I N F O R M A T I K

A Unifying Framework for Integer and Finite Domain Constraint Programming

Alexander Bockmayr   Thomas Kasper

MPI–I–97–2–008   August 1997

FORSCHUNGSBERICHT   RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Im Stadtwald   66123 Saarbrücken   Germany

Authors' Addresses

Alexander Bockmayr
Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany
bockmayr@mpi-sb.mpg.de

Thomas Kasper
Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany
kasper@mpi-sb.mpg.de

Publication Notes

The present report has been submitted for publication elsewhere and will be copyrighted if accepted.

Acknowledgements

This work was partially supported by the ESPRIT Working Group 22457 (CCL II) and the Human Capital and Mobility network CONSOLE.

Abstract

We present a unifying framework for integer linear programming and finite domain constraint programming, which is based on a distinction of primitive and non-primitive constraints and a general notion of branch-and-infer. We compare the two approaches with respect to their modeling and solving capabilities. We introduce symbolic constraint abstractions into integer programming. Finally, we discuss possible combinations of the two approaches.

Keywords

Integer programming, constraint programming, combinatorial optimization, modeling

1 Introduction

Combinatorial problems are ubiquitous in many real world applications like scheduling, planning, transportation, assignment, and many others. Besides special purpose algorithms to compute exact or approximate solutions, there exist also some general approaches to handle these problems. We are interested here in two such approaches:

- Integer linear programming (ILP)
- Finite domain constraint programming (CP(FD))

Integer linear programming has a long tradition in operations research and has produced a large number of impressive results during the last 40 years, see for example [28, 24]. Finite domain constraint programming is a promising new approach for solving complex combinatorial problems, which combines recent progress in programming language design, like constraint logic programming [23] or concurrent constraint programming [30], with efficient constraint solving techniques from mathematics, artificial intelligence, and operations research, see for example [37, 38].

The aim of this paper is to develop a unifying framework for integer linear programming and finite domain constraint programming. On the one hand, we want to clarify the relationship between these two approaches and identify (some of) their strengths and weaknesses. On the other hand, we want to show how each of the two approaches may profit from the other and indicate possible ways towards their integration.

Practical problem solving usually involves two steps [31, 39, 40]:

- Model building
- Model solving

In the first step, we develop a model of the problem in some formal language. In the second step, we solve this model on a computing system, possibly after translating it into a more machine-oriented form. In order to compare integer linear programming and finite domain constraint programming, we ask two fundamental questions, closely related to each other:

- How expressive is the language that we can use to build a model?
  (Declarative view)
- How efficient are the algorithms that support this language when the model is solved?
  (Operational view)

Very roughly, we can say that finite domain constraint programming offers the more powerful language to express combinatorial problems, while integer linear programming supports only a rather small language, for which however very efficient algorithms are available. The overall performance of the two approaches, i.e. the tradeoff between expressivity and efficiency, is of course problem dependent.

The organization of this paper is as follows. We start in Sect. 2 by comparing integer linear programming and finite domain constraint programming from the declarative point of view. We formally define the underlying constraint languages in the framework of first-order predicate logic and give a declarative logical semantics in the standard model of rational numbers. In Sect. 3, we compare the two approaches from the operational point of view. To describe the operational semantics, we develop a unifying framework, *branch-and-infer*, and show how this subsumes the two approaches. In the remaining sections, we use this framework to extend ILP with concepts from CP(FD) and vice versa. In Sect. 4, we show how the symbolic constraint concept of constraint programming might enrich integer programming. In Sect. 5, we discuss how linear programming might enhance finite domain constraint solving and indicate possible ways towards an integration of the two approaches.

2 Modeling combinatorial problems in ILP and CP(FD)

When we solve a combinatorial problem on a computer, we first need a language to formulate the problem. For example, this can be a modeling language from mathematical programming, like AMPL or GAMS [31], or a high-level programming language from computer science, like CHIP [18], ILOG solver [29], or OZ [32]. In order to clarify the relationship of the constraint languages underlying ILP and CP(FD), we propose to use *first-order predicate logic* [9], which gives us a standard syntax and a very well-understood semantics to compare the two approaches. There exist also higher-order notions in finite domain constraint programming, but we do not consider these in the present paper.

In first-order predicate logic, a language is defined by a *signature* $\Sigma = (F, P)$, where F is a set of function symbols and P is a set of predicate symbols with given arities. Function symbols of arity 0 correspond to constants. Furthermore, we need a countably infinite set $V = \{x, y, z, x_1, x_2, \dots\}$ of variable symbols. A *term* t is built from function and variable symbols in the usual way, i.e. a variable or a constant symbol is a term, and if f is an n-ary function symbol and $t_1, \dots, t_n, n \geq 1$, are terms, then $f(t_1, \dots, t_n)$ is a term. The set of all terms over F and V will be denoted by $T(F, V)$. We always assume that F contains the function symbols $0, 1, +, -, \cdot, /$ for the standard arithmetical operations and the list constructors $[\,]$ and $[\cdot|\cdot]$. Here, $[\,]$ stands for the empty list, and $[h|t]$ for a list with head element h and tail t. $[a_1, \dots, a_n]$ is an abbreviation for the list of elements $a_1, \dots, a_n$.

Definition 2.1
A *constraint* is a logical formula of the form $p(t_1, \dots, t_n)$, with an n-ary predicate symbol $p \in P$ and terms $t_1, \dots, t_n \in T(F, V)$. An *arithmetic constraint* is of the form $t_1 \diamond t_2$, with $t_1, t_2 \in T(F, V)$ and $\diamond \in \{=, \leq, \geq, \neq, <, >\}$. An *integrality constraint* is of the form $\texttt{integral}([x_1, \dots, x_n])$, with variables $x_1, \dots, x_n \in V$. All other constraints are called *symbolic*. The *constraint language* associated with a signature Σ is the union

$$L = A \cup I \cup S$$

of the set A of all arithmetic, the set I of all integrality, and the set S of all symbolic constraints. For a constraint set $C \subseteq L$, we denote by $Var(C)$ the set of all variables occurring in some constraint of C.

The constraint language of ILP. Let $\Sigma_{ILP} = (F_{ILP}, P_{ILP})$ be defined by

$$F_{ILP} = \{0, 1, +, -, \cdot, /, [\,], [\cdot|\cdot]\} \text{ and } P_{ILP} = \{\leq, \geq, =, \texttt{integral}\}.$$

The constraint language L_{ILP} of *integer linear programming* is given by

- $A_{ILP} = \{\sum_{i \in I} a_i x_i \leq b \mid a_i, b \in \mathbb{Q}, x_i \in V\} \cup \{\sum_{i \in I} a_i x_i = b \mid a_i, b \in \mathbb{Q}, x_i \in V\}$
- $I_{ILP} = \{\texttt{integral}([x_1, \ldots, x_n]) \mid x_i \in V\}$
- $S_{ILP} = \emptyset$.

This means that we have only linear equations and inequalities, and the `integral` constraint. There are no symbolic constraints.

The constraint language of CP(FD). Consider the signature $\Sigma_{FD} = (F_{FD}, P_{FD})$, where

$$F_{FD} = \{0, 1, +, -, \cdot, /, [\,], [\cdot|\cdot]\} \text{ and } P_{FD} = \{\leq, \geq, =, \neq, >, <, \texttt{integral}, \texttt{alldifferent}\}.$$

A mini constraint language L_{FD} for *finite domain constraint programming* is given by

- $A_{FD} = \{\sum_{i \in I} a_i x_i \diamond b,\ x_i \diamond x_j \mid a_i, b \in \mathbb{Q},\ x_i, x_j \in V,\ \diamond \in \{\leq, \geq, =, \neq, >, <\}\}$
- $I_{FD} = \{\texttt{integral}([x_1, \ldots, x_n]) \mid x_i \in V\}$
- $S_{FD} = \{\texttt{alldifferent}([x_1, \ldots, x_n]) \mid x_i \in V\}$.

The main difference to L_{ILP} is the presence of symbolic constraints. The mini constraint language L_{FD} contains only one symbolic constraint, $\texttt{alldifferent}([x_1, \ldots, x_n])$, which intuitively says that the variables $x_1, \ldots, x_n$ should take different values. In traditional integer programming, a quadratic number of constraints would be needed to express this condition. More realistic finite domain constraint languages will contain various other symbolic constraints. For example, the constraint logic programming language CHIP provides a number of so-called global constraints, e.g. `cumulative`, to express cumulative resource limits over a time period (cf. Example 2.4), `diffn`, for non-overlapping of n-dimensional rectangles, `cycle`, for the number of cycles in a directed graph, or `among` and `sequence`, for various constraints on sequences of finite domain variables (see [1, 11, 10] for more details).

After introducing the syntax, we next give the declarative semantics of our formulas. This is done by interpreting all symbols over the rational numbers. An n-ary function symbol $f \in F$ corresponds to a function $f : \mathbb{Q}^n \to \mathbb{Q}$, an n-ary predicate symbol $p \in P$ to a relation $p \subseteq \mathbb{Q}^n$. Variables are interpreted using an *assignment* $\alpha : V \longrightarrow \mathbb{Q}$, which can be naturally extended to an interpretation of terms $\alpha : T(F,V) \to \mathbb{Q}$. We say that a constraint $p(t_1,\dots,t_n)$ is *valid* or *true* under the assignment α, if $p(\alpha(t_1),\dots,\alpha(t_n))$ holds in $\mathbb{Q}^n$. In particular, an arithmetic constraint $t_1 \diamond t_2$ is true (under α) if $\alpha(t_1) \diamond \alpha(t_2)$ holds in the rational numbers, where $\diamond \in \{=,\leq,\geq,\neq,<,>\}$. The constraint `integral`$([x_1,\dots,x_n])$ holds if $\alpha(x_1),\dots,\alpha(x_n)$ are integer numbers. The constraint `alldifferent`$([x_1,\dots,x_n])$ holds if $\alpha(x_i) \neq \alpha(x_j)$, for all $1 \leq i < j \leq n$. A constraint set C is *satisfiable* or *feasible* if there exists an assignment $\alpha : V \to \mathbb{Q}$ such that all constraints in C become true under α, otherwise it is called *infeasible*. If $Var(C) = \{x_1,\dots,x_n\}$, we call the vector $(\alpha(x_1),\dots,\alpha(x_n)) \in \mathbb{Q}^n$ a *solution* of C. The set of all solutions will be denoted by $sol(C)$. Given two constraint sets C, C' we say that C *entails* C', and write $C \to C'$, if all assignments satisfying C also satisfy C'.

Definition 2.2
Let L be a constraint language. A *combinatorial problem* is given by a finite set $C \subseteq L$ of constraints such that for each variable $x \in \mathrm{Var}(C)$ the set C contains an integrality constraint `integral`$([\dots,x,\dots])$ and a lower and upper bound constraint $x \geq l, x \leq u$, with $l,u \in \mathbb{Z}$. Logically, a combinatorial problem C corresponds to the conjunction of the constraints in C.

Although the constraints are interpreted over the rational numbers, the integrality and bound constraints guarantee that the set of values that a variable can take is always a *finite domain*, i.e. a finite set of integer numbers.

Definition 2.3
Let C be a combinatorial problem and $f : \mathbb{Q}^n \longrightarrow \mathbb{Q}$ a function. We call a problem

$$\mathrm{opt}\{f(x) \mid x \in sol(C)\}, \text{ with } \mathrm{opt} \in \{\max,\min\}$$

a *combinatorial optimization problem*. A solution $x \in sol(C)$ is *optimal* if

$$f(x) \diamond f(y), \text{ for all } y \in sol(C),$$

with $\diamond \in \{\geq,\leq\}$ depending on whether we are in a maximization or in a minimization context. For the rest of the paper, we will assume without loss of generality that combinatorial optimization problems are maximization problems.

Example 2.4
We illustrate the two constraint languages on a small example. The problem is to pack 6 different chemicals into bins, such that the number of bins becomes minimal. The chemicals arise in the following quantities:

Chemicals	E_1	E_2	E_3	E_4	E_5	E_6
Quantities	3	2	1	5	3	4

All bins have the capacity 5. For reasons of security, the chemicals E_1, E_2 and E_3 have to be packed into different bins. Since we have 6 chemicals that arise in quantities less than or equal to 5, we need at most 6 bins.

In our CP(FD) model we use the `alldifferent` constraint and a simple form of the `cumulative` constraint [1]:

`cumulative(Start, Duration, Resource, Limit, End)`

- `Start` is a list $[S_1, \dots, S_n]$ of variables or natural numbers.
- `Duration` is a list $[D_1, \dots, D_n]$ of variables or natural numbers.
- `Resource` is a list $[R_1, \dots, R_n]$ of variables or natural numbers.
- `Limit` is a natural number.
- `End` is a variable or natural number.

The main application of this constraint is in scheduling, where the variables S_i denote the starting time, D_i the duration, and R_i the resource consumption of a task i, for $i = 1, \dots, n$. `End` denotes the total end of the schedule. The constraint expresses that at any time point t the total number of resources required by the tasks does not exceed the given limit, i.e. $\sum_{\{i | S_i \le t \le S_i + D_i - 1\}} R_i \le \texttt{Limit}$.

For bin packing, the `cumulative` constraint can be used in the following way. We introduce for each chemical a variable E_i that can take a value from $\{1, \dots, 6\}$, which corresponds to the bin it is assigned to. For the number of bins we use a further variable B that can also take a value between 1 and 6. Representing the bin packing is now done as follows. We use the variables $E_1, \dots, E_6$ as the starting time variables. Each time point represents a bin. Thus assigning E_1 the value 2 means to pack E_1 into the second bin. A duration of 1 for all the E_i ensures that there will be no overlap between different bins. The quantity of the different chemicals is represented by the resources, and the bin capacity by the total resource limit, which we choose to be 5. Viewed as a schedule, the minimal completion time is exactly the minimal number of bins required. The security requirement is modeled by an `alldifferent` constraint on the variables E_1, E_2 and E_3.

$$
\begin{array}{ll}
\min & B \\
\text{s.t.} & \texttt{cumulative}([E_1, E_2, E_3, E_4, E_5, E_6], [1,1,1,1,1,1], [3,2,1,5,3,4], 5, B), \\
& \texttt{alldifferent}([E_1, E_2, E_3]), \\
& 1 \le E_i \le 6, \quad i \in \{1, \dots, 6\} \\
& 1 \le B \le 6, \\
& \texttt{integral}([E_1, E_2, E_3, E_4, E_5, E_6, B]).
\end{array}
$$

Note that the estimate on the number of bins does not affect the size of the model if we use a `cumulative` constraint. It is only required for the upper bound of the variable B.

In integer programming, the estimate on the number of bins has much more influence on the size of the model. Here we use 0-1 variables x_{ij} indicating that chemical i is packed to bin j, and variables y_j indicating that bin j is used in the packing.

$$
\begin{array}{ll}
\min & \sum_{j=1}^{6} y_j \\
\text{s.t.} & \sum_{j=1}^{6} x_{ij} = 1, \quad i \in \{1, \dots, 6\} \\
& 3x_{1j} + 2x_{2j} + 1x_{3j} + 5x_{4j} + 3x_{5j} + 4x_{6j} \le 5y_j, \quad j \in \{1, \dots, 6\} \\
& x_{1j} + x_{2j} + x_{3j} \le 1, \quad j \in \{1, \dots, 6\} \\
& 0 \le x_{ij} \le 1, \quad i \in \{1, \dots, 6\}, j \in \{1, \dots, 6\} \\
& 0 \le y_j \le 1, \quad j \in \{1, \dots, 6\} \\
& \texttt{integral}([x_{11}, \dots, x_{66}, y_1, \dots, y_6]).
\end{array}
$$

While the integer model uses 42 0-1 variables and 18 arithmetic constraints (without bounds and integrality constraint), the CP(FD) model needs only 7 FD variables and 2 symbolic constraints and thus is much smaller. This is a general observation. Due to the symbolic constraints, CP(FD) models are often much more compact than corresponding ILP models.

3 Solving combinatorial problems by branch-and-infer

After having compared the constraint languages of ILP and CP(FD) from the declarative point of view, we now come to their operational semantics. We develop a general framework, branch-and-infer, that unifies the classical branch-and-cut approach from integer linear programming [28] with the usual operational semantics of finite domain constraint programming [36].

3.1 Primitive and non-primitive constraints

We start from a common distinction in finite domain constraint programming [36] and split the constraint language L into a set $\mathrm{Prim}(L)$ of *primitive* constraints and a set $\mathrm{NPrim}(L)$ of *non-primitive* constraints, such that

$$L = \mathrm{Prim}(L) \cup \mathrm{NPrim}(L) \quad \text{and} \quad \mathrm{Prim}(L) \cap \mathrm{NPrim}(L) = \emptyset.$$

Intuitively, the primitive constraints are those constraints that can be easily solved. In other words, we always assume that for a set of primitive constraints there exist efficient, i.e. at least polynomial, methods for satisfiability, entailment, and optimization. The non-primitive constraints are the difficult constraints, for which such methods do not exist (in conjunction with a set of primitive constraints). Adding non-primitive constraints to a problem makes it hard to solve.

Primitive constraints in ILP. Given the constraint language L_{ILP} of integer linear programming, we define

- $\mathrm{Prim}(L_{ILP}) = A_{ILP}$ and
- $\mathrm{NPrim}(L_{ILP}) = I$.

This means that the primitive constraints are linear equations and inequalities over $\mathbb{Q}$, while the only non-primitive constraint is integrality.

Primitive constraints in CP(FD). For the mini constraint language L_{FD} of finite domain constraint programming, we may choose

- $\mathrm{Prim}(L_{FD}) = \{x \leq u, x \geq l, x \neq v, x = y, \mid x, y \in V, l, u, v \in \mathbb{Z}\} \cup I$
- $\mathrm{NPrim}(L_{FD}) = L_{FD} \setminus \mathrm{Prim}(L_{FD})$.

On the one hand, we have only very simple equations and inequalities, where the left-hand side and the right-hand side is either a variable or a constant. On the other hand, we also admit certain disequalities. Moreover, `integral`$([x_1, \ldots, x_n])$ is also primitive now. Therefore, in finite domain constraint programming, the primitive constraints of a combinatorial problem will be solved over the integers and not over the rationals.

From the viewpoint of integer programming, the set of primitive constraints $\mathrm{Prim}(C)$ of a combinatorial problem C defines a *relaxation* of the problem, i.e. a constraint set $\mathrm{rel}(C)$ such that $C \to \mathrm{rel}(C)$. We say that a relaxation $\mathrm{rel}(C)$ is *stronger* than a relaxation $\mathrm{rel}'(C)$, if $\mathrm{rel}(C) \to \mathrm{rel}'(C)$, and *strictly* stronger, if moreover $\mathrm{rel}'(C) \not\to \mathrm{rel}(C)$. Primitive constraints are in general not powerful enough to express a combinatorial problem. This can be caused by their limited expressivity or, even if their expressivity is sufficient, by the fact that a representation of the problem in terms of primitive constraints is not known. In finite domain constraint programming, the primitive constraints are not expressive enough to describe, e.g., the set $\{(1,0),(0,1),(1,1),(2,1),(1,2)\} \subseteq \mathbb{Q}^2$ or the set $\{(0,2),(1,1),(2,0)\} \subseteq \mathbb{Q}^2$ (see Fig. 1). In integer linear programming, it is theoretically always possible to describe the convex hull of the integer solution set by a system of facet-defining inequalities, but for most practical problems, such a representation is not known.

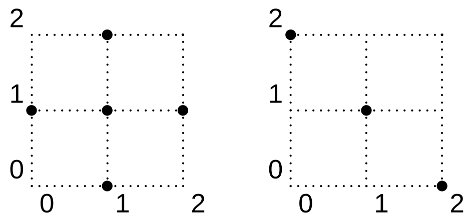

Figure 1: Geometric illustration of the point sets

3.2 Inferring primitive from non-primitive constraints

In general, a combinatorial problem contains both primitive and non-primitive constraints. Since an efficient constraint solver is available only for the primitive constraints, the basic idea is to reduce non-primitive constraints to primitive ones. However, as we have seen before, a complete reduction is in general not possible, i.e. we cannot just replace a non-primitive constraint by an equivalent set of primitive constraints. The only thing that we can do, is a partial reduction, i.e. we can infer from the given primitive constraints and the non-primitive constraint new primitive constraints. In the ideal case, we can derive sufficiently many new primitive constraints so that by solving the strengthened set of primitive constraints, we obtain a solution of the original problem.

This has two consequences. The first one is that each non-primitive constraint does not only have a declarative semantics but also an operational semantics, determining how primitive constraints can be inferred during the solution process. The second consequence is that the inference process can stop, but still a solution of the primitive constraints is not feasible for the whole problem. Thus we need a second technique in order to get a complete solver. This is branching, which will be discussed in Section 3.3.

We now describe the computational setup for handling the inference process [30, 33]. It consists of a *constraint store* that contains the current set of primitive constraints and a number of *inference agents* that are connected to the store, one for each non-primitive constraint (see Fig. 2). We require that the constraint store, i.e. the set of primitive

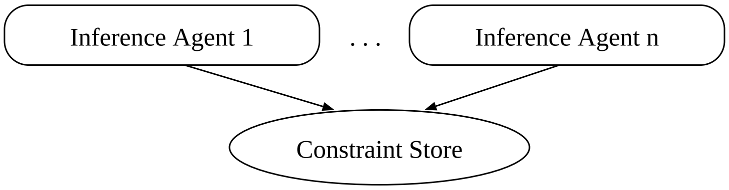

Figure 2: Architecture for constraint-based solving

constraints, is always satisfiable and that we can efficiently compute a feasible solution. For each non-primitive constraint c, the corresponding inference agent tries to infer new primitive constraints p that follow from c and the constraint store.

To describe our branch-and-infer approach in a formal way, we will use *transition rules* of the form

$$\texttt{\_\_:}\quad \frac{\langle P, S\rangle}{\langle P', S'\rangle} \text{ if } \mathit{Cond}$$

saying that from a computation state $\langle P, S\rangle$ we may proceed to a computation state $\langle P', S'\rangle$ if the conditions in *Cond* are satisfied. Here $P = \{C_1, \dots, C_m\}$ (resp. P') denotes a set of combinatorial (sub-)problems $C_1, \dots, C_m$, which logically corresponds to the disjunction $C_1 \vee \cdots \vee C_m$. The set S (resp. S') denotes a set of feasible solutions. For any set T and any element t, we will write $t \uplus T$ instead of $\{t\} \cup T$. When solving a combinatorial problem C, the initial state is $\langle\{C\}, \emptyset\rangle$, and the final state is $\langle\emptyset, \{S\}\rangle$. If $S = \emptyset$, then the problem is infeasible. Transition rules are a standard tool in computational logic. They allow us to separate the *logic* of the constraint solving process from the *control*, i.e. the actual use of the rules, which in general will depend on the implementation.

The operational behaviour of the non-primitive constraints is formalized by the rule

$$\texttt{bi\_infer:}\quad \frac{\langle (c \uplus C) \uplus P, S\rangle}{\langle (p \uplus (c \uplus C)) \uplus P, S\rangle} \text{ if } \begin{array}{c} c \text{ is non-primitive,} \\ p \text{ is primitive,} \\ \mathrm{Prim}(C) \wedge c \rightarrow p, \\ \mathrm{Prim}(C) \not\rightarrow p. \end{array}$$

We say the inference process becomes *stable* if the operational semantics of all the inference agents connected to the store cannot infer more primitive constraints in order to strengthen the relaxation.

Communication through the constraint store. If more than one non-primitive constraint is present, then the different inference agents can communicate with each other through the constraint store, i.e. the set of primitive constraints. This communication comes for free in the constraint-based computation model because each non-primitive constraint can use all the primitive constraints in the store as input for its inference algorithm. In general, communication can happen by exchanging primitive constraints between the different inference algorithms, but also by extracting some global information reflecting the interaction of all the primitive constraints in the store, e.g., by optimizing some objective function. Due to the communication through the store, we can implement inference algorithms independently from each other and combine them freely.

Inference in ILP. Consider the constraint language L_{ILP} of integer linear programming with the primitive and non-primitive constraints defined before. Given a combinatorial problem C, the relaxation obtained by the primitive constraints in C is the standard *linear programming relaxation* of C. Inferring a new primitive constraint corresponds to the generation of a *cutting plane* that cuts off some part of this relaxation. Cutting plane generation has a long history in integer programming. Two fundamental principles for cutting plane generation of general integer programs are the Chvátal-Gomory method and the disjunctive method [28, 3, 4].

Inference in FD. In finite domain constraint programming, the basic inference principle is *domain reduction*. The corresponding *local consistency* techniques have been studied in artificial intelligence for a long time [35]. For each non-primitive constraint, so-called *propagation* algorithms try to remove inconsistent values from the domain of the variables occurring in the constraint. From a logical point of view, this corresponds to the generation of a new primitive constraint of the form $x \leq u, x \geq l$, which is called *bound reasoning*, or $x \neq v$, which is called *domain reasoning*. Whenever the domain of a variable changes, all propagation algorithms of the constraints in which this variable occurs may become active and further reduce the domains of their variables.

In general, on the class of linear equations and inequalities, the propagation algorithms of finite domain constraint programming cannot compete with linear programming techniques. The reason is that the arithmetic constraints are primitive in ILP, whereas they are non-primitive in CP(FD). This means that in CP(FD) each arithmetic constraint is handled individually, while in ILP all the arithmetic constraints are solved together.

3.3 Branching

As we have mentioned before, the reduction of non-primitive constraints to primitive constraints is in general not complete, either because the primitive constraints are not expressive enough or because the complete reduction is computationally not feasible. Therefore, we need a second technique that enforces further strengthening of the relaxation if the inference process on a problem has become stable.

This can be achieved by splitting the problem into subproblems and to process each subproblem independently of the others. Subproblems are obtained by setting up branching constraints and adding to each of them one copy of the problem under consideration.

If the branching constraints are chosen in the right way, the relaxation of a subproblem will be strictly stronger than the relaxation of the father problem. Therefore, the inference agents associated with the non-primitive constraints may become active again and derive new primitive constraints.

The branching operation is described by the rule

$$\texttt{bi\_branch:} \quad \frac{\langle C \uplus P, S\rangle}{\langle \{c_1 \uplus C, \ldots, c_k \uplus C\} \cup P, S\rangle} \quad \text{if} \quad \begin{array}{l} C \equiv C \wedge (\bigvee_{i=1}^{k} c_i) \\ c_i \text{ primitive} \\ \text{Prim}(C) \not\rightarrow c_i, i = 1, \ldots, k. \end{array}$$

The constraints $c_1, \ldots, c_k$ are called *branching constraints*, the problems $\{c_1 \uplus C, \ldots, c_k \uplus C\}$ are the new *subproblems*. Logically, C is equivalent to the disjunction $(C \wedge c_1) \vee \ldots \vee (C \wedge c_k)$, which we denote by $C \equiv C \wedge (\bigvee_{i=1}^{k} c_i)$. In many applications, we will have a binary branching of the form $c_1 \equiv c, c_2 \equiv \neg c$. By repeated application of the branching rule we build up a *search tree*. Eventually, we will get a complete enumeration of all the solutions in $sol(C)$. However, this is computationally feasible only for problems with a very small number of variables.

In practice, the division into subproblems has to be avoided as much as possible. Splitting can be avoided if we know that the (sub-)problem is infeasible. Since deciding the satisfiability of the whole problem is computationally not feasible, we test feasibility only on the primitive constraints, i.e. the relaxation. The next rule describes pruning by the infeasibility of the relaxation, which is denoted by $\perp$.

$$\texttt{bi\_clash:} \quad \frac{\langle C \uplus P, S\rangle}{\langle P, S\rangle} \quad \text{if} \quad \text{Prim}(C) \rightarrow \perp$$

The relaxation plays a crucial role in the branch-and-infer approach. The primitive constraints do not only allow for the communication between different non-primitive constraints, they also link the branch and the infer component. Applying the rule `bi_infer` strengthens the relaxation and thus it becomes more likely that the rule `bi_clash` can be applied. On the other hand, applying `bi_branch` imposes new primitive constraints on the subproblems that may induce further applications of `bi_infer`. Thus, branching and inference work hand in hand in order to solve the problem more efficiently. This will become even more important when solving optimization problems by branch-and-relax resp. branch-and-cut (see Sect. 3.3.2).

The transition rules `bi_infer`, `bi_branch` and `bi_clash` are the basic rules in our branch-and-infer framework. What is still missing are rules that describe when a solution has been obtained. This depends on the type of problem to be solved, i.e. whether we want to find feasible or optimal solutions.

3.3.1 Solving combinatorial problems

Solving a combinatorial problem means deciding whether the problem is satisfiable and if so computing one or more feasible solutions. We will hide the concrete method of computing feasible solutions from the relaxation and the way they are represented in a function extract. This function has to be chosen properly according to whether one wants to compute only one solution or more. For example, if one wants to compute all solutions, one can return the relaxation in a solved form if all non-primitive constraints are entailed. If one is interested in only one solution, then extract can give only one variable assignment. The function extract can be used to derive a feasible solution of the problem even when the non-primitive constraints have not been completely reduced to primitive constraints. All these cases are captured by the rule

$$\texttt{bi\_sol:} \quad \frac{\langle C \uplus P, S\rangle}{\langle P, S \cup S^*\rangle} \ \text{if} \ \begin{array}{l} S^* = \texttt{extract}(\mathrm{Prim}(C)) \\ S^* \rightarrow C. \end{array}$$

If one wants to compute more or all solutions, then repeated application of this rule to the different subproblems will collect the different solution families. Thus the task of finding more solutions is left to the control strategy for the application of the different transition rules.

One might think that integer linear programming cannot be used for satisfiability since it is usually applied in an optimization context. But notice that on the one hand we can simply take an empty objective function. Then linear programming may give us a feasible solution of the relaxation. On the other hand, the user has often an intuition on where feasible solutions may be. Therefore he might set up his own objective function, which can help to direct the search into the neighborhood of a feasible solution. This leads us to optimization problems, which we consider now.

3.3.2 Solving combinatorial optimization problems

In many applications, one would like to compute a feasible solution of a constraint set that is optimal with respect to some objective function. Consider a maximization problem

$$\max\{f(x) \mid x \in sol(C)\}.$$

Note that feasible solutions of $sol(C)$ yield lower bounds for the maximum value of f. To solve optimization problems, there exist two general methods, branch-and-bound, as it is used in finite domain constraint programming, and branch-and-relax resp. branch-and-cut, which are standard techniques in integer linear programming. Note that we follow here the terminology of constraint programming, where branch-and-relax corresponds to what is usually called branch-and-bound in integer linear programming. We now formalize the different approaches in our framework.

Branch-and-Bound. The *branch-and-bound* method is characterized by using only lower bounds to find an optimal solution. Thus we solve a sequence of satisfiability problems leading successively to better solutions. More precisely, we repeatedly compute a feasible solution $s^* \in sol(C)$ and then add the constraint $f(x) \geq f(s^*) + 1$ to all the subproblems of our search tree, which restricts the set of feasible solutions to those that yield better objective function values. The constraint $f(x) \geq f(s^*) + 1$ is called *lower bounding constraint*. If after adding a lower bounding constraint, all the subproblems become infeasible, then the last feasible solution is optimal. We require that f takes always integral values if x is integral since otherwise feasible solutions may be lost and the global optimum cannot be found.

To describe the lower bounding procedure, we extend the inference system consisting of the rules `bi_branch`, `bi_clash` by the rule

$$\texttt{bi\_climb:} \quad \frac{\langle \{C, C_1, \ldots, C_n\}, \{s\} \rangle}{\langle \{c \uplus C, c \uplus C_1, \ldots, c \uplus C_n\}, \{s^*\} \rangle} \quad \text{if} \quad \begin{array}{l} s^* = \texttt{extract}(\mathrm{Prim}(C)) \\ f(s^*) > f(s) \\ c \equiv (f(x) \geq f(s^*) + 1). \end{array}$$

Here, the function `extract` is again responsible for computing a feasible solution of the relaxation. If no feasible solution is known, we assume $f(s) = -\infty$.

Branch-and-Relax. In contrast to branch-and-bound, which uses only lower bounds, *branch-and-relax* works with two bounds. In addition to the global lower bound *glb* obtained from a feasible solution, we compute for each subproblem a local upper bound *lub*. For example, this be can be done by optimizing the objective function subject to the relaxation of a subproblem, i.e. the primitive constraints in the constraint store. We describe branch-and-relax again by an extension of the transition system given by the rules `bi_clash`, `bi_branch`. The local upper bounds allow us to introduce a new rule to prune the search tree. If a local upper bound is smaller than the best known global lower bound, then the corresponding subproblem cannot lead to a better solution and therefore can be discarded.

$$\texttt{bi\_bound:} \quad \frac{\langle C \uplus P, \{s\} \rangle}{\langle P, \{s\} \rangle} \quad \text{if} \quad \max\{f(x) \mid x \in sol(C)\} \leq lub \leq f(s)$$

Furthermore, when computing a local upper bound, we may find an optimal solution of a subproblem that yields a better feasible solution of the whole problem.

$$\texttt{bi\_opt:} \quad \frac{\langle C \uplus P, \{s\} \rangle}{\langle P, \{s^*\} \rangle} \quad \text{if} \quad \begin{array}{l} \max\{f(x) \mid x \in sol(\mathrm{Prim}(C))\} = f(s^*) \\ s^* \in sol(C) \\ f(s^*) > f(s) \end{array}$$

Branch-and-relax is obtained from branch-and-bound by replacing the rule `bi_climb` with the two rules `bi_bound` and `bi_opt`.

To apply branch-and-relax in practice, we must be able to compute local upper bounds in a computationally feasible way. For example, this is possible in integer linear programming, where we can obtain an upper bound by solving the linear programming relaxation. We may even find a feasible solution of the whole problem, so that both rules `bi_bound` and `bi_opt` possibly can be applied.

3.4 Branch-and-Infer

To summarize, the rule system for *branch-and-infer* consists of the rules

`bi_infer, bi_branch, bi_clash`

together with one of the following three alternatives:

- `bi_sol` (satisfiability)
- `bi_climb` (branch-and-bound)
- `bi_bound, bi_opt` (branch-and-relax)

Combinatorial problems in CP(FD). The rule set of finite domain constraint programming for solving combinatorial problems is given by

$$\texttt{FD\_SAT} = \{\ \texttt{bi\_infer, bi\_branch, bi\_clash, bi\_sol}\ \}.$$

Combinatorial optimization problems in CP(FD). The rule set of finite domain constraint programming for solving combinatorial optimization problems is given by

$$\texttt{FD\_OPT} = \{\ \texttt{bi\_infer, bi\_branch, bi\_clash, bi\_climb}\ \}.$$

Combinatorial optimization problems in ILP – Branch-and-Cut. Solving combinatorial optimization problems in integer linear programming by *branch-and-cut* is described by the rule set

$$\texttt{ILP\_OPT} = \{\ \texttt{bi\_infer, bi\_branch, bi\_clash, bi\_bound, bi\_opt}\ \}.$$

Here, the last four rules describe branch-and-relax, while the first rule `bi_infer` allows for the generation of cutting planes.

In branch-and-bound, the only way to prune the search space is to apply the rule `bi_clash`. Therefore it is of great impact whether the lower bounding constraint is primitive or non-primitive in the underlying solver. If the lower bounding constraint is primitive, then it can be added to the constraint store and thus has a direct effect on the relaxation, i.e. the rule `bi_clash` may be applied earlier in the solution process. If the lower bounding constraint is non-primitive, then it may not be possible to reduce it completely to primitive constraints. Therefore, the relaxation may be not strong enough in order to apply the rule `bi_clash` and more subproblems will be generated.

Example 3.1
Consider the constraint set

$$\begin{aligned} C \;=\; \{ \;& x_1 + x_2 + x_3 \leq 1, \\ & x_1 \leq 1, x_2 \leq 1, x_3 \leq 1, x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, \\ & \texttt{integral}([x_1, x_2, x_3]) \quad \} \end{aligned}$$

and the objective function $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$, which has to be maximized. Assume that the `extract`-function generates the feasible (and optimal) solution $(1, 0, 0)$.

If we use an integer programming based solver and add the lower bounding constraint $x_1 + x_2 + x_3 \geq 2$, then the linear programming relaxation $\mathrm{Prim}_{LP}(C')$ of $C' = C \cup \{x_1 + x_2 + x_3 \geq 2\}$ is infeasible. This can be seen by adding $x_1 + x_2 + x_3 \leq 1$ and $x_1 + x_2 + x_3 \geq 2$ resulting in the contradiction $0 \leq -1$. However, if we use a finite domain constraint solver, then the finite domain relaxation $\mathrm{Prim}_{FD}(C') = \{x_1 \leq 1, x_2 \leq 1, x_3 \leq 1, x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, \mathtt{integral}([x_1, x_2, x_3])\}$ is feasible. In particular, no inference can be made by the non-primitive constraints $x_1 + x_2 + x_3 \leq 1$ and $x_1 + x_2 + x_3 \geq 2$. Thus the relaxation will stay feasible and `bi_clash` cannot be applied. In order to detect the infeasibility of C', it is necessary to split the problem into two more subproblems.

An advantage of branch-and-bound compared to branch-and-relax is that whenever a solver supports a non-primitive constraint of the form $g(x) \geq b$, for some function g, then an optimization problem can be set up that uses g as objective function. Thus it is not required to have an algorithm that can optimize g directly subject to the constraints. A major disadvantage of branch-and-bound is that it does not provide any information on the quality of a feasible solution, because no upper bounds are available.

4 Extending ILP by symbolic constraints

After having developed a common framework for ILP and CP(FD), we now show how the idea of symbolic constraints from CP(FD) can be carried over to ILP. In constraint programming, symbolic constraints have been introduced for two reasons. On the one hand, they extend the constraint language and allow to model many problems in a much more natural and compact way. On the other hand, they allow to incorporate efficient algorithms for a specific problem area into a general solver. A typical example is the `cumulative` constraint (cf. Example 2.4). Many scheduling problems can be modeled very naturally with this constraint. On the operational side, powerful algorithms from operations research, e.g. edge-finding [16], can be used in order to reduce the domain of the variables. Thus, symbolic constraints not only increase the expressive power of the constraint language. They are also crucial for the efficiency of the problem solver.

In integer linear programming, symbolic constraints can play a similar role. On the declarative side, they extend the language of linear equations and inequalities. On the operational side, they allow to integrate specialized cutting plane algorithms based on polyhedral combinatorics into a general solver.

4.1 Symbolic constraints in ILP

A first way of using symbolic constraints in integer programming is when a problem is defined by a set of linear inequalities that is too large to be represented in the solver. A typical example is the traveling salesman problem (TSP) with its exponentially many (in the number of cities) subtour elimination constraints [25]. To handle these constraints, we can extend our constraint language by a symbolic `tsp` constraint, e.g.

```
tsp(Adjacencies, Weights)
```

- `Adjacencies`: A list of 0-1 variables $[x_{12}, \ldots, x_{(n-1)n}]$
- `Weights`: A list of non-negative rational numbers $[w_{12}, \ldots, w_{(n-1)n}]$.

The adjacency of two nodes i and j in the graph is represented by a variable $x_{ij}, i < j$, which has value 1 if the edge is used in a tour and 0 otherwise. The weights w_{ij} represent the cost imposed by using the edge between the nodes i and j in a tour.

From the declarative point of view, hiding the exponentially many primitive constraints inside a new symbolic constraint gives us a clear and concise modeling. The key feature of such symbolic constraints, however, comes from their operational semantics. In branch-and-infer, the non-primitive `tsp` constraint will be realized by an inference algorithm for primitive constraints, i.e., a separation algorithm for the problem-defining inequalities. Thus, the symbolic constraint is not just an abbreviation for a huge number of constraints. The associated inference agent will infer only selected inequalities that improve the current formulation. In the traveling salesman problem, these are separators for the degree constraints and subtour elimination constraints. The efficiency of solving such constraints can be drastically increased, if not only separators for the problem-defining inequality classes are built into the symbolic constraint, but also separators for other classes of strong valid inequalities, e.g. facet-defining inequalities for the convex hull of feasible solutions. For the `tsp` constraint, we could add for example separators for comb-inequalities. Problem specific branch-and-cut algorithms have been extremely successful in solving hard combinatorial optimization problems. The concept of symbolic constraints allows us to embed these techniques into the constraint language of a *general* constraint solver.

Next we show how a symbolic constraint can be used in order to increase the expressivity of the constraint language. For example, we can introduce a symbolic constraint for handling non-linear 0-1 inequalities

$$\sum_{I \subseteq \{1,\ldots,n\}} a_I \prod_{i \in I} x_i \leq b, \quad a_I, b \in \mathbb{Q}, \quad x_i \in \{0,1\}. \tag{1}$$

Theoretically, there exists an equivalent set of linear inequalities with the same set of 0-1 solutions, practically however, such a linear inequality description is often not known. Instead of linearizing the non-linear constraint completely at the beginning, the idea is again to represent it by a non-primitive constraint and to linearize it partially during the constraint solving process, by inferring linear inequalities only if they improve the current relaxation in the constraint store. Different linearization procedures have been proposed in the literature, see for example [5, 6]. A method in the spirit of constraint programming has been developed in [7], which takes into account the constraints in the store during the linearization process.

Different non-primitive constraints can communicate through the primitive constraints in the store. The symbolic constraint for non-linear inequalities already illustrates one way of communication, where the primitive constraints in the store are used to enhance the linearization. We now describe another way of communication, where different non-primitive constraints cooperate in an extended ILP solver.

Suppose we introduce a symbolic constraint for set packing. Let M be a set and $F = \{M_1, \ldots, M_n\}$ be a family of subsets of M. The problem of set packing consists in selecting a set $P \subseteq F$ such that each element of M is contained in at most one set of P. We model this situation by a symbolic constraint

`setpack(Sets)`

- `Sets`: A list $[[x_1, [el_{11}, \ldots, el_{1k_1}]], \ldots, [x_n, [el_{n1}, \ldots, el_{nk_n}]]]$, where x_i is a 0-1 variable and el_{ij} is a name for the j-th element of subset i.

The variable x_i takes the value 1 if the elements of set i are used in the packing, and 0 otherwise. Now consider the constraint set

$$\begin{array}{l} \texttt{setpack}([x_1,[a]],[x_2,[a,b]],[x_3,[b]]), \\ x_1 + x_3 \leq 1 \ , \ \ldots \ , \ \texttt{integral}([x_1,x_2,x_3]), \end{array}$$

and suppose that the primitive constraint $x_1 + x_3 \leq 1$ has been inferred by some other non-primitive constraint during the constraint solving process. The inference algorithm of `setpack` can now detect that this inequality fits into the structure of the `setpack` constraint and may infer, e.g., the new primitive constraint $x_1 + x_2 + x_3 \leq 1$.

4.2 A symbolic constraint for assignment problems

When introducing a new symbolic constraint, one has always to keep in mind that it has to be both expressive and efficient. On the one hand, a symbolic constraint should be generic enough in order to apply to many problem situations. On the other hand, there must be enough domain-specific knowledge that can be exploited during the inference process, in order to get a more efficient solution of the problem than without the new constraint. Finding the right balance between these two aspects is not an easy task.

To illustrate the idea of symbolic constraint abstractions in ILP, we propose a new symbolic constraint for assignment problems, where the general task is to assign items from one set to locations from another set. The constraint has the following form:

`assign(Assignments, Weights, Capacities, Indicators)`

- `Assignments`: A list of lists $[[x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}]], m, n \geq 1$, of 0-1 variables x_{ij} or values.
- `Weights`: A list of lists $[[w_{11}, \ldots, w_{1n}], \ldots, [w_{m1}, \ldots, w_{mn}]], m, n \geq 1$, of non-negative rational numbers w_{ij}.
- `Capacities`: A list $[c_1, \ldots, c_n]$ of non-negative rational numbers c_j.
- `Indicators`: A list $[y_1, \ldots, y_n]$ of 0-1 variables or values.

Consider a set of items indexed by $M = \{1, \ldots, m\}$ and a set of locations indexed by $N = \{1, \ldots, n\}$. Each item has to be assigned to exactly one location. An assignment of item $i \in M$ to location $j \in N$ is modeled by an assignment variable

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ is assigned to location } j, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

In order to give the `assign` constraint a broader application spectrum, we include location indicator variables $y_j, j \in N$, with the meaning

$$y_j = \begin{cases} 1 & \text{if and only if } \sum_{i=1}^m x_{ij} \geq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The `assign` constraint can be used to express various kinds of problems like generalized assignment, uncapacitated warehouse location, or (one-dimensional) bin packing. It is also possible to use this constraint for the multiple knapsack problem, if we introduce an additional artificial knapsack that has sufficient capacity to take the items that do not fit into the given knapsacks of the problem.

From the declarative point of view, the `assign` constraint is equivalent to the set of constraints

$$\begin{array}{rcll} \sum_{j=1}^n x_{ij} & = & 1, & i \in M \\ \sum_{i=1}^m w_{ij} x_{ij} & \leq & c_j y_j, & j \in N \\ x_{ij}, y_j & \in & \{0,1\}, & i \in M, j \in N. \end{array}$$

The `assign` constraint can be used in a flexible way. We can fix some variables in advance to the value 0 or 1. In combination with the location indicator variables, we can assign items either statically to a given set of locations or use locations dynamically if they satisfy certain side constraints.

On the operational side, the inference algorithms behind the `assign` constraint may exploit various results from polyhedral combinatorics, e.g. general assignment [22, 21], uncapacitated warehouse location [17, 15], bin packing [14] or multiple knapsack [20, 19], in order to infer strong valid inequalities that strengthen the relaxation in the constraint store.

Example 4.1
Consider a warehouse location problem. Suppose we have m clients, indexed by $M = \{1, \ldots, m\}$, and n potential sites for opening a warehouse, indexed by $N = \{1, \ldots, n\}$. The problem consists of selecting a set of locations to open a warehouse and then assigning an open warehouse to each client. We use the `assign` constraint to model this problem. The variable $x_{ij}, i \in M, j \in N$ indicates whether client i is supplied by warehouse j or not. The goal is to minimize the overall costs, which consist of the fixed costs f_j for opening

warehouse j, and the variable costs v_{ij} for supplying client i by warehouse j.

$$\begin{array}{ll} \min & \sum_{i=1}^{m} \sum_{j=1}^{n} v_{ij} x_{ij} + \sum_{j=1}^{n} f_j y_j \\ \text{s.t.} & \texttt{assign}([[x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}]], \\ & \qquad\qquad [[1, \ldots, 1], \ldots, [1, \ldots, 1]], \\ & \qquad\qquad [m, \ldots, m], [y_1, \ldots, y_n]), \\ & 0 \leq x_{ij} \leq 1, \quad 0 \leq y_j \leq 1, \quad i \in M, j \in N \\ & \texttt{integral}([x_{11}, \ldots, x_{mn}, y_1, \ldots, y_n]). \end{array}$$

Suppose now that there is the additional constraint that some customer p wants to be supplied by the same warehouse as customer r or customer s. Using a non-linear symbolic constraint, we can easily extend the above model to

$$\begin{array}{ll} \min & \sum_{i=1}^{m} \sum_{j=1}^{n} v_{ij} x_{ij} + \sum_{j=1}^{n} f_j y_j \\ \text{s.t.} & \texttt{assign}([[x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}]], \\ & \qquad\qquad [[1, \ldots, 1], \ldots, [1, \ldots, 1]], \\ & \qquad\qquad [m, \ldots, m], [y_1, \ldots, y_n]), \\ & \sum_{t=1}^{n} x_{pt} x_{rt} + x_{pt} x_{st} = 1, \\ & 0 \leq x_{ij} \leq 1, \quad 0 \leq y_j \leq 1, \quad i \in M, j \in N \\ & \texttt{integral}([x_{11}, \ldots, x_{mn}, y_1, \ldots, y_n]). \end{array}$$

Although this model contains two non-primitive constraints, a feasible solution of the whole problem can be computed due to the fact that the two non-primitive constraints communicate over the constraint store by inferring new primitive constraints, i.e. cutting planes.

5 Combining finite domain and ILP techniques

In this section, we present different ways for combining methods from ILP and CP(FD). The basic idea is to handle linear equations and inequalities altogether as in ILP, and not individually as in CP(FD). There exist various possibilities for a combination, which range from using linear programming techniques inside the inference algorithms of non-primitive constraints up to extending the language of primitive constraints in CP(FD) by general linear inequalities. Our aim here is only to show how these alternatives follow naturally from our framework. We do not want to discuss their realization, which is a topic of further research.

5.1 Handling linear inequalities by a symbolic constraint

In a first step, we discuss an integration that leaves the primitive constraints of the finite domain language L_{FD} unchanged. We introduce a new non-primitive constraint `linear` that collects all the linear inequalities occurring in the problem and uses them to derive stronger primitive constraints in $\mathrm{Prim}(L_{FD}^1)$.

`linear(Matrix, Variables, RightHandSide)`

- `Matrix`: A list of lists $[[a_{11}, \ldots, a_{1n}], \ldots, [a_{m1}, \ldots, a_{mn}]], m, n \geq 1$, of rational numbers a_{ij}.
- `Variables`: A list $[x_1, \ldots, x_n]$ of variables x_j.
- `RightHandSide`: A list $[b_1, \ldots, b_m]$ of rational numbers b_i.

specifying the system of linear inequalities $\sum_{j=1}^{n} a_{ij}x_j \leq b_i, i = 1, \ldots, m$.

The *extended finite domain constraint language* $L^1_{FD/LP}$ for handling linear arithmetic by a symbolic constraint is defined as follows:

- $L^1_{FD/LP} = L_{FD} \cup \{\texttt{linear}([[a_{11}, \ldots, a_{1n}], \ldots, [a_{m1}, \ldots, a_{mn}]], [x_1, \ldots, x_n], [b_1, \ldots, b_m]) \mid a_{ij}, b_i \in \mathbb{Q}, x_j \in V\}$
- $\text{Prim}(L^1_{FD/LP}) = \text{Prim}(L_{FD})$
- $\text{NPrim}(L^1_{FD/LP}) = \text{NPrim}(L_{FD}) \cup \{\texttt{linear}([[a_{11}, \ldots, a_{1n}], \ldots, [a_{m1}, \ldots, a_{mn}]], [x_1, \ldots, x_n], [b_1, \ldots, b_m]) \mid a_{ij}, b_i \in \mathbb{Q}, x_j \in V\}$

Note that the integrality constraint is still primitive. The transition rules in the branch-and-infer framework are the same as for standard finite domain constraint programming.

In addition to the usual bound propagation on each inequality, the new non-primitive constraint `linear` allows to apply linear programming techniques on the whole system of inequalities. By taking into account the bound constraints in the store, which are partly inferred by other non-primitive constraints, linear programming can exploit the interaction between all inequalities in order to infer stronger bounds or even to fix a variable to some value.

Example 5.1
Consider the constraint set

$$\begin{aligned} C = \{ \quad & -3x_1 + 2x_2 \leq 0, 3x_1 + 2x_2 \leq 6, \\ & x_1 \leq 2, x_2 \leq 2, x_1 \geq 0, x_2 \geq 0, \texttt{integral}([x_1, x_2]) \quad \}. \end{aligned}$$

Simple bound propagation treating each inequality independently of the others cannot detect that the greatest integral value of x_2 is 1. Now we model the same problem with the new constraint:

$$\begin{aligned} C' = \{ \quad & \texttt{linear}([[-3, 2], [3, 2,]], [x_1, x_2], [0, 6]]), \\ & x_1 \leq 2, x_2 \leq 2, x_1 \geq 0, x_2 \geq 0, \texttt{integral}([x_1, x_2]) \quad \}. \end{aligned}$$

The inference algorithm of `linear` detects (for example by maximizing x_2 subject to the linear inequalities over the rational numbers) that the upper bound of x_2 is 1.5 and thus can be reduced to 1. Therefore we can infer the primitive constraint $x_1 \leq 1$ and add it to the constraint store.

The use of linear programming for improving bounds is discussed in [34]. In [12, 2], linear programming is used to detect fixed variables. Linear programming can also check global consistency over the rational numbers, which can help to detect infeasibility earlier than by local consistency methods.

A main disadvantage of this form of integration is that the linear inequalities are hidden inside a non-primitive constraint. Therefore, they are not visible to the other non-primitive constraints and cannot be exploited by their inference algorithms. Furthermore, in a branch-and-bound context, the lower bounding constraint is still non-primitive, which results in a less powerful pruning (cf. Example 3.1).

5.2 Linear inequalities as primitive constraints

To overcome these disadvantages, we propose a second form of integration $L^2_{FD/LP}$. We extend the primitive constraints from CP(FD) by general linear inequalities. Thus the constraint language L_{FD} itself remains unchanged, but the definition of the primitive and non-primitive constraints changes:

- $L^2_{FD/LP} = L_{FD}$
- $\mathrm{Prim}(L^2_{FD/LP}) = \{x \leq u, x \geq l, x \neq v, x = y, \sum_{i=1}^n a_i x_i \leq b, \sum_{i=1}^n a_i x_i = b \mid x_i, x, y \in V, l, u, v \in \mathbb{Z}, a_i, b \in \mathbb{Q}\}$
- $\mathrm{NPrim}(L^2_{FD/LP}) = I \cup S_{FD}$

Since general linear inequalities are primitive now, we can no longer check in a computationally feasible way whether the store is satisfiable with respect to integer solutions. Therefore the `integral` constraint becomes non-primitive and satisfiability is checked over the rational numbers, which can be done in polynomial time, although the solution set need not be convex anymore [27].

The extension of the notion of primitive constraints allows us on the one hand to combine symbolic constraints of CP(FD), e.g. `alldifferent`, with symbolic constraints of extended ILP, e.g. `assign`. On the other hand, inference algorithms in existing non-primitive constraints may be improved and new non-primitive constraints can be designed, which use the extended primitive constraint set for more powerful inferences.

For example, the presence of disequalities allows us to set up stronger disjunctions than the usual dichotomy on the integral numbers. These stronger disjunctions can be used by an inference algorithm of the `integral` constraint that derives cutting planes by the disjunctive method. The bound reduction algorithms that were accommodated in the previous approach in the `linear` constraint can be used as a further inference algorithm of the `integral` constraint. Linear inequalities allow us to express relations between variables that are part of a non-primitive constraint directly by primitive constraints. If the interaction between the variables is strong enough, then in conjunction with the other primitive constraints in the store, this may lead to an earlier detection of infeasibility. In a branch-and-bound context, handling linear inequalities as primitive constraints makes it possible to place the lower bounding constraint directly into the store, which achieves a better pruning of the search space than by treating the lower bounding constraint as a non-primitive constraint (cf. Example 3.1).

The main drawback of the relaxation in finite domain constraint programming is that it can guide the solution process only in a very limited way, due to the low expressivity of the primitive constraints. Therefore the way branching is done plays an important role. In our extended integration, the linear relaxation may help to guide the solution process in a better way and may lead to better branching strategies, e.g. strong branching (see [26]). Furthermore, we can obtain better upper bounds that we can apply in a branch-and-relax context.

6 Conclusion

We have introduced a unifying framework, branch-and-infer, to describe and compare the languages of integer linear programming and finite domain constraint programming, both from the viewpoint of model building, i.e. their declarative semantics, and model solving, i.e. their operational semantics. Finite domain constraint programming offers a variety of arithmetic and symbolic constraints that allows to model and solve combinatorial problems in many different ways. Integer linear programming admits only linear equations and inequalities, but has developed very efficient methods to handle them. Our framework shows how integer linear programming can be extended with symbolic constraints and how algorithmic techniques from integer programming can be used in combination with finite domain methods.

References

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57 – 73, 1993.

[2] B. De Backer and H. Beringer. Cooperative solvers and global constraints: the case of linear arithmetic constraints. In *Postconference Workshop on Constraints, Databases, and Logic Programming, ILPS'95*, 1995.

[3] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9), 1996.

[4] E. Balas, S. Ceria, G. Cornuéjols, and N. R. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19, 1996.

[5] E. Balas and J.B. Mazzola. Nonlinear 0-1 programming: I. Linearization techniques. *Mathematical Programming*, 30:1–21, 1984.

[6] E. Balas and J.B. Mazzola. Nonlinear 0-1 programming: II. Dominance relations and algorithms. *Mathematical Programming*, 30:22–45, 1984.

[7] P. Barth. *Logic-based 0-1 constraint programming.* Operations Research/Computer Science Interfaces Series. Kluwer, 1996.

[8] P. Barth and A. Bockmayr. Finite domain and cutting plane techniques in CLP($\mathcal{PB}$). In L. Sterling, editor, *Logic Programming. 12th International Conference, ICLP'95, Kanagawa, Japan*, pages 133 – 147. MIT Press, 1995.

[9] J. Barwise. An introduction to first-order logic. In J. Barwise, editor, *Handbook of Mathematical Logic.* North Holland, 1977.

[10] N. Beldiceanu, A. Aggoun, and E. Contejean. Introducing constrained sequences in CHIP. Technical report, COSYTEC S.A., Orsay, France, 1996.

[11] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97 – 123, 1994.

[12] H. Beringer and B. de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic programming: Formal methods and practical applications*, pages 245 – 272. Elsevier, 1995.

[13] A. Bockmayr. Solving pseudo-Boolean constraints. In *Constraint Programming: Basics and Trends*, pages 22 – 38. Springer, LNCS 910, 1995.

[14] A. Caprara. On an integer programming formulation of a class of partitioning problems. Technical Report OR-96-8, DEIS, University of Bologna, 1996.

[15] A. Caprara and J.J. Salazar. A branch-and-cut algorithm for the index selection problem. Technical Report OR-95-9, DEIS, University of Bologna, September 1996.

[16] Y. Caseau and C. Le Pape. Exploiting operations research results in constraint-based scheduling. A tutorial. In *Practical Application of Constraint Technology, PACT'96, London*, 1996.

[17] G. Cornuéjols and J.-M. Thizy. Some facets of the simple plant location problem. *Mathematical Programming*, 23:50–74, 1982.

[18] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Fifth Generation Computer Systems, Tokyo, 1988.* Springer, 1988.

[19] C.E. Ferreira, A. Martin, and R. Weismantel. A cutting plane based algorithm for the multiple knapsack problem. Preprint SC-93-07, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1993.

[20] C.E. Ferreira, A. Martin, and R. Weismantel. Facets for the multiple knapsack problem. Preprint SC-93-04, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1993.

[21] E.S. Gottlieb and M.R. Rao. $(1, k)$-configuration facets for the generalized assignment problem. *Mathematical Programming*, 46:53–60, 1990.

[22] E.S. Gottlieb and M.R. Rao. The generalized assignment problem: Valid inequalities and facets. *Mathematical Programming*, 46:31–52, 1990.

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

[24] E. L. Johnson, G. L. Nemhauser, and M. W. P. Savelsbergh. Progress in integer programming: An exposition. Submitted to INFORMS J. on Computing, 1997.

[25] M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In *Handbook on Operations Research and Management Science*, volume 7, pages 225 – 330. Elsevier, 1995.

[26] M. Jünger, G. Reinelt, and S. Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In W. Cook, L. Lovász, and P. Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS series in discrete mathematics and theoretical computer science*, pages 111 – 152. AMS, 1995.

[27] J.-L. Lassez and K. McAloon. A canonical form for generalized linear constraints. *Journal of Symbolic Computation*, 13:1–24, 1992.

[28] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley, 1988.

[29] J. F. Puget. A C++ implementation of CLP. Technical report, ILOG S. A., 1994. http://www.ilog.com.

[30] V. A. Saraswat. *Concurrent constraint programming*. MIT Press, 1993.

[31] R. Sharda. *Linear & discrete optimization and modeling software*. UNICOM, 1993.

[32] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer, LNCS 1000, 1995.

[33] G. Smolka. Finite domain constraint programming. Draft of the chapter of the Oz Primer, March 1997.

[34] C. Solnon. Coopération de solveurs linéaires sur les réels pour la résolution de problèmes linéaires sur les entiers. In *JFPLC'97, Orléans*. Hermes, 1997.

[35] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[36] P. van Hentenryck and Y. Deville. Operational semantics of constraint logic programming over finite domains. In *Programming language implementation and logic programming, PLILP'91, Passau*. Springer, LNCS 528, 1991.

[37] P. van Hentenryck and V. Saraswat. Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701 – 726, 1996.

[38] M. Wallace. Practical applications of constraint programming. *Constraints*, 1:139 – 168, 1996.

[39] H. P. Williams. *Model building in mathematical programming.* John Wiley, 3rd revised edition, 1993.

[40] H. P. Williams. *Model solving in mathematical programming.* John Wiley, 1993.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-97-2-009	A. Bockmayr, F. Eisenbrand	On the Chvátal Rank of Polytopes in the 0/1 Cube
MPI-I-97-2-008	A. Bockmayr, T. Kasper	A Unifying Framework for Integer and Finite Domain Constraint Programming
MPI-I-97-2-007	P. Blackburn, M. Tzakova	Two Hybrid Logics
MPI-I-97-2-006	S. Vorobyov	Third-order matching in $\lambda\rightarrow$-Curry is undecidable
MPI-I-97-2-005	L. Bachmair, H. Ganzinger	A Theory of Resolution
MPI-I-97-2-003	U. Hustadt, R.A. Schmidt	On evaluating decision procedures for modal logic
MPI-I-97-2-002	R.A. Schmidt	Resolution is a decision procedure for many propositional modal logics
MPI-I-97-2-001	D.A. Basin, S. Matthews, L. Viganò	Labelled modal logics: quantifiers
MPI-I-96-2-010	A. Nonnengart	Strong Skolemization
MPI-I-96-2-009	D.A. Basin, N. Klarlund	Beyond the Finite in Automatic Hardware Verification
MPI-I-96-2-008	S. Vorobyov	On the decision complexity of the bounded theories of trees
MPI-I-96-2-007	A. Herzig	SCAN and Systems of Conditional Logic
MPI-I-96-2-006	D.A. Basin, S. Matthews, L. Viganò	Natural Deduction for Non-Classical Logics
MPI-I-96-2-005	A. Nonnengart	Auxiliary Modal Operators and the Characterization of Modal Frames
MPI-I-96-2-004	G. Struth	Non-Symmetric Rewriting
MPI-I-96-2-003	H. Baumeister	Using Algebraic Specification Languages for Model-Oriented Specifications
MPI-I-96-2-002	D.A. Basin, S. Matthews, L. Viganò	Labelled Propositional Modal Logics: Theory and Practice
MPI-I-96-2-001	H. Ganzinger, U. Waldmann	Theorem Proving in Cancellative Abelian Monoids
MPI-I-95-2-011	P. Barth, A. Bockmayr	Modelling Mixed-Integer Optimisation Problems in Constraint Logic Programming