    ↦fd_singleton
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]),
no_clash(1, [X,Y]), X=1    ↦fd_no_clash
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]), X=1,
notin(Y, [1], 1), no_clash(1, [Y])    ↦fd_no_clash_singleton
domain(X, [(1,0)]), domain(Y, [(1,0), (2,0)]), X=1
notin(Y, [1], 1)    ↦fd_notin_soft
domain(X, [(1,0)]), domain(Y, [(1,-1), (2,0)]), X=1
```

17.3 Generation of Timetables

The generation of a timetable proceeds as follows. Each course is associated with a `domain` constraint allowing for the whole week, the periods being numbered from 0 to 167. It is important to note that, for each course, the initial assessment for all periods is 0, indicating that no period is given preference initially. Then preassignment constraints and availability constraints will be translated into `in` and `notin` constraints. Adding `in` and `notin` constraints may narrow the domains of the courses using the rules presented above. Propagation continues until a fixpoint is reached, that is to say, when further rewriting does not change the store. Usually, the solver is not powerful enough to determine that the constraints are satisfiable. In order to guarantee that a valid solution is found a search procedure is called.

17.4 Evaluation

IfIPlan has been in use at the Computer Science Department of the University of Munich since 1996. It brought down the time necessary for creating a timetable from a few days by hand to a few minutes on a computer. The core of the solver takes no more than 20 lines of code. Due to the declarativity of the approach, IfIPlan can be easily adapted to solve timetabling problems for other universities.

Part IV

Appendix

A. Foundations from Logic

Even though we expect the reader to be familiar with first-order logic [40], we give some definitions in order to introduce our terminology and notation.

A.1 First-Order Logic: Syntax and Semantics

This section introduces *first-order languages* and defines how the elements of such a language are interpreted, i.e., how a meaning is assigned to them.

Definition A.1.1 (Alphabet). *The formulae of a first-order language are constructed from an* alphabet *consisting of*

- *logic symbols:* $\bot$, $\top$, $\neg$, $\wedge$, $\vee$ $\rightarrow$, $\forall$, *and* $\exists$
- *syntactic symbols: "(", ")", and ","*
- *a countably infinite number of variables:* $X, Y, Z, \ldots$
- *function symbols:* $f, g, h, \ldots$
- *predicate symbols:* $p, q, r, \ldots$

These five categories of symbols are pairwise disjoint.

The sets of function and predicate symbols depend on the application. They are given as the signature of the language.

Definition A.1.2 (Signature). *A* signature *of a first-order language consists of*

- *a set of* function symbols, *each with* arity $n \in \mathbb{N}$[1], *and*
- *a set of* predicate symbols, *each with* arity $n \in \mathbb{N}$.

A symbol with arity n is called an n-ary (*or* nullary, unary, binary, ternary *in the case of* $n = 0, 1, 2, 3$) *symbol. Function and predicate symbols with arity 0 are called* constants *and* proposition symbols, *respectively.*

Definition A.1.3 (Term, Formula). *The sets of terms and formulae that can be constructed with a given signature are defined inductively as follows.*

- *A* term *is*

[1] We consider zero a natural number, i.e., $\mathbb{N} = \{0, 1, 2, \ldots\}$.

  - *a variable or*
  - *a* function term $f(t_0, \ldots, t_{n-1})$*, where* f *is an* n*-ary function symbol and the* arguments $t_0, \ldots, t_{n-1}$ *are terms.*
- *A* formula *is*
  - *an* atomic formula *(*atom*)* $p(t_0, \ldots, t_{n-1})$*, where* p *is an* n*-ary predicate symbol and the* arguments $t_0, \ldots, t_{n-1}$ *are terms, or*
  - *the* falsity $\bot$ *or*
  - *the* truth $\top$ *or*
  - *the* negation $\neg F$ *of a formula* F *or*
  - *the* conjunction $(F \wedge F')$*, the* disjunction $(F \vee F')$ *or the* implication $(F \to F')$ *between two formulae* F *and* F'*, or*
  - *a* universally quantified formula $\forall x F$ *or an* existentially quantified formula $\exists x F$*, where* x *is a variable and* F *is a formula.*

Actually, we will use some syntactic sugaring.

- A formula of the form $\forall x_0 \ldots \forall x_{n-1} F$ or $\exists x_0 \ldots \exists x_{n-1} F$ is abbreviated as $\forall x_0 \ldots x_{n-1} F$ or $\exists x_0 \ldots x_{n-1} F$, respectively.
- $\bar{x}$ is an abbreviation for $x_1, \ldots, x_n$ for some integer $n \geq 0$.
- The parentheses around the empty argument list after a constant or proposition symbol are omitted.
- We may write $(F' \leftarrow F)$ for $(F \to F')$.
- Parentheses around formulae may also be omitted according to the following precedence and associativity rules. Negation and quantifiers precede (i.e., bind more closely than) conjunction, conjunction precedes disjunction, and disjunction precedes the two directions of the implication. The implication to the right $(\to)$ associates to the right. The implication to the left $(\leftarrow)$ associates to the left.
- The same name may be used for function symbols and predicate symbols with different arities. From the context it will be clear that these "homonyms" are actually different symbols.

Now we will interpret the elements of a first-order language, i.e., we assign a "meaning" to them.

Definition A.1.4 (Interpretation). *Let* Σ *be a signature of a first-order language. An* interpretation I *of* Σ *consists of*

- *a non-empty set* U*, called the* universe,
- *a function* $I(f) : U^n \to U$ *for every* n*-ary function symbol* f *of* Σ*, and*
- *a relation* $I(p) \in U^n$ *for every* n*-ary predicate symbol* p *of* Σ*.*

Definition A.1.5 (Variable Valuation). *Let* V *be a set of variables and* I *an interpretation. Then a* variable valuation *for* V *w.r.t.* I *is a function from* V *into the universe of* I*.*

We will need an operation on variable valuations that modifies or adds the value for a single variable.

Definition A.1.6. *Given a universe U and a variable valuation $\eta : V \to U$. The function $\eta[x \mapsto u] : V \to U$ is defined as*

$$\eta[x \mapsto u](y) = \begin{cases} \eta(y) & \text{if } y \neq x, \\ u & \text{if } y = x. \end{cases}$$

We say that a quantified formula $\forall x F$ or $\exists x F$ *binds* the variable x within the *scope* F. When a quantified formula is interpreted, the bound variable will have a value assigned locally within the scope, independently of the value assigned to this variable outside the quantified formula. An occurrence of a variable x is *free* if it is not in the scope of a binding for x. In order to interpret a formula, we only need a variable valuation for its *free variables*, i.e., for the variables with a free occurrence. A variable is a *free* variable of a formula if it has a free occurrence in that formula.

Definition A.1.7 (Free Variables). *The set $vars(t)$ of variables of a term t is defined as*

- $vars(v) := \{v\}$ *for a variable* v
- $vars(f(t_0, \ldots, t_{n-1})) := vars(t_0) \cup \ldots \cup vars(t_{n-1})$

The set $free(F)$ of free variables of a formula F is defined as

- $free(p(t_0, \ldots, t_{n-1})) := vars(t_0) \cup \ldots \cup vars(t_{n-1})$
- $free(\top) := free(\bot) := \emptyset$
- $free(\neg F) := free(F)$ *for a formula* F
- $free(F \wedge F') := free(F \vee F') := free(F \to F') := free(F) \cup free(F')$ *for formulae* F *and* F'
- $free(\forall x F) := free(\exists x F) := free(F) \setminus \{x\}$ *for a variable* x *and a formula* F

Given an interpretation for a signature Σ and a valuation for a set V of variables, we can interpret the terms in the set $\mathcal{T}(\Sigma, V)$ and the formulae in the set $\mathcal{F}(\Sigma, V)$, which are defined as follows.

$$\mathcal{T}(\Sigma, V) := \{t \mid t \text{ is a term with function symbols from } \Sigma \text{ and } \operatorname{vars}(t) \subseteq V\}$$
$$\mathcal{F}(\Sigma, V) := \{F \mid F \text{ is a formula with function and predicate symbols from } \Sigma \text{ and } \operatorname{free}(F) \subseteq V\}$$

A *closed formula* is a formula without free variables, i.e., an element of $\mathcal{F}(\Sigma, \emptyset)$. Closed formulae are also called *sentences*. A *theory* is a set of sentences. A term, a formula, or a theory is *ground* if it does not contain any variables. In particular, a *ground term* is an element of $\mathcal{T}(\Sigma, \emptyset)$ and a *ground formula* is a quantifier-free sentence.

To simplify the notation of formulae, we allow the omission of variables for quantifiers in certain cases.

Definition A.1.8 (Universal and Existential Closure). *The* universal closure *(respectively,* existential closure*) of a formula F, denoted $\forall F$ (respectively, $\exists F$), is the sentence $\forall x_1 \forall x_2 \ldots \forall x_n F$ (respectively, $\exists x_1 \exists x_2 \ldots \exists x_n F$), where $x_1, x_2, \ldots, x_n$ are all free variables of F.*

An interpretation gives a meaning to expressions in the logic of interest.

Definition A.1.9 (Interpretation of Terms and Formulae). *Let I be an interpretation of a signature Σ with universe U and $\eta : V \to U$ a variable valuation. Then the function $\eta^I : \mathcal{T}(\Sigma, V) \to U$ is defined by induction on the structure of the term:*

- $\eta^I(v) := \eta(v)$ *for a variable* v
- $\eta^I(f(t_0, \ldots, t_{n-1})) := I(f)(\eta^I(t_0), \ldots, \eta^I(t_{n-1}))$ *for an n-ary function symbol f and terms $t_0, \ldots, t_{n-1}$*

For a formula F in $\mathcal{F}(\Sigma, V)$, it is defined by induction on the structure of F when I and η satisfy *F, written $I, \eta \models F$:*

- $I, \eta \models p(t_0, \ldots, t_{n-1})$ *holds iff* $(\eta^I(t_0), \ldots, \eta^I(t_{n-1})) \in I(p)$.
- $I, \eta \models \bot$ *does not hold.*
- $I, \eta \models \top$ *holds.*
- $I, \eta \models \neg F$ *holds iff* $I, \eta \models F$ *does not hold.*
- $I, \eta \models F \wedge F'$ *holds iff* $I, \eta \models F$ *and* $I, \eta \models F'$ *hold.*
- $I, \eta \models F \vee F'$ *holds iff* $I, \eta \models F$ *or* $I, \eta \models F'$ *holds.*
- $I, \eta \models F \to F'$ *holds iff* $I, \eta \models F$ *does not hold or* $I, \eta \models F'$ *does.*
- $I, \eta \models \forall x F$ *holds iff* $I, \eta[x \mapsto u] \models F$ *holds for all* $u \in U$.
- $I, \eta \models \exists x F$ *holds iff* $I, \eta[x \mapsto u] \models F$ *holds for some* $u \in U$.

Notice that for the question whether $I, \eta \models F$ holds, it does not matter how η maps variables that are not free in F. Sentences can be interpreted without refering to a specific variable valuation at all.

Definition A.1.10 (Interpretation of Sentences and Theories).

- *Let $\eta_\emptyset$ be the variable valuation for the empty set of variables. Then an interpretation I for a signature Σ* satisfies *a sentence F in $\mathcal{F}(\Sigma, \emptyset)$, written $I \models F$, if $I, \eta_\emptyset \models F$ holds. We also say that I is a* model *of F. An interpretation is a* model *of a theory Th if it is a model of each formula in Th.*
- *A sentence is* valid *if it is satisfied by every interpretation. It is* satisfiable *if it is satisfied by some interpretation. It is* unsatisfiable *if it is not satisfied by any interpretation.*

We write $I, \eta \not\models F$ if I and η do not satisfy F, and $I \not\models F$ if I does not satisfy F.

Definition A.1.11 (Logical Consequence). *A sentence or theory Φ is a* logical consequence *of a sentence or theory Ψ, written $\Psi \models \Phi$, if every model of Ψ is also a model of Φ. Two sentences or theories are* equivalent *if they are logical consequences of each other.*

We restrict our attention to a certain class of interpretations. These interpretations have a fixed universe and a fixed interpretation of the function symbols; they differ only in the interpretation of predicate symbols. Thus, we only need to care about the latter when defining a specific interpretation.

Definition A.1.12 (Herbrand Interpretation). *An interpretation I for a signature Σ is a* Herbrand interpretation *if*

- *the universe of I is the set $\mathcal{T}(\Sigma, \emptyset)$ of ground terms, called the* Herbrand universe*, and*
- *for every n-ary function symbol f of Σ, the assigned function $I(f)$ maps a tuple $(t_0, \ldots, t_{n-1})$ of ground terms to the ground term $f(t_0, \ldots, t_{n-1})$.*

A Herbrand model *of a sentence or a theory is a Herbrand interpretation satisfying the sentence or theory.*

The Herbrand base *for a signature Σ is the set $\{p(t_0, \ldots, t_{n-1}) \mid p$ is an n-ary predicate symbol of Σ and $t_0, \ldots, t_{n-1} \in \mathcal{T}(\Sigma, \emptyset)\}$, i.e., the set of ground atoms in $\mathcal{F}(\Sigma, \emptyset)$.*

A Herbrand interpretation is uniquely determined by the set of ground atoms it satisfies. As is frequently done in the literature, we will usually identify such a set I of ground atoms, i.e., a subset of the Herbrand base, with the Herbrand interpretation satisfying exactly the elements of I.

A.2 Basic Calculi and Normal Forms

By a *calculus* we understand a set of rules for the manipulation of formulae. A calculus presupposes a logic and provides syntactic operations to derive new formulae of this logic from given ones. A calculus is used to find out whether some theory is satisfiable or whether some sentence is a logical consequence of some theory. The basis for the operations are so-called *inference rules*, which have the following general form:

$$\frac{F_1, \ldots, F_n}{F}$$

The formulae $F_1, \ldots, F_n$ are called the *premises* of the inference rule, the formula F below is its *conclusion*. An application of the rule is possible if the premises $F_1, \ldots, F_n$ are given or have been derived by previous rule applications; the effect of the application is that the conclusion formula F is derived and added to the formulae.

As in an algorithm, every manipulation step of a calculus should be effectively computable and it should be decidable whether a manipulation step is applicable in a given situation. However, a calculus is usually not deterministic like an algorithm. That is, more than one manipulation step may be applicable in a situation and the calculus need not provide a decision between the steps.

Our interest is in a classical calculus for proving theorems in first-order logic, the resolution calculus. Before we come to this calculus, we introduce some auxiliary notions.

A.2.1 Substitutions

Substitutions are a tool used in many calculi.

Definition A.2.1 (Substitution). *Let V be a set of variables and let $\mathcal{T}(\Sigma, V')$ be set of terms. A* substitution *is a function $\sigma : V \to \mathcal{T}(\Sigma, V')$. A substitution $\sigma : V \to \mathcal{T}(\Sigma, V')$ is* finite *if V is finite. The* identity substitution *will be denoted by ϵ.*

Substitutions are traditionally written as postfix operators. In the following, a substitution will be given as a set of pairs and terms of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where x_is are the variables and t_is are the terms.

A substitution $\sigma : V \to \mathcal{T}(\Sigma, V')$ is implicitly extended homomorphically to a function $\sigma : \mathcal{T}(\Sigma, V) \to \mathcal{T}(\Sigma, V')$ on terms. That is, for an n-ary function symbol f of Σ and $t_0, \ldots, t_{n-1} \in \mathcal{T}(\Sigma, V)$, we define $f(t_0, \ldots, t_{n-1})\sigma := f(t_0\sigma, \ldots, t_{n-1}\sigma)$. This allows us to compose substitutions. In accordance with the postfix notation of substitution application, the composition of substitutions is written as juxtaposition with the substitutions to be applied from left to right.

A substitution $\sigma : V \to \mathcal{T}(\Sigma, V')$ may actually be applied to any term with variables among V. Let Σ' be a signature containing at least all the function and predicate symbols of Σ with appropriate arity, then $\mathcal{T}(\Sigma, V')$ is a subset of $\mathcal{T}(\Sigma', V')$ and σ can be used as a function from V into $\mathcal{T}(\Sigma', V')$.

Substitutions for a set V of variables are also extended to several other types of objects. We introduce a generic name for all these objects.

Definition A.2.2 (Logical Expression). *Logical expressions over a set V of variables are defined inductively as follows. A* logical expression *over V is*

- *a term with variables in V,*
- *a formula with free variables in V,*
- *a substitution from an arbitrary set of variables into some set $\mathcal{T}(\Sigma, V)$ for some signature Σ or*
- *a tuple of logical expressions over V.*

A logical expression is a simple expression *if it does not contain quantifiers and infinite substitutions.*

Applying a substitution to a formula is defined recursively as follows:

- $p(t_0, \ldots, t_{n-1})\sigma := p(t_0\sigma, \ldots, t_{n-1}\sigma)$
- $\bot\sigma := \bot$
- $\top\sigma := \top$
- $(\neg F)\sigma := \neg(F\sigma)$
- $(F \wedge F')\sigma := (F\sigma) \wedge (F'\sigma)$
- $(F \vee F')\sigma := (F\sigma) \vee (F'\sigma)$
- $(F \to F')\sigma := (F\sigma) \to (F'\sigma)$
- $(\forall x F)\sigma := \forall x'(F\sigma[x \mapsto x'])$

- $(\exists x F)\sigma := \exists x'(F\sigma[x \mapsto x'])$

In the latter two cases, x' is an arbitrary variable that does not occur freely in $F(\sigma[x \mapsto c])$ with some constant c.

Applying a substitution σ to a substitution τ is defined as the substitution composition $\tau\sigma$. Substitution composition is associative (like function composition in general) and it also associates with substitution application. This allows us to omit parentheses.

Definition A.2.3 (Instance). *A logical expression e is an* instance *of a logical expression e' if there is a substitution σ such that $e = e'\sigma$. In this case, we also say that e' is* more general *than e.*

Definition A.2.4 (Variable Renaming). *A* variable renaming *for a logical expression e is a substitution σ with*

- *σ is injective*
- *$\sigma(X) \in V$ for all $x \in V$*
- *$\sigma(X)$ does not occur in e for free variables x of e*

Definition A.2.5 (Variants). *Two logical expressions are variants if they are identical modulo a variable renaming, i.e., e and e' are* variants *if there are two substitutions σ and τ such that $e = e'\sigma$ and $e' = e\tau$.*

Several calculi apply substitutions to logical expressions in order to make them equal. The process of finding appropriate substitutions is called *unification*. We will perform unification for simple expressions only.

Definition A.2.6 (Unification).
Let $e_0, \ldots, e_{n-1}$ be simple expressions.

- *A* unifier *for $e_0, \ldots, e_{n-1}$ is a substitution σ such that $e_0\sigma = \ldots = e_{n-1}\sigma$. The simple expressions $e_0, \ldots, e_{n-1}$ are* unifiable *if such a unifier exists. A unifier σ for $e_0, \ldots, e_{n-1}$ is* most general *if every unifier τ for $e_0, \ldots, e_{n-1}$ is an instance of σ, i.e., $\tau = \sigma\rho$ for some substitution ρ.*
- *A* tuple of unifiers *for $e_0, \ldots, e_{n-1}$ is a tuple $(\sigma_0, \ldots, \sigma_{n-1})$ of substitutions such that $e_0\sigma_0 = \ldots = e_{n-1}\sigma_{n-1}$. The simple expressions $e_0, \ldots, e_{n-1}$ are* tuple-unifiable *if such a tuple of unifiers exists. A tuple $(\sigma_0, \ldots, \sigma_{n-1})$ of unifiers for $e_0, \ldots, e_{n-1}$ is* most general *if every tuple $(\tau_0, \ldots, \tau_{n-1})$ of unifiers for $e_0, \ldots, e_{n-1}$ is an instance of $(\sigma_0, \ldots, \sigma_{n-1})$, i.e., $\tau_0 = \sigma_0\rho$ and ... and $\tau_{n-1} = \sigma_{n-1}\rho$ for some substitution ρ.*

A.2.2 Negation Normal Form and Prenex Form

It is sometimes useful to restrict ourselves to certain classes of first-order formulae. This allows us to apply specific calculi or to simplify general calculi.

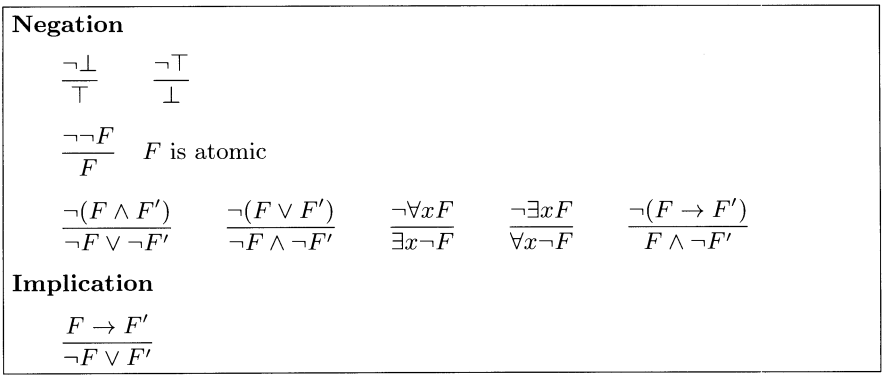


Fig. A.1. Negation and implication rules

Definition A.2.7 (Negation Normal Form). *A formula is in* negation normal form *if it has no subformula of the form $F \to F'$ and in every subformula of the form $\neg F'$ the formula F' is atomic.*

A subformula of a formula F is any formula occurring in F.

For every sentence F, there is an equivalent sentence F_{neg} in negation normal form. We obtain such a F_{neg} from F using the negation and implication rules from Fig. A.1 in a sequence of application of inference rules as follows. If F is not yet in negation normal form, then it has a subformula of the form $G \to G'$ or a subformula of the form $\neg G$ with a non-atomic G. For a given subformula of this form, there is exactly one specialization of the implication rule or one of the negation rules that has the subformula as its premise. We replace the subformula with the conclusion of this specialized rule. Replacements of this kind are performed as long as possible. The process will terminate after a finite number of steps and leaves a formula in negation normal form that is logically equivalent to the original formula F.

Definition A.2.8 (Prenex Form). *A formula F is in* prenex form *if it is of the form $Q_0 x_0 \dots Q_{n-1} x_{n-1} G$, where every Q_i is a quantifier, x_i is a variable, and G is a formula without quantifiers. We call $Q_0 x_0 \dots Q_{n-1} x_{n-1}$ the* quantifier prefix *and G the* matrix.

For every sentence F, there is an equivalent sentence in prenex form and it is possible to compute such a sentence from F.

A.2.3 Skolemization

The next goal is to eliminate the quantifier prefix. If there are only universal quantifiers, we can simply omit the prefix because it is uniquely determined by the variable symbols occuring in the matrix. If the prefix contains existential

quantifiers, we apply a transformation called *Skolemization*: each existentially quantified variable is replaced by a term composed of a new function symbol whose arguments are all the variables of universal quantifiers preceding the respective existential quantifier in the prefix.

Skolemization is defined here only for formulae in negation normal form. This restriction is not needed. However, it simplifies the argumentation in this section and the restricted definition will suffice for this book.

Definition A.2.9 (Skolemization). *Let $F \in \mathcal{F}(\Sigma, V)$ be a formula in negation normal form with an occurrence of a subformula $\exists x G$ with free variables $v_0, \ldots, v_{n-1}$. Let f be an n-ary function symbol not occurring in Σ, let s be the term $f(v_0, \ldots, v_{n-1})$, and let F' be the same formula as F, but with the occurrence of $\exists x G$ replaced by $G[x \mapsto s]$.*

Then F' is a Skolemized form *of F with the* Skolem function *f.*

Skolemization steps can be applied repeatedly to a sentence or a theory in negation normal form, thus eliminating all occurrences of existential quantifiers and introducing several Skolem functions.

A.2.4 Clauses

The resolution calculus given below works on a rather restricted set of first-order formulae, namely clauses:

Definition A.2.10 (Literal, Clause). *A* literal *is an atom or the negation of an atom. An atom is called* positive *literal and the negation of an atom is called a* negative *literal. A positive literal L and its negation $\neg L$ are called* complementary *literals. A* clause *is a formula of the form $\bigvee_{i=0}^{n-1} L_i$ where all the L_i are literals. This notation is called a clause in* disjunctive normal form. *If $n = 0$, then the clause is the* empty clause (empty disjunction). *A clause with exactly one positive literal is called* definite clause.

Thus, a clause is either the empty clause denoting $\bot$ or a non-empty clause $L_0 \vee \ldots \vee L_{n-1}$ with $n > 0$.

We use some syntactic sugaring for clauses. For a clause $\bigvee_{i=0}^{n+m-1} L_i$ with

$$L_i = \begin{cases} \neg B_i & \text{for } i = 0, \ldots, n-1 \\ H_{i-n} & \text{for } i = n, \ldots, n+m-1 \end{cases}$$

for atoms B_j and H_k, we frequently write $\bigwedge_{j=0}^{n-1} B_j \rightarrow \bigvee_{k=0}^{m-1} H_k$. This is called the *implication form* of the clause. $\bigwedge_{j=0}^{n-1} B_j$ is the *body* and $\bigvee_{k=0}^{m-1} H_k$ is the *head* of the clause.

A sentence of the form $\forall x_0, \ldots, x_{n-1} C$, where C is a clause, is called a *closed clause*. Sometimes we sloppily identify a *clause* with its universal closure. A theory is in *clausal form* or a *clausal theory* if it consists of closed clauses.

We combine the normalization steps from Sects. A.2.2, A.2.3, and A.2.4. An arbitrary theory *Th* can be transformed into clausal form as follows

Resolution

$$\frac{R \vee A \qquad R' \vee \neg A'}{R\sigma \vee R'\sigma'} \quad (\sigma, \sigma') \text{ is a most general pair of unifiers for the atoms } A \text{ and } A'$$

Factoring

$$\frac{R \vee L \vee L'}{(R \vee L)\sigma} \quad \sigma \text{ is a most general unifier for the literals } L \text{ and } L'$$

Fig. A.2. Inference rules of the resolution calculus

- Convert every formula in the theory into an equivalent formula in negation normal form.
- Perform Skolemization in order to eliminate all existential quantifiers.
- Convert the resulting theory, which is still in negation normal form, into an equivalent theory in clausal form.

A.2.5 Resolution

The resolution calculus [47] is a calculus for a clausal theory Th (see Fig. A.2).

In the resolution calculus, closed clauses are represented in disjunctive form and the universal quantifiers are omitted for convenience. The disjunction $\vee$ is considered to be associative and commutative and it has the falsity $\bot$ as its neutral element. (These properties could be described by inference rules.)

The resolution calculus is based on two inference rules (Fig. A.2).

- The *resolution rule* takes two clauses C and C' that can be instantiated in such a way that a literal from C and a literal from C' become complementary. Then the remaining literals from the two instantiated clauses are combined into a new clause, which is called the *resolvent* and is added.
- The *factoring rule* takes a clause C that can be instantiated in such a way that two literals of C become equal. Then one of these literals is removed and the remainder of the instantiated clause, which is called the *factor*, is added.

In both inference rules, most general unifiers are used in order to avoid too strong an instantiation. Notice also that tuple unification (actually pair unification) is used in the resolution rule, whereas plain unification is used in the factoring rule. This means that variables from different clauses can be instantiated independently, whereas variables from the same clause have to be instantiated uniformly.

List of Figures

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, Berlin, Heidelberg, New York, 1997. Springer.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the 2nd International Conference on Principles and Practice of Constraint Programming*, 4(2):133–165, 1999.
3. S. Abdennadher and M. Marte. University course timetabling using Constraint Handling Rules. *Journal of Applied Artificial Intelligence*, 14(4):311–326, 2000.
4. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming*, LNCS 1894, Berlin, Heidelberg, New York, 2000. Springer.
5. S. Abdennadher and H. Schütz. CHR$^{\vee}$: A flexible query language. In *Flexible Query Answering Systems*, LNAI 1495, Berlin, Heidelberg, New York, 1998. Springer.
6. K. R. Apt. Some remarks on boolean constraint. In K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints*, LNCS 1865, Berlin, Heidelberg, New York, 2000. Springer.
7. K. R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *5th International Conference on Principles and Practice of Constraint Programming*, LNCS 1713, Berlin, Heidelberg, New York, 1999. Springer.
8. K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of ACM*, 29(3):841–862, 1982.
9. F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *2nd World Congress on Expert Systems*, Estoril, Portugal, 1994.
10. F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, Heidelberg, New York, 1995. Springer.
11. B. Buchberger. Introduction to Groebner bases. In B. Buchberger and F. Winkler, editors, *Groebner Bases and Applications*, pages 3–31. Cambridge University Press, Cambridge, UK, 1998.
12. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.
13. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
14. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.

15. A. Colmerauer. An introduction to Prolog III. In J. W. Lloyd, editor, *Computational Logic: Symposium Proceedings*, pages 37–79. Springer, Berlin, Heidelberg, New York, 1990.
16. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.
17. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
18. F. de Boer, J. Kok, C. Palamidessi, and J. Rutten. Semantic models for concurrent logic languages. *Theoretical Computer Science*, 86(1):3–34, 1991.
19. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. MIT Press, Cambridge, Mass., 1990.
20. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *International Conference on Fifth Generation Computer Systems*, pages 693–702. Institute for New Generation Computer Technology, 1988.
21. S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. A. Valenzuela, and M. H. Wright. WISE design of indoor wireless systems: Practical computation and optimization. *IEEE Computational Science & Engineering*, 2(1):58–68, 1995.
22. T. Frühwirth. Theory and practice of constraint handling rules, Special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
23. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
24. T. Frühwirth and S. Abdennadher. The Munich rent advisor: A success for logic programming on the internet. *Journal on Theory and Practice of Logic Programming, Special Issue on Logic Programming and the Internet*, 1(3), 2001.
25. T. Frühwirth and P. Brisset. Optimal placement of base stations in wireless indoor communication networks. *IEEE Intelligent Systems Magazine, Special Issue on Practical Applications of Constraint Technology*, 15(1):49–53, 2000.
26. M. Henz and J. Würtz. Using Oz for college time tabling. In *First International Conference on the Practice and Theory of Automated Timetabling*, pages 283–296, Edinburgh, UK, 1995.
27. J. Herbrand. Recherches sur la theorie de la demonstrations, PhD thesis, 1930.
28. M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Stuttgart, Germany, Oct. 1988.
29. H. Hong. RISC-CLP(Real): Constraint logic programming over real numbers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research.* MIT Press, Cambridge, Mass., 1993.
30. G. Huet. Resolution d'equations dans les langages d'ordre 1, 2,..., PhD thesis, 1976.
31. J.-L. J. Imbert. Linear constraint solving in clp-languages. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, Heidelberg, New York, 1995. Springer.
32. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.
33. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
34. N. Karmarkar. A polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

35. R. Kowalski. Algorithm = logic + control. *CACM*, 22(7):424–435, 1979.
36. A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–73, 1985.
37. M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *4th International Conference on Logic Programming*, pages 858–876, Cambridge, Mass., 1987. MIT Press.
38. M. J. Maher. Complete axiomatizations of the algebras of finite, rational, and infinite trees. In *3rd Annual IEEE Symposium on Logic in Computer Science LICS'88*, pages 348–357, Los Alamitos, California, 1988. IEEE Computer Society Press.
39. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
40. E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks, Monterey, California, 1987.
41. S. Menju, K. Sakai, Y. Sato, and A. Aiba. A study on boolean constraint solvers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 253–268. MIT Press, Cambridge, Mass., 1993.
42. H. Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *3rd International Conference on the Practical Application of Constraint Technology*, pages 257–272, London, 1997. Practical Application Company Ltd.
43. R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.
44. R. Mohr and G. Masini. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988.
45. M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
46. J.-C. Regin. A filtering algorithm for constraints of difference in csp. In *AAAI National Conference*, pages 362–367, Seattle, Wash., 1994.
47. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.
48. K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Applications. *Journal of Symbolic Computation*, 8(6):589–603, 1989.
49. V. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, Mass., 1993.
50. A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, CWI - Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.
51. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, Chichester, 1986.
52. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.
53. G. Smolka. Residuation and guarded rules for constraint logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 405–419. MIT Press, Cambridge, Mass., 1993.
54. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, LNCS 1000, Berlin, Heidelberg, New York, 1995. Springer.
55. A. Tarski. *A Decision method for elementary algebra and geometry.* University of California Press, Berkeley, California, 1951.
56. P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

57. P. van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization.* MIT Press, Cambridge, Mass., 1997.
58. P. van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(FD). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, Heidelberg, New York, 1995. Springer.
59. M. Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1,2):139–168, 1996.

Index

Cognitive Technologies

Managing Editors: D.M. Gabbay J. Siekmann
