Constraint Solving in Logic Programming and in Automated Deduction: a Comparison*

Alessandro Armando[1], Erica Melis[2], and Silvio Ranise[1]

[1] DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
[2] Universität des Saarlandes, FB Informatik – D-66041 Saarbrücken, Germany

Abstract. Constraint solving has been successfully employed in diverse areas such as Operation Research, Planning, Logic Programming, and Automated Deduction. This has led to the development of a number of specialised approaches as well as to the adoption of different integration schemes and methodologies. In this paper we introduce an approach to incorporate constraint solving in term rewriting and we compare it with the Constraint Logic Programming scheme. We compare the two approaches both at the theoretical and at the implementational level and discuss potentials for cross-fertilisation.

1 Introduction

Constraint solving aims at the development of efficient procedures tailored to restricted domains as well as to the integration of such procedures into general purpose reasoners. Constraint solving has been successfully employed in diverse areas such as Operation Research, Planning, Logic Programming, and Automated Deduction. This has led to the development of a number of specialised approaches as well as to the adoption of different integration schemes and methodologies.

While it is important to develop constraint solving techniques in the domain of interest, it is equally important to compare solutions developed in different settings and to look for opportunities of cross-fertilisation. For instance, [19] describes how Nelson and Oppen's approach to combining decision procedures [16] developed in the context of Automated Deduction can be lifted to a technique for building composite constraint solvers to be used in Constraint Logic Programming (CLP).

In this paper we compare the CLP scheme with a form of term rewriting, called *Constraint Contextual Rewriting* (CCR), which incorporates constraint solving. We compare the two approaches both at the theoretical and at the implementational level and discuss potentials for cross-fertilisation.

* We would like to thank Martin Müller for his feedback on (concurrent) constraint logic programming. The authors are supported in part by *Conferenza dei Rettori delle Università Italiane (CRUI)* in collaboration with *Deutscher Akademischer Austaunschdienst (DAAD)* under the *Vigoni Programme*.

We start by providing a common vocabulary for describing the two techniques (Section 2). A brief introduction to the CLP scheme, its operational semantics, and its implementation in the CLP($\mathcal{R}$) system is then given (Section 3). The CCR scheme is then introduced and its implementation in the context of the simplification mechanism of the Boyer and Moore's prover, NQTHM, is discussed (Section 4). Although CCR is inspired by the simplification mechanism of NQTHM, the abstract characterisation we give in Sect. 4 is new and provides a neat characterisation of the interplay between the constraint solver and the simplification mechanism. This contrasts with the presentation given in [3] where no abstract description is provided. In Sect. 5 we carry out a comparative analysis of the two approaches and explore the potentials for cross-fertilisation. Finally in Sect. 6 we draw some final remarks.

2 Formal Preliminaries

By Σ, Π, and V (possibly subscribed) we denote collections of function symbols, predicate symbols, and variables (with their arity). $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the set of terms and ground terms built on Σ and V. A (Σ, Π, V)-*atom* is either an expression $q(t_1, \ldots, t_n)$ where $q \in \Pi$ and $t_i \in \tau(\Sigma \cup V)$, $i = 1, \ldots, n$ or an expression $t_1 = t_2$ where $t_i \in \tau(\Sigma \cup V)$, $i = 1, 2$. A (Σ, Π, V)-*literal* is either a (Σ, Π, V)-atom or a negated (Σ, Π, V)-atom. We write (Σ, Π)-atom (-literal) instead of $(\Sigma, \Pi, \emptyset)$-atom (-literal). In the following we consider three sets of predicate (function) symbols Π_j, Π_p, and Π_c (Σ_j, Σ_p, and Σ_c) such that $\Pi_c \cap \Pi_p = \emptyset$ and $\Pi_j = \Pi_c \cup \Pi_p$ ($\Sigma_j = \Sigma_c \cup \Sigma_p$).

A *constraint domain* is a (Σ_c, Π_c)-structure $\mathcal{D}$, where a (Σ, Π)-structure consists of a set D and an assignment of functions and relations on D to the symbols of Σ and Π (resp.) which respect the arity of the symbols. A (Σ, Π)-*theory* is a collection of (Σ, Π)-formulae. A *model* of a (Σ, Π)-theory T is a (Σ, Π)-structure under which all the formulae in T evaluate to *true*. A $\mathcal{D}$-*model* of a theory T is a model of T extending $\mathcal{D}$ (this requires that the signature of $\mathcal{D}$ is contained in the signature of T). We write $T \models_{\mathcal{D}} \phi$ to denote that ϕ is valid in all $\mathcal{D}$-models of T. $\exists_{-\tilde{x}}\phi$ denotes the existential closure of ϕ except for the variables in the sequence $\tilde{x}$, whereas $\tilde{\exists}\phi$ denotes the full existential closure of ϕ.

3 Constraint Solving in Logic Programming

A *CLP program* is a set of clauses of the form $p \leftarrow p_1, \ldots, p_n$. where p is a (Σ_p, Π_p, V)-atom and p_i is either a (Σ_p, Π_p, V)-atom or a (Σ_c, Π_c, V)-atom for $i = 1, \ldots, n$ ($n \geq 0$). A *CLP goal* is a clause $\leftarrow p_1, \ldots, p_n$., where p_i is either a (Σ_p, Π_p, V)-atom or a (Σ_c, Π_c, V)-atom for $i = 1, \ldots, n$ ($n \geq 0$). It is assumed that $\Sigma_c \cap \Sigma_p = \emptyset$.

3.1 Operational Semantics of CLP Programs

The operational semantics of CLP programs provides an abstract characterisation of SLD-resolution [14] enhanced with constraint solving. Following [9], the operational semantics is given by means of a transition system whose transitions are $\rightarrow_r$(resolution), $\rightarrow_c$(constraint introduction), $\rightarrow_i$(inference/propagation), and $\rightarrow_s$(solution) and whose states are triples of the form $\langle A, C, S\rangle$, where A is a multiset of (Σ_j, Π_j, V)-atoms, and C and S are multisets of (Σ_c, Π_c, V)-atoms collectively called the *constraint store*. C is the set of so-called *active constraints* or the *constraint store* and S is the set of so-called *passive constraints*. There is also a state denoted by *fail*. The initial goal G is represented as a state $\langle G, \emptyset, \emptyset\rangle$. Final states are states of the form $\langle \emptyset, C, S\rangle$. C_0 is said to be an *answer constraint* for G if $\models_{\mathcal{D}} C_0 \leftrightarrow \exists_{-\tilde{x}}(C \wedge S)$ holds.

Given a CLP program P and a constraint domain $\mathcal{D}$, the transition relation is obtained by combining the following transition rules:

$$\langle A \cup \{a\}, C, S\rangle \rightarrow_r \langle A \cup \{p_1, \ldots, p_n\}, C, S \cup \{(a = p)\}\rangle$$

if $(p \leftarrow p_1, \ldots, p_n.) \in P$, and a and p have the same predicate symbol. $(a = p)$ abbreviates a conjunction of equations between corresponding arguments of a and p thereby encoding the corresponding unification problem.

$$\langle A \cup \{a\}, C, S\rangle \rightarrow_r \mathit{fail}$$

is executed if for each rule $(p \leftarrow p_1, \ldots, p_n.) \in P$, a and p have different predicate symbols.

$$\langle A \cup \{c\}, C, S\rangle \rightarrow_c \langle A, C, S \cup \{c\}\rangle$$

if c is a constraint told to the constraint solver;

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$$

if $\langle C', S'\rangle = \texttt{infer}(C, S)$.

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$$

if $\texttt{consistent}(C)$ or else

$$\langle A, C, S\rangle \rightarrow_s \mathit{fail}$$

$\rightarrow_r$ transitions reduce atoms using the rules of the program; $\rightarrow_c$ transitions introduce constraints into the constraint solver; finally $\rightarrow_s$ transitions test whether the constraint store is consistent, and $\rightarrow_i$ transitions infer more active constraints possibly modifying the passive ones. The predicate `consistent` and the function `infer` model the interface to the constraint solver. They are assumed to enjoy the following properties (where $\mathcal{D}$ is the constraint domain): if $\models_{\mathcal{D}} \tilde{\exists} C$ then $\texttt{consistent}(C)$; if $\texttt{infer}(C, S) = \langle C', S'\rangle$ then $\models_{\mathcal{D}} (C \wedge S) \leftrightarrow (C' \wedge S')$. Such assumptions guarantee that if C_0 is an answer constraint of G w.r.t. a CLP program P, then $P \models_{\mathcal{D}} C_0 \rightarrow G$.

3.2 The CLP($\mathcal{R}$) System's Architecture

In order to attain the performance expected from a programming language it is important that the cost of constraint solving is commensurate with the inherent difficulty of the constraints themselves. To this end the constraint solving capabilities of the CLP($\mathcal{R}$) system are distributed among a set of sub-solvers. The sub-solvers are specialised to tackle constraints of increasing complexity and are hierarchically organised as depicted in Fig. 1 [10]. The lower a module is in the hierarchy the stronger (but also computationally more expensive) are the corresponding solving capabilities.

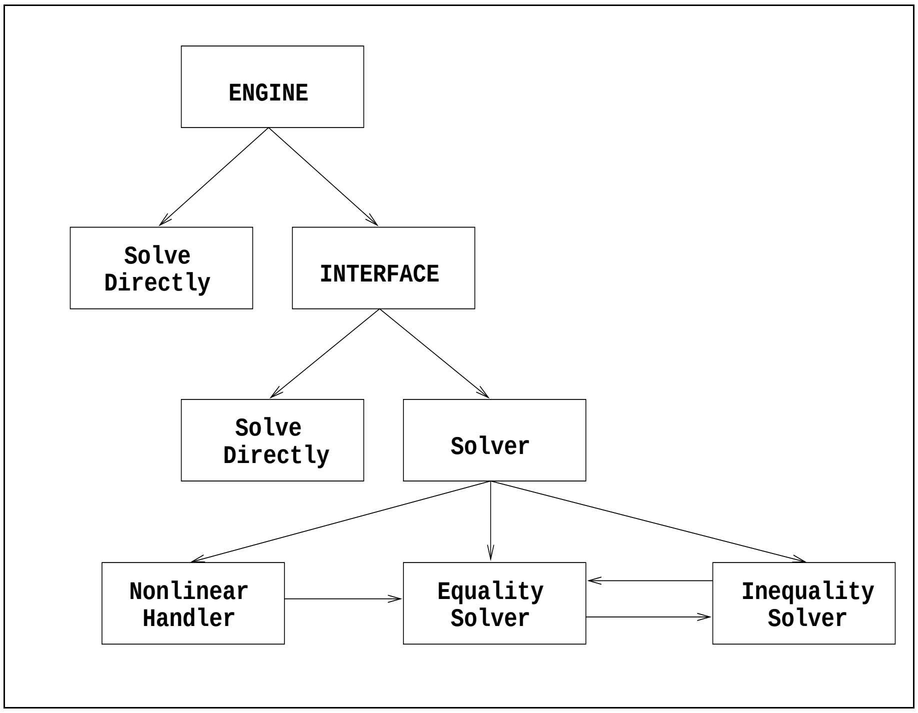

Fig. 1. Architecture of the CLP($\mathcal{R}$) system

The inference `ENGINE` controls the execution of the SLD derivation, maintains the variable bindings, directly solves some simple constraints (e.g. ground constraints), and hands difficult ones to the interface. The `INTERFACE` directly solves some of the input constraints, puts the remaining ones into a normal form and then passes the normalised constraints to the `Solver`. The `Solver` selects and redirects the input constraints to either the `Equality Solver`, the `Inequality Solver`, or to the `Nonlinear Handler`. The `Equality Solver` keeps the equalities in parametric solved form using a refinement of the Gaussian elimination method. The `Inequality Solver` keeps the inequalities in a solved form using an incremental version of Phase I of the Two-Phase Simplex Method (see, e.g., [18]). It is capable of detecting inconsistency (when no solution can be found) as well as implicit equalities, i.e. equalities entailed by the constraint store. Finally the `Nonlinear Handler` stores nonlinear constraints for later use as passive constraints. In fact, nonlinear constraints may subsequently become linear if variables occurring in them get instantiated to grounds terms. The `Nonlinear Handler` detects this situation and issues the new linear constraints to the `Equality Solver`.

A further module (not explicitly depicted in Fig. 1) is the `Output Module`. Task of the `Output Module` is to translate the content of the constraint store into a logically equivalent formula containing only variables occurring in the input goal. To this end a variant of the Fourier's variable-elimination method is used (see, e.g., [18]).

4 Constraint Solving in Automated Deduction

Term rewriting is a fundamental and thoroughly studied technique in automated deduction (see [6] for a survey). Several extensions have been investigated to make term rewriting techniques applicable in complex applications. *Contextual rewriting* [20, 2, 8] is an extension to *conditional rewriting* [11] whereby the context surrounding the term being rewritten is used to relieve the conditions of conditional rewrites. (For instance, if the current focus of rewriting is a subterm of lit in the clause $\{lit\} \cup E$, then the set of the negations of the literals in E can be taken as the context of rewriting.) Contextual rewriting, apparently first introduced in NQTHM [4], is now widely employed, for instance, in the provers RRL [12], SPIKE [2], and STeP [15]. Here we focus on *constraint contextual rewriting* (CCR) which generalises contextual rewriting by incorporating the functionalities provided by a decision procedure. For a different approach to integrating constraint solving with term rewriting see, e.g., [13].

In what follows E and E' are ground clauses, i.e. finite sets of (Σ_j, Π_j)-literals.[1] The disjointness of Σ_p and Σ_c is not required, instead it is assumed that $\Sigma_c \subseteq \Sigma_p = \Sigma_j$. C, C', and C'' are constraint stores, i.e. sets of (Σ_j, Π_c)-atoms to be intended conjunctively. We define $\rho(\Sigma, \Pi)$ to be the union of $\tau(\Sigma)$ and the set of (Σ, Π)-atoms. In the following s, r, and t range over $\rho(\Sigma_j, \Pi_j)$. If p is a (Σ_j, Π_c)-literal then $\overline{p}$ is a set of constraints equivalent to $\neg p$ in $\mathcal{D}$, i.e. $\models_{\mathcal{D}} \overline{p} \leftrightarrow \neg p$, otherwise $\overline{p} = \emptyset$. Furthermore, if $E = \{p_1, \ldots, p_n\}$, then $\overline{E} = \bigcup_{i=1}^{n} \overline{p_i}$. R is a set of *(conditional) equations*, i.e. clauses of the form $p_1, \ldots, p_n \to l = r$, where $l \in \tau(\Sigma_j, V)$, $r \in \tau(\Sigma_j, V')$ with $V' \subseteq V$, and for $i = 1, \ldots, n$ $(n \geq 0)$ p_i is a (Σ_j, Π_j, V_i)-atom with $V_i \subseteq V$. CR is a set of *(conditional) constraint rules*, i.e. clauses of the form $p_1, \ldots, p_n \to c$, where c is a (Σ_j, Π_c, V)-atom, and for $i = 1, \ldots, n$ $(n \geq 0)$ p_i is a (Σ_j, Π_j, V_i)-atom with $V_i \subseteq V$. We assume that the models of $R \cup CR$ are $\mathcal{D}$-models.

4.1 Constraint Contextual Rewriting

In the following, we provide an abstract account of constraint contextual rewriting in the context of a process of clause simplification. Given a set of conditional

[1] In this section we assume that the set of atoms includes the propositional constants *true* and *false* denoting truth and falsity respectively.

rewrite rules R and a set of conditional constraint rules CR, the *simplification* relation $\Longrightarrow$ is the smallest transitive relation closed under the following rules of inference:

$$\frac{}{E \cup \{\mathit{true}\} \Longrightarrow \{\mathit{true}\}}\ (\textit{s-true}) \qquad \frac{}{E \cup \{\mathit{false}\} \Longrightarrow E}\ (\textit{s-false})$$

$$\frac{\overline{E} \mapsto\!\!\!\!\twoheadrightarrow C}{E \Longrightarrow \{\mathit{true}\}}\ (\textit{cl-unsat}) \text{ if } \mathtt{unsat}(C) \qquad \frac{C \mapsto C'}{C \mapsto\!\!\!\!\twoheadrightarrow C'}\ (\textit{csimp})$$

$$\frac{\overline{E} \mapsto_e (l = r)}{E \Longrightarrow E \cup \{(l \neq r)\}}\ (\textit{extract-equ}) \qquad \frac{p \rightarrow_{\overline{E}} p'}{E \cup \{p\} \Longrightarrow E \cup \{p'\}}\ (\textit{simplify})$$

The relations $\mathtt{unsat}(C)$, $C \mapsto_e a = b$, $C \mapsto C'$, and $C \mapsto\!\!\!\!\twoheadrightarrow C'$ model the interface to the constraint solver. $\mathtt{unsat}(C)$ determines the inconsistency of C, $C \mapsto_e a = b$ finds equalities entailed by the constraint store C, and $C \mapsto C'$ (which we assume to be irreflexive) denotes constraint simplification, $C \mapsto\!\!\!\!\twoheadrightarrow C'$, which for the moment coincides with constraints simplification (rule ($\textit{csimp}$)), will be extended subsequently. $s \rightarrow_C t$ models the activity of (constraint contextual) rewriting s to t using the constraint store C as the context of rewriting (see definition below).

The rules ($\textit{s-true}$) and ($\textit{s-false}$) express obvious properties of clause simplification. ($\textit{cl-unsat}$) says that if the constraint solver simplifies $\overline{E}$ to an inconsistent constraint store, then E can be simplified to the trivially valid clause $\{\mathit{true}\}$. ($\textit{csimp}$) encodes the constraint simplification mechanism. ($\textit{extract-equ}$) says that if an equality is entailed by the constraint store, then its negation can be added to the clause at hand. ($\textit{simplify}$) says that simplification of a literal p in a clause $E \cup \{p\}$ is achieved by constraint conditional rewriting p using $\overline{E}$ as the context of rewriting.

Given a set of conditional rewrite rules R, a set of conditional constraint rules CR, and a constraint store C, the *constraint contextual rewriting* relation $\rightarrow_C$ is the smallest reflexive and transitive relation over $\rho(\Sigma_j, \Pi_j)$ closed under the following rules (where $a \neq b$ abbreviates $\neg(a = b)$):

$$\frac{}{s = s \rightarrow_C \mathit{true}}\ (\textit{r-true}) \qquad \frac{}{s \neq s \rightarrow_C \mathit{false}}\ (\textit{r-false})$$

$$\frac{C \cup \overline{p} \mapsto\!\!\!\!\twoheadrightarrow C'}{p \rightarrow_C \mathit{true}}\ (\textit{true-lit}) \text{ if } \mathtt{unsat}(C')$$

$$\frac{p_1\sigma \rightarrow_C \mathit{true} \ \cdots \ p_n\sigma \rightarrow_C \mathit{true}}{s[l\sigma]_u \rightarrow_C s[r\sigma]_u}\ (\textit{crew}) \text{ if } (p_1, \ldots, p_n \rightarrow l = r) \in R$$

where $t[s]_p$ denotes the expression t with the subexpression at position p replaced by s.

($\textit{r-true}$) and ($\textit{r-false}$) say that identities and negated identities can be rewritten to *true* and *false* respectively regardless of the constraint store at hand. ($\textit{crew}$) says that in order to use a conditional equation in rewriting we must relieve its hypotheses first.

However, as illustrated by the following example, we can infer more than this. Let $\Pi_c = \{\leq\}$, $\Sigma_c = \{-, +\}$, $\Sigma_p = \{min, max\}$, and let the constraint domain be $\mathcal{R}_{lin}$, i.e. the domain of linear arithmetics over the real numbers. Furthermore, let us assume a constraint solver that is based on Fourier's elimination method.[2] Let C_0 be the constraint store $\{l - min(a) \leq 0, 1 - k \leq 0, -l + k + max(a) \leq 0\}$. The constraint solver simplifies C_0 to $C_1 = \{1 - min(a) + max(a) \leq 0\}$. Because of the presence of the uninterpreted function symbols min and max neither the consistency nor the inconsistency of C_1 can be concluded by the constraint solver. Actually, C_1 is not satisfiable, but its unsatisfiability is not a consequence of linear arithmetic reasoning only. However, if $min(X) - max(X) \leq 0$ is in CR, then it can be instantiated to $min(a) - max(a) \leq 0$ and added to the constraint store. Finally, constraint simplification yields the trivially inconsistent constraint store $\{1 \leq 0\}$. This kind of reasoning is formalised by the following rule (which extends the definition of $\mapsto\!\!\!\!\twoheadrightarrow$):

$$\frac{p_1\sigma \rightarrow_C true \ \cdots \ p_n\sigma \rightarrow_C true \quad c\sigma \rightarrow_C c' \quad C \cup \{c'\} \mapsto\!\!\!\!\twoheadrightarrow C'}{C \mapsto\!\!\!\!\twoheadrightarrow C'} \ (augment)$$

$$\text{if } (p_1, \ldots, p_n \rightarrow c) \in CR \text{ and } \texttt{csub}(C, c, \sigma)$$

where `csub` is a relation such that if $\texttt{csub}(C, c, \sigma)$ then there exists C' such that $C \cup \{c\sigma\} \mapsto C'$. `csub` models the capability of the constraint solver of finding instances of the conclusions of conditional constraint rules whose addition to the current constraint store promotes further constraint simplification. Therefore the condition $\texttt{csub}(C, c, \sigma)$ has the effect of selecting from CR a set of candidate constraint rules. The premises of the rule specify that the hypotheses of the selected constraint rule must be relieved for the constraint rule to be applicable, and that the instantiated conclusion of the constraint rule $c\sigma$ is first rewritten into c' and then added to the current constraint store which is finally simplified into C'. This gives us a stronger form of constraint simplification which is capable to exploit the knowledge about user-defined symbols encoded in CR.

For the soundness of the above calculus it suffices to assume that the relations `unsat`, $\mapsto_e$, and $\mapsto$ enjoy the following properties: if $\texttt{unsat}(C)$ then $\models_{\mathcal{D}} \neg C$; if $C \mapsto C'$, then $\models_{\mathcal{D}} C \leftrightarrow C'$; if $C \mapsto_e (a = b)$, then $C \models_{\mathcal{D}} (a = b)$. Under such assumptions it is possible to conclude that if $E \Longrightarrow E'$ then $R, CR \models_{\mathcal{D}} E' \leftrightarrow E$ and that if $s \rightarrow_C t$ then $R, CR \models_{\mathcal{D}} C \rightarrow (s \sim t)$, where "$\sim$" is "$=$" if s and t are literals and "$\sim$" is "$\leftrightarrow$" if s and t are terms.

4.2 The Simplifier of NQTHM: an Architectural View

CCR is inspired by the simplifier of NQTHM [5] which incorporates a decision procedure for linear arithmetics. An abstract description of NQTHM's simplifier is depicted in Fig. 2 (borrowed from [1]). The `SIMPLIFIER` starts by sending the

[2] For our purposes it suffices to know that the method simplifies the constraint store by linearly combining the available constraints and then eliminating the parent constraints. The exhaustive application of the method yields a constraint store whose (in)consistency can be easily checked.

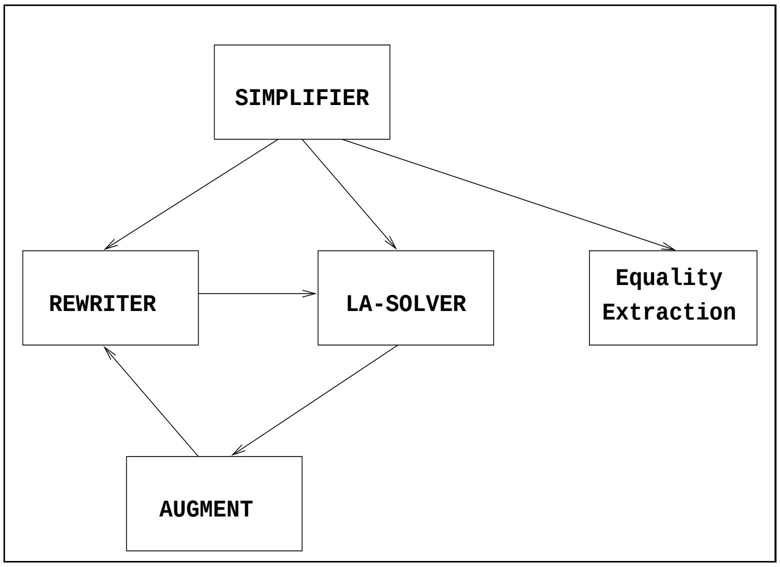

Fig. 2. The architecture of the simplifier of NQTHM

negated literals of the input clause to the solver. If inconsistency is detected, then *true* is returned (rule (*cl-unsat*)). Otherwise the equalities entailed by the constraint store are detected by the `Equality Extraction` module and then added to the clause (rule (*extract-equ*)). Then the `SIMPLIFIER` goes on by repeatedly rewriting the literals in the clause by using contextual information (rule (*simplify*)). If $\{p\} \cup E$ is the clause to be simplified and p the selected literal, then the `REWRITER` rewrites p in the context of $\overline{E}$. If the result of rewriting p is p', then the clause $\{p\} \cup E$ is simplified to $\{p'\} \cup E$ and simplification continues on the next literal in the clause.

The `REWRITER` rewrites the expression at hand either via conditional rewriting (rule (*crew*)) or by invoking the `LA-SOLVER` (rule (*true-lit*)). The `LA-SOLVER` implements a satisfiability procedure based on the Fourier elimination method. If the `LA-SOLVER` fails to detect inconsistency, then it invokes the `AUGMENT` module. The task of the `AUGMENT` module is to derive new constraints from the available constraint rules. Since constraint rules may be conditional, the `AUGMENT` module may in turn invoke the `REWRITER` to relieve the conditions of the constraint rules (rule (*augment*). The `REWRITER`, the `LA-SOLVER`, and the `AUGMENT` module are thus mutually interfaced.

5 A Comparative Analysis

As pointed out in [9], the crucial feature of the CLP scheme is that it provides a logic programming language whose operational and declarative semantics as well as the relationship between them can be parametrised by the domain of computation and constraints. The same applies to the CCR scheme, as it provides a term rewriting framework whose model-theoretic and operational semantics as well the relationship between them can be parametrised in a similar way. When we rationally reconstructed CCR from the implementation of the simplifier of NQTHM, we borrowed both the semantic framework and the methodology developed in the CLP setting. This is already an important example of cross-fertilisation between the two areas.

In what follows we compare the distinguished features of the two schemes and discuss potentials for further cross-fertilisation.

Solvers. The parametrisation of the CLP and the CCR schemes w.r.t. the constraint domain is reflected at the architectural level in a neat distinction between a general purpose inference engine and the constraint solver. However, pragmatic considerations, e.g. performance requirements, may lead to the choice of different constraint solvers. For instance, the constraint solver of the CLP($\mathcal{R}$) system consists of several specialised sub-solvers, whereas NQTHM uses a single solver based on Fourier's elimination. Even though it is well known that the Simplex Method is far more efficient than Fourier's elimination, Boyer and Moore motivated the adoption of the latter on the ground that the time spent by their integrated procedure is negligible compared to the time spent by the whole simplifier. Hence, a faster decision procedure would not affect significantly the performance of the prover. However, in order to cope with the number of linear inequalities generated by their method, they had to restrict heuristically the inference capability of their procedure. As a consequence, their solver may fail to detect inconsistency and this may lead to otherwise unnecessary recursive invocations of the rewriter (indirectly invoked by the `AUGMENT` module). A more effective decision procedure is therefore likely to yield a better performance.

A further possibility (not exploited in NQTHM and in the CCR scheme we presented) is to return a simplified representation of the constraint store as a result of the simplification process. This can be useful if other theorem proving procedures are available. For instance, take again the example from Sect. 4 where $C_0 = \{l - min(a) \leq 0, 1 - k \leq 0, -l + k + max(a) \leq 0\}$ is simplified to $C_1 = \{1 - min(a) + max(a) \leq 0\}$. C_1 can be translated back into the clause $E' = \{min(a) - max(a) \leq 0\}$[3] which can be easily proved by induction, thereby avoiding the need of resorting to user-provided constraint rules.

Logic Variables. The CLP($\mathcal{R}$) system implements a sophisticated architecture capable of dealing with instantiation of logic variables in a distributed fashion. Since no logic variables are allowed in the clause being rewritten, the problem does not occur in the CCR scheme we presented. However, in extensions to

[3] Remember that the constraint store encodes the negations of the literals in the input clause. Furthermore, since Fourier's elimination works in the constraint domain of the rationals numbers and in our example the multiplicands are integers, it is sound to translate the constraint $X < 1 + Y$ into the literal $X \leq Y$.

rewriting where logic variables are allowed, e.g. in equational logic programming [17], implementations of the CCR Schema could benefit from the techniques devised in the CLP setting.

Interpreted Function Symbols. The CCR technique of augmenting the constraint store by means of constraint rules encoding knowledge about interpreted symbols can be employed in the CLP scheme too. For instance, in a hypothetical extension of the CLP($\mathcal{R}$) system enhanced with augmentation and provided with the following constraint rule:

$$min(A) \leq max(A)$$

the goal $\leftarrow X < min(L), max(L) < X$. finitely fails.

User-Defined Constraints. Constraint solvers are usually "built in" to the host system. There are several proposals to extend CLP languages to allow the user to define new constraints. For instance, in [7] new constraints are predicates and a set of guarded rules (called *constraint simplification rules*) specify how the constraint store is to be simplified. The constraint store is then kept closed under the application of the constraint simplification rules. From an automated deduction perspective this amounts to having a general purpose solver which can be customised by means of an appropriate set of constraint simplification rules. This approach to building user-defined solvers could be easily applied in automated theorem proving and is a further promising area of cross-fertilisation.

6 Conclusions

This paper has introduced constraint contextual rewriting (CCR) as a particular integration of constraint solving and term rewriting. We have compared the application of constraint solving in CLP and in CCR at the methodological, theoretical, and implementation level. We have also given evidence that a comparative analysis of the two approaches facilitates the theoretical analysis of the different schemes and paves the way to challenge extensions.

References

1. A. Armando and S. Ranise. From Integrated Reasoning Specialists to "Plug-and-Play" Reasoning Components. In *Fourth International Conference on Artificial Intelligence and Symbolic Computation (AISC98). Plattsburgh, NY, USA. September 16-18*, 1998.
2. A. Bouhoula and M. Rusinowitch. SPIKE: A system for automatic inductive proofs. *Lecture Notes in Computer Science*, 936, 1995.
3. R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence (Logic and the Acquisition of Knowledge)*, 11:83–124, 1988.

4. R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.
5. R.S. Boyer and J.S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.
6. N. Dershowitz and J.P. Jouannaud. Rewriting systems. In *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Publishers, Amsterdam, 1990.
7. T. Fruehwirth. Constraint handling rules. *Lecture Notes in Computer Science*, 910, 1995.
8. Harald Ganzinger. A completion procedure for conditional equations. *Journal of Symbolic Computation*, 11(1-2):51–81, January–February 1991.
9. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, (19/20):503–581, 1994.
10. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *TOPLAS*, 14(3):339–395, July 1992.
11. S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2-3):175–193, October 1984.
12. Deepak Kapur and Hantao Zhang. RRL: A rewrite rule laboratory. In E. Lusk; R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 768–769, Berlin, May 1988. Springer.
13. H. Kirchner. On the use of constraints in automated deduction. *Lecture Notes in Computer Science*, 910:128–??, 1995.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
15. Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The stanford temporal prover. Technical Report CS-TR-94-1518, Stanford University, June 1994.
16. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. Technical Report STAN-CS-78-652, Stanford Computer Science Department, April 1978.
17. Michael J. O'Donnell. Equational logic programming. In C. J. Hogger D. M. Gabbay and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, *Logic Programming*, chapter 3. Oxford University Press, 1995.
18. A. Schrijver. *Theory of Linear and Integer Programming*. A Wiley-Interscience publication. Wiley, Chichester, New York, 1986.
19. Cesare Tinelli and Mehdi Harandi. Constraint logic programming over unions of constraint theories. *Journal of Logic and Functional Programming*, 1998. (to appear).
20. H. Zhang and J. L. Remy. Contextual rewriting. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 46–62, Dijon, France, May 1985. Springer.