Principles of Constraint Systems and Constraint Solvers

Thom Frühwirth
Faculty of Computer Science, University of Ulm, Germany
Thom.Fruehwirth@informatik.uni-ulm.de

Slim Abdennadher
Department of Computer Science, German University Cairo, Egypt
Slim.Abdennadher@guc.edu.eg

February 1, 2005

Abstract

In this compact overview, we introduce the most common constraint systems used in constraint programming languages and algorithms to solve them. Constraint systems 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. Algorithms are presented as logical inference rules that are directly executable in the Constraint Handling Rules language.

1 Introduction

The idea behind 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 one, and not a prime number. Combining the pieces of partial information expressed by these constraints (digit, greater than one, odd, not prime) we are able to derive that the digit we are looking for is "9".

As it runs, a constraint program successively generates constraints. 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. In the constraint solver, efficient special-purpose algorithms are employed for specific classes of constraints.

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.

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.

The advantages of constraint-based 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 possible.

Contents. After the preliminaries, we introduce the common constraint systems, i.e. classes of constraints together with their solvers. This presentation is closely based on [FA03]. For each constraint system, we will give its allowed constraints, its constraint theory, a typical algorithm to solve the constraints, properties of the algorithm (termination and complexity), and an example of a typical application. Complexity analysis will be based on semi-naive implementations of CHR, while recent CHR compilers and other more low level implementations may achieve better complexities.

The domains of the constraint systems we consider are numbers (integers or reals), truth values and terms (trees). In particular, we will describe constraints for Booleans, finite enumeration and interval domains, linear polynomial equations and rational trees. Algorithms will be based on local propagation methods or variable elimination.

We will use the Constraint Handling Rules (CHR) [Frü98, Frü04] language extended with disjunction, CHR$^{\vee}$, to specify and implement the algorithms. Due to space limitations, we will introduce the CHR language only by means of concrete code. With CHR, we can describe the algorithms in a concise and compact manner by executable inference rules.

It should be emphasized that our view of constraint programming concentrates on the principles, and this not only for didactic and space reasons. For each constraint system we introduce, a richer set of constraints, more sophisticated algorithms and more efficient implementations than we give here are known and commonly used in constraint programming languages, be they academic or commercial. Still, constraint solving can be understood from the underlying principles as presented here. For lack of space we cannot discuss constraint-based optimization and global constraints in this paper. We also do not give an overview of current constraint-based systems or languages. For further reading we recommend [FA03, MS98, Apt03].

2 Preliminaries

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

2.1 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 for reasoning in the context of constraint programming languages.

The following definition is based on Höhfeld and Smolka [HS88] and Jaffar and Maher [JM94]. We assume some familiarity with first order predicate logic.

Definition 2.1 *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* $=$*, 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 $=$ play a prominent role. In constraint programming, we are mainly concerned with conjunctions of atomic constraints whose variables are (implicitly) existentially quantified. In addition, the atomic constraints are syntactically restricted so that they can be solved efficiently. Examples of constraint systems will be given in the next sections.

2.2 Constraint Solvers

A *constraint solver* implements an algorithm for solving allowed constraints in accordance with the constraint theory. The solver collects the constraints that arrive *incrementally* from one or more running programs. It puts them into the *constraint store*, a data structure for constraints. It tests their satisfiability, simplifies and if possible solves them. The final constraint that results from a computation is called the *answer (constraint)*.

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 $CT \models \forall(C \leftrightarrow D)$

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

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

Incrementality means that simplifying C and then simplifying the result together with newly arrived constraints D should give the same result as simplifying $C \wedge D$ (up to reordering of constraints). Ideally, the incremental computation $solve(solve(C) \wedge D)$ should not be more costly than $solve(C \wedge D)$.

When used from within a constraint programming language, 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, i.e. if $CT \models \neg\exists C$.

Example. $X{>}Y \wedge Y{>}X$ is inconsistent, and $X{\geq}Y \wedge Y{\geq}X$ is not.

This reasoning service corresponds to satisfaction-completeness. The solver implements a *decision procedure* for satisfiability of allowed constraints. The detection of inconsistency is essential in constraint programming, in particular it can avoid unnecessary continued search for solutions.

Simplification

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

Example. $X{\leq}2 \wedge X{\leq}4$ is simplified into $X{\leq}2$, and $2{*}X{=}6$ 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. 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 v, i.e. $CT \models \forall(C \rightarrow X{=}v)$, where v is a constant.

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

This special case of simplification is important for representing answers 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 resulting in D (which does not contain X), and where $CT \models \exists X C \leftrightarrow D$.

Example. Projection of $\exists Y(X{<}Y \land Y{<}Z)$ onto X and Z results in $X{<}Z$ over the reals, but in $X{+}1{<}Z$ over the integers. In the syntactic equation $\exists Y(X{=}f(Y))$, 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, projection may yield a significant increase in the size and number of constraints.

The reasoning services can be regarded and implemented as variations of simplification that maintains a *normal form* of the constraints. The normal form then also determines what "simpler" means. The constraint solver is expected to implement constraint simplification 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 content with *incomplete* implementations that when complete would take exponential time.

2.3 Constraint Solving Algorithms

A variety of algorithms exist for constraint systems, mostly adapted from artificial intelligence, graph theory and operations research. For didactic reasons, we will concentrate on the basic principles of these algorithms. They will only be of use if they have polynomial complexity (except for search, of course).

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 typically have to be interleaved with *search* to achieve completeness. A clear distinction between the two approaches is not always possible.

2.3.1 Variable Elimination

The allowed constraints are typically *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 $X{=}Y{+}Z \land Z{>}0$. (Fourier's algorithm works directly on linear inequalities, but has exponential complexity.)

Given an equation $e_1{=}e_2$, we call e_1 the *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 constant. 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 (Section 5). For example, in $X{=}7{-}Y \ \wedge \ X{=}3{+}Y$, we can remove the second occurrence of X. This results in $X{=}7{-}Y \ \wedge \ Y{=}2$. Removing Y finally leads to the solution $X{=}5 \ \wedge \ Y{=}2$.

2.3.2 Local Propagation for Local Consistency

The given problem is broken down into small fixed-size overlapping sub-problems. (By size we mean the number of constraints and variables.) These sub-problems are considered repeatedly until a fixpoint is reached. The constraints of the sub-problems are tightened (simplified), and new implied (redundant) constraints are computed (propagated) from them. The constraints are added hoping that they cause further simplification for the overlapping sub-problems.

For example, we may consider sub-problems consisting of two constraints. Given $X{\geq}Y \ \wedge \ Y{\geq}Z \ \wedge \ Z{\geq}X$, the first two constraints imply $X{\geq}Z$. This constraint tightens the third initial constraint $Z{\geq}X$ into $Z{=}X$.

For any given problem, there is only a polynomial number of fixed-size sub-problems. So if we can deal with sub-problems in polynomial time, there is a chance to have a polynomial algorithm to simplify the overall problem. (It need not be so if the the atomic constraints of the problem allow an exponential or worse number of tightenings.) For example, if the constraints are simple inequalities between two variables, we can compute implied and tightened constraints by a simple table look-up. Since there is a fixed number of different kinds of inequalities, tightening can only be performed a constant number of times per constraint.

While local propagation thus can be efficient, in general it only provides an approximation algorithm for the set of solutions. That is, *local consistency* can be achieved, ensuring that each sub-problem has a solution, but not necessarily *global consistency*, meaning that the overall problem has a solution (see also next subsection).

Classical consistency algorithms were first explored for *constraint networks* in artificial intelligence research in the late 1960's. The main algorithms are *arc* *consistency* (Subsection 4.1) and *path consistency*. Originally, the algorithms involved unary and binary constraints over finite sets of values only (so-called finite domains, see Section 4). Since these algorithms tighten constraints by removing values from the sets, they are polynomial.

Local consistency methods may require that expressions are in *flat normal form*, where distinct variables are the only arguments of functions (i.e., functions are not allowed to be nested). An expression can be *flattened* by performing the opposite of variable elimination. Each non-variable sub-expression and each repeated occurence of a variable is replaced by a new variable that is equated with the replaced expression. For example, $2X{+}Y{>}5$ is flattened into $W{>}F \wedge X{+}V{=}W \wedge X{+}Y{=}V \wedge F{=}5$. Note that this transformation is only necessary once, before the actual computation starts.

In this paper we choose flat normal forms mainly for didactic reasons. 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 and constraints. Consistency methods are sensitive to the representation of the constraints. (There is no efficient and general way to find the representation that enables most simplification most efficiently, even though progress has been reported for linear integer arithmetic constraints [HS03].)

2.3.3 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* associated with the computation. Search is a *case analysis*, that is *case splitting* by introducing *choices*. Search techniques most used in constraint programming are depth-first search, and branch and bound for optimisation.

Usually, search is interleaved with constraint solving. A minimal 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*. 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 searched as $X{<}Y \vee X{>}Y$.

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 may choose the variable that occurs most for labeling in the hope that this will cause most simplification. This heuristic is an example of a *static* order. Choosing the variable with the smallest set of possible values first is called *first-fail principle*, since we may expect that labeling this variable will lead to failure quickly, thus *pruning* branches in the search tree early. This heuritic results in a *dynamic* order. Similarly, since the next value for labeling a variable must be chosen, there are also *value ordering* heuristics. For an example, see the Boolean constraint solver (Section 3).

2.4 Constraint Handling Rules

We will use the Constraint Handling Rules (CHR) [Frü98] language extended with disjunction, CHR$^{\vee}$, to specify and implement the constraint solver algorithms. CHR was initially developed for writing constraint solvers, but has matured into a general-purpose concurrent constraint language over the last decade. Its main features are a kind of multi-set rewriting combined with propagation rules. The clean logical semantics of CHR facilitates non-trivial program analysis and transformation. Implementations of CHR now exist in many Prolog systems, also in Haskell and Java. Besides constraint solvers, applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis [Frü04].

Due to space limitations, we will introduce CHR only very briefly. On the web-site for CHR [Frü04], more information and executable constraint solvers for online experimentation are available.

There are two types of rules in CHR, *simplification rules* of the form `H <=> G | B` and *propagation rules* of the form `H ==> G | B`. The l.h.s. `H` is a conjunction of CHR (i.e. user-defined, rule-defined) constraints, the *guard* `G` is a conjunction of *built-in* (i.e. pre-defined) constraints, and finally, the r.h.s. `B` a conjunction (or disjunction in CHR$^{\vee}$) of arbitrary constraints. An empty, trivial guard can be omitted. An optional name may be prepended to the rule with the symbol `@`. The CHR rules have an immediate logical reading, where the guard implies a logical equality or implication between l.h.s. and r.h.s. of a rule.

The programs will use concrete syntax of Prolog implementations of CHR$^{\vee}$: Conjunction $\wedge$ is written as comma `','`. Disjunction $\vee$ is written as semi-colon `';'`. Variables start with upper-case letters, constraint and function symbols with lower-case letters. We also use some typical Prolog *built-ins*, such as syntactic equality `=` that encodes term unification. For uniformity, these built-ins are regarded as built-in constraints and will be explained at their use.

Operationally, a simplification rule `H <=> G | B` *replaces* instances of `H` by `B` provided the guard holds. A propagation rule `H ==> G | B` instead *adds* `B` to `H`. CHR is a concurrent committed-choice language, that means that rules can be applied in parallel and that (unlike Prolog) a rule application is never undone. CHR rules are applied exhaustively, until a fixed-point is reached, to the global conjunction (considered as multi-set) of constraints. To avoid trivial non-termination, a CHR propagation rule is never applied a second time to the syntactically same conjunction of constraints. If new constraints arrive, rule applications are restarted. (Thus any correct, terminating and confluent constraint solver written in CHR will be incremental by nature.)

3 Boolean Algebra B

We start with the Boolean constraint system that admits a simple local consistency algorithm to solve constraints [MSSA93].

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
Boolean algebra according to the following truth table.

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

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

where X, Y, and Z are variables or truth values and where $\odot \in \{\sqcap, \sqcup, \oplus, \rightarrow, \leftrightarrow\}$.

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.

For simplicity, 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*.

3.1 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 whenever possible. Unlike in Section 6, we assume here that syntactic equality `=` is a *built-in constraint*. 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 into syntactic equalities.

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.

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.

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

Complexity. We will give an informal derivation of the worst case time complexity. Let c be the number of atomic Boolean constraints in a constraint problem. Each rule application removes one constraint. Hence, there can be at most c rule applications.

Before applying a rule, we have to find it. In the worst case, there are *c rule tries (rule application attempts)*, since we may have to check each of the at most c constraints in the current state of computation against the given rules until we find an applicable one. Each check can be done in quasi-constant time using the union-find algorithm for built-in syntactic equality (see Subsection 6.1).

Thus the worst case time complexity of applying the above rules is slightly worse than $O(c^2)$, the complexity observed in experiments is usually linear [Frü02].

3.1.1 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 (cf. Subsection 2.3.3) 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).
```

Lists have a special syntax, 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`.

An efficient implementation has to make sure that constraint solving and search are interleaved. For example, consider the constraint problem `and(X,Y,Z), and(X,Y,W), neg(Z,W), enum([X,Y,Z,W])`. The computation 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 computation fails finitely and *false* is returned as a result.

To improve the labeling, we can introduce a *variable ordering*. We choose the variable for labeling that occurs most in the problem - in the hope that this will cause most simplification. This heuristic is called *first-fail principle*.

We can also introduce a *value ordering*. We count the cases in which the values `0` and `1` cause simplification. For example, 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 (Section 4) that are solved using consistency techniques. This increases expressiveness, since arithmetic functions are available.

- *Integer Programming*

  This technique from operations research uses linear programming methods to solve Boolean problems expressed as linear polynomial equations (Section 5) 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.

- *Theorem Proving*

  The famous *SAT problems* can be regarded as propositional Boolean constraint problems in clausal form. Variants of resolution are employed to solve problems in clausal form. (Resolution can be considered as constraint solving [Dum95].) Already the 3-SAT problem (conjunction of clauses with at most three variables) is NP-complete.

- *Boolean Unification*

  An extension of syntactic unification (Section 6) is used to solve Boolean equalities. Boolean unification computes a single, most general solution. Boolean variable elimination requires the introduction of auxiliary variables, possibly leading to an exponential blow-up in size.

3.2 Application Example: Digital Circuits

Digital circuits are usually modeled using Boolean constraints. They are applied to generate, specialize, simulate, and analyze (verify and test) the circuits. Of special importance is fault analysis. A digital circuit consist of *(logical) gates*, which correspond to allowed atomic constraints.

We briefly show how to model the classical *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. 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 computation 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`.

4 Finite Domains *FD*

Finite domains are one of the success stories of constraint logic programming (CLP). 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. Many real-life combinatorial problems can be expressed in this constraint system, most prominently scheduling and planning applications. Finite domains appeared in one of the first CLP languages CHIP [DVS$^+$88]. It was the result of a synthesis of logic programming (Prolog) and constraint networks as explored in artificial intelligence research. Other influential CLP languages with finite domains are clp(FD) [CD96] and cc(FD) [vHSD95].

Constraint System *FD*

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

Signature

- Function symbols.
  - The integers 0 and 1
  - Unary prefix operator $-$ (minus sign)
  - Binary infix operators $+$, and .. for intervals, and lists `[...]`
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbols $=, <, \leq, >, \geq, \neq$, and *in* for domains

Constraint theory
Presburger arithmetic extended by

- $X + (-X) = 0, X{\leq}Y \leftrightarrow \exists Z X + Z = Y (Z \text{ nonnegative}), \ldots$
- $X\ in\ n..m \leftrightarrow n{\leq}X \wedge X{\leq}m$
- $X\ in\ [k_1, \ldots, k_l] \leftrightarrow X{=}k_1 \vee \ldots \vee X{=}k_l$

Allowed atomic constraints
Linear equations and inequations:

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

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.

The theory is Presburger arithmetic extended to accommodate the additional constraints and negative numbers. It is mentioned here for completeness, its knowledge is not necessary to understand the constraint system. Presburger arithmetic is complete and decidable, it axiomatizes the linear fragment of arithmetic over natural numbers with + and =. (Linearity means that there is no multiplication between variables.) The theory only refers to the numbers 0 and 1, which are in the signature. The interpretation will map arithmetic expressions to the integers, which form the constraint domain.

The *domain constraint* $X\ \mathit{in}\ D$ means that the variable X takes its value from the given finite domain D. More precisely, $X\ \mathit{in}\ [k_1, \ldots, k_l]$ denotes an *enumeration domain constraint*, where the possible values of X are explicitly enumerated. $X\ \mathit{in}\ n..m$ 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, $X\ \mathit{in}\ []$ or $X\ \mathit{in}\ n..m\ (n > m)$, 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 $X\ \mathit{in}\ [1, 2, 3] \wedge X{\neq}2$ we can derive the tighter domain constraint $X\ \mathit{in}\ [1, 3]$, while from $X\ \mathit{in}\ 1..3 \wedge X{\neq}2$ 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 $X\ \mathit{in}\ [v]$ or $X\ \mathit{in}\ v..v$.

Any linear polynomial equation with rational coefficients can be expressed as a conjunction of allowed constraints. First, the coefficients of the polynomial are multiplied with a number such that they all become integers. Then the multiplications are rewritten as sums, e.g., $3X$ becomes $X + X + X$. Then, the resulting expression is flattened. For example, $X{+}X{+}Y{>}5$ is flattened into $W{>}F \wedge X{+}V{=}W \wedge X{+}Y{=}V \wedge F\ \mathit{in}\ [5]$. While therefore our allowed constraints are of sufficient expressibility, in practice one will use coefficients and linear polynomials in allowed constraints to improve representation and propagation. It is easy to extend the constraint solver given below in this way. (Since the number of constraints and variables changes then, the complexity formula changes accordingly).

4.1 Arc Consistency and Bounds 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.

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. A conjunction of constraints can be made arc consistent by making each atomic constraint arc consistent. Obviously, this approach describes a local consistency algorithm (Section 2.3.2), because we consider sub-problems of one atomic constraint together with the domain constraints of its variables.

The worst case time complexity of arc consistency is $O(cd^n)$ for arbitrary n-ary constraints [MM88], where c is the number of constraints and d is the size of the largest domain.

For interval domains, a weaker but analogous form of arc consistency proves useful. We just require that all interval bounds participate in a solution of the constraint. A conjunction of constraints is bounds consistent if each atomic constraint in it is bounds consistent. Constraints can be made bounds consistent by tightening their interval domains.

Note that local consistency does not imply global consistency. For example, the inconsistent constraint $X\ in\ D \ \wedge\ Y\ in\ D \ \wedge\ X{\neq}Y \ \wedge\ X{=}Y$ is arc and bounds consistent for all domains D with more than one value.

4.2 Local Propagation Constraint Solver

For simplicity, we start with the bounds consistency algorithm for interval constraints [vHDT92, Ben95]. The implementation is based on interval arithmetic.

4.2.1 Interval Domains

In the solver, `in`, `le`, `eq`, `ne`, 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 local consistency affect the interval constraints (`in`) only, the other constraints 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 denotes a unique value that happens to be the bound of the other interval.

Example 4.1 Here is a sample computation involving `le`:

```
        U in 2..3, V in 1..2, U le V
 ↦le    V in 1..2, U le V, U in 2..2
 ↦le    U le V, U in 2..2, V in 2..2.
```

Finally, $X+Y=Z$ is represented as `add(X,Y,Z)` in the rule:

```
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. Here these built-in constraints describe when addition is bounds consistent.

Example 4.2 Here is an example computation involving `add`:

```
    U in 1..3, V in 2..4, W in 0..4, add(U,V,W)  ↦add
    add(U,V,W), U in 1..2, V in 2..3, W 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.

Complexity. 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. 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. There may be up to $O(c)$ rule tries. Each rule try and each rule application take constant time if the arithmetic built-ins take constant time (which is the case for bounded numbers). So the worst case time complexity is $O(cvw)$.

4.2.2 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 all values 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 4.3 The constraint problem `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 problem `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 problem `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 arguments for termination 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 $O(cvd^3)$.

4.2.3 Search

To achieve satisfaction-completeness, search must be employed (Subsection 2.3.3). We implement the search routine analogous to the one for Boolean constraints (Subsection 3.1.1).

```
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. Calling `indomain(X)` in the second disjunct ensures that subsequently, the next value for `X` from the list `L` will be tried.

For interval domains, search is usually done by splitting intervals in two halves. This splitting is 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. Note that `indomain(X)` is called outside of the disjunction, because both interval halves can be split further in the future.

4.3 Application Example: 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 find a feasible schedule that minimizes the overall production time that is needed to accomplish all the tasks. This problem can be expressed as a finite-domain constraint problem. (Optimization aspects are not discussed for lack of space.)

Each *task* T_i is associated with a constraint

$$S_i + d_i = E_i,$$

where S_i is the starting time of the task, d_i is its duration (usually known) and E_i is its end time. These temporal variables 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

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

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, as e.g. in CHIP.

Additional constraints can model set-up times, release times, deadlines, as well as renewable resources, and non-availability of resources at certain times. These conditions are usually implemented as *global constraints* that can take an arbitrary number of variables as argument, which results in better propagation.

5 Linear Polynomial Equations $\Re$

One motivation for introducing constraints in 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($\Re$) [JMSY92]) or rational numbers (Prolog-III [Col90], CHIP [DVS$^+$88]). The constraint system $\Re$ relies on variable elimination. Since this algorithm is complete, no search is necessary.

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:

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

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 the *(linear) polynomial.*

Tarski's theory of real closed fields covers linear and non-linear polynomials. It is mentioned here for completeness, its knowledge is not necessary to understand the constraint system. The theory only refers to the real numbers 0 and 1, which are 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.

5.1 Variable Elimination Constraint Solver

Typically, in constraint solvers, incremental variants of classical variable elimination algorithms [Imb95] 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.

To illustrate the principle of *variable elimination*, we first consider equations only. A conjunction of equations 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 need to 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 `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 `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 5.1 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
```

The `eliminate` rule is now applicable to the equations in reversed order, i.e. `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
```

So `X` is determined to be `-2` and `Y` is `-1`.

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

```
determine @ A*X+B eq 0 <=> number(B) | X is -B/A.
```

The built-in `V is E` computes the result of the arithmetic expression `E` and equates it with the variable `V`.

Termination. The solver terminates, because 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.

Complexity. Consider a problem with c equations and v different variables. Since each constraint can contain at most all v variables, there can be at most cv occurrences of variables in any state of the computation. Each rule application either removes a variable or an equation. There are at most $O(cv)$ rule applications, provided a removed variable will never be re-introduced into the equation. In an incremental setting this is achieved by first removing variables from a newly arrived equation and only then use the new equation for removing variables in the old equations. This behavior is implicit in most CHR implementations, because they prefer removal of new constraints. For a new constraint, there are $O(c)$ rule tries in the worst case. Trying to apply the `eliminate` rule to given constraints has complexity $O(v)$, the `empty` rule takes constant time. Hence, the overall complexity is $O(c^2v^2)$, i.e., quadratic in the maximal size of the problem.

Inequations. We can extend our solver to inequations. As in the Simplex algorithm, an inequation is flattened into an equation and a simple inequation on a single new variable, which is called *slack variable.* 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 $\odot \in \{<, \leq, >, \geq, \neq\}$.

However, the given rules do not suffice to detect inconsistency between equations consisting only of slack variables. 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, one can add rules that either enforce a more strict solved form (as in CHIP), or do more variable elimination on such equations (as in CLP($\Re$)).

5.2 Application Example: Finance

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$^\vee$, where

- `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 constraint problem `mortgage(100000,360,0.01,1025,S)` results in `S=12625.90` (rounded). This demonstrates the effect of accumulation of interest: even though we have paid back 360 times 1025 (= 369000) over time, there is still a final debt of 12625.90.

With the same rule, we can also compute what initial loan we can pay back completely under the conditions above: the problem `mortgage(D,360,0.01,1025,0)` results in `D=99648.79`, only a slightly lower amount.

But how much longer would we have to pay for the original loan of 100000? The problem `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 problem `-1025 lt S, S le 0, mortgage(100000,T,0.01,1025,S)` results in `T=374, S=-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 problem `mortgage(D,360,0.01,R,0)` results in `R eq 0.0102861198*D`, i.e., the monthly repayment is just above the 1% interest rate 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.

6 Rational Trees *RT*

Syntactic equality of first order 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. (We therefore omit a special section on application examples.)

In early Prolog implementations, the *occur-check* was omitted from syntactic equality 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 [Col82]. 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., $X{=}f(X)$.

Constraint System *RT*

Domain
Herbrand universe

Signature

- Infinitely many function symbols.
- Constraint symbols.
  - Nullary symbols *true*, *false*
  - Binary symbol =

Constraint theory
Clark's Equality Theory
Reflexivity: $\forall(\mathit{true} \rightarrow x{=}x)$
Symmetry: $\forall(x{=}y \rightarrow y{=}x)$
Transitivity: $\forall(x{=}y \wedge y{=}z \rightarrow x{=}z)$
Compatibility: $\forall(x_1{=}y_1 \wedge \ldots \wedge x_n{=}y_n \rightarrow f(x_1,\ldots,x_n){=}f(y_1,\ldots,y_n))$
Decomposition: $\forall(f(x_1,\ldots,x_n){=}f(y_1,\ldots,y_n) \rightarrow x_1{=}y_1 \wedge \ldots \wedge x_n{=}y_n)$
Contradiction: $\forall(f(x_1,\ldots,x_n){=}g(y_1,\ldots,y_m) \rightarrow \mathit{false})$ if $f \neq g$ or $n \neq m$

Allowed atomic constraints

$$C ::= \mathit{true} \mid \mathit{false} \mid s{=}t$$

where s and t are terms over the signature Σ.

The associated domain is the Herbrand universe, i.e. all terms that can be built out of the function symbols in the signature. The constraint theory is decidable, but not complete. For example, $\exists X, Y\ (X{=}f(X) \wedge Y{=}f(Y) \wedge \neg X{=}Y)$ does not follow from the theory, nor does its negation. One more axiom concerning implied equalities is needed for a complete theory [Mah88].

Let X_i be variables and t_j be arbitrary terms ($1 \leq i,j \leq n$). A conjunction of equations is *solved (in solved normal form)* if it is of the form

$$X_1{=}t_2 \wedge X_2{=}t_3 \wedge \ldots \wedge X_{n-1}{=}t_n \quad (n \geq 0),$$

where $X_i \neq X_j$ and $X_i \neq t_j$ for all ($1 \leq i < j \leq n$).

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

From the solved form we can read off the most general unifier of the given set of initial equations by interpreting each equation $X_i{=}t_i$ as a substitution that replaces X_i by t_i.

6.1 Variable Elimination Constraint Solver

The following algorithm to solve equations over rational trees is similar to the one in [Col82], 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).

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. The *size of a term* is the number of occurrences of function symbols in the term. For different variables x and y, either x`@<`y or y`@<`x, they cannot be the same in the order. The built-in constraint `X@=<Y` holds if `Y@<X` does not hold. (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+1}, since `@<` is transitive and implies $\neq$.)

We need some more auxiliary built-ins 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 using `eq`.

```
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.
```

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. Note the relationship with the `eliminate` rule in the constraint solver for $\Re$ (Section 5).

The solver is satisfaction-complete. 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 6.1 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`.

$$\begin{array}{ll}
 & \underline{\texttt{h(Y,f(a),g(X,a)) eq h(f(U),Y,g(h(Y),U)))}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{Y eq f(U), }\underline{\texttt{f(a) eq Y}}\texttt{, g(X,a) eq g(h(Y),U)} \\
\mapsto_{\texttt{orientation}} & \texttt{Y eq f(U), Y eq f(a), }\underline{\texttt{g(X,a) eq g(h(Y),U)}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{Y eq f(U), Y eq f(a), X eq h(Y), }\underline{\texttt{a eq U}} \\
\mapsto_{\texttt{orientation}} & \underline{\texttt{Y eq f(U)}}\texttt{, }\underline{\texttt{Y eq f(a)}}\texttt{, X eq h(Y), U eq a} \\
\mapsto_{\texttt{confrontation}} & \texttt{Y eq f(U), }\underline{\texttt{f(U) eq f(a)}}\texttt{, X eq h(Y), U eq a} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{Y eq f(U), }\underline{\texttt{U eq a}}\texttt{, X eq h(Y), }\underline{\texttt{U eq a}} \\
\mapsto_{\texttt{confrontation}} & \texttt{Y eq f(U), U eq a, X eq h(Y), }\underline{\texttt{a eq a}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{Y eq f(U), U eq a, X eq h(Y)}
\end{array}$$

Example 6.2 Here is a simple example involving infinite rational trees.

$$\begin{array}{ll}
 & \underline{\texttt{X eq f(X)}}\texttt{, }\underline{\texttt{X eq f(f(X))}} \\
\mapsto_{\texttt{confrontation}} & \texttt{X eq f(X), }\underline{\texttt{f(X) eq f(f(X))}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \underline{\texttt{X eq f(X)}}\texttt{, }\underline{\texttt{X eq f(X)}} \\
\mapsto_{\texttt{confrontation}} & \texttt{X eq f(X), }\underline{\texttt{f(X) eq f(X)}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{X eq f(X), }\underline{\texttt{X eq X}} \\
\mapsto_{\texttt{reflexivity}} & \texttt{X eq f(X)}
\end{array}$$

i.e., the second constraint is redundant.

Termination. The solver terminates, the proof of [Col82] 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 applicable at most once to an equation.
- The `decomposition` rule leads to equations between smaller terms.
- 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 `@<`. 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.

Complexity. The built-in constraints can be implemented such that they take constant time. Due to the confrontation rule, the complexity of the solver is worse than linear. The intricate interaction between the `decomposition` rule and the `confrontation` rule in the case of infinite terms makes it hard to determine the exact worst case time complexity of the solver. We conjecture that it is quadratic in the number of function symbols and variables.

Classical and Optimal Algorithms. In 1930, Herbrand gave an informal description of a unification algorithm. Robinson rediscovered a similar algorithm when he introduced the resolution procedure for first-order logic in 1965. Since the late 70s, there are quasi-linear time algorithms for unification. They can be considered as extensions of the union-find algorithm from constants to trees. Indeed, the CHR implementation of union-find [SF05] can be combined with the constraint solver rules given here to give a time-optimal algorithm.

7 Conclusions

In this compact overview, we introduced the most common constraint systems used in constraint programming languages and algorithms to solve them. We described constraints for Booleans, finite enumeration and interval domains, linear polynomial equations and rational trees.

For each constraint system, we gave its allowed constraints, its constraint theory, an algorithm to implement it, discussed the algorithm's termination and worst-case time complexity, and an example of a typical application. Algorithms were presented as efficiently executable logical inference rules of the Constraint Handling Rules (CHR) language.

References

[Apt03] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, UK, 2003.

[Ben95] F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, Heidelberg, New York, 1995. Springer.

[CD96] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.

[Col82] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.

[Col90] 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.

[Dum95] E. Dumbill. Application of resolution and backtracking to the solution of constraint satisfaction problems, project report, 1995.

[DVS$^+$88] 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.

[FA03] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.

[Frü98] 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.

[Frü02] 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.

[Frü04] T. Frühwirth. CHR web-pages, www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/chr.html, 2004.

[HS88] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Stuttgart, Germany, October 1988.

[HS03] W. Harvey and P. J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints Journal*, 8(2):173–207, 2003.

[Imb95] 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.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.

[JMSY92] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp($\Re$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[Mah88] 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.

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *8th European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988.

[MS98] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, Cambridge, Mass., 1998.

[MSSA93] 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.

[SF05] T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming (TPLP)*, to appear 2005.

[vHDT92] P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[vHSD95] 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.