Chapter 5

An Analysis of Arithmetic Constraints on Integer Intervals

In this chapter we demonstrate how OpenSolver can be configured for solving arithmetic constraints on variables with integer interval domains. We study a number of approaches to implement constraint propagation for these constraints. To describe them we introduce integer interval arithmetic. Each approach is explained using appropriate proof rules that reduce the variable domains.

Our goal is to determine which approach can be expected to give the best performance. To this end, we compare them on a set of benchmark problems. For the most promising approach we provide results that characterize the effect of constraint propagation.

5.1 Introduction

5.1.1 Motivation

The subject of arithmetic constraints on reals has attracted a great deal of attention in the literature. In contrast, arithmetic constraints on integer intervals have not been studied even though they are supported in a number of constraint programming systems. In fact, constraint propagation for them is present in ECLiPSe, SICStus Prolog, GNU Prolog, ILOG Solver and undoubtedly most of the systems that support constraint propagation for linear constraints on integer intervals. Yet, in contrast to the case of linear constraints — see notably [HS03] — we did not encounter in the literature any analysis of this form of constraint propagation.

In this chapter we study these constraints in a systematic way. It turns out that in contrast to linear constraints on integer intervals there are a number of natural approaches to constraint propagation for these constraints.

It could be argued that since integer arithmetic is a special case of real arithmetic, specialized constraint propagation methods for integer arithmetic con-
Chapter 5. Integer Arithmetic

Constraints are not needed. Indeed, a constraint satisfaction problem (CSP) involving arithmetic constraints on integer variables can be solved using any known method for constraints on the reals, with additional constraints ensuring that the variables assume only integer values. This was suggested in [BO97] and implemented for example in RealPaver [Gra04b]. However, a dedicated study and implementation of the integer case is beneficial for a number of reasons.

- In some cases the knowledge that we are dealing with integers yields a stronger propagation than the approach through the propagation for arithmetic constraints on the reals.

- The 'indirect' approach through the reals is based on floating point numbers, which are of limited precision. With a library like GNU MP (Multiple Precision, [Gra04a]) arbitrary precision floating point numbers can be used. However, for each problem the precision has to be chosen separately.

In contrast, for integer variables, we can use arbitrary length integers. These are limited only by available memory, and do not involve setting any parameters, making this approach more flexible and natural.

- Since arithmetic constraints on integer intervals are supported in a number of constraint programming systems, it is natural to investigate in a systematic way various approaches to their implementation. The direct approaches based on the integers are amenable for a clear theoretical analysis. In particular, in Section 5.8 and Subsection 5.9.1 we provide the characterization results that clarify the effect of constraint propagation for the approach that emerged in our studies as the fastest.

An example that supports the first argument is the constraint \( x \cdot y = z \), where \(-3 \leq x \leq 3\), \(-1 \leq y \leq 1\), and \(1 \leq z \leq 2\). When all variables are integers, there are no solutions having \( x = 3 \) or \( x = -3 \), and the constraint propagation methods that we consider here will actually remove these values from the domain of \( x \). However, if these variables are considered to be reals, these values may not be removed, and solving the integer problem through constraint propagation methods for constraints on the reals may lead to a larger search space.

As an indication that integer representation is not entirely a theoretical issue, consider the following benchmark from [BO97]. Find \( n \) integers \( x_1, \ldots, x_n \), \( 1 \leq x_i \leq n \), verifying the conditions

\[
\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} i, \quad \prod_{i=1}^{n} x_i = \prod_{i=1}^{n} i, \quad x_1 \leq x_2 \leq \ldots \leq x_n.
\]

For \( n = 10 \) the initial maximum value of the left-hand side expression of the second constraint equals \( 10^{10} \), which exceeds \( 2^{32} \), the number of values that can be represented as 32-bit integers. For \( n = 16 \), there is already no signed integer
representation of this bound in 64 bits. A small program written specifically for this benchmark indicates that on our machines, hardware integer operations are slightly faster than floating-point operations, and using arbitrary length integers costs less than a factor 4.

5.1.2 Outline of the Chapter

In the next section we define arithmetic constraints, and we relate our analysis to the model of constraint solving of Section 2.3. The unifying tool in our analysis is integer interval arithmetic that is modeled after the real interval arithmetic, see e.g., [HJvE01]. There are, however, essential differences since we deal with integers instead of reals. For example, multiplication of two integer intervals does not need to be an integer interval. In Section 5.3 we introduce the integer interval arithmetic and establish the basic results. Then in Section 5.4 we show that using integer interval arithmetic we can define succinctly the well-known constraint propagation for linear constraints on integer intervals.

The next three sections, 5.5, 5.6 and 5.7, form the main part of the chapter. We introduce there three approaches to constraint propagation for arithmetic constraints on integer intervals. They differ in the way the constraints are treated: either they are left intact, or the multiple occurrences of variables are eliminated, or the constraints are decomposed into a set of atomic constraints.

Then in Section 5.8 we characterize the effect of constraint propagation for the last approach. In Section 5.9 we discuss in detail our implementation of the alternative approaches, and in Section 5.10 we describe the experiments that were performed to compare them. They indicate that an optimized version of the third approach is superior to the other approaches. Finally, in Section 5.11 we provide the conclusions.

This chapter is based on joint work with Krzysztof Apt. Preliminary results were reported in [Apt03] and [AZ04].

5.2 Preliminaries

5.2.1 Arithmetic Constraints

To define the arithmetic constraints use the alphabet that comprises

- variables,

- two constants, 0 and 1,

- the unary minus function symbol \('-\',

- three binary function symbols, \'+\', \'-\', and \'-\', all written in the infix notation.
By an arithmetic expression we mean a term formed in this alphabet and by an arithmetic constraint a formula of the form

$$s \ op \ t,$$

where $s$ and $t$ are arithmetic expressions and $op \in \{<, \leq, =, \neq, \geq, >\}$. For example

$$x^5 \cdot y^2 \cdot z^4 + 3x \cdot y^3 \cdot z^5 \leq 10 + 4x^4 \cdot y^6 \cdot z^2 - y^2 \cdot x^5 \cdot z^4 \quad (5.1)$$

is an arithmetic constraint. Here $x^5$ is an abbreviation for $x \cdot x \cdot x \cdot x \cdot x$ and similarly with the other expressions. If '·' is not used in an arithmetic constraint, we call it a linear constraint.

By an extended arithmetic expression we mean a term formed in the above alphabet extended by the unary function symbols '⁻', 'ⁿ' and '√⁻' for each $n \geq 1$ and the binary function symbol '⁻' written in the infix notation. For example

$$\sqrt{(y^2 \cdot z^4)/(x^2 \cdot y^5)} \quad (5.2)$$

is an extended arithmetic expression. Here, unlike in (5.1), $x^5$ is a term obtained by applying the function symbol '⁻' to the variable $x$. The extended arithmetic expressions will be used only to define constraint propagation for the arithmetic constraints.

Fix now some arbitrary linear ordering $<$ on the variables of the language. By a monomial we mean an integer or a term of the form

$$a \cdot x_1^{n_1} \cdots x_k^{n_k}$$

where $k > 0$, $x_1, \ldots, x_k$ are different variables ordered w.r.t. $<$, and $a$ is a non-zero integer and $n_1, \ldots, n_k$ are positive integers. We call then $x_1^{n_1} \cdot \cdots \cdot x_k^{n_k}$ the power product of this monomial.

Next, by a polynomial we mean a term of the form

$$\Sigma_{i=1}^n m_i,$$

where $n > 0$, at most one monomial $m_i$ is an integer, and the power products of the monomials $m_1, \ldots, m_n$ are pairwise different. Finally, by a polynomial constraint we mean an arithmetic constraint of the form $s \ op \ b$, where $s$ is a polynomial with no monomial being an integer, $op \in \{<, \leq, =, \neq, \geq, >\}$, and $b$ is an integer. It is clear that by means of appropriate transformation rules we can transform any arithmetic constraint to a polynomial constraint. For example, assuming the ordering $x < y < z$ on the variables, the arithmetic constraint (5.1) can be transformed to the polynomial constraint

$$2x^5 \cdot y^2 \cdot z^4 - 4x^4 \cdot y^6 \cdot z^2 + 3x \cdot y^3 \cdot z^5 \leq 10$$

So, without loss of generality, from now on we shall limit our attention to the polynomial constraints.
5.2.2 Constraint Solving

In this chapter, the arithmetic constraints are interpreted over elements of domain type $\mathcal{I}$. Recall from Section 2.2.4 that the domains in $\mathcal{I}$ are integer intervals, or intervals for short, having the form

$$[a..b]$$

where $a$ and $b$ are integers; $[a..b]$ denotes the set of all integers between $a$ and $b$, including $a$ and $b$. If $a > b$, we call $[a..b]$ the empty interval and denote it by $\emptyset$. Further, by a range we mean an expression of the form

$$x \in I$$

where $x$ is a variable and $I$ is an interval.

As final domains we will be using $[\mathcal{I}]$, containing all domains with a single integer. In Sections 5.6 and 5.7 we will be rewriting CSPs to eliminate multiple occurrences of variables, or to decompose constraints into atomic constraints. This rewriting involves the introduction of new variables, and for these we will consider all domains in $\mathcal{I} - \{\emptyset\}$ as final domains. In other words, the new variables introduced by rewriting constraints are auxiliary variables. As a result, branching never takes place on these variables. This is justified in Section 5.6.

With this information, an explicit reference to ECSPs is not necessary. Also, all approaches to solving arithmetic constraints considered in this chapter have the property that if domains are singleton sets and the corresponding assignment of values to variables is not a solution, a failure is deduced, so all solved forms correspond to CSP solutions.

Finally, to conform our notation to that of [Apt03] and [AZ04], in this chapter we describe constraint propagation by means of proof rules that act on CSPs and preserve equivalence. An interested reader can consult [Apt98] or [Apt03] for a precise explanation of this approach to describing constraint propagation. In general it allows the description of transformations of CSPs beyond those of Section 2.3, but here we will consider only rules of the form

$$\langle C ; x_1 \in D_1, \ldots, x_n \in D_n \rangle$$
$$\langle C' ; x_1 \in D'_1, \ldots, x_n \in D'_n \rangle$$

that modify the domain of a single variable $x_j$, and have $D'_i = D_i$ for $i \neq j$, and $C' = C[D'_1, \ldots, D'_n]$, the new domain projected on the constraints. In presence of integer interval domains, such rules correspond to domain reduction functions of signature

$$f : \mathcal{I}^n \rightarrow \mathcal{I}$$

with input scheme $\langle 1, \ldots, n \rangle$ and output scheme $\langle j \rangle$, having

$$f(D_1, \ldots, D_n) = D'_j.$$ 

A CSP that is closed under application of such rules corresponds then to a common fixed point of (the domains extensions of) the corresponding DRFs.
5.3 Integer Set Arithmetic

To reason about the arithmetic constraints we employ a generalization of the arithmetic operations to the sets of integers. Here and elsewhere $\mathbb{Z}$ denotes the set of all integers.

5.3.1 Definitions

For $X, Y$ sets of integers we define the following operations:

- **addition:**
  \[ X + Y := \{ x + y \mid x \in X, y \in Y \}. \]

- **subtraction:**
  \[ X - Y := \{ x - y \mid x \in X, y \in Y \}. \]

- **multiplication:**
  \[ X \cdot Y := \{ x \cdot y \mid x \in X, y \in Y \}. \]

- **division:**
  \[ X/Y := \{ u \in \mathbb{Z} \mid \exists x \in X \exists y \in Y u - y = x \}. \]

- **exponentiation:**
  for each natural number $n > 0$,
  \[ X^n := \{ x^n \mid x \in X \}, \]

- **root extraction:**
  for each natural number $n > 0$,
  \[ \sqrt[n]{X} := \{ x \in \mathbb{Z} \mid x^n \in X \}. \]

All the operations except division are defined in the expected way. We shall return to it at the end of Section 5.7. At the moment it suffices to note the division operation is defined for all sets of integers, including $Y = \emptyset$ and $Y = \{0\}$. This division operation corresponds to the following division operation on the sets of reals introduced in [Rat96]:

\[ X \odot Y := \{ u \in \mathbb{R} \mid \exists x \in X \exists y \in Y u \cdot y = x \}. \]

For an integer or a real number $a$ and $op \in \{+,-,\cdot,./,\odot\}$ we identify $a \, op \, X$ with $\{a\} \, op \, X$ and $X \, op \, a$ with $X \, op \, \{a\}$.

To present the rules we are interested in we shall also use the addition and division operations on the sets of real numbers. Addition is defined in the same way as for the sets of integers, and division is defined above. In [HJvE01] it
5.3. Integer Set Arithmetic

is explained how to implement these operations on, possibly unbounded, real intervals.

Further, given a set $A$ of integers or reals, we define

$$\leq A := \{ x \in \mathbb{Z} \mid \exists a \in A \ x \leq a \},$$
$$\geq A := \{ x \in \mathbb{Z} \mid \exists a \in A \ x \geq a \}.$$

When limiting our attention to intervals of integers the following simple observation is of importance.

5.3.1. NOTE. For $X, Y$ integer intervals and $a$ an integer the following holds:

- $X \cap Y, X + Y, X - Y$ are integer intervals.
- $X/\{a\}$ is an integer interval.
- $X \cdot Y$ does not have to be an integer interval, even if $X = \{a\}$ or $Y = \{a\}$.
- $X/Y$ does not have to be an integer interval.
- For each $n > 1$ $X^n$ does not have to be an integer interval.
- For odd $n > 1$ $\sqrt[n]{X}$ is an integer interval.
- For even $n > 1$ $\sqrt[n]{X}$ is an integer interval or a disjoint union of two integer intervals.

For example we have

$$[2..4] + [3..8] = [5..12],$$
$$[3..7] - [1..8] = [-5..6],$$
$$[3..3] \cdot [1..2] = \{3.6\},$$
$$[3..5]/[-1..2] = \{-5, -4, -3, 2, 3, 4, 5\},$$
$$[-3..5]/[-1..2] = \mathbb{Z},$$
$$[1..2]^{1/2} = \{1, 4\},$$
$$\sqrt[3]{[-30..100]} = [-3..4],$$
$$\sqrt[3]{[-100..9]} = [-3..3],$$
$$\sqrt[3]{[1..9]} = [-3.. -1] \cup [1..3].$$

To deal with the problem that non-interval domains can be produced by some of the operations we introduce the following operation on the sets of integers:

$$\text{int}(X) := \mathcal{I}(X)$$

It is the integer interval counterpart of the hull operation on floating-point intervals, introduced in Section 2.2.4, and has the property that

$$\text{int}(X) = \begin{cases} 
\text{smallest integer interval containing } X & \text{if } X \text{ is finite,} \\
\mathbb{Z} & \text{otherwise.}
\end{cases}$$

For example $\text{int}([3..5]/[-1..2]) = [-5..5]$ and $\text{int}([-3..5]/[-1..2]) = \mathbb{Z}$. 
5.3.2 Implementation

To define constraint propagation for the arithmetic constraints on integer intervals we shall use the integer set arithmetic, mainly limited to the integer intervals. This brings us to the discussion of how to implement the introduced operations on the integer intervals. Since we are only interested in maintaining the property that the sets remain integer intervals or the set of integers $\mathbb{Z}$ we shall clarify how to implement the intersection, addition, subtraction and root extraction operations of the integer intervals and the $\text{int}(.)$ closure of the multiplication, division and exponentiation operations on the integer intervals. The case when one of the intervals is empty is easy to deal with. So we assume that we deal with non-empty intervals $[a..b]$ and $[c..d]$, i.e., $a \leq b$ and $c \leq d$.

Intersection, addition and subtraction. It is easy to see that

$$[a..b] \cap [c..d] = [\max(a, c) .. \min(b, d)],$$

$$[a..b] + [c..d] = [a + c .. b + d],$$

$$[a..b] - [c..d] = [a - d .. b - c].$$

So the interval intersection, addition, and subtraction are straightforward to implement.

Root extraction. The outcome of the root extraction operator applied to an integer interval will be an integer interval or a disjoint union of two integer intervals. We shall explain in Section 5.5 why it is advantageous not to apply $\text{int}(.)$ to the outcome. This operator can be implemented by means of the following case analysis.

Case 1. Suppose $n$ is odd. Then

$$\sqrt{[a..b]} = \left[ \left\lfloor \sqrt{a} \right\rfloor .. \left\lceil \sqrt{b} \right\rceil \right].$$

Case 2. Suppose $n$ is even and $b < 0$. Then

$$\sqrt{[a..b]} = \emptyset.$$

Case 3. Suppose $n$ is even and $b \geq 0$. Then

$$\sqrt{[a..b]} = \left[ -\left\lfloor \sqrt{b} \right\rfloor .. -\left\lceil \sqrt{a^+} \right\rceil \right] \cup \left[ \left\lfloor \sqrt{a^+} \right\rfloor .. \left\lceil \sqrt{b} \right\rceil \right],$$

where $a^+ := \max(0, a)$.
5.3. Integer Set Arithmetic

**Multiplication.** For the remaining operations we only need to explain how to implement the \texttt{int(.)} closure of the outcome. First note that

\[
\text{int}([a..b] \cdot [c..d]) = [\min(A) \cdot \max(A)],
\]

where \(A = \{ a \cdot c, a \cdot d, b \cdot c, b \cdot d \} \).

Using an appropriate case analysis we can actually compute the bounds of \(\text{int}([a..b] \cdot [c..d])\) directly in terms of the bounds of the constituent intervals.

**Division.** In contrast, the \texttt{int(.)} closure of the interval division is not so straightforward to compute. The reason is that, as we shall see in a moment, we cannot express the result in terms of some simple operations on the interval bounds.

Consider non-empty integer intervals \([a..b]\) and \([c..d]\). In analyzing the outcome of \(\text{int}([a..b]/[c..d])\) we distinguish the following cases.

**Case 1.** Suppose \(0 \in [a..b]\) and \(0 \in [c..d]\).
Then by definition \(\text{int}([a..b]/[c..d]) = \mathbb{Z} \). For example,

\[
\text{int}([-1..100]/[-2..8]) = \mathbb{Z}.
\]

**Case 2.** Suppose \(0 \not\in [a..b]\) and \(c = d = 0\).
Then by definition \(\text{int}([a..b]/[c..d]) = \emptyset\). For example,

\[
\text{int}([10..100]/[0..0]) = \emptyset.
\]

**Case 3.** Suppose \(0 \not\in [a..b]\) and \(c < 0\) and \(0 < d\).
It is easy to see that then

\[
\text{int}([a..b]/[c..d]) = [-e..e],
\]

where \(e = \max(|a|, |b|)\). For example,

\[
\text{int}([-100, -10]/[-2.5]) = [-100..100].
\]

**Case 4.** Suppose \(0 \not\in [a..b]\) and either \(c = 0\) and \(d \neq 0\) or \(c \neq 0\) and \(d = 0\).
Then \(\text{int}([a..b]/[c..d]) = \text{int}([a..b]/([c..d] \setminus \{0\}))\). For example

\[
\text{int}([1..100]/[-7.0]) = \text{int}([1..100]/[-7..-1]).
\]

This allows us to reduce this case to Case 5 below.

**Case 5.** Suppose \(0 \not\in [c..d]\).
This is the only case when we need to compute \(\text{int}([a..b]/[c..d])\) indirectly. First, observe that we have

\[
\text{int}([a..b]/[c..d]) \subseteq [\min(A) \cdot [\max(A))],
\]

where \(A = \{ a/c, a/d, b/c, b/d \} \).
However, the equality does not need to hold here. Indeed, note for example that \(\text{int}([155..161]/[9..11]) = [16..16]\), whereas for \(A = \{155/9, 155/11, 161/9, 161/11\}\) we have \(\lfloor \min(A) \rfloor = 15\) and \(\lceil \max(A) \rceil = 17\). The problem is that the value 16 is obtained by dividing 160 by 10 and none of these two values is an interval bound.

This complication can be solved by preprocessing the interval \([c..d]\) so that its bounds are actual divisors of an element of \([a..b]\). First, we look for the least \(c' \in [c..d]\) such that \(\exists x \in [a..b] \forall u \in \mathbb{Z} u \cdot c' = x\). Using a case analysis, the latter property can be established without search. Suppose for example that \(a > 0\) and \(c > 0\). In this case, if \(c' \cdot \lfloor b/c' \rfloor \geq a\), then \(c'\) has the required property. Similarly, we look for the largest \(d' \in [c..d]\) for which an analogous condition holds. Now \(\text{int}([a..b]/[c..d]) = [\lfloor \min(A) \rfloor .. \lceil \max(A) \rceil]\), where \(A = \{a/c', a/d', b/c', b/d'\}\).

In view of the auxiliary computation in case \(0 \notin [c..d]\), we shall introduce in Section 5.9 a modified division operation with a more direct implementation.

**Exponentiation.** The \(\text{int}(.)\) closure of the interval exponentiation is straightforward to implement by distinguishing the following cases.

**Case 1.** Suppose \(n\) is odd. Then
\[
\text{int}([a..b]^n) = [a^n..b^n].
\]

**Case 2.** Suppose \(n\) is even and \(0 \leq a\). Then
\[
\text{int}([a..b]^n) = [a^n..b^n].
\]

**Case 3.** Suppose \(n\) is even and \(b \leq 0\). Then
\[
\text{int}([a..b]^n) = [b^n..a^n].
\]

**Case 4.** Suppose \(n\) is even and \(a < 0\) and \(0 < b\). Then
\[
\text{int}([a..b]^n) = [0..\max(a^n, b^n)].
\]

5.3.3 **Correctness Lemma**

Given now an extended arithmetic expression \(s\) each variable of which ranges over an integer interval, we define \(\text{int}(s)\) as the integer interval or the set \(\mathbb{Z}\) obtained by systematically replacing each function symbol by the application of the \(\text{int}(.)\) operation to the corresponding integer set operation. For example, for the extended arithmetic expression \(s := \sqrt[3]{(y^2 \cdot z^3)/(x^2 \cdot u^3)}\) of (5.2) we have
\[
\text{int}(s) = \text{int}(\sqrt[3]{\text{int}(y^2) \cdot \text{int}(z^3)}/\text{int}(x^2) \cdot \text{int}(u^3))).
\]
where we assume that \(x\) ranges over \(X\), etc.
5.4. An Intermezzo: Linear Constraints

The discussion in the previous subsection shows how to compute \( \text{int}(s) \) given an extended arithmetic expression \( s \) and the integer interval domains of its variables.

The following lemma is crucial for our considerations. It is a counterpart of the so-called ‘Fundamental Theorem of Interval Arithmetic’ established in [Moo66]. Because we deal here with the integer domains an additional assumption is needed to establish the desired conclusion.

5.3.2. Lemma (Correctness). Let \( s \) be an extended arithmetic expression with the variables \( x_1, \ldots, x_n \). Assume that each variable \( x_i \) of \( s \) ranges over an integer interval \( X_i \). Choose \( a_i \in X_i \) for \( i \in [1..n] \) and denote by \( s(a_1, \ldots, a_n) \) the result of replacing in \( s \) each occurrence of a variable \( x_i \) by \( a_i \).

Suppose that each subexpression of \( s(a_1, \ldots, a_n) \) evaluates to an integer. Then the result of evaluating \( s(a_1, \ldots, a_n) \) is an element of \( \text{int}(s) \).

Proof. The proof follows by a straightforward induction on the structure of \( s \). \( \Box \)

5.4 An Intermezzo: Constraint Propagation for Linear Constraints

Even though we focus here on arithmetic constraints on integer intervals, it is helpful to realize that the integer interval arithmetic is also useful to define in a succinct way the well-known rules for constraint propagation for linear constraints (studied in detail in [HS03]). To this end consider first a constraint \( \Sigma_{i=1}^n a_i \cdot x_i = b \), where \( n \geq 0 \), \( a_1, \ldots, a_n \) are non-zero integers, \( x_1, \ldots, x_n \) are different variables, and \( b \) is an integer. To reason about it we can use the following rule parametrized by \( j \in [1..n] \):

\[
\text{LINEAR EQUALITY}
\]

\[
\begin{align*}
\{ \Sigma_{i=1}^n a_i \cdot x_i = b ; \ x_1 \in D_1, \ldots, x_n \in D_n \} \\
\{ \Sigma_{i=1}^n a_i \cdot x_i = b ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \}
\end{align*}
\]

where

- for \( i \neq j \)
  \[
  D'_i := D_i.
  \]

- \[
  D'_j := D_j \cap \text{int}(\left( b - \Sigma_{i \in \{1..n\} - \{j\}} a_i \cdot x_i \right)/a_j).
  \]
Note that by virtue of Note 5.3.1

\[ D'_j = D_j \cap (b - \sum_{i \in [1..n] \setminus \{j\}} \text{int}(a_i \cdot D_i)) / a_j. \]

To see that this rule preserves equivalence, first note that taking the intersection implies \( D'_j \subseteq D_j \), i.e., the domain is not extended by application of the rule. Further, suppose that for some \( d_1 \in D_1, \ldots, d_n \in D_n \) we have \( \sum_{i=1}^n a_i \cdot d_i = b \). Then for \( j \in [1..n] \) we have

\[ d_j = (b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot d_i) / a_j \]

which by the Correctness Lemma 5.3.2 implies that

\[ d_j \in \text{int}\left((b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot x_i) / a_j \right), \]

i.e., \( d_j \in D'_j \).

Next, consider a constraint \( \sum_{i=1}^n a_i \cdot x_i \leq b \), where \( a_1, \ldots, a_n, x_1, \ldots, x_n \) and \( b \) are as above. To reason about it we can use the following rule parametrized by \( j \in [1..n] \):

**LINEAR INEQUALITY**

\[ \langle \sum_{i=1}^n a_i \cdot x_i \leq b ; x_1 \in D_1, \ldots, x_n \in D_n \rangle \]

where

1. for \( i \neq j \)
   
   \[ D'_i := D_i, \]

2. for \( j \in [1..n] \)
   
   \[ D'_j := D_j \cap \left( \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot x_i / a_j \right) \]

To see that this rule preserves equivalence, first note that \( D'_j \subseteq D_j \). Further, suppose that for some \( d_1 \in D_1, \ldots, d_n \in D_n \) we have \( \sum_{i=1}^n a_i \cdot d_i \leq b \). Then \( a_j \cdot d_j \leq b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot d_i \). By the Correctness Lemma 5.3.2

\[ b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot d_i \in \text{int}(b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot x_i), \]

so by definition

\[ a_j \cdot d_j \in \text{int}(b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot x_i) \]

and consequently

\[ d_j \in \text{int}(b - \sum_{i \in [1..n] \setminus \{j\}} a_i \cdot x_i) / a_j \]

This implies that \( d_j \in D'_j \).
5.5 Constraint Propagation: First Approach

We now move on to a discussion of constraint propagation for the arithmetic constraints on integer intervals. The following example illustrates our first approach. Consider the constraint

\[ x^3 y - x \leq 40 \]

and the ranges \( x \in [1..100] \) and \( y \in [1..100] \). We can rewrite it as

\[ x \leq \left\lfloor \sqrt[3]{40 + x} / y \right\rfloor \tag{5.3} \]

since \( x \) assumes integer values. The maximum value that the expression on the right-hand side can take is \( \lfloor \sqrt[3]{140} \rfloor \), so we conclude \( x \leq 5 \). By reusing (5.3), now with the information that \( x \in [1..5] \), we conclude that the maximum value the expression on the right-hand side of (5.3) can take is actually \( \lfloor \sqrt[4]{45} \rfloor \), from which it follows that \( x \leq 3 \).

In the case of \( y \) we can isolate it by rewriting the original constraint as \( y \leq 40/x^3 + 1/x^2 \) from which it follows that \( y \leq 41 \), since by assumption \( x \geq 1 \). So we could reduce the domain of \( x \) to \([1..3]\) and the domain of \( y \) to \([1..41]\). This interval reduction is optimal, since \( x = 1, y = 41 \) and \( x = 3, y = 1 \) are both solutions to the original constraint \( x^3 y - x \leq 40 \).

More formally, we consider a polynomial constraint \( \sum_{i=1}^m m_i = b \) where \( m > 0 \), no monomial \( m_i \) is an integer, the power products of the monomials are pairwise different and \( b \) is an integer. Suppose that \( x_1, \ldots, x_n \) are its variables ordered w.r.t. \( < \).

Select a non-integer monomial \( m_i \) and assume it is of the form

\[ a \cdot y_1^{n_1} \cdots y_k^{n_k}, \]

where \( k > 0, y_1, \ldots, y_k \) are different variables ordered w.r.t. \( < \), \( a \) is a non-zero integer and \( n_1, \ldots, n_k \) are positive integers. So each \( y_i \) variable equals to some variable in \( \{x_1, \ldots, x_n\} \). Suppose that \( y_p \) equals to \( x_j \). We introduce the following proof rule:

**POLYNOMIAL EQUALITY**

\[
\begin{align*}
\langle \sum_{i=1}^m m_i = b : x_1 \in D_1, \ldots, x_n \in D_n \rangle \\
\langle \sum_{i=1}^m m_i = b : x_1 \in D'_1, \ldots, x_n \in D'_n \rangle
\end{align*}
\]

where

- for \( i \neq j \)

\[ D'_i := D_i. \]
Chapter 5. Integer Arithmetic

\[ D'_j := \text{int} \left( D_j \cap \sqrt[n]{\text{int} \left( (b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \} \} \right) / s \right) \right) \]

and

\[ s := a \cdot y_1^n \cdot \ldots \cdot y_{p-1}^{n_{p-1}} \cdot y_{p+1}^{n_{p+1}} \ldots y_k^{n_k}. \]

To see that this rule preserves equivalence, first note that taking the intersection implies \( D'_j \subseteq D_j \), i.e., the domain is not extended by application of the rule. Next, choose some \( d_1 \in D_1, \ldots, d_n \in D_n \). To simplify the notation, given an extended arithmetic expression \( t \) denote by \( t' \) the result of evaluating \( t \) after each occurrence of a variable \( x_i \) is replaced by \( d_i \).

Suppose that \( \sum_{i=1}^{m} m'_i = b \). Then

\[ d'^{np} \cdot s' = b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \} \] so by the Correctness Lemma 5.3.2 applied to \( b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \} \) and to \( s \)

\[ d'^{np} \in \text{int}(b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \}) / \text{int}(s). \]

Hence

\[ d_j \in \sqrt[n]{\text{int}(b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \}) / \text{int}(s)} \]

and consequently

\[ d_j \in \text{int} \left( D_j \cap \sqrt[n]{\text{int} \left( (b - \sum_{i \in [1..m]} \{ \{ \ldots - \{ \{ \} \} \}) / s \right) \right) \right) \]

i.e., \( d_j \in D'_j \).

Note that we do not apply \( \text{int}(\cdot) \) to the outcome of the root extraction operation. For even \( n_p \) this means that the second operand of the intersection can be a union of two intervals, instead of a single interval. To see why this is desirable, consider the constraint \( x^2 - y = 0 \) in the presence of ranges \( x \in [0..10] \), \( y \in [25..100] \). Using the \( \text{int}(\cdot) \) closure of the root extraction we would not be able to update the lower bound of \( x \) to 5.

Next, consider a polynomial constraint \( \sum_{i=1}^{m} m_i \leq b \). Below we adopt the assumptions and notation used when defining the POLYNOMIAL EQUALITY rule. To formulate the appropriate rule we stipulate that for the extended arithmetic expressions \( s \) and \( t \)

\[ \text{int}((\leq s)/t) := \geq Q \cap \leq Q, \]

with \( Q = (\leq \text{int}(s))/\text{int}(t) \).

To reason about this constraint we use the following rule:
POLYNOMIAL INEQUALITY

\[
\begin{align*}
\langle \Sigma_{i=1}^n m_i \leq b : x_1 \in D_1, \ldots, x_n \in D_n \rangle \\
\langle \Sigma_{i=1}^n m_i \leq b : x_1 \in D'_1, \ldots, x_n \in D'_n \rangle
\end{align*}
\]

where

- for \( i \neq j \)

\[D'_i := D_i,
\]

- \( D'_j := \text{int} \left( D_j \cap \sqrt[n]{\text{int} \left( \frac{\leq \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right)}{s} \right)} \right)\)

To prove that this rule preserves equivalence, first note that \( D'_j \subseteq D_j \). Next, choose some \( d_1 \in D_1, \ldots, d_n \in D_n \). As above given an extended arithmetic expression \( t \) we denote by \( t' \) the result of evaluating \( t \) when each occurrence of a variable \( x_i \) in \( t \) is replaced by \( d_i \).

Suppose that \( \Sigma_{i=1}^n m'_i \leq b \). Then

\[d_{i, p}^n \cdot s' \leq b - \Sigma_{i \in [1..m]-\{j\}} m'_i.\]

By the Correctness Lemma 5.3.2

\[b - \Sigma_{i \in [1..m]-\{j\}} m'_i \in \text{int} \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right),\]

so by definition

\[d_{i, p}^n \cdot s' \leq \text{int} \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right).
\]

Hence by the definition of the division operation on the sets of integers

\[d_{i, p}^n \leq \text{int} \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right) / \text{int}(s)\]

Consequently

\[d_j \in \sqrt[n]{\text{int} \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right) / \text{int}(s)}\]

This implies that \( d_j \in D'_j \).

Note that the set \( \leq \text{int} \left( b - \Sigma_{i \in [1..m]-\{j\}} m_i \right) \) appearing in the definition of \( D'_j \) is not an interval. So to properly implement this rule we need to extend the implementation of the division operation discussed in Subsection 5.3.2 to the case when the numerator is an extended interval. Our implementation takes care of this.

In an optimized version of this approach we simplify the fractions of two polynomials by splitting the division over addition and subtraction and by dividing out common powers of variables and greatest common divisors of the constant factors. Subsequently, fractions whose denominators have identical power products are added. We used this optimization in the initial example by simplifying
(40 + x)/x^3 to 40/x^3 + 1/x^2. The reader may check that without this simplification step we can only deduce that y ≤ 43.

To provide details of this optimization, given two monomials s and t, we denote by

\[ \frac{[s]}{t} \]

the result of performing this simplification operation on s and t. For example, \[ \frac{(2 \cdot x^3 \cdot y)/(4 \cdot x^2)}{= (x \cdot y)/2, \text{ whereas } (4 \cdot x^3 \cdot y)/(2 \cdot y^2)}{= (2 \cdot x^3)/y.} \]

In this approach we assume that the domains of the variables y_1, ..., y_{p-1}, y_{p+1}, ..., y_n of m_i do not contain 0. (One can easily show that this restriction is necessary here). For a monomial s involving variables ranging over the integer intervals that do not contain 0, the set int(s) either contains only positive numbers or only negative numbers. In the first case we write sign(s) = + and in the second case we write sign(s) = −.

The new domain of the variable x_j in the POLYNOMIAL INEQUALITY rule is defined using two sequences \( m'_0, ..., m'_n \) and \( s'_0, ..., s'_n \) of extended arithmetic expressions such that

\[ m'_i/s'_i = [b/s] \text{ and } m'_i/s'_i = -[m_i/s] \text{ for } i \in [1..m]. \]

Let \( S := \{s'_i \mid i \in [0..m] - \{l\} \} \) and for an extended arithmetic expression \( t \in S \) let \( I_i := \{i \in [0..m] - \{l\} \mid s'_i = t\} \). We denote then by \( p_t \) the polynomial \( \sum_{i \in I_t} m'_i \).

The new domains are then defined by

\[ D'_j := \text{int} \left( D_j \cap \left( \sqrt[\frac{\sum_{i \in S} p_t \otimes t}{\sum_{i \in \text{int}} (\sum_{i \in S} p_t \otimes t)} \right) \right) \]

if sign(s) = +, and by

\[ D'_j := \text{int} \left( D_j \cap \left( \sqrt[\frac{\sum_{i \in S} p_t \otimes t}{\sum_{i \in \text{int}} (\sum_{i \in S} p_t \otimes t)} \right) \right) \]

if sign(s) = −. Here the int(s) notation used in the Correctness Lemma 5.3.2 is extended to expressions involving the division operator \( \otimes \) on real intervals in the obvious way. We define the int(.) operator applied to a bounded set of real numbers, as produced by the division and addition operators in the above two expressions for \( D'_j \), to denote the smallest interval of real numbers containing that set.

### 5.6 Constraint Propagation: Second Approach

In this approach we limit our attention to a special type of polynomial constraints, namely the ones of the form \( s \ op b \), where \( s \) is a polynomial in which each variable occurs at most once and where \( b \) is an integer. We call such a constraint a simple polynomial constraint. By introducing variables that are equated
with appropriate monomials we can rewrite any polynomial constraint into a sequence of simple polynomial constraints. We apply then to the simple polynomial constraints the rules introduced in the previous section.

To see that the restriction to simple polynomial constraints can make a difference consider the constraint

$$100x \cdot y - 10y \cdot z = 212$$

in presence of the ranges $x, y, z \in [1..9]$. We rewrite it into the sequence

$$u = x \cdot y, \quad v = y \cdot z, \quad 100u - 10v = 212,$$

where $u, v$ are new variables, each with the domain $[1..81]$.

It is easy to check that the POLYNOMIAL EQUALITY rule introduced in the previous section does not yield any domain reduction when applied to the original constraint $100x \cdot y - 10y \cdot z = 212$. In presence of the discussed optimization the domain of $x$ gets reduced to $[1..3]$.

However, if we repeatedly apply the POLYNOMIAL EQUALITY rule to the simple polynomial constraint $100u - 10v = 212$, we eventually reduce the domain of $u$ to the empty set (since this constraint has no integer solution in the ranges $u, v \in [1..81]$) and consequently can conclude that the original constraint $100x \cdot y - 10y \cdot z = 212$ has no solution in the ranges $x, y, z \in [1..9]$, without performing any search.

The integer interval domains of the introduced variables are fully determined by the integer interval domains of the original variables. For this reason, no branching on these variables is needed: when the domains of the original variables are reduced to singleton sets, the domains of the introduced variables will be singleton sets as well. It is not efficient to branch on the new variables either. Consider for example that we have problem variables $x, y \in [1..10]$, with a variable $u \in [1..100]$ introduced to represent their product $u = x \cdot y$. If the domain of $u$ is bisected, two branches are created in which the domains of the problem variables $x$ and $y$ have an overlap after propagation of the constraint $u = x \cdot y$:

- left branch: $\begin{align*}
  u &\in [1..50] \\
  x &\in [1..10] \\
  y &\in [1..10]
\end{align*}$

- right branch: $\begin{align*}
  u &\in [51..100] \\
  x &\in [6..10] \\
  y &\in [6..10]
\end{align*}$

This overlap can lead to a larger search space, so branching on the introduced variables should be avoided. For the search process, such variables can be considered auxiliary variables, as introduced in Section 2.2.5.
Chapter 5. Integer Arithmetic

5.7 Constraint Propagation: Third Approach

In this approach we focus on a small set of 'atomic' arithmetic constraints. We call an arithmetic constraint \textit{atomic} if it is in one of the following two forms:

- a linear constraint,
- \(x \cdot y = z\).

It is easy to see that using appropriate transformation rules involving auxiliary variables we can transform any arithmetic constraint into a sequence of atomic arithmetic constraints. In this transformation, as in the second approach, the auxiliary variables are equated with monomials so we can easily compute their domains.

The transformation to atomic constraints can strengthen the reduction. Consider for example the constraint

\[
u \cdot x \cdot y + 1 = v \cdot x \cdot y
\]

and ranges \(u \in [1..2]\), \(v \in [3..4]\), and \(x, y \in [1..4]\). The first approach without optimization and the second approach cannot find a solution without search. If, as a first step in transforming this constraint into a linear constraint, we introduce an auxiliary variable \(w\) to replace \(x \cdot y\), we are effectively solving the constraint

\[
u \cdot w + 1 = v \cdot w
\]

with the additional range \(w \in [1..16]\), resulting in only one duplicate occurrence of a variable instead of two. With variable \(w\) introduced (or using the optimized version of the first approach) constraint propagation alone finds the solution \(u = 2\), \(v = 3\), \(x = 1\), \(y = 1\).

We explained already in Section 5.4 how to reason about linear constraints. (We omitted there the treatment of the disequalities which is routine.) Next, we focus on the reasoning for the multiplication constraint \(x \cdot y = z\) in presence of the non-empty ranges \(x \in D_x\), \(y \in D_y\) and \(z \in D_z\). To this end we introduce the following three domain reduction rules:

\textit{MULTIPLICATION 1}

\[
\frac{(x \cdot y = z \mid x \in D_x, y \in D_y, z \in D_z)}{(x \cdot y = z \mid x \in D_x, y \in D_y, z \in \text{int}(D_x \cdot D_y))}
\]

\textit{MULTIPLICATION 2}

\[
\frac{(x \cdot y = z \mid x \in D_x, y \in D_y, z \in D_z)}{(x \cdot y = z \mid x \in D_x \cap \text{int}(D_x/ D_y), y \in D_y, z \in D_z)}
\]
MULTIPLICATION 3

\[
\{x \cdot y = z ; x \in D_x, y \in D_y, z \in D_z\} 
\subseteq \{x \cdot y = z ; x \in D_x, y \in D_y \cap \text{int}(D_z/D_x), z \in D_z\}
\]

The way we defined the multiplication and the division of the integer intervals ensures that the MULTIPLICATION rules 1, 2, and 3 are equivalence preserving. Consider for example the MULTIPLICATION 2 rule. Take some \(a \in D_x, b \in D_y\) and \(c \in D_z\) such that \(a \cdot b = c\). Then \(a \in \{x \in \mathbb{Z} \mid \exists z \in D_z \exists y \in D_y x \cdot y = z\}\), so \(a \in D_x/D_y\) and a fortiori \(a \in \text{int}(D_x/D_y)\). Consequently \(a \in D_x \cap \text{int}(D_x/D_y)\). Because we also have \((D_x \cap \text{int}(D_x/D_y)) \subseteq D_x\), this shows that the MULTIPLICATION 2 rule is equivalence preserving.

The following example from [Apt03] shows an interaction between all three MULTIPLICATION rules.

5.7.1. EXAMPLE. Consider the CSP

\[
\{x \cdot y = z ; x \in [1..20], y \in [9..11], z \in [155..161]\}. \quad (5.4)
\]

To facilitate the reading we underline the modified domains. An application of the MULTIPLICATION 2 rule yields

\[
\{x \cdot y = z ; x \in [16..16], y \in [9..11], z \in [155..161]\}
\]

since, as already noted in Subsection 5.3.2, \([155..161]/[9..11] = [16..16]\), and \([1..20] \cap \text{int}([16..16]) = [16..16]\). Applying now the MULTIPLICATION 3 rule we obtain

\[
\{x \cdot y = z ; x \in [16..16], y \in [10..10], z \in [155..161]\}
\]

since \([155..161]/[16..16] = [10..10]\) and \([9..11] \cap \text{int}([10..10]) = [10..10]\). Next, by the application of the MULTIPLICATION 1 rule we obtain

\[
\{x \cdot y = z ; x \in [16..16], y \in [10..10], z \in [160..160]\}
\]

since \([16..16] \cdot [10..10] = [160..160]\) and \([155..161] \cap \text{int}([160..160]) = [160..160]\).

So using all three multiplication rules we could solve the CSP (5.4). \(\square\)

Now let us clarify why we did not define the division of the sets of integers \(Z\) and \(Y\) by

\[
Z/Y := \{z/y \in \mathbb{Z} \mid y \in Y, z \in Z, y \neq 0\}.
\]

The reason is that in that case for any set of integers \(Z\) we would have \(Z/\{0\} = 0\). Consequently, if we adopted this definition of the division of the integer intervals, the resulting MULTIPLICATION 2 and 3 rules would not be equivalence preserving anymore. Indeed, consider the CSP

\[
\{x \cdot y = z ; x \in [-2..1], y \in [0..0], z \in [-8..10]\}.
\]
Then we would have \([-8..10]/[0..0] = \emptyset\) and consequently by the MULTIPLICA-
TION 2 rule we could conclude

\[
\langle x \cdot y = z ; \ x \in \emptyset, y \in [0..0], z \in [-8..10] \rangle.
\]

So we reached an inconsistent CSP while the original CSP is consistent.

In the remainder of the chapter we will also consider variants of this third
approach that allow squaring and exponentiation as atomic constraints. For this
purpose we explain the reasoning for the constraint \(x = y^n\) in presence of the
non-empty ranges \(x \in D_x\) and \(y \in D_y\), and for \(n > 1\). To this end we introduce
the following two rules in which to maintain the property that the domains are
intervals we use the \(\text{int}(\cdot)\) operation of Section 5.3:

**EXPRESSIONATION**

\[
\frac{\langle x = y^n ; \ x \in D_x, y \in D_y \rangle}{\langle x = y^n ; \ x \in D_x \cap \text{int}(D_y^n), y \in D_y \rangle}
\]

**ROOT EXTRACTION**

\[
\frac{\langle x = y^n ; \ x \in D_x, y \in D_y \rangle}{\langle x = y^n ; \ x \in D_x, y \in \text{int}(D_y \cap \sqrt[D_y]{D_x}) \rangle}
\]

To prove that these rules are equivalence preserving suppose that for some
\(a \in D_x\) and \(b \in D_y\) we have \(a = b^n\). Then \(a \in D_y^n\), so \(a \in \text{int}(D_y^n)\) and conse-
quently \(a \in D_x \cap \text{int}(D_y^n)\). Also \(b \in \sqrt[D_y]{D_x}\), so \(b \in D_y \cap \sqrt[D_y]{D_x}\), and consequently
\(b \in \text{int}(D_y \cap \sqrt[D_y]{D_x})\). The set intersection operation prevents the extension of the
domains, as usual.

### 5.8 A Characterization of the MULTIPLICA-
TION Rules

It is useful to reflect on the effect of the proof rules used to achieve constraint
propagation. In this section, by way of example, we focus on the MULTIPLICA-
TION rules and characterize their effect using the notion of bounds consistency
of [VHSD98]. Let us recall first the definition that we adopt here to the mul-
tiplication constraint. Given an integer interval \([l..h]\) we denote by \([l, h]\) the
corresponding real interval.

**5.8.1. Definition.** The CSP \(\langle x \cdot y = z ; \ x \in [l_x..h_x], y \in [l_y..h_y], z \in [l_z..h_z] \rangle\) is
called **bounds consistent** if

- \(\forall a \in [l_x, h_x] \exists b \in [l_y, h_y] \exists c \in [l_z, h_z] \ a \cdot b = c.\)
5.8. A Characterization of the MULTIPLICATION Rules

- \( \forall b \in \{l_y, h_y\} \ \exists a \in [l_x, h_x] \ \exists c \in [l_z, h_z] \ a \cdot b = c, \)
- \( \forall c \in \{l_z, h_z\} \ \exists a \in [l_x, h_x] \ \exists b \in \{l_y, h_y\} \ a \cdot b = c. \)

Then we have the following result.

5.8.2. Theorem (Bounds consistency). Suppose a CSP \( \langle x \cdot y = z ; x \in D_x, y \in D_y, z \in D_z \rangle \) with integer interval domains is bounds consistent. Then it is closed under the applications of the MULTIPLICATION 1, 2 and 3 rules.

**Proof.** See the Appendix.

The converse of the above result does not hold. Here is an example. Consider the CSP

\[ \langle x \cdot y = z ; x \in [-2..1], y \in [-3..10], z \in [8..10] \rangle. \]

It is not bounds consistent, since for \( y = -3 \) no real values \( a \in [-2,1] \) and \( c \in [8,10] \) exist such that \( a \cdot (-3) = c \). Indeed, it is easy to check that

\[ \{ y \in \mathbb{R} \mid \exists x \in [-2,1] \exists z \in [8,10] x \cdot y = z \} = (-\infty, -4] \cup [8, \infty). \]

However, this CSP is closed (see Section 5.2.2) under the applications of the MULTIPLICATION 1, 2 and 3 rules since

- \( [8..10] \subseteq \text{int}([-2..1] \cdot [-3..10]) \), as \( \text{int}([-2..1] \cdot [-3..10]) = [-20..10] \),
- \( [-2..1] \subseteq \text{int}([8..10]/[-3..10]) \) as \( \text{int}([8..10]/[-3..10]) = [-10..10] \), and
- \( [-3..10] \subseteq \text{int}([8..10]/[-2..1]) \) as \( \text{int}([8..10]/[-2..1]) = [-10..10] \).

The following result clarifies that this example identifies the only cause of discrepancy between the closure under the MULTIPLICATION rules and bound consistency. Here, given an integer interval \( D := [l..h] \) we define

\[ \langle D \rangle := \{ x \in \mathbb{Z} \mid l < x < h \}. \]

5.8.3. Theorem (Bounds consistency 2). Consider a CSP \( \phi := \langle x \cdot y = z ; x \in D_x, y \in D_y, z \in D_z \rangle \) with non-empty integer interval domains and such that

\[ 0 \in \langle D_x \rangle \cap \langle D_y \rangle \] implies \( 0 \in D_z. \) \hfill (5.5)

Suppose \( \phi \) is closed under the applications of the MULTIPLICATION 1, 2, and 3 rules. Then it is bounds consistent.

**Proof.** See the Appendix.

Let us mention here that to deal with the constraint \( x \cdot y = z \) in [SS01] similar rules to our MULTIPLICATION rules were proposed. These rules were defined without the use of interval arithmetic. The rules for the variables \( x \) and \( y \) are different and more complex (also from an implementation point of view) than our MULTIPLICATION rules 2 and 3. As a result they achieve bounds consistency for the constraint \( x \cdot y = z \) for arbitrary integer interval domains.
5.9 Implementation Details

5.9.1 Weak Division

We already mentioned in Section 5.3 that the division operation on the intervals does not admit an efficient implementation. The reason is that the int(.) closure of the interval division \([a..b] / [c..d]\) requires an auxiliary computation in case when \(0 \not\in [c..d]\). The preprocessing of \([c..d]\) becomes impractical for small intervals \([a..b]\), and large \([c..d]\), occurring for example for the constraint \(\prod_{i=1}^{n} x_i = \prod_{i=1}^{n} i\), of the benchmark problem mentioned in Subsection 5.1.1. To remedy this problem we have used in our implementation another division operation. We call it \textit{weak division} since it yields a larger set (and so is ‘weaker’). This operation is defined as follows:

\[
[a..b] : [c..d] := \begin{cases} 
\text{if } 0 \not\in [c..d], \text{ or } \not\in [a..b] \text{ and } 0 \in \{c..d\} \text{ and } c < d, \\
\{a/c',a/d',b/c',b/d'\} \text{ otherwise}
\end{cases}
\]

where \(A = \{a/c',a/d',b/c',b/d'\}\), and \([c'.d'] = [c..d] - \{0\}\). Then \(\text{int}(\{a..b\} : [c..d])\) can be computed by a straightforward case analysis already used for \(\text{int}(\{a..b\}/[c..d])\) but now without any auxiliary computation.

In particular, in our implementation we used the following counterparts of the \textit{MULTIPLICATION} rules 2 and 3:

\[
\text{MULTIPLICATION 2w} \quad \frac{\langle x \cdot y = z : x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z : x \in D_x \cap \text{int}(D_z : D_y), y \in D_y, z \in D_z \rangle}
\]

\[
\text{MULTIPLICATION 3w} \quad \frac{\langle x \cdot y = z : x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z : x \in D_x, y \in D_y \cap \text{int}(D_z : D_y), z \in D_z \rangle}
\]

In the assumed framework based on constraint propagation and tree search, all domains become eventually singletons or empty sets. It can easily be verified that both division operations are then equal, i.e., \([a..b] : [c..d] = [a..b]/[c..d]\), for \(a \geq b\) and \(c \geq d\). For this reason, we can safely replace any of the reduction rules introduced in this chapter, notably \textit{POLYNOMIAL EQUALITY}, \textit{POLYNOMIAL INEQUALITY}, and \textit{MULTIPLICATION 2} and 3, by their counterparts based on the weak division. For the \textit{MULTIPLICATION} rules specifically, the following theorem states that both sets of rules actually achieve the same constraint propagation.
5.9. **Implementation Details**

5.9.1. **Theorem (MULTIPLICATION).** A CSP \( x \cdot y = z ; x \in D_x, y \in D_y, z \in D_z \) with integer interval domains is closed under the applications of the MULTIPLICATION 1, 2 and 3 rules iff it is closed under the applications of the MULTIPLICATION 1, 2w and 3w rules.

**Proof.** See the Appendix. □

Let us clarify now the relation between the MULTIPLICATION rules and the corresponding rules based on real interval arithmetic coupled with the rounding of the resulting real intervals inwards to the largest integer intervals. The CSP \( x \cdot y = z ; x \in [-3..3], y \in [-1..1], z \in [1..2] \), which we already discussed in the introduction, shows that these approaches yield different results. Indeed, using the MULTIPLICATION rule 2 we can reduce the domain of \( x \) to \([-2..2]\), while the second approach yields no reduction.

On the other hand, the applications of the MULTIPLICATION rules 2w and 3w to \( (x \cdot y = z ; x \in D_x, y \in D_y, z \in D_z) \) such that int\( (D_x : D_y) \neq \) int\( (D_z : D_x) \) and int\( (D_x : D_y) \neq \) int\( (D_z : D_y) \) (so in cases when the use of the weak interval division differs from the use of the interval division) do coincide with the just discussed approach based on real interval arithmetic and inward rounding. This is a consequence of the way the multiplication and division of the real intervals are defined, see [HJvE01]. We did not implement these instances of the MULTIPLICATION rules 2w and 3w through a detour via the rules for real intervals for the reasons explained in the introduction.

5.9.2 **Implementation**

**Constraint Propagation**

Integer intervals in OpenSolver are implemented by the `IntegerInterval` domain type plug-in. This plug-in, and the interval arithmetic operations on it are built using the `mpz` type of the GNU MP library [Gra04a], which supports arbitrary precision (or rather arbitrary length) integers. Domains of type `IntegerInterval` consist of an indication of the type of the set (bounded, unbounded, left/right-bounded, or empty), and the appropriate number (0, 1, or 2) of bounds.

Left-bounded and right-bounded sets have the respective forms \( \{ x \in \mathbb{Z} | x > l \} \) and \( \{ x \in \mathbb{Z} | x < h \} \), which are not integer intervals. Therefore, instead of \( I \), `IntegerInterval` is a (rather crude) implementation of the domain type containing all sets \([l..h]\), with \( l, h \in \mathbb{Z} \cup \{-\infty, \infty\} \), where \( \mathbb{Z} \) is a finite subset of \( \mathbb{Z} \) containing all integers that can be represented on a particular machine, using type `mpz`.

The reduction rules are implemented by a plug-in `IIARule` (Integer Interval Arithmetic Rule). Its specifier string has the form

\[ x_j^{op}(s) \]
where \( op \in \{\leq, =\} \), \( s \) is a monomial, and \( q \) is a polynomial. When the domain of a variable in \( s \) or \( p \) is modified, \texttt{IIARule} will set the domain \( D_j \) of \( x_j \) to

\[
\text{int}(D_j \cap \sqrt[\text{op}]{\text{int}(q/s)})
\]

if \( op \) is the symbol \( = \), or to

\[
\text{int}(D_j \cap \sqrt[\text{op}]{\text{int}(\leq(q/s))})
\]

if \( op \) is the symbol \( \leq \). With \( q \) set to \( b - \sum_{i \in \{1, \ldots, m\} \setminus \{i\}} m_i \), this implements the \textit{POLYNOMIAL EQUALITY} and \textit{POLYNOMIAL INEQUALITY} rules of Section 5.5, of which all other rules are instances.

As an example of its use in a solver configuration, the following three operators implement the constraint \( x^3y - x \leq 40 \).

\[
\begin{align*}
\text{DRF IIARule } \{ & x^3 \ast (1 \ast y) \leq 1 \ast x + 40 \}; \\
\text{DRF IIARule } \{ & y^1 \ast (1 \ast x^3) \leq 1 \ast x + 40 \}; \\
\text{DRF IIARule } \{ & x^1 \ast (-1) \leq -1 \ast x^3 \ast y + 40 \};
\end{align*}
\]

The \( ! \) prefix in the second specifier string activates the optimization described at the end of Section 5.5, which entails that common power products in \( s \) and \( q \) are eliminated. This cannot be implemented as a preprocessing stage, because \texttt{IIARule} needs to know the full monomial \( s \), so that it can select the appropriate case for \( \text{sign}(s) \).

### Scheduling Reduction Operators

For the second and third approach, we make use of the scheduling facilities of the operator-based scheduler, described in Section 4.1.1. We distinguish user constraints from the constraints that are introduced to define the values of auxiliary variables. Before considering for execution a DRF \( f \) that is part of the implementation of a user constraint, it is ensured that all auxiliary variables that \( f \) relies on are updated. For this purpose, the indices of the DRFs that update these variables precede the index of \( f \) in the schedule. If \( f \) can change the value of an auxiliary variable, its index is followed by the indices of the DRFs that propagate back these changes to the variables that define the value of this auxiliary variable.

As an example, the following operators (prefixed by a sequence number that is not part of the configuration) enforce the constraint \( 100x \cdot y - 10y \cdot z = 212 \).

\[
\begin{align*}
0. \ & \text{DRF IIARule } \{ & \text{aux}_xy^1 \ast (1) = x \ast y \}; \\
1. \ & \text{DRF IIARule } \{ & \text{aux}_yz^1 \ast (1) = y \ast z \}; \\
2. \ & \text{DRF IIARule } \{ & x^1 \ast (y) = \text{aux}_xy \}; \\
3. \ & \text{DRF IIARule } \{ & y^1 \ast (x) = \text{aux}_xy \}; \\
4. \ & \text{DRF IIARule } \{ & y^1 \ast (z) = \text{aux}_yz \}; \\
5. \ & \text{DRF IIARule } \{ & z^1 \ast (y) = \text{aux}_yz \}; \\
6. \ & \text{DRF IIARule } \{ & \text{aux}_xy^1 \ast (100) = 10 \ast \text{aux}_yz + 212 \}; \\
7. \ & \text{DRF IIARule } \{ & \text{aux}_yz^1 \ast (-10) = -100 \ast \text{aux}_xy + 212 \};
\end{align*}
\]
The operators with sequence number 6 and 7 correspond to the user constraint. Operators 1, 2, and 3 constrain the auxiliary variable aux.xy to be equal to \( x \cdot y \), and 0, 4, and 5 do the same for aux.yz and \( y \cdot z \). The following schedule is generated along with the decomposition.

\[
\text{SCHEDULER ChangeScheduler \{ schedule = \{ 1,6,2,3,0,7,4,5 \} \};}
\]

It specifies that in principle, operators 6 and 7 are applied in sequence, but operator 1 is considered before operator 6 in order that aux.xy, which appears in the right-hand side expression for operator 6, is updated before operator 6 is applied. If this modifies aux.yz, operators 2 and 3 will propagate this modification back to \( x \) and \( y \) before operator 7 is applied. The example is artificial because without other constraints on \( x \) and \( y \), the “problem” could be solved entirely on the auxiliary variables, and evaluation and back propagation would only have to be applied once.

For the third approach, there can be hierarchical dependencies between auxiliary variables. Much like the HC4 algorithm of [BGGP99] (see also Section 4.5), the generated schedule specifies a bottom-up traversal of this hierarchy in a forward evaluation phase and a top-down traversal in a backward propagation phase before and after applying a DRF of a user constraint, respectively. In the forward evaluation phase, the DRFs that are executed correspond to the \texttt{MULTIPLICATION 1} and \texttt{EXPONENTIATION} rules. The DRFs of the backward propagation phase correspond to the \texttt{MULTIPLICATION 2} and 3, and \texttt{ROOT EXTRACTION} rules. It is easy to construct examples showing that the use of hierarchical schedules can be beneficial compared to cycling through the rules.

**Optimization**

One of our benchmark problems is an \texttt{optimization} problem, where we want to find the assignment of values to decision variables that yields the optimal value for an objective function. Our approach to optimization problems is to introduce a variable for the outcome of an objective function, which can then be evaluated by constraint propagation. An optimization operator (a particular form of reduction operator, discussed in Section 3.2.2) then monitors this variable. It records the best value seen for any solution, and applies the dynamic constraint that new solutions must improve on this value.

For integer objective functions this is implemented by the \texttt{Optimize} reduction operator:

\[
\text{DRF Optimize \{ -x \};}
\]

Its specifier string is the name of an \texttt{IntegerInterval} variable, prefixed with - for minimization. or + for maximization. If the objective function is composed of arithmetic operations, it can be evaluated using the \texttt{IIARule} reduction operator.
This yields a particular form of \textit{branch-and-bound} search (see for example [Dec03]). Branch-and-bound algorithms maintain an estimation for the outcome of the objective function in the subtree that is currently being explored. This estimation is a \textit{bound} for the outcome of the objective function: a lower bound for minimization, and an upper bound for optimization. Based on this estimation, it may be possible to conclude that a particular branch of the search tree will never be able to improve on the current best solution. Such subtrees can then be pruned away. In our case the estimation is an interval that is guaranteed to contain the outcome of the objective function for any solutions that descend from the current node of the search tree.

\textbf{Approaches}

The proposed approaches were implemented by first rewriting arithmetic constraints to polynomial constraints, and then to a sequence of DRFs that correspond with the rules of the approach used. We considered the following methods:

\begin{enumerate}
\item[1a] the first approach, discussed in Section 5.5;
\item[1b] the optimization of the first approach discussed at the end of Section 5.5 that involves dividing out common powers of variables;
\item[2a] the second approach, discussed in Section 5.6. The conversion to simple polynomial constraints is implemented by introducing an auxiliary variable for every nonlinear monomial. This procedure may introduce more auxiliary variables than necessary;
\item[2b] an optimized version of approach 2a, where we stop introducing auxiliary variables as soon as the constraints contain no more duplicate occurrences of variables;
\item[3a] the third approach, discussed in Section 5.7, allowing only linear constraints and multiplication as atomic constraints:
\item[3b] idem, but also allowing \( x = y^2 \) as an atomic constraint;
\item[3c] idem, allowing \( x = y^n \) for all \( n > 1 \) as an atomic constraint.
\end{enumerate}

Approaches 2 and 3 involve an extra rewrite step, where the auxiliary variables are introduced. The resulting CSP is then rewritten according to approach 1a. During the first rewrite step the hierarchical relations between the auxiliary variables are recorded and the schedules are generated as a part of the second rewrite step. For approaches 2b and 3 the question of which auxiliary variables to introduce is an optimization problem in itself. Some choices result in more auxiliary variables than others. We have not treated this issue as an optimization
5.10. Experiments

5.10.1 Problems

In our experiments we used the following benchmarks.

**Cubes.** The problem is to find all natural numbers $n \leq 100000$ that are a sum of four different cubes, for example

\[ 1^3 + 2^3 + 3^3 + 4^3 = 100. \]

This problem is modeled as follows\(^1\):

\[
\begin{align*}
1 \leq x_1, & \quad x_1 \leq x_2 - 1, \quad x_2 \leq x_3 - 1, \quad x_3 \leq x_4 - 1, \quad x_4 \leq n, \\
x_1^3 + x_2^3 + x_3^3 + x_4^3 &= n; \quad n \in [1..100000], \quad x_1, x_2, x_3, x_4 \in \mathbb{Z}
\end{align*}
\]

**Opt.** We are interested in finding a solution to the constraint $x^3 + y^2 = z^3$ in the integer interval $[1..100000]$ for which the value of $2x \cdot y - z$ is maximal.

Program 3.1 on page 38 shows the OpenSolver configuration script for solving this problem according to approach 2a, which in this case is identical to that for approach 3c.

**Fractions.** This problem is taken from [SS02]: find distinct nonzero digits such that the following equation holds:

\[
\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1
\]

There is a variable for each letter. The initial domains are $[1..9]$. To avoid symmetric solutions an ordering is imposed:

\[
\frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI}
\]

---

\(^1\)Note that because the inequality constraints update only one bound, this works only because the domain type implemented by `IntegerInterval` supports domains of the form \(\{x \in \mathbb{Z} \mid x \geq l\}\) and \(\{x \in \mathbb{Z} \mid x \leq h\}\), with \(\mathbb{Z}\) a finite subset of \(\mathbb{Z}\). For using domain type `I` we would have to provide initial bounds for \(x_1, x_2, x_3, \text{and} \ x_4\).
Also two redundant constraints are added:

\[ \frac{3A}{BC} \geq 1 \quad \text{and} \quad \frac{G}{HI} \leq 1 \]

Because division is not present in our arithmetic expressions, the above constraints are multiplied by the denominators of the fractions to obtain arithmetic constraints. Two representations for this problem were studied:

- *fractions1* in which five constraints are used: one equality and four inequalities for the ordering and the redundant constraints,
- *fractions2*, used in [SS02], in which three auxiliary variables, \( BC, EF \) and \( HI \), are introduced to simplify the arithmetic constraints: \( BC = 10B + C \), \( EF = 10E + F \), and \( HI = 10H + I \).

Additionally, in both representations, 36 disequalities \( A \neq B, A \neq C, \ldots, H \neq I \) are used.

**Kyoto.** The problem (from [DS95]) is to find the number \( n \) such that the alphanumeric equation

\[
\begin{array}{cccc}
K & Y & O & T \\
K & Y & O & T \\
+ & K & Y & O & T \\
\hline
T & O & K & Y & O
\end{array}
\]

has a solution in the base-\( n \) number system. Our representation uses a variable for each letter and one variable for the base number. The variables \( K \) and \( T \) may not be zero. There is one large constraint for the addition, 6 disequalities \( K \neq Y \ldots T \neq O \) and four constraints stating that the individual digits \( K, Y, O, T \), are smaller than the base number. To spend some CPU time, we searched base numbers 2..100.

**Sumprod.** This is the problem cited in Subsection 5.1.1, for \( n = 14 \). We use the following representation:

\[
(x_1 + \ldots + x_n = c_1 + \ldots + c_n, \\
x_1 \ldots x_n = c_1 \ldots c_n, \\
x_1 \leq x_2, x_2 \leq x_3, \ldots, x_{n-1} \leq x_n; \\
x_1, \ldots, x_n \in [1..n], \\
c_1 \in \{1\}, c_2 \in \{2\}, \ldots, c_n \in \{n\})
\]

For \( n = 14 \), the value of the expression \( \prod_{i=1}^{n} i \) equals \( 14! \), which exceeds \( 2^{32} \), and to avoid problems with the input of large numbers, we used bound variables \( c_1, \ldots, c_n \) and constraint propagation to evaluate it.
5.10.2 Results

Tables 5.1 and 5.2 compare the proposed approaches on the problems defined in the previous subsection. We used a chronological variable selection strategy and a bisection branching, and in all experiments we searched for all solutions, traversing the entire search tree by means of depth-first leftmost-first chronological backtracking. The first two columns of table 5.1 list the number of variables and the DRFs that were used. Column nodes lists the size of the search tree, including failures and solutions. The next two columns list the number of times that a DRF was executed, and the percentage of these activations that the domain of a variable was actually modified. For the opt problem, the DRF that implements the optimization is not counted, and its activation is not taken into account. The elapsed times in the last column are the minimum times (in seconds) recorded for 5 runs on a 1200 MHz Athlon CPU.

Table 5.2 lists measured numbers of basic interval operations. Note that for approach 1b, there are two versions of the division and addition operations: one for integer intervals, and one for intervals of reals of which the bounds are rational numbers (marked Q). Columns multI and multF list the numbers of multiplications of two integer intervals, and of an integer interval and an integer factor, respectively. These are different operations in our implementation.

For the cubes, opt, and sumprod problems, the constraints are already in simple form, so approaches 1a, 1b and 2b are identical. For cubes and opt all nonlinear terms involve a single multiplication or exponentiation, so for these experiments also approaches 2a and 3c are the same. For both versions of the fractions problem, and for sumprod, no exponentiations are used, so versions a, b, and c of approach 3 are identical. The results of these experiments clearly show the disadvantage of implementing exponentiation by means of multiplication: the search space grows because we increase the number of variable occurrences and lose the information that it is the same number that is being multiplied. For opt and approach 3a, the run did not complete within reasonable time and was aborted.

Columns E and I of table 5.1 compare the propagation achieved by our approaches with two other systems, respectively ECL\textsuperscript{PS}$^c$ Version 5.6 [WNS97] using the ic library, and ILOG Solver 5.1 [Ilo01] using type ILOINT. For this purpose we ran the test problems without search, and compared the results of constraint propagation. A mark '=' means that the computed domains are the same, '+' that our approach achieved stronger propagation than the solver that we compare with, and '-' that propagation is weaker. For cubes, ECL\textsuperscript{PS}$^c$ computes the same domains as those computed according to approach 3b, so here the reduction is stronger than for 3a, but weaker than for the other approaches. For opt ECL\textsuperscript{PS}$^c$ and ILOG Solver compute the same domains. These domains are narrower than those computed according to approaches 3a and 3b, but the other approaches achieve stronger reduction. In all other cases except for kyoto
### Chapter 5. Integer Arithmetic

<table>
<thead>
<tr>
<th></th>
<th>nvar</th>
<th>nDRF</th>
<th>nodes</th>
<th>activated</th>
<th>%eff.</th>
<th>elapsed (sec.)</th>
<th>E</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cubes</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2b</td>
<td>5</td>
<td>14</td>
<td>169,755</td>
<td>1,876,192</td>
<td>9.52</td>
<td>9.60</td>
<td>+</td>
<td>=</td>
</tr>
<tr>
<td>2a,3c</td>
<td>9</td>
<td>22</td>
<td>169,755</td>
<td>2,237,590</td>
<td>16.28</td>
<td>6.46</td>
<td>+</td>
<td>=</td>
</tr>
<tr>
<td>3a</td>
<td>13</td>
<td>34</td>
<td>206,405</td>
<td>3,011,749</td>
<td>20.02</td>
<td>8.46</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3b</td>
<td>13</td>
<td>34</td>
<td>178,781</td>
<td>2,895,717</td>
<td>20.62</td>
<td>8.61</td>
<td>=</td>
<td>-</td>
</tr>
<tr>
<td><strong>opt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2b</td>
<td>4</td>
<td>7</td>
<td>115,469</td>
<td>5,187,002</td>
<td>42.16</td>
<td>20.97</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2a,3c</td>
<td>8</td>
<td>15</td>
<td>115,469</td>
<td>9,800,017</td>
<td>60.00</td>
<td>21.47</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3a</td>
<td>10</td>
<td>21</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3b</td>
<td>10</td>
<td>21</td>
<td>5,065,195</td>
<td>156,906,444</td>
<td>46.49</td>
<td>406.39</td>
<td>=</td>
<td>-</td>
</tr>
<tr>
<td><strong>fractions 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1a</td>
<td>9</td>
<td>154</td>
<td>11,289</td>
<td>1,193,579</td>
<td>3.65</td>
<td>16.00</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>1b</td>
<td>9</td>
<td>154</td>
<td>7,879</td>
<td>734,980</td>
<td>3.45</td>
<td>17.99</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2a</td>
<td>37</td>
<td>210</td>
<td>11,289</td>
<td>1,410,436</td>
<td>23.27</td>
<td>4.95</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2b</td>
<td>32</td>
<td>200</td>
<td>11,289</td>
<td>1,385,933</td>
<td>21.65</td>
<td>5.41</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>208</td>
<td>11,131</td>
<td>1,426,204</td>
<td>27.76</td>
<td>5.07</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td><strong>fractions 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1a</td>
<td>12</td>
<td>105</td>
<td>2,449</td>
<td>270,843</td>
<td>9.72</td>
<td>0.58</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>1b</td>
<td>12</td>
<td>105</td>
<td>989</td>
<td>94,894</td>
<td>9.12</td>
<td>0.55</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2a</td>
<td>20</td>
<td>121</td>
<td>2,449</td>
<td>350,390</td>
<td>22.19</td>
<td>0.48</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2b</td>
<td>15</td>
<td>111</td>
<td>2,449</td>
<td>301,865</td>
<td>17.50</td>
<td>0.46</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>123</td>
<td>1,525</td>
<td>293,051</td>
<td>27.33</td>
<td>0.41</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td><strong>kyoto</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1a</td>
<td>5</td>
<td>37</td>
<td>87,085</td>
<td>3,299,814</td>
<td>6.09</td>
<td>21.80</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>1b</td>
<td>5</td>
<td>37</td>
<td>87,085</td>
<td>3,288,461</td>
<td>5.94</td>
<td>46.12</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2a</td>
<td>13</td>
<td>53</td>
<td>87,085</td>
<td>3,781,514</td>
<td>23.02</td>
<td>10.73</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2b</td>
<td>12</td>
<td>51</td>
<td>87,085</td>
<td>3,622,461</td>
<td>21.45</td>
<td>11.06</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3a</td>
<td>16</td>
<td>60</td>
<td>87,087</td>
<td>4,276,066</td>
<td>26.70</td>
<td>10.25</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3b</td>
<td>16</td>
<td>60</td>
<td>87,085</td>
<td>4,275,957</td>
<td>26.70</td>
<td>10.34</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3c</td>
<td>16</td>
<td>59</td>
<td>87,085</td>
<td>3,746,532</td>
<td>23.26</td>
<td>9.33</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td><strong>sumprod</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2b</td>
<td>28</td>
<td>82</td>
<td>230,233</td>
<td>10,910,441</td>
<td>7.91</td>
<td>112.25</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>2a</td>
<td>30</td>
<td>86</td>
<td>230,233</td>
<td>9,196,772</td>
<td>9.39</td>
<td>89.37</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>3</td>
<td>54</td>
<td>134</td>
<td>55,385</td>
<td>3,078,649</td>
<td>18.01</td>
<td>26.57</td>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

Table 5.1: Statistics and comparison with other solvers
### Experiments

<table>
<thead>
<tr>
<th></th>
<th>root</th>
<th>exp</th>
<th>div</th>
<th>multI</th>
<th>multF</th>
<th>sum</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cubes</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,2b</td>
<td>1,182</td>
<td>4,224</td>
<td>0</td>
<td>0</td>
<td>4,756</td>
<td>4,245</td>
<td>14,408</td>
</tr>
<tr>
<td>2a,3c</td>
<td>180</td>
<td>181</td>
<td>0</td>
<td>0</td>
<td>4,756</td>
<td>4,245</td>
<td>9,363</td>
</tr>
<tr>
<td>3a</td>
<td>0</td>
<td>0</td>
<td>589</td>
<td>438</td>
<td>4,927</td>
<td>4,363</td>
<td>10,317</td>
</tr>
<tr>
<td>3b</td>
<td>192</td>
<td>198</td>
<td>384</td>
<td>198</td>
<td>4,842</td>
<td>4,305</td>
<td>10,121</td>
</tr>
</tbody>
</table>

| **opt** |      |     |     |       |       |      |       |
| 1,2b   | 2,299| 4,599| 1,443|1,444 |11,064 | 5,187| 26,037|
| 2a,3c  | 1,636| 1,538|2,150 |738   |8,138  | 4,445| 18,645|
| 3b     | 21,066|18,106|54,172|18,285|106,652|57,470|275,751|

| **fractions1** |      |     |     |       |       |      |       |
| 1a     | 0    | 0   | 868 | 28,916|14,238 |13,444|57,466 |
| 1b     | 0    | 0   | 51  | 11,892| 8,010 | 6,727|29,584 |
|         |      |     |     | 1,550 | 1,355 |      |      |
| 2a     | 0    | 0   | 734 | 933   | 4,736 | 4,669|11,071 |
| 2b     | 0    | 0   | 776 | 1,509 | 5,292 | 5,147|12,725 |
| 3      | 0    | 0   | 693 | 339   | 4,835 | 4,769|10,636 |

| **fractions2** |      |     |     |       |       |      |       |
| 1a     | 0    | 0   | 142 | 690   | 304   | 212  | 1,348 |
| 1b     | 0    | 0   | 19  | 127   | 59    | 26   | 344   |
|         |      |     |     | 65    | 49    |      |      |
| 2a     | 0    | 0   | 124 | 149   | 138   | 94   | 505   |
| 2b     | 0    | 0   | 124 | 206   | 210   | 118  | 658   |
| 3      | 0    | 0   | 114 | 46    | 142   | 101  | 403   |

| **kyoto** |      |     |     |       |       |      |       |
| 1a      | 735  | 11,041|1,963|13,853 |10,853 |13,946|52,390 |
| 1b      | 735  | 8,146|218  |8,955  |12,516 |10,592|48,749 |
|         |      |     |     | 4,310 | 3,277 |      |      |
| 2a      | 383  | 759 |1,591|484    |5,324  | 7,504|16,044 |
| 2b      | 383  | 759 |1,597|1,360  |5,756  | 8,008|17,863 |
| 3a      | 0    | 0   |1,991|578    |5,324  | 7,505|15,398 |
| 3b      | < 0.5| < 0.5|1,990|578    |5,324  | 7,504|15,397 |
| 3c      | 1    | 1   |1,554|484    |5,324  | 7,504|14,868 |

| **sumprod** |      |     |     |       |       |      |       |
| 1,2b     | 0    | 0   | 4,032|100,791|85,419 |149,479|339,721|
| 2a       | 0    | 0   | 2,186|27,948 |81,728 |149,479|261,340|
| 3        | 0    | 0   | 609  |205    |25,799 |46,960|73,573 |

Table 5.2: Measured numbers (thousands) of interval operations
and approach 1b the results of all three solvers are the same.

For the fractions puzzle, the symbolic manipulation of approach 1b reduces the search tree by a factor 0.70 for the first representation, and by a factor 0.40 for the second. However, this reduction is not reflected in the timings. For _fractions1_ the elapsed time even increases. The reason is that computing the domain updates involves adding intervals of real numbers. The arithmetic operations on such intervals are more expensive than their counterparts on integer intervals, because the bounds have to be maintained as rational numbers. Arithmetic operations on rational numbers are more expensive because they involve the computation of greatest common divisors. For _kyoto_ the symbolic manipulation did not reduce the size of the search tree, so the effect is even more severe.

In general, the introduction of auxiliary variables leads to a reduction of the number of interval operations compared to approach 1a. The reason is that auxiliary variables prevent the evaluation of subexpressions that did not change. This effect is strongest for _fractions1_, where the main constraint contains a large number of different power products. Without auxiliary variables all power products are evaluated for every _POLYNOMIAL EQUALITY_ rule defined by this constraint, even those power products the variable domains of which did not change. With auxiliary variables the intervals for such unmodified terms are available immediately, which leads to a significant reduction of the number of interval multiplications. For _sumprod_, the difference between approaches 1a and 2a is a bit artificial, because the operations that are saved involve the computation of the constant term $c_1 \cdots c_n$. A comparable number of interval additions can be saved if we introduce a variable for the constant term $c_1 + \ldots + c_n$. If we add these variables to the CSP all variants of approaches 1 and 2 are essentially the same.

The effect that stronger reduction is achieved as a result of introducing auxiliary variables, mentioned in Section 5.7, is seen for both representations of the _fractions_ benchmark, and prominently for _sumprod_. In the latter case, this effect depends on a decomposition of the term $\prod_{i=1}^{n} x_i$ as $x_1 \cdot (x_2 \cdot \ldots \cdot (x_{n-1} \cdot x_n) \ldots))$. with an auxiliary variable per pair of brackets. The decomposition then matches the chronological ordering used to select the variable for branching. If the ordering is reversed, the number of nodes is equal to that of the other approaches. The effect described in Section 5.6 is not demonstrated by these experiments.

If we do not consider the symbolic manipulation of approach 1b, then approach 3c leads to the smallest total number of interval operations in all cases, but the scheduling mechanism discussed in Section 5.9 is essential for a consistent good performance. If for example the schedule is omitted for _opt_, the number of interval operations almost triples, and performance of approach 2a and 3c is then much worse than that of approach 1a.

The total numbers of interval operations in table 5.2 do not fully explain all differences in elapsed times. One of the reasons is that different interval operations have different costs. Also some overhead is involved in applying a DRF, so if the number of applications differs significantly for two experiments, this influences
the elapsed times as well (opt, 1a, 2a, fractions2, 2a, 2b). The elapsed times are not the only measure that is subject to implementation details. For example, we implemented division by a constant interval \([-1..1\]) as multiplication by a constant, which is more efficient in our implementation. Such decisions are also reflected in the numbers reported in table 5.2.

5.11 Conclusions

In this chapter we discussed a number of approaches to constraint propagation for arithmetic constraints on integer intervals. To assess them we implemented them using the OpenSolver framework, and compared their performance on a number of benchmark problems. We can conclude that:

- Implementation of exponentiation by multiplication gives weak reduction. In our third approach \(x = y^n\) should be used as an atomic constraint.

- The optimization of the first approach, where common powers of variables are divided out, can significantly reduce the size of the search tree, but the resulting reduction steps rely heavily on the division and addition of rational numbers. These operations are more expensive than their integer counterparts, because they involve the computation of greatest common divisors. As a result, our implementation of this approach was inefficient.

- Introducing auxiliary variables can be beneficial in two ways: it may strengthen the propagation, as discussed in Sections 5.6 and 5.7, and it may prevent the evaluation of subexpressions the variable domains of which did not change.

- As a result, given a proper scheduling of the rules, the second and third approach perform better than the first approach without the optimization, in terms of numbers of interval operations. Actual performance depends on many implementation aspects. However for our test problems the performance of variants 2a, 2b and 3c does not differ much, except for one case where the decomposition of a single multiplication of all variables significantly reduced the size of the search tree.

Because of the inherent simplicity of the reduction rules and the potential additional reduction of the search tree, approach 3c is our method of choice. We decompose polynomial constraints into multiplication, exponentiation, and linear constraints. A hierarchical scheduling of the resulting reduction rules is essential for improving the performance of approach 1a. As we noted at the end of Section 5.9.2, it may be possible to improve the performance further by treating the decomposition into atomic constraints as an optimization problem, minimizing the number of auxiliary variables.
Given that approach 1b can achieve a significant reduction of the search tree, it would be interesting to combine it with approach 3c. Depending on the effect of the symbolic manipulation, a selection of the optimized rules that enforce a particular constraint according to approach 1b could be used as redundant rules. In this case, the internal computations need not be precise, and we could maintain the rational bounds as floating-point numbers, thus avoiding the expensive computation of greatest common divisors.

Note that our characterization of the third approach is limited to version 3a. A characterization of the linear equality constraints can be found, for example, in [Apt03], but the atomic constraint $x = y^n$ is not covered. Also a proper characterization of the first and second approach may help us formalize the improved reduction observed in Sections 5.6 and 5.7. Because of the lengthy proofs involved, we have left this as an opportunity for future work.

We would like to point out that the operators studied in this chapter are similar to those for enforcing hull consistency, which we discussed in Section 4.5 for floating-point intervals. In Chapter 7 we will implement a stronger notion of consistency called box consistency, and apply this both to floating-point intervals and integer intervals. The operators for enforcing box consistency will be composed from the facilities introduced here and in Section 4.5, plus a generic operator for nested search.

So far we have only seen examples of constraint solvers on a single domain type. In the next chapter we will study a hybrid solver, where some of the facilities introduced here, namely those for optimization, are combined with reduction operators on special-purpose domain types.