Fundamenta Informaticae 39 (1999) 167–187
IOS Press

An Open Automated Framework for Constraint Solver Extension: the **SoleX** Approach

Eric Monfroy*
CWI, P.O. Box 94079
NL-1090 GB, Amsterdam, the Netherlands.
e-mail: Eric.Monfroy@cwi.nl

Christophe Ringeissen†
LORIA - INRIA, 615 rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France.
e-mail: Christophe.Ringeissen@loria.fr

Abstract. In declarative programming languages based on the constraint programming paradigm, computations can be viewed as deductions enhanced with the use of constraint solvers. However, admissible constraints are restricted to formulae handled by solvers and thus, declarativity may be jeopardized. We propose a domain-independent scheme to extend constraint solvers so that they can handle *alien constraints*, i.e., constraint involving new function symbols. This mechanism, called **SoleX**, consists of a set of symbolic rule-based transformations: they add and deduce syntactical as well as semantic information related to alien constraints, complete the computation domain, and purify constraints in order to allow solvers to cope with alien constraints. These transformations can be seen as elementary solvers, and thus, **SoleX** is a collaboration of these several solvers with the initial solver. Some extensions of computation domains have already been studied to demonstrate the broad scope of **SoleX** potential applications.

Keywords: Solver Extension, Constraint Solver Design, Constraint Solving, Programming with Constraints.

*Address for correspondence: CWI, P.O. Box 94079, NL-1090 GB, Amsterdam, the Netherlands
†Address for correspondence: LORIA - INRIA, 615 rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy Cedex, France

1. Introduction

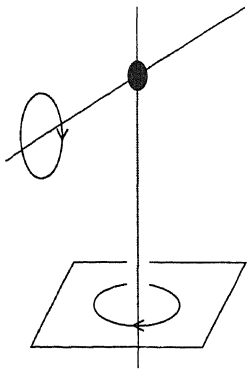
Since the mid-eighties, constraint programming (CP) [12, 7, 25] has emerged as an interesting style of programming. The framework of this new paradigm is based on the separation between a programming language to generate requirements (the *constraints*) on objects (the *computation domain*), and a mechanism for computing solutions of these constraints (the *solver*). CP has to face the dilemma "declarativity vs. efficiency": "efficient" solvers are generally specialized and restricted to some classes of constraints, while "declarative" solvers can handle wider classes of problems but are generally inefficient. Thus, a compromise solution consists in considering solvers that cannot always handle all the constraints the user manipulate in the programming languages. A solver is said to be *complete* if it is able to solve any constraint defined by the language. However, for declarativity reasons, solvers of CP systems are not always complete: for example CLP($\mathcal{R}$) [14, 11, 13] allows one to specify but not to solve non-linear constraints, i.e., they are suspended till they eventually become linear. Although this kind of technique is sufficient for some applications, it is not satisfactory in the general case.

Designing a solver that handles (and preferably efficiently) all the constraints provided in the programming language is a hard, tedious, and lengthy task. Possibly, there may be no solver for this computation domain. Thus, in order to increase the declarativity of a constraint programming system without jeopardizing its efficiency, we are concerned with a general framework and mechanisms for extending (and eventually completing) efficient solvers so they can handle new function symbols. In [9], a decision procedure on $\mathcal{R}$ is extended to a decision algorithm on $\mathcal{R}+\mathcal{M}$ [1]. In this paper, we extend the method of [9] with other syntactical manipulations and semantic transformations as well. Moreover, our framework is independent from the computation domain and the programming language.

The aim of this paper is to present **SoleX**, a generic (i.e., independent from the domain computation and from the extra function symbols) scheme for extending constraint solvers ; generalizes and formalizes some of the previous works. **SoleX** enables one increasing the declarativity of CP systems without jeopardizing the completeness of the solvers, and without designing new solvers from scratch. The aim of **SoleX** is to enrich solvers with symbolic computation so they can treat new function symbols called *alien* symbols. **SoleX** consists of a set of *solver extensions* (i.e., meta transformation rules) together with a scheduler to control their application and the execution of the solver to be extended (called the *initial solver*). The solver extensions aim at processing alien terms and constraints using alien terms such that these terms and constraints can be handled by the initial solver (syntactical solver extensions), or such that information carried by these terms and constraints becomes understandable by the initial solver (semantic solver extensions). The solver extensions are either derived from some standard features/properties of the domain of computation, or fed with transformation rules specified by the user for a given domain, solver, and alien function symbols.

The paper is organized as follows. Section 3 formalizes our framework. Section 4 describes

[1] CLP($\mathcal{R}+\mathcal{M}$) is obtained by extending the domain of CLP($\mathcal{R}$) with some special nonarithmetic function symbols.



$$
\begin{cases}
\cos(\delta_1) * \cos(\delta_2) - \cos(\phi) * \cos(\theta) * \cos(\psi) + \sin(\phi) * \sin(\psi) = 0 \\
\sin(\delta_1) * \cos(\delta_2) - \sin(\phi) * \cos(\theta) * \cos(\psi) - \cos(\phi) * \sin(\psi) = 0 \\
\sin(\delta_2) + \sin(\theta) * \cos(\psi) = 0 \\
-\cos(\delta_1) * \sin(\delta_2) - \cos(\phi) * \cos(\theta) * \sin(\psi) + \sin(\phi) * \cos(\psi) = 0 \\
-\sin(\delta_1) * \sin(\delta_2) + \sin(\phi) * \cos(\theta) * \sin(\psi) - \cos(\phi) * \cos(\psi) = 0 \\
\cos(\delta_2) - \sin(\theta) * \sin(\psi) = 0 \\
\sin(\delta_1) - \cos(\phi) * \sin(\theta) = 0 \\
-\cos(\delta_1) - \sin(\phi) * \sin(\theta) = 0 \\
\cos(\theta) = 0 \\
l_2 * \cos(\delta_1) * \cos(\delta_2) - p_x = 0 \\
l_2 * \sin(\delta_1) * \cos(\delta_2) - P_y = 0 \\
l_2 * \sin(\delta_2) + l_1 - p_z = 0
\end{cases}
$$

Figure 1. Robot-arm with two degrees of freedom

the (rule-based) elementary solvers: semantic solvers (Section 4.2) and syntactical solvers (Section 4.1). The related transformation rules are formally given in Section 4.3. We then examine (Section 5) the control of solvers. Section 6 describes some applications of **SoleX** over different domains. Finally, comparisons, conclusions and future works are discussed in Section 7.

2. Motivations

The general problematics is the following: we have a solver available (the initial solver) that is able to handle a definite language of constraints. For some reasons (such as declarativity of the programming language), we want to use some extra function symbols (the alien symbols). Thus, we need a solver that can manipulate a larger language of constraints. However, we do not want to implement a new solver from scratch, but we want to extend the solver we have at disposal without modifying it (we consider it as a black-box).

The inverse robot kinematics problem [4] illustrates our motivations. We want to determine, for a given robot, a position and an orientation of the end-effector, the distances at the prismatic joints and the angles at the revolute joints (see Figure 1). The problem for a robot having two degrees of freedom can be described by a system of equations (see Figure 1) that also involves trigonometric functions[2]. The solution we look for is a symbolic expression describing the relation between parameters and variables. However, neither trigonometric solvers nor trigonometric simplifications automatically return the solution we expect. Thus, we would like to extend a solver for non-linear polynomial constraints (such as Gröbner bases computation) with the trigonometric functions *sine* and *cosine*. We could solve "by hand" this example in the following way. First, for each angle α, we replace (abstract) $\cos(\alpha)$ and $\sin(\alpha)$ by two new variables C_α and S_α respectively. Then, for each C_α and S_α, we add a new constraint $C_\alpha^2 + S_\alpha^2 = 1$. We apply Gröbner bases computation on this new set of constraints. Finally, we

[2]The parameters and variables are described in Section 6.

remove all the constraints we have added, and we replace C_α and S_α with $\cos(\alpha)$ and $\sin(\alpha)$ respectively. Hence, we obtain the expected solution. Since the problem is rather small and the transformations are quite simple, we can imagine doing it by hand, and **SoleX** is just a way to automate the mechanism. However, when dealing with bigger problems, and numerous more complex transformations, and when solving the problem requires reaching a fixed point of such mechanisms, it is no more conceivable that the solver extension can be realized by hand, and this justify an automated process.

Let us give a second example. One may have to consider unification problems together with constraints on depths of ground terms, i.e., constraints such as

$$depth(X) = 4 - depth(g(Z)) \ \wedge \ g(g(Z)) = g(g(g(b))) \ \wedge \ g(Y) = g(g(X)).$$

We have here two disjoint sorts: terms (solved by unification) and integers. Since no solver over the integers can handle the function *depth*, we want to extend a Diophantine solver (or a finite domain like solver) with the *depth* function. Let us forget that solving this problem by hand requires numerous transformations. We can determine what kinds of transformations can be used (such as the *depth* of a constant is 1, or the $depth(g(X))$ is $1 + depth(X)$), but we do not know if they are sufficient, and in which order to apply them. Thus, we are not sure to reach a fixed-point, neither to get a convenient solution. That introduces two other requirements: the framework for solver extensions must ease the integration and the classification of several kinds of transformations, and must provide some tools to ensure termination of the extension process.

Nowadays, some methods may be investigated for extending solvers. Solver combination methods [1, 16, 22, 23] aim at designing a general solver for a union of theories (that corresponds to a new mixed domain which is a conservative extension of the original ones) based on the cooperation of elementary solvers, each one already defined for one of the theories. Since we want to stay on the same interpretation domain, such frameworks are not well suited. Independently of these theoretical results, more practical issues have been explored for the cooperation of several solvers on a single domain [21, 5, 2, 8, 17], or on several domains [18, 20, 19]. However, in such systems, extra function symbols cannot be directly handled.

The problem of integrating deduction techniques into computer algebra has attracted considerable interest [24, 10, 3], and standard computer algebra systems (such as Mathematica [26]) already provides some equation simplification tools. Although they are powerful, no methods/techniques are available for designing a solver extension or to ensure its soundness. Similar comments can be done about CHRs [6] and ELAN [15][3]. Some works were also conducted in the area of constraint transformation [18], but these techniques act only as a pre-processing.

To overcome the problems of solver extension, we designed **SoleX**, a framework together with a mechanism for extending and completing constraint solvers. **SoleX** facilitates the design and the classification of symbolic transformations and also automates their application, in order to enrich solvers so they can treat alien symbols. The semantics of the alien symbols that

[3] However, these systems are really well suited for implementing our framework.

can be introduced using **SoleX** can be of different kinds. First, they can be syntactic sugar (or parameterization) to replace the extensional definition of a function (e.g., $3.x^2 + 2.x + 1$ may be named $p(x)$. Then, $3.y^2 + 2.y + 1$ will be denoted $p(y)$.). Second, they can be standard functions not handled by the solver. For example, usual solvers for arithmetic constraints cannot manipulate the functions *sine* and *cosine*. Unlike to the previous two cases, the last class of alien symbols corresponds to functions with no defined meaning on the domain. For example, a function can be characterized by experimental measures that can be expressed as constraints. The solved form may define the extensional definition of the function (or of a class of functions).

SoleX is the ordered application of four phases (collections of solver extensions fed with transformation rules designed by the user) to process the alien function symbols and deduce related information. The Reduction phase reduces the search space by adding semantic and syntactical information carried by the alien functions. The Expansion phase completes the constraints with always valid (w.r.t. the extended domain) constraint, i.e., characteristics of the functions (e.g., an absolute value is always greater than or equal to zero). Then, before the Solving phase (application of the initial solver), the constraint store is purified by abstracting remaining alien function symbols. After application of the initial solver, the Contraction phase replaces abstraction variables with their related alien terms (this is the "opposite" of abstraction) and removes "redundancies" added by the expansion phase. Several applications of **SoleX** may be necessary to reach a fixed point and to solve the constraints.

3. Basic Concepts

Let us first introduce some standard notations about terms and substitutions of variables by terms. Given a first-order signature Σ and a denumerable set $\mathcal{V}$ of variables, $T(\Sigma, \mathcal{V})$ denotes the set of $\mathcal{F}_\Sigma$-terms with variables in $\mathcal{V}$. Terms (resp. variables) are denoted by $t_1, \ldots, t_n$ (resp. $x_1, \ldots, x_n$). A *ground* term is a term without variables. The terms $t_{|\omega}$, $t[s]_\omega$ and $t[\omega \leftarrow s]$ denote respectively the subterm of t at the position ω, the term t with the subterm s at the position ω and the replacement in t of $t_{|\omega}$ by s. The symbol of t occurring at the position ω (resp. the top symbol of t) are written $t(\omega)$ (resp. $t(\epsilon)$). The term $t[s]$ denotes a term t with some subterm s. The term $t[s \leftarrow u]$ denotes the term where s is replaced by u in *all* occurrences of s in t. $\mathcal{V}(t)$ denotes the set of variables occurring in the term t. A *substitution* $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an assignment from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$. We use letters $\sigma, \mu, \gamma, \phi, \ldots$ to denote substitutions. The application of a substitution σ to a term t is written in postfix notation $t\sigma$.
We now define the objects handled by **SoleX**: solvers, and constraint systems.

Definition 3.1. (Constraint system) A constraint system is a 4-uple $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ where:

- Σ is a first-order signature given by a set of function symbols $\mathcal{F}_\Sigma$, and a set of predicate symbols $\mathcal{P}_\Sigma$,
- $\mathcal{D}$ is a Σ-structure (its domain is denoted by $|\mathcal{D}|$),
- $\mathcal{V}$ is an infinite denumerable set of variables,

- $\mathcal{L}$ is a set of *constraints*: it is a non-empty set of $(\Sigma, \mathcal{V})$-atomic formulas closed under conjunction and disjunction. The unsatisfiable constraint is denoted by $\bot$ and the truth constraint is denoted by $\top$. An assignment is a mapping $\alpha : \mathcal{V} \to |\mathcal{D}|$. The set of all assignments is denoted by $ASS_{\mathcal{D}}^{\mathcal{V}}$. An assignment α extends uniquely to a homomorphism $\underline{\alpha} : T(\Sigma, \mathcal{V}) \to \mathcal{D}$. The set of solutions of a constraint $c \in \mathcal{L}$ is the set $Sol_{\mathcal{D}}(c)$ of assignments $\alpha \in ASS_{\mathcal{D}}^{\mathcal{V}}$ such that $\underline{\alpha}(c)$ holds. A constraint c is valid in $\mathcal{D}$ (denoted by $\mathcal{D} \models c$) if $Sol_{\mathcal{D}}(c) = ASS_{\mathcal{D}}^{\mathcal{V}}$.

The enrichment of a constraint system CS consists of some additional functions defined on the original domain. The interpretation of symbols previously defined in CS is unchanged.

Definition 3.2. (Constraint system enrichment) Let $CS^+ = (\Sigma^+, \mathcal{D}^+, \mathcal{V}^+, \mathcal{L}^+)$ and $CS = (\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ be two constraint systems. Then, CS^+ is an enrichment of CS if:

- $\mathcal{F}_\Sigma \subseteq \mathcal{F}_{\Sigma^+}$ and $\mathcal{P}_\Sigma = \mathcal{P}_{\Sigma^+}$
- $|\mathcal{D}| = |\mathcal{D}^+|$ and $\forall r \in \Sigma$, $r_{\mathcal{D}^+} = r_{\mathcal{D}}$, where $r_{\mathcal{D}}$ (resp. $r_{\mathcal{D}^+}$) represents the interpretation of r on the Σ-structure $\mathcal{D}$ (resp. $\mathcal{D}^+$).
- $\mathcal{V} = \mathcal{V}^+$, $\mathcal{L} \subseteq \mathcal{L}^+$

The notations $t[g]$, $t[g \hookleftarrow u]$ defined on terms can be extended on constraints as follows: $C[g]$ means that g is a term occurring in C and $C[g \hookleftarrow u]$ is the constraint obtained from C by replacing g by u. In the same way, constraints can be viewed as terms built on atomic constraints and binary connectives $\wedge, \vee$. This leads to the notion of positions and subconstraints. The constraint $C[[c]]_\omega$ denotes a constraint with a subconstraint c occurring at the position ω, and $C[[\omega \hookleftarrow c']]$ is the constraint obtained by replacing the subconstraint occurring in C at the position ω by c' . The connectives $\wedge, \vee$ satisfy the following equational axioms:

$$ACD = \begin{cases} C_1 \wedge C_2 & = & C_2 \wedge C_1 \\ C_1 \wedge (C_2 \wedge C_3) & = & (C_1 \wedge C_2) \wedge C_3 \\ C_1 \vee C_2 & = & C_2 \vee C_1 \\ C_1 \vee (C_2 \vee C_3) & = & (C_1 \vee C_2) \vee C_3 \\ C_1 \wedge (C_2 \vee C_3) & = & (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \end{cases}$$

A set of constraints $\{c_1, \ldots, c_n\}$ where $c_i \in \mathcal{L}^+$ for $i = 1, \ldots, n$ (a constraint store) is represented by a conjunction of constraints $c_1 \wedge \cdots \wedge c_n$, where c_i's are not necessarily atomic constraints. This conjunction can be split into an impure component in $\mathcal{L}^+$ and a pure component in $\mathcal{L}$. This explains why we represent any constraint in $\mathcal{L}^+$ by a pair (C, P) where $C \in \mathcal{L}^+$ and $P \in \mathcal{L}$, and (C, P) means the conjunction $C \wedge P$. If C is in $\mathcal{L}$, then (C, P) is said *pure*.

Definition 3.3. (Aliens, pure and impure constraints) A *pure* constraint (resp. term) is a constraint (resp. term) in $\mathcal{L}$. An *alien* subterm in a term t is a term with a top-symbol in $\Sigma^+ \backslash \Sigma$ such that its super-terms (whenever they exist) have top-symbols in Σ. The set of aliens in C denoted by $Alien(C)$ is the set of alien subterms of terms occurring as arguments of atomic constraints in C. A constraint C is *impure* if $Alien(C)$ is non-empty.

Intuitively, a component solver is an algorithm which transforms a constraint C into a new constraint C' "simpler" than C, but equivalent to C in the structure $\mathcal{D}$ (a solver preserves the solutions). Moreover, the repeated application of a solver always reaches a fixed-point which is a constraint in *solved form*.

Definition 3.4. (Component Solver) A *component solver* (or *solver* in short) for a constraint system $(\Sigma, \mathcal{D}, \mathcal{V}, \mathcal{L})$ is a computable function $S : \mathcal{L} \to \mathcal{L}$ s.t.:

1. $\forall C \in \mathcal{L},\ Sol_{\mathcal{D}}(S(C)) \subseteq Sol_{\mathcal{D}}(C)$ (correctness)
2. $\forall C \in \mathcal{L},\ Sol_{\mathcal{D}}(C) \subseteq Sol_{\mathcal{D}}(S(C))$ (completeness)
3. $\forall C \in \mathcal{L},\ \exists n \in \mathbb{N},\ S^{n+1}(C) = S^n(C)$

A constraint C is in solved form w.r.t. S if $S(C) = C$. A *partial* solver is a computable function S satisfying only conditions (1) and (3). A *non-deterministic solver* is $S = (S_i)_{i=1,\ldots,m}$ such that S_i is a partial solver for $i = 1, \ldots, m$ and

$$\forall C \in \mathcal{L},\ Sol_{\mathcal{D}}(C) = \bigcup_{i=1}^{m} Sol_{\mathcal{D}}(S_i(C)).$$

A constraint C is in solved form w.r.t. $S = (S_i)_{i=1,\ldots,m}$ if $S_i(C) = C$ for $i = 1, \ldots, m$. We denote by S^+, the n-th iteration S^n of S for some (unspecified) $n \geq 0$. Similarly S^* denotes the repeated application of S till reaching the solved form. If $S = (S_i)_{i,\ldots,m}$ then $S^n = (S_i^n)_{i=1,\ldots,m}$ and $S^* = (S_i^*)_{i=1,\ldots,m}$.

In the following, we represent a solver by a rule of the form:

Rule $\dfrac{C}{C'}$ if Cond

The different possibilities of Rule application induce a relation $\to_{\mathsf{Rule}}$ on $\mathcal{L} \times \mathcal{L}$. Whenever there is no infinite chain $C \to_{\mathsf{Rule}} \cdots \to_{\mathsf{Rule}} \cdots$, it is very natural to define a solver S from the relation $\to_{\mathsf{Rule}}$, where a constraint C usually relates to finitely many constraints $C_1, \ldots, C_m$ such that $C \to_{\mathsf{Rule}} C_1, \ldots, C \to_{\mathsf{Rule}} C_m$. Similarly to the definition of solvers, two kind of constraint selections can be distinguished. If $Sol_{\mathcal{D}}(C) = Sol_{\mathcal{D}}(C_i)$ for any $C \in \mathcal{L}$ and $i = 1, \ldots, m$, then the *don't care* application of Rule chooses an arbitrary C_k for $k \in [1, m]$, and $S(C) := C_k$. Otherwise, if $Sol_{\mathcal{D}}(C) = \bigcup_{i=1}^{m} Sol_{\mathcal{D}}(C_i)$ for any $C \in \mathcal{L}$ and $i = 1, \ldots, m$, then the *don't know* application of Rule corresponds to a non-deterministic solver $S = (S_i)_{i=1,\ldots,m}$ such that $S_i(C) = C_i$ for $i = 1, \ldots, m$. In both cases, if there is no constraint C' such that $C \to_{\mathsf{Rule}} C'$, then $S(C) := C$.

For sake of simplicity, we do not consider in the following non-deterministic solvers, but one should be aware that the same approach could be investigated with such solvers but at the cost of more complicated notations. It is also important to note that the non-determinism of solvers is usually implemented via a backtracking mechanism.

Assumption 3.1. S is a solver for CS.

We are interested in the design of a *rule-based solver* for an enrichment CS^+ of CS which extends the solver S known for CS. This rule-based solver is given by a set of rules transforming a constraint in $\mathcal{L}^+$, seen as a pair (C, P) made of an impure constraint C and a pure one P.

Example 3.1. The following rule defines a solver for CS^+ if S is a solver for CS.

Solve
$$\frac{(C, P)}{(C, S(P))}$$

In the case of a non-deterministic solver $S = (S_i)_{i=1,\ldots,m}$ for CS, we need the following rule to be applied with a *don't know* strategy:

SolveNonDet
$$\frac{(C, P)}{(C, S_i(P))}\ i = 1, \ldots, m$$

We will develop a rule-based solver for CS^+ using the rule Solve. In addition to Solve, some other solvers (solver extensions, see Section 4) will be applied on C.

4. Solver Extensions

The solver extensions have been grouped together w.r.t. the kind of action they have on the constraint store. On one hand, semantic rules (Section 4.2) make use of the properties of the domain or of the properties of the alien functions. On the other hand syntactical rules (Section 4.1) are based on syntactical transformations which are always valid in any constraint system. A formal description of transformation rules is given in Section 4.3.

4.1. Syntactical Solver Extensions

The following transformation rules mainly deal with either equations occurring in the constraint store, or equations introduced in the constraint store. There are also rules to maintain the partition of pure and impure constraints.

Variable Abstraction The rule Abstraction transforms impure constraints into pure ones by adding new variables to name aliens. These variables X_u replace alien subterms u and the related equations $X_u = u$ are added to the constraint store. Then, equations $X_u = u$ are no more transformed and remain in the constraint store. To replace alien subterms, we use a bijective mapping which associates to each non-variable term u a unique variable X_u. According to this convention, two occurrences of the same term are automatically replaced by the same variable.

Example 4.1. Consider the constraint $C = (\sin^2(x+y) + \cos^2(x+y)) = 1 - \sin^3(2x) * (\sin(x+y) + \cos(x+y))$. Abstraction* transforms C into $C' = (X^2 + Y^2 = 1 - Z^3 * (X + Y)) \wedge ((X = \sin(x+y)) \wedge (Y = \cos(x+y)) \wedge (Z = \sin(2x)))$.

Alien Replacement We have seen above how to purify the constraint store with Abstraction. Thus, we get a pure constraint together with a conjunction of equations $X_u = u$ involving new variables called *abstraction variables*. Then, Solve (see Example 3.1) can obviously be used to simplify this pure constraint. After that, the idea is to re-built an impure constraint without abstraction variables in order to be able to apply another kind of solver extensions (described later on). To this end, AlienRep consists in replacing abstraction variables by their related alien subterms. Obviously, this rule is the converse of Abstraction and so (AlienRep* ∘ Abstraction*) is the identity solver, where ∘ represents the usual composition of functions.

Inter-Reduction The idea of this transformation rule InterRed is to perform the replacement of a term by another one, provided there is an equation between these two terms in the constraint store. Thus, we use an ordering $\prec$ on heterogeneous terms in $T(\Sigma^+ \cup \mathcal{V}^+)$ to orient equations as rewrite rules. This ordering satisfy some requirements: an impure (resp. non-ground) term cannot be less than a pure (resp. ground) term with respect to $\prec$. These requirements can be easily understood by the fact that we want to be able to purify constraints and to compute the truth value of constraints.

Example 4.2. Consider the constraint $C = (y \leq \sin 2x + \sin z) \wedge (\sin z = 1) \wedge (\sin 2x = 0)$. InterRed transforms C into $(y \leq 0 + 1) \wedge (\sin z = 1) \wedge (\sin 2x = 0)$ provided that $\sin z \succ 1$ and $\sin 2x \succ 0$.

Moving constraints As said before, a constraint store is represented by a couple (C, P). When a constraint c in C becomes pure (e.g., after Abstraction), it can be carried to P, the pure part of the store. This is realized with the rule ToPure. In a similar way, constraints in P that become impure (e.g., after AlienRep) are moved to C with the rule FromPure.

4.2. Semantic Solver Extensions

Rewriting is an ubiquitous concept for reducing expressions by simpler but equivalent ones. In this section, some meta-transformation rules (mainly based on rewriting) are proposed. These transformations must be regarded as solvers and aim to integrate properties relevant to functions and predicates in CS^+. Our aim is to apply rewriting techniques for transforming both terms and constraints occurring in the constraint store. Hence, we need to consider a database of rewrite rules for the terms reduction (TGR) and another one for the constraints reduction (CGR). In the following, we define how to apply such rules coming from a database of properties. In our framework, a rule is guarded by a constraint, which means that the rule is applied only if the related constraint is entailed by the current constraint store. Hence, we have chosen here to use a form of contextual rewriting, where the rewriting context is processed by the deduction mechanism available in the constraint system enrichment. At this point, another question should be addressed: how to match a term with the left-hand side of a rule? For sake of expressiveness, this matching cannot be simply syntactic and we improve it in two directions.

First, for the constraints matching, we assume that logical connectives satisfy some equational properties. Second, for the terms matching, our idea is to normalize terms to be rewritten before applying rules and thus before the matching process. Since a solver S is usually provided with a normalization process on terms in CS (which is more or less its internal representation), we must also study the problem of extending this normalization to terms in CS^+.

Term Dependent Guarded Reduction The term rewrite system TGR is a finite set of guarded rules $(l \to r \| g)$ where g is a constraint in CS^+ and l, r are terms such that $\mathcal{D}^+ \models g \Rightarrow l = r$. An instance of the term l, say $l\sigma$, occurring in the constraint store C can be replaced by $r\sigma$ when $g\sigma$ is entailed by C. This transformation rule is called TermRed.

Example 4.3. Consider the guarded rules $(|x| \to x \| x \geq 0)$ and $(|x| \to -x \| x < 0)$. The constraint $C = (|y-2| = x + |x| + 1) \wedge (y \geq 3) \wedge (x * y < 0)$ can be reduced to $y = 3 \wedge x < 0$ thanks to TermRed and Solve (see Example 3.1).

Constraint Dependent Guarded Reduction The constraint rewrite system CGR consists of a finite set of guarded rules $(L \to R \| G)$ where G is a constraint in CS^+ and L, R are conjunctions of atomic constraints in CS^+ such that $\mathcal{D}^+ \models G \Rightarrow (L \Leftrightarrow R)$. A rewrite relation is defined as previously, except that matching is now performed modulo the associativity-commutativity of $\wedge$,$\vee$ and the distributivity of $\wedge$ with respect to $\vee$. The corresponding transformation rule is called ConsRed.

Example 4.4. Consider the guarded rule $(\sqrt{x} = y \to x = y^2 \| x \geq 0)$. The constraint $(x >$
$) \wedge ((x-1) * (y-3) > 0) \wedge (\sqrt{y-2} = y-4)$ can be reduced to $(x > 2) \wedge ((x-1) * (y-3) >$
$\wedge (y-2) = (y-4)^2$ since $y - 2 \geq 0$. Finally, we get solutions for y using the initial solver.

Remark 4.1. Formally, checking the implication (entailment) requires a validity checker for the enriched constraint system. If such a decision algorithm is not provided, then the semantic entailment can be approximated by a syntactic constraint inclusion test.

Domain Dependent Completion/Deletion In order to Complete/Delete the information encoded in the constraint store, we consider a database of valid facts, i.e. a finite set DDR of valid conjunctions of constraints in CS^+. This leads to a pair of quite opposite transformation rules, namely DomComp and DomDel. DomComp completes the constraint store C by an instance of a constraint $C' \in DDR$ provided this instance is not yet entailed by C. Conversely DomDel deletes an instance of $C' \in DDR$ occurring in the constraint store. For trigonometric functions, examples of valid constraints in DDR are: $-1 \leq \sin x \leq 1$, $-1 \leq \cos x \leq 1$, $\cos^2(x) + \sin^2(x) = 1$.

Example 4.5. Consider the constraint $C = (1 - \sin 2x = y)$ and the valid constraint $(-1 \leq \sin X \leq 1) \in DDR$. Applying DomComp on C yields $C \wedge (-1 \leq \sin 2x \leq 1)$ thanks to the instantiation of X by $2x$. DomComp does not apply on $C' = (0 \leq y \leq 1) \wedge C$ since $(0 \leq y \leq 1) \wedge (1 - V = y)$ already implies $(-1 \leq V \leq 1)$, where V stands for $\sin 2x$.

Normalization We assume that the solver S is equipped with a normalizing mapping NF, that is an idempotent computable mapping $NF : T(\Sigma, \mathcal{V}) \to T(\Sigma, \mathcal{V})$ such that $\forall t \in T(\Sigma, \mathcal{V})$, $\mathcal{D} \models t = NF(t)$. Moreover, the computation of $NF(t)$ does not depend on the names of variables in $\mathcal{V}(t)$ but depends on the ordering of variables given by $\prec$. Formally, this assumption can be stated as follows: for any term $t \in T(\Sigma, \mathcal{V})$ and any variable renaming $\xi : \mathcal{V}(t) \to \mathcal{V}$, $(\forall x, y \in \mathcal{V}(t),\ x\xi \prec y\xi \Leftrightarrow x \prec y)$ implies $NF(t\xi) = NF(t)\xi$.

Let us now define the extension NF^+ of the normalizing mapping NF.

Definition 4.1. The mapping $NF^+ : T(\Sigma^+, \mathcal{V}^+) \to T(\Sigma^+, \mathcal{V}^+)$ is defined by:

- $NF^+(f(t_1, \ldots, t_m)) = f(NF^+(t_1), \ldots, NF^+(t_m))$ if $f \in \Sigma^+ \backslash \Sigma$.
- If t is pure, then $NF^+(t) = NF(t)$
- If t is an impure term with a top symbol in Σ, then $NF^+(t)$ is recursively obtained:

  1. Compute $b_i := NF^+(a_i)$ for every alien $a_i \in Alien(t)$.
  2. Compute the term t^π obtained by replacing in t aliens $a_1, \ldots a_n$ by new variables $x_1, \ldots, x_n$ such that $x_k \prec x_l \Leftrightarrow b_k \prec b_l$ and $x_k = x_l \Leftrightarrow b_k = b_l$ for $1 \leq k, l \leq n$.
  3. $NF^+(t)$ is $NF(t^\pi)\{x_k \mapsto b_k\}_{k=1,\ldots,n}$.

The transformation rule Normalize consists in the replacement of any term t by its normal form $NF^+(t)$ following an innermost strategy.

Example 4.6. Let us consider the function symbols $+, *, -$, the predicate symbol $\leq$ in Σ (interpreted as usual over reals), the new function symbol f in Σ^+, and the normalizing mapping NF such that: $NF(x-x) = 0$, $NF(x+0) = x$ and $NF(1*x) = x$. Then $NF^+(f(1*x)-f(x+0)) = 0$ and applying Normalize on the constraint $C = (f(1*x) - f(x+0) \leq x - y)$ leads to $0 \leq x - y$. The initial solver can now treat this constraint and so we get the solution $y \leq x$.

4.3. Solver Extensions as Solvers

The transformation rules (Figure 2) formalize the solver extensions described in the previous sections, as well as the constraints (pure and/or impure constraints) they handle. These rules make precise the behavior/application of the solver extensions, and could be directly prototyped in programming languages like ELAN [15] and CHRS [6]. It is a routine examination to prove that each rule transforms a constraint into an equivalent one, and so it is correct and complete. However, in order to state that rules correspond to solvers, we still have to take care about the termination of their repeated applications.

Proposition 4.1. Solve, InterRed, Abstraction, AlienRep, Normalize, DomComp, DomDel, ToPure, FromPure *are solvers.*

Proof:

- Abstraction strictly decreases the number of aliens u in C without any occurrence of the corresponding abstraction variable X_u.

- AlienRep strictly decreases the number of abstraction variables X_u in C.
- InterRed replaces all occurrences but one of a term by a smaller term also occurring in (C, P). Such kind of replacements cannot be performed infinitely many times.
- ToPure strictly decreases the number of conjuncts in the first component C.
- FromPure strictly decreases the number of conjuncts in the second component P.
- DomComp strictly decreases the number of constraints in DDR which have no instance in the constraint C.
- DomDel strictly decreases the number of instances of DDR constraints in C.
- Applying repeatedly Normalize always terminates since NF is idempotent and so is NF^+.
- In the same way, the repeated application of Solve terminates since S is a solver.

□

Since TermRed and ConsRed are parameterized by arbitrary rewrite rules, we cannot in general ensure their termination.

Assumption 4.1. We assume that the rule system TGR (resp. CGR) is defined in such a way that the repeated application of TermRed (resp. ConsRed) terminates. Moreover left-hand sides of rules in TGR (resp. CGR) are impure terms (resp. impure constraints) and valid constraints in DDR are impure and have only one conjunct (this can be assumed without loss of generality).

Under these assumptions on TGR, CGR and DDR, we get the following facts:

Fact 4.1. *According to Assumption 4.1,* TermRed *and* ConsRed *are solvers.*

Fact 4.2. *According to Assumption 4.1,* InterRed, Normalize, TermRed *and* ConsRed *do not apply on pure constraints.*

5. SoleX: the Solver Collaboration

The solver for CS^+ is described as a set of transformation rules (presented in the previous section) together with control. The basic operation we used for combining solvers is the composition (of functions). The extended solving process **SoleX** is (Contraction ∘ Solve ∘ Expansion ∘ Reduction) where the four phases are as follows:

- Reduction phase (Reduction = (ConsRed$^+$ ∘ TermRed$^+$ ∘ InterRed$^+$ ∘ Normalize$^+$)): the constraint store is transformed thanks to semantic and syntactical solver extensions introduced in the two previous sections.
- Expansion phase (Expansion = (ToPure* ∘ Abstraction* ∘ DomComp*)) : the constraint store is completed by valid constraints which may be helpful in the next phase and may be purified thanks to Abstraction.
- Solve phase (Solve): the initial solver is applied on the pure part of the constraint store.

Solve	$\dfrac{(C, P)}{(C, S(P))}$
InterRed	$\dfrac{(C[g] \wedge g = d, P)}{(C[g \hookleftarrow d] \wedge g = d, P)}$ if g is impure, $d \prec g$
Abstraction	$\dfrac{(C[u], P)}{(C[u \hookleftarrow X_u] \wedge X_u = u, P)}$ if $\begin{cases} u \in Alien(C), \\ X_u \text{ is a new variable.} \end{cases}$
AlienRep	$\dfrac{(C \wedge X_u = u, P)}{(C\{X_u \mapsto u\}, P)}$ if X is an abstraction variable
Normalize	$\dfrac{(C[g], P)}{(C[g \hookleftarrow NF^+(g)], P)}$ if C is impure, $g \neq NF^+(g)$
TermRed	$\dfrac{(C[l\sigma]_\omega, P)}{(C[\omega \hookleftarrow r\sigma], P)}$ if $(l \to r \| g) \in TGR, \mathcal{D}^+ \models (C \wedge P) \Rightarrow g\sigma$
ConsRed	$\dfrac{(C, P)}{(C'[[\omega \hookleftarrow R\sigma]], P)}$ if $(L \to R \| G) \in CGR, C =_{ACD} C'[[L\sigma]]_\omega, \mathcal{D}^+ \models (C \wedge P) \Rightarrow G\sigma$
DomComp	$\dfrac{(C, P)}{(C \wedge C'\sigma, P)}$ if $C' \in DDR$ and $\sigma : \begin{cases} Alien(C')\sigma \subseteq Alien(C) \\ C'\sigma \text{ is not a conjunct of } C \end{cases}$
DomDel	$\dfrac{(C \wedge C'\sigma, P)}{(C, P)}$ if $C' \in DDR : Alien(C')\sigma \subseteq Alien(C)$
ToPure	$\dfrac{(C \wedge c, P)}{(C, P \wedge c)}$ if c is pure
FromPure	$\dfrac{(C, P \wedge c)}{(C \wedge c, P)}$ if c contains some abstraction variable

Figure 2. Solver Extensions

- Contraction phase (Contraction = (DomDel* ∘ AlienRep* ∘ FromPure*)): the impure equations introduced in the second phase are merged with the new pure part of the constraint store. The persistent valid constraints added in the same phase are removed.

It is important to notice that a transformation rule is not necessarily a solver since its repeated application may not terminate. For the same termination problem, a composition of two solvers yields a new function which is not necessarily a solver (Definition 3.4). For proving the termination of a composition of solvers, we may need to embed all orderings related to elementary solvers into a Noetherian ordering $<$. As solver extensions are parameterized by $NF, \prec$, rewrite rules in TGR and CGR, or by valid constraints in DDR, we have to give more precise sufficient conditions in order to ensure the termination of the **SoleX** process.

Theorem 5.1. *Let $\geq$ be a quasi-ordering on $\mathcal{L}^+\backslash\mathcal{L}$ such that $>$ is Noetherian.* **SoleX** *is a solver for CS^+ if the following conditions are satisfied:*

1. TermRed *and* ConsRed *are solvers,*
2. *For any* $E \in \{\text{Normalize}, \text{InterRed}, \text{TermRed}, \text{ConsRed}\}$, *we have*

$$E(C, P) = (C', P) \text{ and } C' \in \mathcal{L}^+\backslash\mathcal{L} \Rightarrow C \geq C'$$

3. ExtSolve = Contraction $\circ$ Solve $\circ$ Expansion *is a solver such that*

$$\text{ExtSolve}(C, P) = (C', P') \Rightarrow C > C'$$

The proof is quite obvious since **SoleX** is ExtSolve $\circ$ Reduction and we assume a complexity measure that does not increase by Reduction but strictly decreases by ExtSolve.

Remark 5.1. **SoleX** is just a partial solver if S is partial. Moreover, if $S = (S_i)_{i=1,\ldots,m}$ is a non-deterministic solver, then **SoleX** becomes a non-deterministic solver if Solve is replaced by SolveNonDet which is applied with a *don't know* mechanism.

Remark 5.2. **SoleX** becomes a solver when DDR, TGR, CGR, $\prec$ are empty, and NF is the identity mapping.

It remains to show how ExtSolve can be turned into a solver. In the case of an empty set DDR, ExtSolve simply corresponds to Solve and so it is obviously a solver. In general, introducing a valid constraint to the constraint store could also introduce some disturbance in the future solver application and we must take care of this situation. On the contrary, our aim is to express the idea that the Expansion phase may help significantly in the *simplification* step already performed by the solver. In order to formalize the intuitive notion of *further simplification*, let us introduce yet another ordering $\rhd$. This ordering $\rhd$ is now defined on pure constraints and is complementary to the ordering $>$ already introduced for impure constraints. The first requirement consists in choosing $\rhd$ in such a way that S does not compute a more complex constraint w.r.t. $\rhd$.

Proposition 5.1. ExtSolve = Contraction $\circ$ Solve $\circ$ Expansion *is a solver if there exists a quasi-ordering $\trianglerighteq$ on $\mathcal{L}$ such that $\rhd$ is Noetherian and*

1. $P \trianglerighteq S(P)$ *and* $P \wedge P'' \trianglerighteq P$
2. *(a)* $S(P) \rhd S(P \wedge P'')$

   (b) or else $S(P \wedge P'') = S(P) \wedge P''$

where P'' is a pure constraint derived from the abstraction of a constraint in DDR, that is formally $\exists C' \in DDR\ \exists\sigma$, $\text{AlienRep}^*(P'') = C'\sigma$.

Item 2. makes sure that a pure constraint P'' coming from a DDR constraint permits to strictly simplify the second component P of the store or is not used by the solver S (the corresponding DDR constraint will then be removed by DomDel). The ordering $\rhd$ is defined on constraints in $\mathcal{L}$ and may be chosen for instance according to the number of "unsolved" variables.

Proof:
ExtSolve is the composition of several solvers. Hence, ExtSolve obviously preserves the solutions (items 1 and 2 of Definition 3.4).

Let $\pi(C,P)$ (resp. $\iota(C,P)$) be the pure (resp. impure) component of (C,P) defined in the following way:

$$\pi(C,P) = P' \text{ (resp. } \iota(C,P) = C') \text{ where } (C',P') = \mathsf{ToPure}^* \circ \mathsf{Abstraction}^*(C,P)$$

Let E be a solver from $\{\mathsf{ToPure}, \mathsf{Abstraction}, \mathsf{AlienRep}, \mathsf{FromPure}\}$. Then, by definition, E does not modify the result of π i.e., $\pi(C,P) = \pi(E(C,P))$.

Let $>_S$ be the Noetherian ordering defined by: $P >_S P'$ if $P' = S(P)$ and $P' \neq P$. For sake of simplicity, we consider in the following that DDR is a singleton but the same kind of proof can be done by induction on the size of DDR at the cost of more complicated notations. Let C' be the constraint added to C by DomComp. This constraint is abstracted to P'' and added to P thanks to Abstraction followed by ToPure:

$$\begin{aligned} \mathsf{Solve} \circ \mathsf{Expansion}(C,P) &= \mathsf{Solve} \circ \mathsf{ToPure}^* \circ \mathsf{Abstraction}^* \circ \mathsf{DomComp}^*(C,P) \\ &= \mathsf{Solve} \circ \mathsf{ToPure}^* \circ \mathsf{Abstraction}^*(C \wedge C', P) \\ &= \mathsf{Solve}(\iota(C,P), \pi(C,P) \wedge P'') \qquad (1) \end{aligned}$$

Then, two cases corresponding to the restrictions of Proposition 5.1 can happen.

1. Restriction 2.a of Proposition 5.1: $S(\pi(C,P)) \rhd S(\pi(C,P) \wedge P'')$. From Item 1 of Proposition 5.1, we have: $\pi(C,P) \trianglerighteq S(\pi(C,P))$ (2). From Equality (1) and the definitions of the Solve rule and the π function, we obtain:

   $$\pi(\mathsf{Solve} \circ \mathsf{Expansion}(C,P)) = S(\pi(C,P) \wedge P'') \quad (3)$$

   Using restriction 2.a and (3), we get: $S(\pi(C,P)) \rhd \pi(\mathsf{Solve} \circ \mathsf{Expansion}(C,P))$ (4). By transitivity on (2) and (4), we get: $\pi(C,P) \rhd \pi(\mathsf{Solve} \circ \mathsf{Expansion}(C,P))$ (5). Since Contraction may remove some constraints, we also have:

   $$\pi(\mathsf{Solve} \circ \mathsf{Expansion}(C,P)) \trianglerighteq \pi(\mathsf{ExtSolve}(C,P)) \quad (6)$$

   Again by transitivity ((5) and (6)), we get: $\pi(C,P) \rhd \pi(\mathsf{ExtSolve}(C,P))$ (I)
2. Restriction 2.b of Proposition 5.1: $S(\pi(C,P) \wedge P'') = S(\pi(C,P)) \wedge P''$.

   First, we develop $\mathsf{ExtSolve}(C,P)$ (when DomComp adds a single constraint C') using Equality (1):

   $$\begin{aligned} \mathsf{ExtSolve}(C,P) &= \mathsf{DomDel}^* \circ \mathsf{AlienRep}^* \circ \mathsf{FromPure}^*(\iota(C,P), S(\pi(C,P)) \wedge P'') \\ &= \mathsf{DomDel}^* \circ \mathsf{AlienRep}^* \circ \mathsf{FromPure}^*(\iota(C,P) \wedge C', S(\pi(C,P))) \\ &= \mathsf{AlienRep}^* \circ \mathsf{FromPure}^*(\iota(C,P), S(\pi(C,P))) \qquad (7) \end{aligned}$$

Using the definition of π, and Equality (7), we obtain: $\pi(\mathsf{ExtSolve}(C,P)) = S(\pi(C,P))$ (8). Using (8) and the definition of $>_S$, we get:

$$\begin{array}{ll} \text{if } \pi(C,P) \neq S(\pi(C,P)) & \text{then } \pi(C,P) >_S \pi(\mathsf{ExtSolve}(C,P)) \quad (II) \\ & \text{otherwise } \mathsf{ExtSolve}(C,P) = (C,P) \quad (III) \end{array}$$

Finally, consider the relation $\gg$ on $\mathcal{L}$ defined by: $P \gg P'$ if $P \rhd P'$ or ($P \trianglerighteq P'$ and $P >_S P'$). It is a Noetherian ordering since $\rhd$ and $>_S$ are Noetherian, and $>_S$ is included in $\trianglerighteq$ (by definition).

Since $\pi(C,P) \gg \pi(\mathsf{ExtSolve}(C,P))$ ((I) and (II)) or $\mathsf{ExtSolve}(C,P) = (C,P)$ ((III)), we can conclude that the repeated application of ExtSolve always terminates and so ExtSolve is a solver. □

6. Applications

Inverse Robot Kinematics We can now solve the problem [4] briefly described in Section 2. This problem for a robot (see Figure 1) having two revolute joints (degrees of freedom) can be described by the system of equations presented in Figure 1 where l_1, l_2, p_x, p_z are parameters and $P_y, \delta_1, \delta_2, \phi, \theta, \psi$ are variables. l_1, l_2 are the lengths of the robot arms, (p_x, P_y, p_z) is the position of the end-effector, ϕ, θ, ψ are the Euler angles of the orientation of the end-effector, and δ_1, δ_2 are the rotation angles of the revolute joints. The expected solution is a symbolic expression describing the dependence of the joint variables on the geometrical and position parameters. For this application, neither trigonometric solvers nor trigonometric simplifications automatically turn a symbolic solution expressing the relation between parameters and variables. Thus, we need a solver working on the domain of non-linear polynomial constraints (namely Gröbner bases which simplify polynomial equations and return relations between the variables) with trigonometric functions (*sine* and *cosine*). Hence, let DDR be $\{\sin^2(X) + \cos^2(X) - 1 = 0\}$. The DomComp solver completes the system by adding for each angle $(\delta_1, \delta_2, \phi, \theta, \psi)$ the property of sine and cosine $(\sin^2(X) + \cos^2(X) = 1)$. The Abstraction solver replaces every remaining sine and cosine with new variables. Finally after Solve and AlienRep, **SoleX** reaches a fixed-point which is the desired solution:

$$\begin{array}{lll} & \sin(\psi) + k_1 * \cos(\phi) * \sin(\delta_1) * \cos(\delta_1) = 0 & \\ \wedge & \sin(\theta) + k_2 * \cos(\phi) * \sin(\delta_1) = 0 & \wedge \quad P_y + k_6 * \sin(\delta_1) * \cos(\delta_1) = 0 \\ \wedge & \sin(\phi) + k_3 * \cos(\phi) * \sin(\delta_1) * \cos(\delta_1) = 0 & \wedge \quad \sin(\delta_2) + k_7 = 0 \\ \wedge & \cos(\psi) + k_4 * \cos(\phi) * \sin(\delta_1) = 0 & \wedge \quad \sin^2(\delta_1) + k_8 = 0 \\ \wedge & \cos(\theta) = 0 & \wedge \quad \cos(\delta_2) + k_9 * \cos(\delta_1) = 0 \\ \wedge & \cos^2(\phi) + k_5 = 0 & \wedge \quad \cos^2(\delta_1) + k_{10} = 0 \end{array}$$

where $k_1, \ldots, k_{10}$ are constants, depending on the parameters l_1, l_2, p_x, p_z.

Constraint solving over integers and terms This example illustrates how to extend a constraint solver working on conjunctions of two-sorted constraints: the integers and the terms. Constraints over integers are equations and inequations between linear polynomials with integer

coefficients. Constraints over terms are equations between terms. Formally, the signature is as follows:

$$\Sigma = \begin{cases} \leq : \mathbb{Z} \times \mathbb{Z}; \quad +, - : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}; \quad 0, 1 : \mathbb{Z} \\ f : \mathbb{T} \times \mathbb{T} \to \mathbb{T}; \quad g : \mathbb{T} \to \mathbb{T}; \quad a : \mathbb{T} \end{cases}$$

$\mathbb{T}$ (resp. $\mathbb{Z}$) denotes ground terms (resp. integers) and the function symbols and the predicate symbol $\leq$ are interpreted as usual. We consider two new function symbols $depth : \mathbb{T} \to \mathbb{Z}$ and $max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, interpreted respectively as the depth of a term and the maximum of two integers. Since we want to extend the constraint solvers associated to integers and terms, we choose the sets of Term-dependent (Constraint-dependent) Guarded Reduction and DDR as follows:

$$TGR = \begin{cases} (1) & depth(f(X,Y)) \to 1 + max(depth(X), depth(Y)) \\ (2) & depth(g(X)) \to 1 + depth(X) \\ (3) & depth(a) \to 1 \\ (4) & (max(x,y) \to y || x \leq y) \\ (5) & (max(x,y) \to x || y \leq x) \end{cases}$$

$CGR = \{(1)\ depth(X) = 1 \to X = a, (2)\ depth(X) < 1 \to \bot\}, DDR = \{1 \leq depth(X)\}$ where a rule $l \to r$ is an abbreviation for $(l \to r || \top)$. Consider the repeated application of **SoleX** to

$$\begin{cases} (I) & z' - z & = & depth(g(Y)) - depth(Y) \\ (II) & max(z', z) - z & = & u \\ (III) & 0 & \leq & 1 - v + u \\ (IV) & depth(g(X)) & \leq & v \\ (V) & f(W, W') & = & f(g(W'), f(X, X)) \end{cases}$$

First, TermRed applies rule (2) of TGR on equation (I). After Normalize, (I) becomes $z' = z + 1$. TermRed applies rule (4) of TGR on (II), and after Normalize (II) becomes $u = 1$. Then, after ToPure, Solve applies the solver for integers and (III) is transformed into $v \leq 2$. TermRed can now apply rule (2) of TGR on (IV): then application of Normalize, DomComp and ConsRed (rule (1)) leads to $X = a$. Finally, after Normalize and ToPure, Solve (solver for terms, i.e., unification) transforms (V) into $W = g(f(a, a)) \wedge W' = f(a, a)$. The solved form is:
$z' = z + 1 \wedge u = 1 \wedge v \leq 2 \wedge X = a \wedge W = g(f(a, a)) \wedge W' = f(a, a)$
Here, the complexity measure for proving the termination of **SoleX** is based on a combination of elementary measures corresponding to the number of *depth* and *max* occurrences and to the multiset of sizes of *depth* arguments.

Extending Syntactic Matching with the Commutativity The matching process is a crucial mechanism in the implementation of functional programming, as well as unification is at the heart of logic programming. Given a non-ground term s and a ground term t, the aim of matching is to instantiate variables occurring in s in such a way that the instantiation of s becomes equal to t. Thus, a matching problem may be viewed as a specific non-symmetric unification problem, and so, it is still constraint solving over terms. For example, the matching

problem $cons(x, y) = cons(a, cons(b, cons(c, nil)))$ admits $x = a \wedge y = cons(b, cons(c, nil))$ a unique solution. Here, *cons* may be interpreted as the list constructor, and *nil* as the empt list. The operators *cons* and *nil* do no satisfy equational properties, they are free and we tall about *syntactic matching*. Due to its relevance for applying rules, syntactic matching must b implemented very efficiently in rule-based programming languages like ELAN [15]. In this con text, it is also very important to handle more complicated terms such as a list of "equations' $cons(eq(x, a), cons(eq(x, y), nil))$, where *eq* should be now declared as a commutative operato since the equality relation is clearly symmetric. But then, syntactic matching is no more suffi cient to handle such kind of terms. Usually, this combination problem is solved by combining a syntactic matching algorithm with a commutative matching algorithm. Thanks to the **SoleX** ap proach, extending a syntactic matching solver with the commutativity is also possible, provide that CGR consists of the following rewrite rules:

$$\begin{aligned} eq(x_1, x_2) &= eq(y_1, y_2) \rightarrow x_1 = y_1 \wedge x_2 = y_2 \\ eq(x_1, x_2) &= eq(y_1, y_2) \rightarrow x_1 = y_2 \wedge x_2 = y_1 \\ eq(x_1, x_2) &= f(y_1, \ldots, y_m) \rightarrow \bot \quad \text{for any } f \in \Sigma \end{aligned}$$

Let us describe how **SoleX** would proceed on a very simple matching problem:

$$cons(eq(f(x, y), a), z) = cons(eq(a, f(b, c)), cons(eq(b, c), cons(eq(b, d), nil)))$$

where f, a, b, c, d are free operators, eq is commutative and x, y, z are variables. First, Ab traction applies, and we get the pure constraint $cons(X, z) = cons(A, cons(B, cons(C, nil)))$. is pure constraint is solved with Solve, and yields $X = A \wedge z = cons(B, cons(C, nil))$. Af AlienRep, we obtain again a new impure constraint: $eq(f(x, y), a) = eq(a, f(b, c)) \wedge z = cons(eq(b, c), cons(eq(b, d), nil))$. Then, ConsRed can be applied on the first equation as a non deterministic solver generating two branches. The first branch contains the constraint $f(x, y) = a$ which leads to a failure by Solve. In the second branch, we have $a = a \wedge f(x, y) = f(b, c)$ which is transformed into $x = b \wedge y = c$ by Solve. So, we get the expected solution, that is $x = b \wedge y = c \wedge z = cons(eq(b, c), cons(eq(b, d), nil))$.

7. Conclusion

SoleX enables one extending constraint solvers in order to handle alien function symbols, i.e., an initial solver (seen as a black-box solver) is completed with a glass-box mechanism. The extension is composed of several solvers split into: (1) syntactical solvers that process the constraints independently from the computation domain and (2) semantic solvers that enrich constraints with information on the domain or on the function interpretation.

In [9], Heintze & al. propose an extension of the solver of CLP($\mathcal{R}$) for constraints over $\mathcal{R}+\mathcal{M}$. This extension is based on two methods: (1) simplifications that are similar to our notion of normal form extension, and (2) substitutions that can be seen as our rule InterRed. However, our framework for extension is more complete since we also propose some other syntactic rules,

as well as semantic rules. Moreover, we can extend every kind of domains whereas in [9] novel constraint solvers, simplification algorithms or computation domains are always related to $\mathcal{R}$. On the other hand, Heintze & al. not only extend the solver, but also the programming language. Thus, their work also enables applications such as debuggers or prototyping of novel CP systems.

A first implementation of **SoleX** has been realized into CoSAc [21] where two solvers (one based upon a Gröbner bases algorithm for simplifying polynomial equations and the other one based on Gaussian elimination) are extended with new function symbols (such as *sin*, *cos* and $\sqrt{}$ respectively corresponding to the usual trigonometric *sine* and *cosine*, and square root of polynomials). Hence, significant problems like the *Inverse Robot Kinematics* problem [4] (which is originally expressed with trigonometric functions) and the *Robot in a Corridor* problem [21] (which makes use of square roots of polynomials) can be solved automatically.

In this paper, **SoleX** has been presented as a set of transformation rules plus an initial solver. Therefore, we could imagine to prototype this rule-based extended solver with a rule-based programming language. In this context, ELAN [15] is a very good candidate for prototyping issues since it provides facilities to express strategies for applying rules and to call external solvers. However, we believe that a more efficient implementation should be based on a collaboration of several component solvers running concurrently. This explains why a more complete implementation is currently under way with BALI [18, 20, 19] which provides a logical framework for managing constraints (realized with ECLiPSe and the CHRs [6]) and a language for designing and executing solver collaborations. In fact BALI and **SoleX** have similarities that have to be studied in order to completely merge the two concepts for realizing a framework including either solver collaboration and solver extension. Furthermore the extension of solvers with new sorts and new predicates (i.e. constraints) will enable to design solvers on totally different domains. Thus, extending a well-known solver on a "simple" domain could lead to realize solvers on complex domains thanks to solver extension of **SoleX** and solver collaboration of BALI.

References

[1] F. Baader and K. Schulz. On the combination of symbolic constraints, solution domains, and constraint solvers. In *Proc. of CP'95*, volume 976 of *Lecture Notes in Computer Science*, 1995.

[2] F. Benhamou and L. Granvilliers. Combining local consistency, symbolic rewriting, and interval methods. In J. Pfalzgraf, editor, *Proc. AISMC-3*, volume 1138 of *Lecture Notes in Computer Science*, Steyr, Austria, Sep. 1996. Springer-Verlag.

[3] P. G. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and Integration of Theorem Provers and Computer Algebra Systems. In *Proc. of the International Conference Artificial Intelligence and Symbolic Computation (AISC'98), Plattsburgh (New York, U.S.A.)*, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 94–106. Springer-Verlag, Sep. 1998.

[4] B. Buchberger. Applications of Gröbner Bases in Non-Linear Computational Geometry. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 413–446. MIT Press, 1989.

[5] O. Caprotti. Extending risc-clp(cf) to handle symbolic functions. In A. Miola, editor, *Proc. of DISCO'93*, volume 722 of *Lecture Notes in Computer Science*. Springer-Verlag, Sep. 1993.

[6] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends* , volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[7] T. Frühwirth, A. Herold, V. Kuechenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint Logic Programming - An Informal Introduction. In G. Comyn, M. Ratcliffe, and N. Fuchs, editors, *Logic Programming in Action : LPSS'92, Zurich, Switzerland*, volume 636 of *Lecture Notes in Computer Science*. Springer-Verlag, Sep. 1992.

[8] L. Granvilliers. *Consistances locales et transformations symboliques de contraintes d'intervalles*. Phd thesis, University of Orléans, France, December 1998. In French.

[9] N. Heintze, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. Meta-Programming in CLP($\mathcal{R}$). *Journal of Logic Programming*, pages 221–259, 1997.

[10] K. Homann and J. Calmet. Combining Theorem Proving and Symbolic Mathematical Computing. In J.A. Campbell J. Calmet, editor, *Proc. of AISMC-2*, volume 814 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, 1995.

[11] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of the 14th ACM Symposium POPL, Munich, Germany*, pages 111–119. ACM, Jan. 1987.

[12] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[1] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Output in CLP($\mathcal{R}$). In ICOT Staff, editor, *Proc. of FGCS'92*, pages 987–995, Tokyo (Japan), Jun. 1992. IOS Press.

J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[15] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995.

[16] H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[17] P. Marti and M. Rueher. A Distributed Cooperating Constraints Solving System. *International Journal on AI Tools*, 4(1&2):93–113, 1995.

[18] E. Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes*. Phd thesis, Université Henri Poincaré - Nancy 1, Nov. 1996. Also available in english. Available on-line at `http://www.cwi.nl/~eric/Private/Publications/index.html`.

[19] E. Monfroy. An environment for designing/executing constraint solver collaborations. In *Proc. of the 2nd International Workshop on Constraint programming for time critical applications and multi-agent systems (Cotic'98), Nice (France)*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[20] E. Monfroy. The Constraint Solver Collaboration Language of **BALI**. In *Proc. of Frontiers of Combining Systems (FroCoS'98), Amsterdam (The Netherlands)*, 1998.

[21] E. Monfroy, M. Rusinowitch, and R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In *Proc. of ACM SAC'96*, pages 63–72, Feb. 1996.

[22] C. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), 1979.

[23] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In *Frontiers of Combining Systems*, Applied Logic, pages 121–140. Kluwer Academic Publishers, 1996.

[24] The Calculemus Project. *Calculemus and Types '98*, Eindhoven, The Netherlands, Jul. 1998.

[25] P. Van Hentenryck and V. Saraswat. Strategic Directions in Constraint Programming. *ACM Computing Surveys*, 28(4):701–726, Dec. 1996.

[26] S. Wolfram. *The Mathematica Book, 3rd ed.* Cambridge University Press, 1996.