Reasoning on the properties of numerical constraints

Lucas Bordeaux, Eric Monfroy, and Frédéric Benhamou
{bordeaux,monfroy,benhamou@irin.univ-nantes.fr}

Institut de Recherche en Informatique de Nantes (IRIN), France

Abstract. Numerical Constraint-Satisfaction Problems (NCSPs) are stated as (in)-equalities between numerical functions. The properties of these functions are a most useful information, which determines the use of specialized algorithms. We propose a framework to reason on a set of function properties which integrates *monotonicity* and *convexity*, and aims at being general and extensible. Properties are seen as *abstractions* of the function curves, and we propose *deduction rules* to reason on these abstractions. We give guidelines on how this tool can cooperate with existing or customized constraint-solvers.

1 Introduction

Among the most fundamental computational problems involving numerical functions are *optimization*, *i.e.,* the task of finding an optimal value *w.r.t.* a numerical criterion, and *constraint solving*, *i.e.,* the task of finding values that satisfy a set of equalities or inequalities. Arising from the field of combinatorial search in artificial intelligence, constraint-propagation methods [13] recently turned-out to be a general and competitive approach for solving these continuous problems. The transition between discrete and continuous domains was made possible using *intervals* to approximate the ranges of the values permitted for the variables of the problem [5, 11, 1].

It is well-known that the tractability of constraint-satisfaction strongly depends on the properties satisfied by the functions involved. For instance, *linear* functions (*i.e.,* of the form $\sum_i a_i x_i$) can be handled efficiently using linear programming techniques. More general though still not too difficult classes of functions are the *monotonic* and *convex* ones, for which there exists specialized algorithms in the non-linear programming [10, 16], and constraint programming literature [17].

However, and despite the constant interest in solving particular classes of functions, the issue of devising logical tools to represent properties and perform reasoning on these properties has, as far as we know, not been considered in the literature.

We define a framework where properties are seen as constraint *abstractions*. The Abstract Interpretation framework [3] helps defining a sound representation with a clear semantics. We then show how *rules* can be used to reason on these properties. This framework is in some sense a dynamic generalization of the static inference of *signs* in the context of program-analysis [3, 14]. We give guidelines on how this reasoning on properties can be integrated in constraint solvers.

A simple example

To motivate our advocacy that curve properties matter, consider the toy problem of finding the minimum value of the curve x/y (Figure 1) when x and y range over $[1, 100]$. It turns out that this curve is monotonic in the following sense: whatever point we consider in $[1, 100]^2$, increasing x always yields higher values for the result, while increasing y yields lower values. This information prevents us from using costly techniques for this problem, since the minimum is obtained when x takes its lowest value (namely 1) and y takes its highest value (100).

Determining that function x/y be monotonic on $[1, 100]^2$ was easy since division is a basic operation. It becomes more challenging to determine properties of complex functions constructed by *composition* of basic operations. However, this is not always a difficult task. For instance, raising a function to the exponential does not change its monotonicity, and the minimum for function $\exp(x/y)$ is still obtained for $x = 1,\ y = 100$. On the contrary, the minimum of function $-\exp(x/y)$ is reached on $x = 100,\ y = 1$. This kind of reasoning over the syntactical structure of functions is what this paper is about.

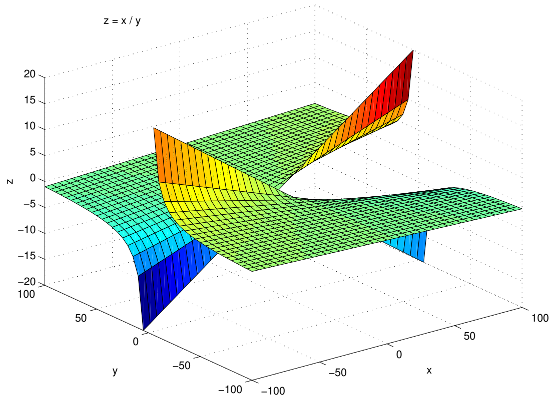

Fig. 1. The curve for division shows 4 monotonic parts: on $\mathbb{R}^- \times \mathbb{R}^-$, on $\mathbb{R}^- \times \mathbb{R}^+$, on $\mathbb{R}^+ \times \mathbb{R}^-$, and on $\mathbb{R}^+ \times \mathbb{R}^+$ (matlab ©).

Outline of the paper

The next Section (2) introduces basic material and notations on functions and expressions. We list some properties of interest and introduce abstractions to reify these properties (Section 3). These abstractions allow some kind of automated reasoning, which is briefly formalized in Section 4, and whose rules are further exemplified in Section 5. Section 6 discusses the practical relevance of this reasoning, and the paper ends up with a conclusion (Section 7).

2 Basic concepts and notations

Let $\mathbb{D}$ be an ordered *domain* of computation (reals, integers, ...), and $\mathbb{D}^+$ (*resp.* $\mathbb{D}^-$) be the set of its positive (*resp.* negative) values. Let $\mathcal{V} = \{x_1, \ldots, x_n\}$ be a set of *variable* names.

An expression like $x + \exp(y)$ can be seen in two different ways: as a *term* built over some symbols and as a *function* of x and y. It is usually clear from the context whether one vision is used or the other.

Expressions

Numerical functions are defined by syntactical means, as expressions built over the variables of the problem and a set of symbols $\exp, +, \ldots$, which are either unary or binary.

As usual in the context of rewriting [12], an expression e is said to *subsume* an expression f if there exists a substitution σ *s.t.* $f = \sigma e$. The notation $f \preceq e$ denotes the *subsumption order*. (*e.g.,* $1-2.z \preceq x-y$, with $\sigma = \{x \leftarrow 1, y \leftarrow 2.z\}$).

Functions and tuples

A *tuple* t maps each variable x of $\mathcal{V}$ to a value t_x ranging over $\mathbb{D}$ (it is hence a closed substitution). A *(total-) function* f maps each tuple t to a value in $\mathbb{D}$, noted $f(t)$.

Laws of addition and product by an element of $\mathbb{D}$ are piecewise-defined on tuples, using the ring operations on $\mathbb{D}$. We associate to each variable x a *unit* tuple $\boldsymbol{x}$ with value 1 on x and 0 elsewhere. Every tuple t may be written in the form $\sum t_x.\boldsymbol{x}$. and the tuple $t + \lambda\boldsymbol{x}$ is obtained by increasing the value of t on x by λ.

Interval Evaluation

In some places we shall need to compute the *range* of a function f. We shall use upper case (F) to denote this range. Computing the exact range is a difficult task in general, but *overestimates* shall be suitable for our needs. Safe overestimates can be obtained through Interval Arithmetics [15, 10], which associates an interval peer to each arithmetic operation (*e.g.,* $[a, b] - [c, d] = [a - d, b - c]$).

3 Abstractions for constraint properties

We select a set of properties which are of special relevance to numerical problem-solving, namely *monotonicity*, *convexity*, and *injectivity*. We define representations for these properties.

3.1 Properties considered

Next figure (2) illustrates these definitions in the univariate case.

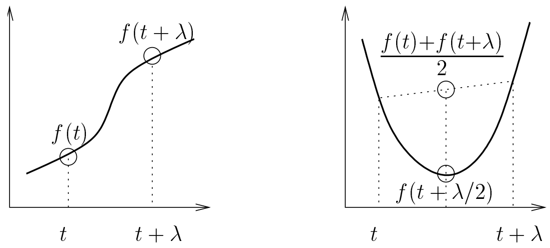

Fig. 2. An *increasing* function (left) - which is also *injective* - and an *upper-convex* one (right).

Monotonicity

Monotonicity means that increasing the value of a tuple on some variable x always yields higher (*resp.* lower) values - informally: "when x increases, so does (*resp.* so does not) $f(x)$". More formally:

Definition 1. [Monotonicity] *A function f is said to be:*

- increasing *on x if (for each tuple t)*
  $$\forall \lambda > 0,\ f(t + \lambda \boldsymbol{x}) \geq f(t)$$
- decreasing *on x if (for each tuple t)*
  $$\forall \lambda > 0,\ f(t + \lambda \boldsymbol{x}) \leq f(t)$$
- independent *on x if it is both increasing and decreasing*
  $$\forall \lambda > 0,\ f(t + \lambda \boldsymbol{x}) = f(t)$$
- monotonic *on x it is either increasing or decreasing.*

Convexity

Convexity is a second-order property of monotonicity, which states that the *slopes* of the function are monotonic - informally: convex functions have a specific $\cup$ or $\cap$ shape.

Definition 2. [Convexity] *A function f is said to be:*

- upper-convex *on x if (for each tuple t)*
  $$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2}\boldsymbol{x}) \ \geq\ \tfrac{1}{2}.(f(t) + f(t + \lambda \boldsymbol{x}))$$
- lower-convex *on x if (for each tuple t)*
  $$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2}\boldsymbol{x}) \ \leq\ \tfrac{1}{2}.(f(t) + f(t + \lambda \boldsymbol{x}))$$
- affine *on x if it is both lower- and upper- convex*
  $$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2}\boldsymbol{x}) \ =\ \tfrac{1}{2}(f(t) + f(t + \lambda \boldsymbol{x}))$$

– convex *on x if it is either upper- or lower- convex.*

Injectivity

The ability to express *strict* inequalities is recovered via the *injectivity* property, stating that no two points have the same image, and which we define as a separate property for technical convenience:

Definition 3. [**Injectivity**] *A function f is said to be:*

– *injective on x if (for each tuple t)*
$$\forall \lambda > 0,\ f(t + \lambda \boldsymbol{x}) \neq f(t)$$

Note that all these definitions are n-dimensional generalizations of the univariate definitions illustrated by figure 2, where the property considered holds for a given *axis*. In particular, our definition of the *convexity* property is weaker than the one used in non-linear optimization. It corresponds to several notions (*row-*, *interval-*, and *axis-* convexity [6, 1, 17]) investigated in the literature on constraints.

3.2 Representing properties

Properties represent classes of functions. For instance, several classes are defined with respect to monotonicity on a given variable: increasing and decreasing functions, functions which have none of these properties and functions for which both properties hold (*i.e., independent* functions).

To reason on properties, classes of functions are represented by *abstract symbols*. Operations on values can be extended to these symbols while preserving well-defined semantic properties. Following [3], the meaning of each abstract symbol we introduce is defined as the *concrete* class of functions it represents.

Definition 4. [**Abstract Symbol**] *We define the following sets of abstract symbols (the associated semantics is stated informally):*

Property	*Abs. Symb.*	*Meaning*
Monotonicity	$\oslash$	*increasing*
	$\obslash$	*decreasing*
	$\circledequal$	*independent*
Convexity	$\circledcup$	*upper-convex*
	$\circledcap$	*lower-convex*
	$\otimes$	*affine*
Injectivity	$\circledneq$	*injective*

Each set is completed with the $\circledtop$ and $\circledbot$ symbols, which respectively represent the class of all functions (with no special property) and the empty class of functions. Symbols for each property are partially ordered as depicted in Figure 3.

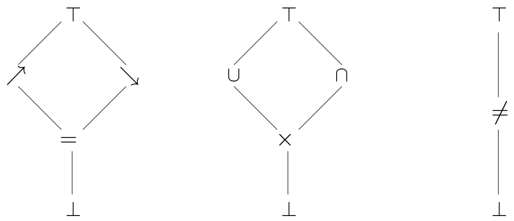

Fig. 3. Abstract lattices for the properties of monotonicity (left), convexity (middle), and injectivity (right).

We use the circled notation throughout the text to avoid confusion between the property symbols and the other signs used in the paper. Note that the ordering on symbols mimics the inclusion order on the classes they represent - for instance the class of affine functions is contained in the class of upper-convex ones.

This ordering allows us to define the usual lattice operations: the least upper bound or *lub* ($a \vee b = \min\{c \mid c \geq a \text{ and } c \geq b\}$) and the greatest lower bound or *glb* ($a \wedge b = \max\{c \mid c \leq a \text{ and } c \leq b\}$). For instance, on the lattice of monotonicity properties, $\obslash \vee \oslash$ gives $\textcircled{\scriptsize $\top$}$ and $\obslash \wedge ⊜$ gives ⊜.

Since our lattices are symmetric, we denote $-\alpha$ the symmetric of property α; for instance $-\oslash$ gives $\obslash$, and $-$⊜ gives ⊜.

Combining properties

In Abstract Interpretation, one can use the cartesian product of several lattices to obtain a finer set of properties [4]. Figure 4 gives a sample (in the univariate case) of the wide range of function properties expressed in our framework using a *triplet* of symbols, standing for monotonicity, convexity, and injectivity respectively.

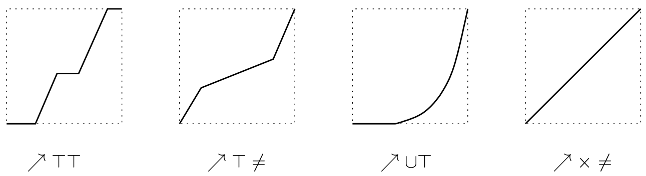

Fig. 4. Several kinds of increasing functions.

4 Reasoning on properties

Now that properties are reified as symbols on which anything we need is defined (in our case, a partial ordering), we can manipulate these objects and make deductions. Quite naturally, the aim of these deductions shall be to infer properties which *provably* hold for some functions. There exists several ways to ensure *correctness* of the deductions. We shall adopt the following definition (which is strongly related to the correctness of constraint-solvers defined, for instance, in [1]):

Definition 5. **[Correctness]** *Let α be an* abstraction *function and γ be the* concretization *function, which maps each symbol to the set of functions it represents. The abstraction is correct if α and γ form a* Galois connection *[3] between the concrete and abstract lattices i.e., (F stands for a class of functions, s is a symbol)*

$$\alpha(F) \leq s \quad\Leftrightarrow\quad F \subseteq \gamma(s) \tag{1}$$

4.1 The rule-based framework

Deductions over properties are performed using a rule-based framework which we now describe. Rules manipulate *facts* of the following form:

Definition 6. **[Fact]**. *A* fact *is a sentence of the form*

$$f \texttt{ is } P \texttt{ on } x \texttt{ when } C$$

where f is an expression, P is an abstract symbol (property), x is a problem variable, and C is a list of conditions of the form $y \in [l, r]$.

Note that we use a pseudo-natural notation for these predicates to better reflect their intended meaning and to ease readability — a more machine-oriented notation could be $P\texttt{(}f\texttt{,}x\texttt{,}C\texttt{)}$.

The *correctness* of fact is obtained from equation 1: $P \leq P' \Leftrightarrow \phi \in \gamma(P')$, where ϕ is the restriction of function f to the space delimited by the restraints of C. An example of (correct) fact is "e^x `is` $\varnothing$ $\textcircled{\cup}$ $\textcircled{\neq}$ `on` x `when` $x \in [-\infty, +\infty]$" — we use several properties P at once to shorten the notations.

Definition 7. **[Matching]**. *A fact a:"f `is` P_1 `on` x `when` $y_n \in I_n$" matches a fact b: "g `is` P_2 `on` x `when` $z_n \in J_n$" if $f \succeq g$, $P_1 \geq P_2$ and if the interval bounding each variable in fact a is tighter ($\subseteq$) than the one in fact b (if any).*

Intuitively: a fact holds if another fact stating a property at least as strong does, and if it does over an interval at least as large. For instance the fact a: "f `is` $\textcircled{\cup}$ `on` x `when` $[1, 2]$" matches the fact b: "f `is` $\otimes$ `on` x `when` $[0, \infty]$", because if b holds then so does a, and so does any consequence of a. This allows us to define *rules*:

Definition 8. **[Rule]**. *A rule is a Horn clause on facts, i.e., a formula of the form $F_1, \ldots, F_n \Rightarrow G$, where G and all F_i's are facts.*

A rule applies *if all its premises match some facts which are known from the knowledge base. This application infers a new fact which is added to the base.*

Of course, *correct* rules infer correct facts from correct facts. Adding a fact to the base can be seen as a process which cumulates information in a monotonic fashion, with the $\wedge$ operation used as a conjunction. Our framework consists in a set of *elementary facts* (axioms) and a set of *rules*. A fact is *provable* in our framework if it is deduced from the facts by a finite number of rules applications.

4.2 Elementary facts

Primitive functions like addition or square root can easily be decomposed into a small number of parts where properties can be identified. Figure 5 summarizes the elementary facts for these functions.

exp.	*part*	*abs. on x*	*exp.*	*part*	*abs. on x*
e^x	$\mathbb{D}$	$\oslash\circledcup\circledneq$	$\log(x)$	$\mathbb{D}^+$	$\oslash\circledcap\circledneq$
x^{2a}	$\mathbb{D}^-$	$\obslash\circledcup\circledneq$	x^{2a+1}	$\mathbb{D}^-$	$\oslash\circledcap\circledneq$
	$\mathbb{D}^+$	$\oslash\circledcup\circledneq$		$\mathbb{D}^+$	$\oslash\circledcup\circledneq$
$\sqrt[2a]{x}$	$\mathbb{D}^+$	$\oslash\circledcap\circledneq$	$\sqrt[2a+1]{x}$	$\mathbb{D}^-$	$\oslash\circledcup\circledneq$
				$\mathbb{D}^+$	$\oslash\circledcap\circledneq$

exp.	*part*	*abs. on x*	*abs. on y*
$x+y$	$\mathbb{D}\times\mathbb{D}$	$\oslash\otimes\circledneq$	$\oslash\otimes\circledneq$
$x-y$	$\mathbb{D}\times\mathbb{D}$	$\oslash\otimes\circledneq$	$\obslash\otimes\circledneq$
$x.y$	$\mathbb{D}^+\times\mathbb{D}^+$	$\oslash\otimes\circledneq$	$\oslash\otimes\circledneq$
	$\mathbb{D}^+\times\mathbb{D}^-$	$\obslash\otimes\circledneq$	$\oslash\otimes\circledneq$
	$\mathbb{D}^-\times\mathbb{D}^+$	$\oslash\otimes\circledneq$	$\obslash\otimes\circledneq$
	$\mathbb{D}^-\times\mathbb{D}^-$	$\obslash\otimes\circledneq$	$\obslash\otimes\circledneq$

Fig. 5. Abstractions for the basic operators

We just give one example of how to prove these results. The generalization is straightforward:

Rule 1 $x.y$ `is` $\oslash\otimes\circledneq$ `on` x `when` $x \in \mathbb{D}^+$, $y \in \mathbb{D}^+$.

Proof. Consider particular values $x, x' \in \mathbb{D}^+$ *and* $y \in \mathbb{D}^+$, *with* $x < x'$. *Since the product is positive, we have* $x.y \leq x'.y$ *(*$\oslash$*); furthermore, since both* x *and* y *are non-zero, we have* $x.y \neq x'.y$ *(*$\circledneq$*); Last, we have* $(x.y + x'.y)/2 = (x.y)/2 + (x'.y)/2$ *(*$\otimes$*).*

5 Deduction rules

Rules make it possible to deduce more facts from the elementary ones. We describe two categories: rules to deduce properties over domains containing 0 (these are typically not handled in Figure 5), and rules for complex functions (more categories could be conceived).

5.1 Rules for extended domains

When some properties hold over two contiguous intervals $[l, m]$ and $[m, r]$, facts can sometimes be asserted for $[l, r]$. This is easy for monotonicity, less easy for convexity, and false for injectivity.

Rule 2 f `is` $\oslash$ `on` x `when` $x \in [l, m]$,
f `is` $\oslash$ `on` x `when` $x \in [m, r] \Rightarrow$
f `is` $\oslash$ `on` x `when` $x \in [l, r]$.
(a similar rule holds for $\obslash$*).*

Proof. *Straightforward.*

Rule 3 f `is` $\obslash\textcircled{$\cup$}$ `on` x `when` $x \in [l, m]$,
f `is` $\oslash\textcircled{$\cup$}$ `on` x `when` $x \in [m, r] \Rightarrow$
f `is` $\textcircled{$\cup$}$ `on` x `when` $x \in [l, r]$.
(same rule for $\textcircled{$\cap$}$*, reversing* $\textcircled{$\cup$}$ *and* $\textcircled{$\cap$}$*).*

Proof. *(sketched) Upper-convexity means the slope is increasing. Since it is negative on* $[l, m]$*, positive on* $[m, r]$*, and increasing on both, it is increasing all over* $[l, r]$*.*

5.2 Rules for complex functions

Unary functions

Complex functions are built by "plugging" basic functions from Figure 5 one into another (recall that $f(g)$ denotes "g plugged into f", while $f(x)$ denotes a unary expression linked to variable x). Some properties are preserved by this plug-in, for instance the composition of injective functions is injective:

Rule 4 g `is` $\textcircled{$\neq$}$ `on` x `when` $x \in I$,
$f(y)$ `is` $\textcircled{$\neq$}$ `on` y `when` $y \in G(I) \Rightarrow$
$f(g)$ `is` $\textcircled{$\neq$}$ `on` x `when` $x \in I$.

Proof. *Consider any instantiation of the variables other than* x *(*g *is hence considered a unary function of* x*). If we take two different values for* x*, then the results for* $g(x)$ *are also different. Since the term* g *is "plugged" into each variable* y*, the results for* f *are also different, and* $f(g)$ *is injective on* x*.*

Rule 5 *For each monotonicity property* M*, we have the rule:*
g `is` M `on` x `when` $x \in I$,
$f(y)$ `is` $\obslash$ `on` y `when` $y \in G(I) \Rightarrow$
$f(g)$ `is` $-M$ `on` x `when` $x \in I$.
(same rule for $\oslash$ *with "*$f(g)$ *is* M*").*

Proof. *Same idea as for rule 4; let* f *and* g *be (say) both decreasing on* x *and* y*. Increasing* x *decreases* y*, which increases* $f(g)$*.*

Rule 6 g `is` $\circledcup$ `on` x `when` $x \in I$,
$f(y)$ `is` $\circledcup\varnothing$ `on` y `when` $y \in G(I) \Rightarrow$
$f(g)$ `is` $\circledcup$ `on` x `when` $x \in I$.
(same rule for $\circledcap$ with $\circledbslash$).

Proof. (sketched) For each $t \in \Omega$ and $\lambda > 0$, we note $t' = t + \lambda \boldsymbol{x}$ and $avg(t, t') = (t + t')/2$. We have $avg(fgx, fgy) \geq f(avg(gx, gy)) \geq f(g(avg(x, y)))$.

Binary functions

In the easy case, *i.e.,* when the considered variable occurs in just one of its leaves, binary functions are treated exactly like a unary one: for each unary rule of the form:

$f(x)$ `is` M_1 `on` x `when` $x \in I, \ldots \Rightarrow$
$f(g)$ `is` M_2 `on` x `when` $x \in J$

we can add left- and right-handed binary counterparts, of the following form (x should not appear as a subterm of h):

$f(x, h)$ `is` M_1 `on` x `when` $x \in I, \ldots \Rightarrow$
$f(g, h)$ `is` M_2 `on` x `when` $x \in J$

And similarly for the right-hand side. Things get a little tricky when variable x occurs in both leaves of f. For instance, increasing x may increase one leaf and decrease the other. For some operations, the overall properties can become unpredictable.

Rule 7 *Let M_1 and M_2 be 2 monotonicity or 2 convexity properties:*
$f(g, z)$ `is` M_1 `on` x `when` $x \in I$,
$f(y, h)$ `is` M_2 `on` x `when` $x \in I \Rightarrow$
$f(g, h)$ `is` $M_1 \vee M_2$ `on` x `when` $x \in I$

Proof. (sketched) If $f(y, h)$ and $f(g, z)$ are (say) both increasing on x, this means that increasing x changes g and h "in the right direction", i.e., the direction which increases f. The same reasoning holds for convexity.

5.3 Examples of deductions

We now give a simple example of the kind of deductions performed by our rules.

Example 1. Consider $f : (x, y) \to x.(y + \log x)$, where x and y range over $[1, 10]$. f is:

increasing on x: Since $\log(x)$ is $\varnothing$ on x when $x \in [1, 10]$ (elementary fact) and $y + z$ is $\varnothing$ on $y \in [1, 10], z \in [0, 1]$, we deduce that $y + \log x$ is $\varnothing$ on x when $x \in [1, 10], y \in [1, 10]$ (rule 5, binary). $x.z$ is also $\varnothing$ on x over $[1, 10]$, hence (rule 7), $x.(y + \log x)$ is $\varnothing \vee \varnothing = \varnothing$.
upper-convex on y: Since the term $\log(x)$ does not contain any occurrence of y, $y + \log(x)$ is $\otimes$ on y. $x.z$ is $\varnothing\otimes$ on z, hence $x.(y + \log(x))$ is $\varnothing$ (rule 6, binary).

Increasing monotonicity on y and upper-convexity on x can also be deduced using similar applications of the rules.

5.4 Discussion

We have restricted the paper to a simple and clearly defined type of rules, but further precision can be gained by simple extensions of the framework. For instance, integrating reasoning on *partial derivatives* requires rules of the following kind:

$$\begin{array}{l} \frac{\delta f}{\delta x} \texttt{ is } \bigoplus \texttt{ when } x \in I \Rightarrow \\ \quad f \texttt{ is } \varnothing \texttt{ on } x \texttt{ when } x \in I \end{array}$$

where $\bigoplus$ is an additional abstract symbol for *signs* [14], which represents positiveness. Injectivity can be detected by *strict* positiveness/negativeness, while rules for convexity require second-order derivatives. Interval evaluation of these derivatives can be used to provide guaranteed computations [2].

6 Practical relevance

Our research was motivated by a practical goal, namely the integration of knowedge on properties within the constraint-solving process, but we have mainly been discussing the property-inference part. We review some of the existing specialized algorithms that fit in with our framework, and we give examples of some of the new applications we foresee.

Integrating specialized solvers

The issue of determining classes of constraint problems which can be solved efficiently by specialized algorithms goes back to [7]. On numerical problems, the properties of monotonicity and convexity have naturally been exploited. Convex constraints have been considered in [17, 6]; especially relevant to numerical domains is the $(3, 2)$-relational-consistency technique [17], which guanrantees backtrack-free search. Improvements of Arc-Consistency for monotonic constraints can be found in *e.g.,* [18]. Both kinds of properties have been widely studied in non-linear optimization, for instance *local* optimization algorithms become *global* when a strong notion of convexity holds [16].

Making several algorithms cooperate is known to be a key feature for efficient constraint solving [9], and the integration of such customized algorithms raises interesting issues. A more efficient cooperation could be obtained if the solver were able to deduce which properties hold for which sub-problems. We hope that our contribution shall be a step towards these intelligent cooperation architectures.

Novel applications

Having properties defined as objects makes it easier to define new algorithms which manipulate this information. As an example of the possible algorithms that can be devised, we briefly describe a simple improvement of optimization and interval evaluation, which we illustrate by the following example:

Example 2. Consider the function $f : (x, y) \rightarrow 2x.y + y$ on $[-10, 10] \times [1, 2]$. The function turns out to be increasing on x over these intervals. We can hence deduce that its global minimum is obtained for $x = -10$.

When we instantiate x to value -10, f becomes the simpler function $g : (x, y) \rightarrow -20.y+y$. Function g now turns out to be decreasing on y when $x = -10$ and $y \in [1, 2]$, and the global minimum is hence obtained when $y = \min_y = 2$.

We have proved, without any use of local search or enumeration, that the global minimum of the function is the tuple $(-10, 2)$. This gives a reliable and precise lower bound for interval evaluation.

The idea behind this example is related to monotonicity: if a function is monotonic on some variable x, we can easily determine the bound of the interval of values for x where the optimal value is found. This allows eliminating the variable, which entails a simplified problem, and we can even start the process over recursively.

7 Conclusions

Constraint properties are a meaningful information which determines the practical tractability of a Constraint Satisfaction Problem. Integrating this information in constraint solvers requires a formal representation (or "reification") of the properties. The contribution of this paper was to show that some form of automated reasoning was possible over properties, leading to perspectives in "property-guided" constraint-solving.

We have shown that *abstraction* provides a convenient representation with a clear semantics. Our main reference was Abstract Interpretation, though other works on abstraction [8] could also be considered. Inference on these abstractions has been formalized using simple deduction rules. We have exemplified this principle with simple syntax-driven rules, and we have suggested how the theory can be enriched. Finaly, we have overviewed some of the actual or possible applications of property-inference for numerical constraint-solving and optimization. It is the aim of our outgoing work to develop these applications.

Acknowledgments

We thank E. Petrov for helpful discussions and advice.

References

1. F. Benhamou and W. Older, 'Applying Interval Arithmetic to real, integer and boolean constraints', *J. of Logic Programming*, **32**(1), 1–24, (1997).
2. G. F. Corliss and L. B. Rall, 'Bounding derivative ranges', in *Encyclopedia of Optimization*, eds., P. M. Pardalos and C. A. Floudas, Kluwer, Dordrecht, (1999).
3. P. Cousot and R. Cousot, 'Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints', in *Conf. Record of the 4th POPL*, pp. 238–252, Los Angeles, California, (1977). ACM Press, New York, NY.

4. P. Cousot and R. Cousot, 'Abstract interpretation and application to logic programs', *J. of Logic Programming*, **13**(2–3), 103–179, (1992).
5. E. Davis, 'Constraint propagation with interval labels', *Artif. Intel.*, **32**, 281–331, (1987).
6. Y. Deville, O. Barette, and P. Van Hentenryck, 'Constraint satisfaction over connected row convex constraints', *Artif. Intel.*, **109**, 243–271, (1999).
7. E. Freuder, 'A sufficient condition for backtrack-free search', *J. of the ACM*, **29**, 24–32, (1982).
8. F. Giunchiglia and T. Walsh, 'A theory of abstraction', *Artif. Intel.*, **56**(2-3), 323–390, (1992).
9. L. Granvilliers, E. Monfroy, and F. Benhamou, 'Symbolic-interval cooperation in constraint programming', in *Int. Symp. on Symbolic and Algebraic Computation*, London, Ontario, (2001). ACM Press.
10. E. R. Hansen, *Global Optimization Using Interval Analysis*, Pure and Applied Mathematics, Marcel Dekker Inc., 1992.
11. E. Hyvönen, 'Constraint reasoning based on interval arithmetic', in *Int. Joint Conf. On Artif. Intel.*, pp. 1193–1198, Detroit, US, (1989). Morgan Kaufmann.
12. C. Kirchner and H. Kirchner, 'Rewriting, Solving, Proving', Technical report, LORIA, (2001). draft of a textbook.
13. A. Mackworth, 'Consistency in networks of relations', *Artif. Intel.*, **1**(8), 99–118, (1977).
14. K. Marriott and P.-J. Stuckey, 'Approximating interaction between linear arithmetic constraints', in *Int. Symp. on Logic Programming*, ed., M. Bruynooghe, pp. 571–585, Ithaca, NY, (1994). MIT Press.
15. R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1966.
16. S. Nash and A. Sofer, *Linear and Nonlinear Programming*, McGraw Hill, 1996.
17. D. Sam-Haroud and B. V. Faltings, 'Consistency techniques for continuous constraints', *Constraints*, **1**(1-2), 85–118, (1996).
18. Z. Yuanlin and H. C. Yap, 'Arc-consistency on n-ary monotonic and linear constraints', in *6th Int. Conf. on Constraint Programming*, ed., R. Dechter, pp. 470–483, Singapore, (2000). LNCS.