System Description: RDL
Rewrite and Decision procedure Laboratory

Alessandro Armando, Luca Compagna, and Silvio Ranise

DIST – Università di Genova, Viale Causa 13 – 16145 Genova, Italia

1 Introduction

RDL[1] simplifies clauses in a quantifier-free first-order logic with equality using a tight integration between rewriting and decision procedures. On the one hand, this kind of integration is considered the key ingredient for the success of state-of-the-art verification systems, such as ACL2's [10], STEP [8], Tecton [9], and *Simplify* [7]. On the other hand, obtaining a principled and effective integration poses some difficult problems. Firstly, there are no formal accounts of the incorporation of decision procedures in rewriting. This makes it difficult to reason about basic properties such as soundness and termination of the implementation of the proposed schema. Secondly, most integration schemas are targeted to a given decision procedure and they do not allow to easily plug new decision procedures in the rewriting activity. Thirdly, only a tiny portion of the proof obligations arising in many practical verification efforts falls exactly into the theory decided by the available decision procedure. **RDL** solves the problems above as follows:

1. **RDL** is based on CCR (Constraint Contextual Rewriting) [1, 2], a formally specified integration schema between (ordered) conditional rewriting and a satisfiability decision procedure [11]. As a consequence, **RDL** is sound [1], terminating [2] and fully automatic.
2. **RDL** is an open system which can be modularly extended with new decision procedures provided these offer certain interface functionalities (see [2] for details).
   In its current version, **RDL** offers *'plug-and-play' decision procedures* for the theories of Universal Presburger Arithmetic over Integers (UPAI), Universal Theory of Equality (UTE), and UPAI extended with uninterpreted function symbols [13].
3. **RDL** implements instances of a *generic extension schema* for decision procedures [3]. The key ingredient of such a schema is a *lemma speculation mechanism* which 'reduces' the validity problem of a given theory to the validity problem of one of its sub-theory for which a decision procedure is available.

Since extensions of quantifier-free first-order logic with equality are useful in practically all verification efforts, **RDL** can be seen as an open reasoning module which can be integrated in larger verification systems. In fact, most state-of-the-art verification systems feature similar components, e.g. ACL2's simplifier, STEP validity checker, Tecton's integration of contextual rewriting and a decision procedure for UPAI, and *Simplify* developed within the Extended Static Checking project.

2 A Motivating Example

Consider the problem of showing the termination of a function to normalize conditional expressions in propositional logic as described in Chap. IV of [5]. The argument in the proof of termination is based on exhibiting a measure function that decreases (according to a given ordering) at each function's recursive call. ms (reported in [12]) is one such a function: $ms(a)$=1 and $ms(\mathsf{If}(x,y,z))$=$ms(x) + ms(x)ms(y)+$

[1] The system is available via the *Constraint Contextual Rewriting Project Home Page* at `http://www.mrg.dist.unige.it/ccr`.

$ms(x)ms(z)$, where If is the ternary constructor for non-atomic conditional expressions, a is an atomic conditional expression, x, y, and z are conditional expressions, and justapposition denotes multiplication. One of the proof obligation formalizing the 'decreaseness' argument above is

$$ms(\mathsf{If}(u, \mathsf{If}(v, y, z), \mathsf{If}(w, y, z))) < ms(\mathsf{If}(\mathsf{If}(u, v, w), y, z)), \tag{1}$$

where $<$ is the 'less-than' relation over integers. In order to prove the validity of (1), we check the unsatisfiability of its negation. By rewriting the l.h.s. and the r.h.s. (of the negation of (1)) with the definition of ms, we obtain:

$$\begin{array}{l} u + uv + uvy + uvz + uw + uwy + uwz \geq \\ u + uv + uw + uy + uvy + uwy + uz + uvz + uwz \end{array} \tag{2}$$

where u, v, y, w and z abbreviates $ms(u)$, $ms(v)$, $ms(y)$, and $ms(z)$, respectively. Then, we perform all the possible cancellations in (2) and we obtain:

$$ms(u)ms(y) + ms(u)ms(z) \leq 0. \tag{3}$$

We assume the availability of the following two facts:

$$ms(E) > 0, \tag{4}$$

$$(X > 0 \wedge Y > 0) \Rightarrow XY > 0. \tag{5}$$

for each conditional expression E and for each pair of numbers X and Y. Then, consider the following two instances of (5):

$$(ms(u) > 0 \wedge ms(y) > 0) \Rightarrow ms(u)ms(y) > 0 \tag{6}$$

$$(ms(u) > 0 \wedge ms(z) > 0) \Rightarrow ms(u)ms(z) > 0. \tag{7}$$

In order to relieve the hypotheses of (6) and (7), it is sufficient to instantiate (4) three times, namely $ms(u) > 0$, $ms(y) > 0$, and $ms(z) > 0$. Finally, it is trivial to detect the unsatisfiability of (3) and the conclusions of (6) and (7).

Three (cooperating) reasoning capabilities are required to automate the above reasoning: (*i*) *rewriting*, (*ii*) *satisfiability checking* and *normalization in a given theory*, and (*iii*) *ground lemma speculation* (in a sense that will be made clear later). The first is used to simplify formulae, e.g. unfolding the definition of ms in (1). The second presents two aspects: the simplification of a literal, e.g. canceling out common terms in (2), and the check for the unsatisfiability of (conjunctions of) literals, e.g. the conclusions of lemmas (6) and (7) with (3). The third is the capability of supplying instances of valid facts to (partially) interpret user-defined function symbols occurring in the current formula, e.g. two instances of (4) are used to relieve hypotheses of (6).

3 Architecture

RDL features a tight integration (based on CCR) of three modules implementing the reasoning capabilities mentioned above: a module for *ordered conditional rewriting*, a *satisfiability decision procedure*, and a module for *lemma speculation*.

In the following, let cl be the clause to be simplified and p be a literal in cl which is going to be rewritten. The *context* C associated to p is the conjunction of the negation of the literals occurring in cl except p. Let T be the theory decided by the decision procedure.

The decision procedure. For efficiency reasons, this module is required to be state-based, incremental, and resettable [11]. The context C is stored by a specialized data structure in the state of the decision procedure. There are three functionalities. First, `cs-unsat` characterizes a set of inconsistent (in T) contexts whose inconsistency can be checked by means of computationally inexpensive checks. Second, given a literal l and the current context C, `cs-simp` computes the new context C' resulting from the addition of l to C in such a way that C' is entailed by the conjunction of l and C in T. Third, `cs-norm` computes a normal representation p' of p w.r.t. T and the information stored in C. This functionality must be compatible with rewriting, i.e. it is required that $p' \prec p$ where $\prec$ denotes a total term ordering on ground literals. (The actual implementation provides a fixed total ordering on ground **RDL** literals.)

Constraint Contextual Rewriting. The rewriter provides the functionality `ccr`. It handles conditional rules of the form $h_1 \wedge \cdots \wedge h_n \Rightarrow (l = r)$, where l and r are **RDL** terms, and $h_1, ..., h_n$ are **RDL** literals. Assume $r\sigma \prec l\sigma$ for a ground substitution σ (otherwise, if $l\sigma$ is different from $r\sigma$, swap l with r in the following). Given $p[l\sigma]$, `ccr` returns $p[r\sigma]$ if $h_1\sigma, ..., h_n\sigma$, and $p[r\sigma]$ are smaller (w.r.t. $\prec$) than $p[l\sigma]$, and for $i = 1, ..., n$ either $h_i\sigma$ is (recursively) rewritten to *true* by invoking `ccr`[2] or by checking whether $h_i\sigma$ is entailed by C (this is done by invoking `cs-unsat` so to check that the negation of $h_i\sigma$ is inconsistent with C). There are two other means of rewriting. Firstly, p is rewritten to $false$ (*true*) if `cs-unsat` checks that p (the negation of p, resp.) is inconsistent with C. Secondly, p is rewritten to p' if p' has been obtained by invoking `cs-norm`.

Lemma speculation. Three instances of the lemma speculation mechanism described in [3] are implemented in **RDL**. All the instances share the goal of feeding the decision procedure with new facts about function symbols which are otherwise uninterpreted in T. More precisely, they inspect the context C and return a set of ground facts entailed by C using T as the background theory. Furthermore, these facts must enjoy some properties to ensure termination (see [3, 2] for details).

The simplest form of lemma speculation is `augment` [6, 1–3], which consists of selecting and instantiating lemmas from a set of available valid formulae in order to obtain ground facts whose conclusions can be readily used by the decision procedure. More precisely, `augment` finds instances of the conclusions among the conditional lemmas which can promote further inference steps in the decision procedure. There are two crucial problems. Firstly, we must relieve hypotheses of lemmas in order to be able to send their conclusions to the decision procedure. We solve this problem by rewriting each hypothesis to *true* (if possible). This is done by invoking `ccr` and it implies that the rewriter and the decision procedure are mutually recursive. The other problem is the presence of extra variables in the hypotheses (w.r.t. the conclusion) of lemmas. **RDL** avoids this problem by requiring that the conclusion contains all the variables occurring in the lemma and that all the variables get instantiated by matching the conclusion of the lemma against the largest (according to $\prec$) literal in C. As an example of how `augment` works, recall that (6) and (7) are generated by matching the conclusion of lemma (5) against (3) twice.

If a suitable set of lemmas is defined, `augment` increases dramatically the effectiveness of the decision procedure. Unfortunately, devising such a suitable set is a time consuming activity. This problem can be solved in some important special cases. In the actual version of **RDL**, `affinize` implements the 'on-the-fly' generation of lemmas about multiplication over integers. To understand how `affinize` works, consider the non-linear inequality $X\,Y \leq -1$ (where X and Y range over integers). By resorting to its geometrical interpretation, it is easy to verify that $X\,Y \leq -1$ is equivalent to $(X \geq 1 \wedge Y \leq -1) \vee (X \leq -1 \wedge Y \geq 1)$. To avoid case splitting, we observe that the semi-planes represented by $X \geq 1$ and $X \leq -1$ as those represented by $Y \leq -1$ and $Y \geq 1$ are non-intersecting. This allows to derive the following four lemmas: $X \geq 1 \Rightarrow Y \leq -1$, $X \leq -1 \Rightarrow Y \geq 1$, $Y \geq 1 \Rightarrow X \leq -1$, and $Y \leq -1 \Rightarrow X \geq 1$. This process can be generalized to non-linear inequalities which can be put in the form $X\,Y \leq K$ (where K is an integer) by factorization. The generated (conditional) lemmas are used as for `augment`.

On the one hand `affinize` can be seen as a significant improvement over `augment` since it does not require any user intervention. On the other hand it fails to apply when inequalities cannot be transformed

[2] **RDL** performs no case splitting while releaving hypotheses of conditional lemmas.

#	PROBLEM	RDL
1	$f(A) = f(B) \Rightarrow (r(g(A,B),A) = A) \vdash$ $r(g(y,z),x) = x \vee \neg(g(x,y) = g(y,z)) \vee \neg(y = x)$	26
2	$A * B = B * A, (\neg(C = 0)) \Rightarrow (rem(C * D, C) = 0) \vdash$ $rem(y * z, x) = 0 \vee \neg(x * y = z * y) \vee x = 0$	109
3	$(A > 0) \Rightarrow (rem(A * B, A) = 0) \vdash \ rem(x * y, x) = 0 \vee x \leq 0$	12
4	$min(A) \leq max(A) \vdash$ $\neg(k \geq 0) \vee \neg(l \geq 0) \vee \neg(l \leq min(b)) \vee \neg(0 < k) \vee l < max(b) + k$	12
5	$(memb(A,B)) \Rightarrow (len(del(A,B)) < len(B)) \vdash$ $\neg(w \geq 0) \vee \neg(k \geq 0) \vee \neg(z \geq 0) \vee \neg(v \geq 0) \vee \neg(memb(z,b))$ $\vee\neg(w + len(b) \leq k) \vee w + len(del(z,b)) < k + v$	17
6	$(0 < A) \Rightarrow (B \leq A * B), 0 < ms(C) \vdash$ $ms(c) + ms(d)^2 + ms(b)^2 < ms(c) + ms(b)^2 + 2ms(d)^2 * ms(b) + ms(d)^4$	72
7	$A \geq 4 \Rightarrow (A^2 \leq 2^A) \vdash \neg(c \geq 4) \vee \neg(b \leq c^2) \vee \neg(2^c < b)$	14
8	$(max(A,B) = A) \Rightarrow (min(A,B) = B), (p(C)) \Rightarrow (f(C) \leq g(C)) \vdash$ $\neg(p(x)) \vee \neg(z \leq f(max(x,y))) \vee \neg(0 < min(x,y)) \vee \neg(x \leq max(x,y)) \vee$ $\neg(max(x,y) \leq x) \vee z < g(x) + y$	114
9	$0 < ms(C) \vdash$ $ms(c) + ms(d)^2 + ms(b)^2 < ms(c) + ms(b)^2 + 2ms(d)^2 * ms(b) + ms(d)^4$	63
10	$\vdash \ x \geq 0 \Rightarrow x^2 - x + 1 \neq 0$	40

Table 1. Experimental Results

into a form suitable for affinization. **RDL** combines augmentation and affinization by considering the function symbols occurring in the context C, i.e. the top-most function symbol of the largest (according to $\prec$) literal in C triggers the invocation of either affinization or augmentation.

4 Experiments

RDL must be judged w.r.t. its effectiveness in simplifying (and possibly checking the validity of) proof obligations arising in practical verification efforts where decision procedures play a crucial role. Hence, standard benchmarks for theorem provers (e.g. TPTP) are not in the scope of **RDL**. We are currently building a corpus of proof obligations extracted from the literature as well as examples available for similar components integrated in verification systems. The problems selected for the corpus are *representative* of disparate verification scenarios and are considered *difficult* for current state-of-the-art verification systems.

Table 1 reports the results of our computer experiments. PROBLEM lists the available lemmas[3] (if any) and the formula to be decided. $\vdash$ is the binary relation characterizing the deductive capability of **RDL** (we have that $\vdash$ is contained in $\models_T$, where T is the theory decided by the available decision procedure extended with the available facts). The last column record the successful (time is expressed in msec) or the unsuccessful (–) attempt to solve a problem by **RDL**.[4]

RDL solves problems 1 and 2 with a decision procedure for UTE. In the former, the decision procedure is used to derive equalities entailed by the context which are used as rewrite rules and enable the use of the available lemma. The ordered rewriting engine implemented by **RDL** is a key feature to successfully solve problem 2 since this form of rewriting allows to handle usually non-orientable rewrite rules such as $A * B = B * A$. **RDL** solves problem 3 with a decision procedure for UPAI. Infact, the available lemma is applied once its instantiated condition, namely $x > 0$, is relieved by the decision

[3] Capitalized letters denote implicitly universally quantified variables.

[4] Benchmarks run on a 600 MHz Pentium III running Linux. **RDL** is implemented in Prolog and it was compiled using Sicstus Prolog, version 3.8.

procedure (it is straightforward to check the inconsistency of $x > 0$ and the literal $x \leq 0$ in the context). **RDL** solves problems 4, 5, 6, and 7 with a decision procedure for UPAI and `augment`. In particular, the formula of problem 6 is a non-linear formula whose validity is successfully established by **RDL** in a similar way of the example in Section 2. **RDL** solves problem 8 with the combination of a decision procedure for UPAI and for UTE. **RDL** solves problems 9 and 10 with the combination of a decision procedure for UPAI, `augment` and `affinize`. The lemma about multiplication (i.e. $0 < I \Rightarrow J \leq I * J$) is supplied in problem 6 but it is not in problem 9. Only the combination of `augment` and `affinize` can solve problem 9. Finally, problem 10 shows the importance of the context in which proof obligations are proved (since **RDL** does not case-split). In fact, without $x \geq 0$ `augment` and `affinize` would not be able to solve problem 10. This shows the importance of the context in which proof obligations are proved (since **RDL** does not case-split).

As a matter of fact, the online version of STeP fails to solve all of the problems reported in Table 1. However, most of the problems are successfully solved by the improved version of STeP described in [4]. This version of STeP solves problems 9 and 10 by resorting to a partial method for quantifier elimination (see [4] for details). Instead, our extension schema is able to prove the formula with simpler mathematical techniques. The comparison is somewhat difficult since the method used by STeP works over the rationals and our affinization technique only works over integers. However, our affinization technique can be used also over rationals to approximate classes of non-linear inequalities. *Simplify* successfully solves problems 1 to 8 thanks to a Nelson-Oppen combination of decision procedure and an incomplete matching algorithm which is capable of instantiating (valid) universally quantified clauses. However, it does not solve problems 9 and 10 since it is unable to handle non-linear facts without user-supplied lemmas (such as, e.g., $0 < I \Rightarrow J \leq I * J$ in problem 6). Finally, SVC fails to solve all the problems involving augmentation and affinization since it does not provide a mechanism to take into account facts which partially interpret user-defined function symbols.

References

1. A. Armando and S. Ranise. Constraint Contextual Rewriting. In *Proc. of the 2nd Intl. Workshop on First Order Theorem Proving (FTP'98), Vienna (Austria)*, pages 65–75, 1998.
2. A. Armando and S. Ranise. Termination of Constraint Contextual Rewriting. In *Proc. of the 3rd Intl. Workshop on Frontiers of Combining Systems (FroCos'2000)*, March 2000.
3. Alessandro Armando and Silvio Ranise. A Practical Extension Mechanism for Decision Procedures: the Case Study of Universal Presburger Arithmetic. *Formal Methods and Tools, special issue of J. of Universal Computer Science (To appear)*, 2001.
4. N. S. Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Computer Science Department, Stanford University, 1998.
5. R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
6. 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.
7. D. L. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. Technical report, DEC, 1996.
8. Z. Manna *et. al.* STeP: The Stanford Temporal Prover. Technical Report CS-TR-94-1518, Stanford University, June 1994.
9. D. Kapur, D.R. Musser, and X. Nie. An Overview of the Tecton Proof System. *Theoretical Computer Science*, Vol. 133, October 1994.
10. M. Kaufmann and J S. Moore. Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. on Software Engineering*, 23(4):203–213, April 1997.
11. G. Nelson and D. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. of the ACM*, 27(2):356–364, April 1980.
12. L. C Paulson. Proving termination of normalization functions for conditional expressions. *J. of Automated Reasoning*, pages 63–74, 1986.
13. R.E. Shostak. Deciding Combination of Theories. *Journal of the ACM*, 31(1):1–12, 1984.