A Reconstruction and Extension of Maple’s Assume Facility via Constraint Contextual Rewriting

Alessandro Armando

DIST, Università di Genova, Viale Causa 13, 16145 Genova, Italy

Clemens Ballarin

Fakultät für Informatik, Technische Universität München, 85748 Garching, Germany

Abstract

Maple’s symbolic evaluator, together with a feature that is usually known as the assume facility, implements a powerful form of conditional rewriting. In a previous paper the authors showed that Maple’s evaluation process can be recast as constraint contextual rewriting (CCR), a form of conditional rewriting that incorporates the services provided by a decision procedure through a well-specified interface. In the present paper, this analysis is extended to a component of the assume facility that deals with problems beyond linear arithmetic and that we call the general solver. This led to the discovery of a fault that causes Maple to return wrong results with some contexts. The reason for this is that the facility wrongly assumes that the general solver is complete in the sense that it uses all the available assumptions in the context. While a simple fix to this problem would reduce the logical strength of the assume facility, we show that a more general approach inspired by techniques available in CCR do not suffer from the problem and naturally lead to stronger forms of simplification.

1 Introduction

Computer algebra and automated reasoning both study problems in the domain of symbolic computation, and there is ample scope for cross-fertilisation.

Email addresses: armando@dist.unige.it (Alessandro Armando), ballarin@in.tum.de (Clemens Ballarin).

Preprint submitted to Journal of Symbolic Computation 30 September 2003
Computer algebra systems are used in by theorem provers as libraries of efficient algorithms to carry out sophisticated computations in algebraic domains (Ballarin and Paulson, 1999; Harrison and Théry, 1998; Armando and Zini, 2001).

On another, conceptual level, techniques from equational reasoning can be used to analyse algebraic simplifiers. The subject of this paper is the reconstruction of Maple’s symbolic evaluator and its assume facility, introduced by Weibel and Gonnet (1993) to solve inconsistencies arising from rules like $\sqrt{x^2} = x$. This rule is often wrong — for example if $x$ denotes a real number and $x < 0$. Removing the rule makes the simplifier correct, but also less powerful — for example if $x \geq 0$. The assume facility provides a way out of the dilemma: it maintains a context which enables the user to specify properties of terms, and the rule is applied to an expression $\sqrt{a^2}$ only if $a \geq 0$ can be derived from the context.

The notion of context plays also a key role in Constraint Contextual Rewriting (CCR, for short, see Armando and Ranise, 2003). CCR is a powerful form of conditional rewriting which incorporates the services provided by a decision procedure. In CCR contextual information is stored and manipulated by the decision procedure whose interface functionalities are neatly specified in an abstract way. CCR is at the core of the simplifier of RDL (Armando et al., 2001), a fully automatic theorem prover for the quantifier-free fragment of first-order logic featuring a tight integration between rewriting and the available decision procedures.

The authors have shown (Armando and Ballarin, 2001) that Maple’s evaluation process can, in general, be recast as CCR. While that work was restricted to the solving activity used by the assume facility as abstractly described in (Weibel and Gonnet, 1993), the present paper extends this analysis and provides an account of the solving capabilities based on an inspection of the actual implementation. This led to the discovery of a fault that causes Maple to return wrong results with some contexts. The reason for this is that the facility is based on the assumption that one of its modules — the general solver — is complete in the sense that it uses all the available assumptions in the context. This is not the case.

Our CCR-based reconstruction of the assume facility does not suffer from the problem as it fails in the above cases (as Maple should do) instead of returning a wrong answer. Moreover, we show that by enabling a powerful mechanism available in CCR, called augmentation, we get a new, strengthened form of simplification that — among many other things — handles successfully the above cases by returning a correct answer.
Structure of the paper. We start in Section 2 by providing a brief description of Maple’s evaluator and the assume facility. In Section 3 we introduce CCR, and then, in Section 4, provide our CCR-based reconstruction of Maple’s evaluator and the assume facility and show that — by enabling augmentation — we get a stronger form of simplification. We conclude in Section 5 with some final remarks.

2 Maple’s Evaluator and the Assume Facility

The assume facility is used by Maple’s symbolic evaluator. It is called in order to resolve branching problems, and to enable transformations that are not generally valid. The facility consists of two sub-modules: the property reasoner and the solver. In the following, the focus is on the interplay between evaluator and the modules of the assume facility. A first view of this, obtained by inspecting the code, is depicted in Figure 1. The solver consists of a module for linear programming problems and a more general solver. The latter is invoked when linear programming fails. The database contains user-supplied assumptions as well as built-in knowledge about the predefined functions. It is possible to extend this database for user-supplied functions, but to do so, the user needs to know about the assume facility’s internals.
Maple’s symbolic evaluator is closely intertwined with the language interpreter, which is also known as evaluator. Symbolic expressions are evaluated, either after parsing or by calling the function \texttt{eval}.

The internal working of \texttt{eval} is complex. Before evaluation proper, an internal simplifier (different from the library function \texttt{simplify}) is called. This carries out arithmetic reasoning on some numeric types including integers, rational and floating point numbers, and puts expressions into a simple normal form: rational numbers are cancelled, sums and products are flattened, and some logic terms are simplified. Further, common numeric factors are pulled out of sums, and powers with common bases are grouped together in products.

The simplified statement is then evaluated. Notably, evaluation deals with function application, and this is how symbolic evaluation is implemented in Maple. Every operation symbol \( f \) has an associated procedure (whose name is also \( f \)) that implements the operation. We call these procedures \emph{evaluation functions} and denote with \( f_e \) the evaluation function of the operation symbol \( f \). Some evaluation functions — for example, \( + \) — are built in and cannot be changed by the user. Others are part of the library — for example, \sin. As evaluation functions are invoked by the evaluator, user-provided functions have the same status as functions in the library. By defining a procedure with name \( f \) the user provides a new definition for the function symbol \( f \). Function symbols without associated evaluation function remain uninterpreted. Also, evaluation functions can return unevaluated expressions. This occurs, for instance, when \( a + b \) is evaluated where \( a \) and \( b \) are distinct uninterpreted symbols, as this expression cannot be simplified. In this case, the evaluation function \( + \) returns \( a + b \). Compound expressions are evaluated recursively, where function arguments are evaluated in an eager fashion, that is, before entering the function.

Because an evaluation function \( f_e \) is a procedure it can call arbitrary procedures. This facility is used in particular to perform conditional evaluation. Either conditions are again evaluated, or queries to the context are made by invoking the property reasoner through its function \texttt{is}. Table 1 shows a few of the simplifications performed by Maple’s square root function. It uses signum, the sign function for real and complex expressions, to decide whether its argument is negative or not in the current context.

The abstract view of Maple’s symbolic evaluator that we adopt for the purpose of this paper is that of a conditional rewrite system, together with a suitable strategy that specifies in which order rewrite rules are applied. This is certainly sufficient to describe the symbolic evaluator, because any computable function
Table 1

Some simplifications performed by Maple’s functions \( \sqrt{\cdot} \) and \( \text{signum} \).

<table>
<thead>
<tr>
<th>Simplification</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sqrt{x^{2n}} = x^n )</td>
<td>( n &gt; 0, n \in \mathbb{Z} ) ( \text{signum}(x) = 1 )</td>
</tr>
<tr>
<td>( \sqrt{x^{2n+1}} = x^n x^{\frac{1}{2^n}} )</td>
<td>( n &gt; 0, n \in \mathbb{Z} ) ( \text{signum}(x) = 1 )</td>
</tr>
<tr>
<td>( \sqrt{x^{2n}} = (-x)^n )</td>
<td>( n &gt; 0, n \in \mathbb{Z} ) ( \text{signum}(x) = -1 )</td>
</tr>
<tr>
<td>( \sqrt{x^{2n+1}} = (-x)^n i(-x)^{\frac{1}{2n}} )</td>
<td>( n &gt; 0, n \in \mathbb{Z} ) ( \text{signum}(x) = -1 )</td>
</tr>
<tr>
<td>( \text{signum}(x^n) = 1 )</td>
<td>( n ) even ( \text{Im}(x) = 0 ) ( x \neq 0 )</td>
</tr>
<tr>
<td>( \text{signum}(x^n) = \text{signum}(x) )</td>
<td>( n ) odd ( \text{Im}(x) = 0 )</td>
</tr>
<tr>
<td>( \text{signum}(x) = 1 )</td>
<td>( x : \text{real}, 0 &lt; x )</td>
</tr>
<tr>
<td>( \text{signum}(x) = -1 )</td>
<td>( x : \text{real}, x &lt; 0 )</td>
</tr>
<tr>
<td>( \text{signum}(x) = 0 )</td>
<td>( x : \text{real}, x = 0 )</td>
</tr>
<tr>
<td>( \text{signum}(x) = \text{normal}(\frac{x}{\text{abs}(\text{Im}(x))}) )</td>
<td>Re and ( \text{Re}(x) = x_r ), ( \text{abs}(\text{Im}(x)) \neq 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{Im occur} ) ( \text{Im}(x) = x_i ),</td>
</tr>
<tr>
<td></td>
<td>( \text{neither in} ) ( x_r = 0 ),</td>
</tr>
<tr>
<td></td>
<td>( x_i ) ( \text{or} ) ( x_r ) ( \text{abs}(\text{Im}(x)) \neq 0 )</td>
</tr>
</tbody>
</table>

can be written as a rewrite system, including operations on floating point numbers and arbitrary-precision integers. This view is also adequate for our purposes: firstly, it describes closely what happens conceptually in Maple. Secondly, it is open to user-defined functions, which can be modelled by adding rewrite rules.

2.2 The Property Reasoner

Maple’s assume facility is centred around the semantics notions of objects and properties. The former comprise Maple’s objects of computation, e.g., the real number \( \pi \) or the square root function \( \sqrt{x} \), the latter are sets of objects, e.g., the interval \([0, +\infty)\) or the set of continuous functions. Objects and properties have their syntactical counterparts in object and property terms respectively.

The set of object terms comprises the usual mathematical expressions, e.g. \( \sin(x + \frac{\pi}{2}) \) and \( \int e^x dx \), with their usual interpretation. Property terms are either

(i) atomic symbols denoting basic properties, e.g. real, positive, monotonic,
(ii) object terms denoting numbers,
(iii) expressions of the form \( (l, r), [l, r), (l, r], \) and \( [l, r] \) where \( l \) and \( r \) are either object terms denoting numbers or the symbols \( -\infty, +\infty, \) or \( \vee \).
(iv) are built out of simpler property terms using the constructors \( \neg, \wedge, \) and \( \vee \).

Since it is always clear from the context whether an object term denotes an
object or a property both sorts are conceptually distinct. We write \( \{t\} \) if \( t \) is an object term denoting a property.

Since property terms denote sets of objects it is natural to interpret them in terms of a Boolean algebra, i.e. a complemented lattice with a top \( \top \) (the set of all objects), a bottom \( \bot \) (the empty set of objects), a partial ordering \( \preceq \) (set inclusion), and the induced lattice operators \( \land \) (set intersection) and \( \lor \) (set union). Here are examples of facts in this algebra:

\[
\begin{align*}
\text{positive} & \preceq \text{real} \\
(\text{positive} \land \text{real}) & \preceq \text{positive} \\
((-\infty, 0] \lor [0, \infty)) & \preceq \text{real} \\
(\text{positive} \land \text{rational}) & \preceq [0, \infty)
\end{align*}
\]

If \( t \) is an object term and \( p \) a property term, then we say that \( t \) has property \( p \), in symbols \( t : p \), if and only if the object denoted by \( t \) is in the set of objects denoted by \( p \).

The context of Maple’s evaluator is a set of assumptions of the form \( t : p \). The context is updated via the commands \texttt{assume}(\( t, p \)) and \texttt{additionally}(\( t, p \)) whose effect is that of adding \( t : p \) to the current context. For the user’s convenience, single, Boolean-valued expressions are allowed as arguments to \texttt{assume} and \texttt{additionally}; these commands are treated as equivalent to ones with two arguments. For instance, \texttt{assume}(\( a \geq 0 \)) has the same effect as \texttt{assume}(\( a, [0, +\infty) \)). The main functionality of the assume facility is to determine whether \( t \) has a property \( p \) in the current context and it is accessible via the query \texttt{is}(\( t, p \)).

The features of the assume facility we have described so far already support simple forms of reasoning, but Maple goes beyond that. For instance, with what we have described so far, it is possible to determine that \( a : \text{real} \) holds in a context containing the assumption \( a : \text{positive} \), but it is not possible to establish that \( a \cdot a : \text{non-negative} \) follows from a context containing \( a : \text{real} \). What is missing in this case is the knowledge about the relation between the properties of the operands of the multiplication operation and the property of the returned value. Maple solves this problem by associating property functions to the function symbols occurring in objects terms.

Given a function symbol \( f \), a property function associated with \( f \) is a procedure \( \bar{f} \) that maps tuples of properties into properties such that

\[
f(t_1, \ldots, t_n) : \bar{f}(p_1, \ldots, p_n) \text{ for all object terms } t_1 : p_1, \ldots, t_n : p_n.
\]

In order to determine whether the term \( f(t_1, \ldots, t_n) \) has property \( p \) in a context \( C \supseteq \{t_1 : p_1, \ldots, t_n : p_n\} \), the property reasoner simply checks whether \( \bar{f}(p_1, \ldots, p_n) \preceq p \). If the check succeeds then a positive answer is returned.
Table 2
Some simplifications performed by the property functions \(\sqcup, \tau,\) and \(\text{sin}\).

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bot)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(a)</td>
<td>(a)</td>
</tr>
<tr>
<td>({0})</td>
<td>({0})</td>
</tr>
<tr>
<td>([l_1, r_1])</td>
<td>([l_2, r_2])</td>
</tr>
<tr>
<td>(\text{irrat})</td>
<td>(\text{irrat})</td>
</tr>
<tr>
<td>(\text{complex} + a)</td>
<td>(\text{complex, if } b \subseteq \text{complex})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(a)</td>
<td>({0})</td>
</tr>
<tr>
<td>({0})</td>
<td>({0})</td>
</tr>
<tr>
<td>({1})</td>
<td>({1})</td>
</tr>
<tr>
<td>([l_1, r_1])</td>
<td>([l_2, r_2])</td>
</tr>
<tr>
<td>(\text{irrat})</td>
<td>(\text{irrat})</td>
</tr>
<tr>
<td>(\text{complex} \tau b)</td>
<td>(\text{complex, if } b \subseteq \text{complex})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>({0})</td>
<td>({0})</td>
</tr>
<tr>
<td>([l, r])</td>
<td>([\min R, \max R])</td>
</tr>
<tr>
<td>([l, \infty))</td>
<td>([-1, 1])</td>
</tr>
<tr>
<td>((\infty, r])</td>
<td>([-1, 1])</td>
</tr>
<tr>
<td>(\text{real})</td>
<td>([1, 1])</td>
</tr>
<tr>
<td>(\text{complex})</td>
<td>(\text{complex})</td>
</tr>
</tbody>
</table>

Examples of property functions and their definitions in the assume facility are listed in Table 2.

The situation is a bit more involved when the property reasoner is asked to determine whether a term \(\phi[t_1, \ldots, t_n]\) built out of multiple function symbols and the terms \(t_1, \ldots, t_n\) has property \(p\) in a context \(C \supseteq \{t_1 : p_1, \ldots, t_n : p_n\}\). In this case, the following general properties of property functions are used together with property functions to compute a property \(p_0\) such that \(\phi[t_1, \ldots, t_n] : p_0\):

\[
a \preceq b \implies \tilde{f}(\ldots, a, \ldots) \preceq \tilde{f}(\ldots, b, \ldots)
\]
\[
\tilde{f}(\ldots, a \lor b, \ldots) = \tilde{f}(\ldots, a, \ldots) \lor \tilde{f}(\ldots, b, \ldots)
\]
\[
\tilde{f}(\ldots, a \land b, \ldots) \preceq \tilde{f}(\ldots, a, \ldots) \land \tilde{f}(\ldots, b, \ldots)
\]
A final check \( p_0 \preceq p \) determines the answer returned by the property reasoner. For some \( f \) full \( \wedge \)-distributivity, i.e.

\[
\tilde{f}(\ldots, a \land b, \ldots) = \tilde{f}(\ldots, a, \ldots) \land \tilde{f}(\ldots, b, \ldots)
\]

holds. If the strong form holds for all functions occurring in \( \phi \), then \( p_0 \) is the least property of \( t \) in the current context.

The built-in domain knowledge used by the property reasoner consists of

- the partial order \( \preceq \) on basic properties,
- equations in the property lattice, e.g. \( \text{rational} \lor \text{irrat} = \text{real} \), and
- property functions that evaluate properties.

2.3 The Solver

When the property reasoner is asked to determine the property of an object term \( \phi[s_1, \ldots, s_m] \), it might be the case that not all the terms \( s_1, \ldots, s_m \) occurring in it are explicitly available in the current context, say \( C = \{ t_1 : p_1, \ldots, t_n : p_n \} \). In such a case it is likely that the procedure for determining properties of object terms presented in Section 2.2 will be unable to find a property of \( \phi[s_1, \ldots, s_m] \). When not all terms are available in the context, the solver is invoked with the task of reformulating \( \phi[s_1, \ldots, s_m] \) into one or more equivalent object terms of the form \( \phi'[t_{j_1}, \ldots, t_{j_k}] \) where \( \{ t_{j_1}, \ldots, t_{j_k} \} \subseteq \{ t_1, \ldots, t_n \} \). This is done by observing that even if the object terms available in the context are not identical to those occurring in \( \phi[s_1, \ldots, s_m] \), they may still contain occurrences of them. Thus the solver may think of the context as stating properties about \( s_1, \ldots, s_m \) and this can be made explicit by writing \( C = \{ t_1[s_1], \ldots, s_m : p_1, \ldots, t_n[s_1], \ldots, s_m : p_n \} \).

For arbitrary \( t_1, \ldots, t_n \) it is difficult to infer properties of \( s_1, \ldots, s_m \) from the assumptions. The assume facility is at the core of Maple in the sense that it is called by the symbolic evaluator frequently. It is therefore restricted to two important classes of problems that can be solved quickly.

2.3.1 Linear Programming

If \( \phi \) and \( t_1, \ldots, t_n \) are linear combinations of \( s_1, \ldots, s_m \) and \( p_1, \ldots, p_n \) are real intervals, then the query can be solved by linear programming. The assumptions collectively define a polyhedron. Its vertices are computed via Gaussian elimination. Minimum and maximum of the objective function \( \phi \) are attained on vertices and provide the range \( r \) of values \( \phi \) can take. A query \( \phi : p \) returns \text{true} if \( r \preceq p \) and \text{false} otherwise.
Example 1 Let $x \geq 0$, $y \geq 0$ and $x + y \leq \frac{\pi}{2}$ be assumptions on $x$ and $y$, and let $0 \leq 2 \cdot x + y \leq \pi$ be the query.

The polyhedron defined by the assumptions is a triangle with vertices $(0, 0)$, $(0, \frac{\pi}{2})$, and $(\frac{\pi}{2}, 0)$. The values of the objective function $2 \cdot x + y$ at the vertices are $0$, $\frac{\pi}{2}$, and $\pi$, respectively. Hence the range $r$ is $[0, \pi]$ and linear programming returns true.

2.3.2 General Solving

If the query is not a linear programming problem then the solver introduces new constants $c_1, \ldots, c_n$ and defines them by

$$
\begin{align*}
  t_1[s_1, \ldots, s_m] &= c_1 \\
  &\vdots \\
  t_n[s_1, \ldots, s_m] &= c_n
\end{align*}
$$

The context $C = \{t_1[s_1, \ldots, s_m] : p_1, \ldots, t_n[s_1, \ldots, s_m] : p_n\}$ can thus be reformulated as $C' = \{c_1 : p_1, \ldots, c_n : p_n\}$. The solver then considers (1) as a system of equations in the indeterminates $s_1, \ldots, s_m$ and tries to solve it. If (1) is under-determined, then the solver fails. Otherwise the solver considers subsystems of (1), i.e., systems of the form:

$$
\begin{align*}
  t_{j_1}[s_1, \ldots, s_m] &= c_{j_1} \\
  &\vdots \\
  t_{j_k}[s_1, \ldots, s_m] &= c_{j_k}
\end{align*}
$$

where $\{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}$ and tries to solve them. If a subsystem (2) is solvable then its solution is of the form:

$$
\begin{align*}
  s_1 &= t'_1[c_{j_1}, \ldots, c_{j_k}] \\
  &\vdots \\
  s_m &= t'_m[c_{j_1}, \ldots, c_{j_k}]
\end{align*}
$$

By using the equations in (3) as rewrite rules, $\phi[s_1, \ldots, s_m]$ is readily reformulated into an equivalent object term $\phi'[c_{j_1}, \ldots, c_{j_k}]$ whose property $p'$ can be determined using the information available in $C' = \{c_1 : p_1, \ldots, c_n : p_n\}$.

The solver considers subsystems in the light of the query $\phi : p$ and distinguishes three cases.

(Case 1) If for one subsystem $p' \preceq p$ can be shown with the property reasoner then the solver returns true.

(Case 2) If for one subsystem the computation of the solution or of the bound fails then the solver fails.
(Case 3) If for all subsystems \( p' \not\preceq p \) then the solver returns \textit{false}. This is generally wrong, as illustrated by Example 3 below.

The solver is superior to linear programming in that object terms and terms occurring in the context need not be linear combinations in \( s_1, \ldots, s_m \). However, solving may fail for non-linear equations. These occur for object terms that are not linear combinations. The solver can deal with a mix of linear and non-linear equations. In the linear case, Gauß-Jordan elimination is used. For non-linear equations Maple’s general solver \texttt{solve} is invoked in an attempt to isolate a variable.

The following example shows a successful invocation of the solver that could not have been solved by linear programming, despite all the occurring equations are linear.

**Example 2** Let \( x \geq 0, x \leq \frac{\pi}{2}, x + y \geq 0 \) and \( x + y \leq \frac{\pi}{2} \) be the assumptions on \( x \) and \( y \). The corresponding property judgements are \( x : [0, \frac{\pi}{2}] \) and \( x+y : [0, \frac{\pi}{2}] \).

In order to decide \( \sin(2 \cdot x + y) \geq 0 \), which is a consequence of the assumptions, the solver is invoked with the task of expressing \( \sin(2 \cdot x + y) \) in terms of \( c_1 = x \) and \( c_2 = x + y \). The only solvable subsystem is the whole system, and elimination yields the object \( \sin(c_1 + c_2) \). This is evaluated to \( [0, 1] \) by the property reasoner and thus \( \sin(2 \cdot x + y) \geq 0 \) is shown.

The solver is, however, inferior to linear programming in the sense that it cannot always use all assumptions. This is illustrated by the next example.

**Example 3** Let \( x \geq 0, y \geq 0, \) and \( x + y \leq \frac{\pi}{2} \) be the assumptions. By Example 1, \( 0 \leq 2 \cdot x + y \leq \pi \), and it thus follows that \( \sin(2 \cdot x + y) \geq 0 \) also holds here. The corresponding property judgements of the assumptions are \( x : [0, \infty), y : [0, \infty), \) and \( x+y : (-\infty, \frac{\pi}{2}] \).

In order to decide \( \sin(2 \cdot x + y) \geq 0 \), the solver is invoked with the task of expressing \( \sin(2 \cdot x + y) \) in terms of \( c_1 = x, c_2 = y, \) and \( c_3 = x + y \). Three subsystems lead to the objects \( \sin(2 \cdot c_1 + c_2), \sin(2 \cdot c_3 - c_2), \) and \( \sin(c_3 + c_1) \). All are evaluated to \( [-1, 1] \) by the property reasoner and therefore \( \sin(2 \cdot x + y) \geq 0 \) is not shown. Moreover, because \( [-1, 1] \not\subseteq [0, \infty) \) for all subsystems, the solver wrongly returns \textit{false}.

Property terms may not contain object variables. Hence expressing the assumptions from Example 3 with only two property judgements is not possible. It follows that the combination of the solver and the property reasoner is not complete for problems with linear assumptions but non-linear queries, and the system must fail in Case 3 instead of returning \textit{false}. The behaviour of Example 3 was observed by all versions of Maple available to the authors, including Version 9.
3 Constraint Contextual Rewriting

This form of conditional rewriting is an extension to contextual rewriting and maintains the context by means of a separate module that implements a (semi-)decision procedure. CCR provides an effective integration of the decision procedure with rewriting. The aim of this work is to recast Maple’s symbolic evaluation process as CCR, and hence we give a brief introduction.

Typically, expressions admissible to the decision procedure are a subset of the expressions admissible to the rewriter, and also the theory of the decision procedure is only a fragment of the theory defined by the rewrite system. In particular, care has to be exercised to distinguish expressions admissible to the context and to the rewriter. The integration provided by CCR is effective, because it makes facts of the larger theory available to the decision procedure through a technique called augmentation.

Let $\Sigma$, $\Pi$ (possibly subscripted) denote finite sets of function and predicate symbols (with their arity), respectively. A signature is a pair of the form $(\Sigma, \Pi)$. $V$ (possibly subscripted) denotes a finite set of variables. A $(\Sigma, V)$-term is a term built out of the symbols in $\Sigma$ and the variables in $V$ in the usual way. A $(\Sigma, \Pi, V)$-atom is either an expression $q(t_1, \ldots, t_n)$ where $q \in \Pi$ and $t_i$ is a $(\Sigma, V)$-term ($i = 1, \ldots, n$) or one of the propositional constants true and false denoting truth and falsity respectively. The usual definitions of formula, clause, literal as given in textbooks on mathematical logic, e.g., Mendelson (1964), and those of substitution and position in an expression, e.g., as given in Dershowitz and Jouanna (1990), are assumed.

If $a$ is an atom, then $\hat{a}$ abbreviates $\neg a$ and $\neg a$ stands for $a$. If $Q$ is a set of literals, then $\hat{Q}$ abbreviates $\{\hat{q} : q \in Q\}$, $Q \implies p$ abbreviates the clause $\hat{Q} \cup \{p\}$, and $\land Q$ stands for a conjunction of the literals in $Q$. To simplify notation we write $q_1, \ldots, q_n \implies q$ in place of $\{q_1, \ldots, q_n\} \implies q$.

If $\phi$ is a formula and $\Gamma$ is a set of formulæ, then $\phi$ is a logical consequence of $\Gamma$ if and only if $\Gamma \models \phi$, where $\models$ denotes entailment in classical predicate logic with equality. A $(\Sigma, \Pi, V)$-theory is a set of $(\Sigma, \Pi, V)$-formulæ closed under logical consequence. If $T$ is a theory, then $\Gamma \models_T \phi$ abbreviates $T \cup \Gamma \models \phi$. $\phi$ is $T$-satisfiable if and only if there exists a model of $T \cup \{\phi\}$, and $T$-unsatisfiable otherwise. $\phi$ is $T$-valid if and only if $\phi$ is a logical consequence of $T$ or, equivalently, if and only if $\phi \in T$.

It is convenient to model the functionalities provided by the reasoning modules by means of contextual reduction relations. Given two sets of expressions $\mathcal{C}$ and $\mathcal{E}$, a contextual reduction relation is a ternary relation $\rightarrow : \mathcal{C} \times \mathcal{E} \times \mathcal{E}$ enjoying the following two properties:
**Reflexivity:** $c :: e \vdash e$ for all $c \in C$ and all $e \in \mathcal{E}$,

**Transitivity:** if $c :: e \vdash e'$ and $c :: e' \vdash e''$, then $c :: e \vdash e''$ for all $c \in C$ and all $e, e', e'' \in \mathcal{E}$.

In the following, two theories $T_c$ and $T_j$ of signature $(\Sigma_c, \Pi_c)$ and $(\Sigma_j, \Pi_j)$ respectively are considered, s.t. $\Sigma_c \subseteq \Sigma_j$, $\Pi_c \subseteq \Pi_j$, and $T_c \subseteq T_j$. The objective will be to simplify $(\Sigma_j, \Pi_j)$-expressions using a decision procedure for $T_c$.

### 3.1 Reasoning Specialist

According to the usual definition, a **decision procedure** for $T_c$ is a procedure which takes a $(\Sigma_c, \Pi_c)$-formula as input and returns a ‘yes-or-no’ answer indicating whether the input formula is $T_c$-satisfiable or not. Unfortunately, although simple and conceptually elegant, this definition is seldom adequate in practical applications. Efficiency considerations require the procedure to be **incremental**, i.e. capable of processing parts of the input problem as soon as they become available. This generalised notion of decision procedure is captured by the notion of reasoning specialist. A **reasoning specialist** is a state-based procedure whose states (called **constraint stores**) are finite sets of $(\Sigma_c, \Pi_c)$-literals represented in some internal form and whose functionalities are abstractly characterised in the following way:

- **cs-init($C$):** **true** only if $C$ is $T_c$-valid, e.g. the empty constraint store.
- **cs-unsat($C$):** **true** only if $C$ is $T_c$-unsatisfiable.
- $P :: C \xrightarrow{\text{cs-simp}} C'$: this is the main functionality of the reasoning specialist, i.e. the activity of adding a finite set of $(\Sigma_j, \Pi_j)$-literals $P$ to $C$ yielding a new constraint store $C'$. For soundness it is required that $(\text{sound.cs-simp})$ if $P :: C \xrightarrow{\text{cs-simp}} C'$ then $P, C \models T_c \land C'$.

Note that **cs-simp** must map $(\Sigma_j, \Pi_j)$-literals into $(\Sigma_c, \Pi_c)$-literals in a suitable way.

**Example 4 (A reasoning specialist for total orders)** Let $\Pi_c = \{=, \leq\}$ and let $T_c$ be a $(\Sigma_c, \Pi_c)$-theory for total orders. Constraint stores are finite sets of $(\Sigma_c, \Pi_c)$-literals of the form $s \leq t$, $s = t$, or $s \neq t$ closed under the following inference rules:

1. (transitivity) $s \leq t$, $t \leq u \vdash s \leq u$
2. (antisymmetry) $s \leq t$, $t \leq s \vdash s = t$

**cs-init($C$) holds if and only if $C = \emptyset$.** *cs-unsat($C$) holds if and only if $C$ contains two literals of the form $s = t$ and $s \neq t$, or of the form $s = t$ and $t \neq s$.*

Let $\Sigma_j = \Sigma_c$, $\Pi_j = \Pi_c \cup \{<, >, \geq\}$, and let $\nu$ be the function mapping
Table 3
Definition of $\nu(c)$ (see Example 4)

<table>
<thead>
<tr>
<th>$c$</th>
<th>$\nu(c)$</th>
<th>$c$</th>
<th>$\nu(c)$</th>
<th>$c$</th>
<th>$\nu(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \leq t_2$</td>
<td>${t_1 \leq t_2}$</td>
<td>$t_1 \leq t_2$</td>
<td>${t_1 \leq t_2, t_1 \neq t_2}$</td>
<td>$t_1 \geq t_2$</td>
<td>$\nu(t_2 \leq t_1)$</td>
</tr>
<tr>
<td>$t_1 \neq t_2$</td>
<td>${t_1 \neq t_2}$</td>
<td>$t_1 \leq t_2$</td>
<td>${t_2 \leq t_1, t_1 \neq t_2}$</td>
<td>$t_1 \neq t_2$</td>
<td>$\nu(t_1 &lt; t_2)$</td>
</tr>
<tr>
<td>$t_1 \neq t_2$</td>
<td>${t_2 \leq t_1}$</td>
<td>$t_1 &gt; t_2$</td>
<td>$\nu(t_2 &lt; t_1)$</td>
<td>$\text{otherwise}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

$(\Sigma_c, \Pi_j)$-literals to sets of $(\Sigma_c, \Pi_c)$-literals as defined in Table 3. $\nu$ is extended to sets of $(\Sigma_c, \Pi_j)$-literals, say $P$, by $\nu(P) = \bigcup_{c \in P} \nu(c)$. $P \xrightarrow{\text{cs-simp}} C'$ holds if and only if $C'$ is the result of closing $C \cup \nu(P)$ w.r.t. the application of (transitivity) and (antisymmetry). It is straightforward to verify that $P :: C \xrightarrow{\text{cs-simp}} C'$ enjoys (sound.cs-simp). To illustrate, if $0, a, b, \sin, \cos \in \Sigma_c$ and $P = \{\sin a > 0, \cos a \geq \sin a\}$, then $P :: \emptyset \xrightarrow{\text{cs-simp}} \{0 \leq \sin a, 0 \neq \sin a, \sin a \leq \cos a, 0 \leq \cos a\}$.

### 3.2 Constraint Contextual Rewriting

Let $C$ and $R$ be a constraint store and a set of rewrite rules respectively. Then $e$ rewrites to $e'$ in context $C$, in symbols $C :: e \xrightarrow{\text{ct}} e'$, if and only if either

(i) $e$ is a literal, $\{\hat{e}\} :: C \xrightarrow{\text{cs-simp}} C'$, $\text{cs-unsat}(C')$, and $e' = \text{true}$; formally

\[
\text{(ext-entails)} \quad \{\hat{p}\} :: C \xrightarrow{\text{cs-simp}} C' \quad \text{if } p \text{ is a literal and } C :: p \xrightarrow{\text{ct}} \text{true} \quad \text{cs-unsat}(C')
\]

or

(ii) $(Q \Rightarrow l = r) \in R$, $e_u = l \sigma$, $e' = e[r \sigma]_u$, and $C :: q \sigma \xrightarrow{\text{ct}} \text{true}$ for all $q \in Q$; formally

\[
\text{(e-rew)} \quad C :: q \xrightarrow{\text{ct}} \text{true} \quad \text{for all } q \in Q \quad \text{if } (Q \Rightarrow l = r) \in R \text{ and } \sigma \text{ is a ground substitution}
\]

In the second rule, which describes conditional rewriting, $l \sigma$ denotes application of substitution $\sigma$ to term $l$, $e_u$ denotes the subterm of $e$ at position $u$, and $e[t]_u$ is the term obtained from $e$ by replacing the subterm at position $u$ with $t$.

Notice that the context in the above rules coincides with the constraint store of the reasoning specialist.

**Example 5 (CCR)** Let us consider the reasoning specialist of Example 4

\[13\]
and let R consist of the following rewrite rule:

\[ X \geq 0 \implies \sqrt{X^2} = X \] (4)

It is easy to verify that \( \sqrt{\cos^2 a} \) rewrites to \( \cos a \) in context \( C = \{ 0 \leq \sin a, 0 \neq \sin a, \sin a \leq \cos a \} \) as witnessed by the following derivation:

\[
\begin{align*}
\{ \cos a \geq 0 \} & \quad \vdash \quad C \\
\{ \cos a \geq 0 \} & \quad \vdash \quad C' \\
C & \quad \vdash \quad \text{true}
\end{align*}
\]

where \( C' = C \cup \{ 0 \leq \cos a, \cos a = 0, 0 \neq \cos a \} \) is readily found \( T_c \)-unsatisfiable by \( \text{cs-unsat} \).

3.3 Augmenting the Constraint Store

Although the notion of CCR defined so far is already a significant improvement over the usual forms of (conditional) rewriting, there is still room for improvement. A serious limitation is revealed by the situation in which the rewriting context is \( T_j \)-unsatisfiable but not \( T_c \)-unsatisfiable. When this is the case, the \( T_j \)-unsatisfiability of the rewriting context cannot possibly be detected by the reasoning specialist. The occurrence in the rewriting context of (function) symbols interpreted in \( T_j \) but not in \( T_c \) is the main cause of the problem. The following example illustrates this.

Example 6 (Augmentation) Let us consider the reasoning specialist of Example 4 and let \( R \) contain (4) and the following fact:

\[ X \geq 0, X \leq \pi \implies \sin X \geq 0 \] (5)

\( \sqrt{\cos^2 a} \) cannot possibly be rewritten to \( \cos a \) in context \( C = \{ 0 \leq a, a \leq \pi, \sin a \leq \cos a \} \) as there does not exist any \( T_c \)-unsatisfiable constraint store \( C' \) such that \( \{ \cos a \geq 0 \} \vdash C' \) holds. This is because \( 0 \leq \sin a \) is not \( T_c \)-entailed by \( C \). On the other hand, since the facts in \( R \) are assumed to be \( T_j \)-valid, \( 0 \leq \sin a \) is \( T_j \)-entailed by \( C \).

A way out of the problem is to extend the rewriting context with \( T_j \)-valid facts, thereby informing the reasoning specialist about properties of function symbols it is not aware of. By adding \( T_j \)-valid facts to the rewriting context, the approach aims at generating a \( T_j \)-equivalent but \( T_c \)-unsatisfiable context whose \( T_j \)-unsatisfiability can therefore be detected by the reasoning specialist. The selection of suitable \( T_j \)-valid facts is done by looking through the available
lemmas. This is formalised by the inference rule

\[
\begin{align*}
C :: q\sigma &\rightarrow \text{true } (\forall q \in Q) \quad \{c\sigma\} :: C \xrightarrow{\text{cs - simp}} C' \quad (Q \rightarrow c) \in R \\
\text{(augment)} &
P :: C \xrightarrow{\text{cs - simp}} C'
\end{align*}
\]

This inference rule states that the context can be extended, for arbitrary \( P \), with a new literal \( \sigma \) provided that a clause of the form \((Q \rightarrow c)\) is in \( R \), \( \sigma \) is a ground substitution, and the instantiated hypotheses \( Q\sigma \) can be established.

**Example 7 (Augmentation—continued)** Thanks to (augment) we now have that \( \{\cos a \not\geq 0\} :: C \xrightarrow{\text{cs - simp}} C' \) where \( C' \) is detected to be \( T_i \)-unsatisfiable by \( \text{cs-unsat} \). From this, it readily follows that \( \sqrt{\cos^2 a} \) rewrites to \( \cos a \) in context \( C \).

It is fairly simple to show the soundness of CCR. Technically this amounts to showing that

(i) if \( C :: e \rightarrow e' \), then \( C \models_{T_j} e \sim e' \) and

(ii) if \( P :: C \xrightarrow{\text{cs - simp}} C' \), then \( P, C \models_{T_j} \land C' \).

where \( e \sim e' \) stands for \( e = e' \) if \( e \) and \( e' \) are terms, and for \( e \leftrightarrow e' \) if \( e \) and \( e' \) are formulae, respectively.

The problem with the augmentation rule is both the selection and the instantiation of the facts from \( R \). In the general case this can lead to the exploration of huge (even infinite) search spaces. Suitable control strategies must therefore be put in place to constrain the applicability of the rule.\(^1\) However in many cases simple strategies based on heuristic criteria for the selection and instantiation of the rules work well in practice (cf. Example 9 below).

Notice that the augmentation heuristics is crucial to obtain an effective integration: in Boyer and Moore (1988) it is shown that without the heuristics the reasoning specialist is of limited use, whereas its introduction improves dramatically the performance of the prover (both in speed and in decreased user interaction). It is also worth emphasising that both the calculus and its properties are independent from the theory decided by the reasoning specialist. Therefore, CCR can be applied in a wide variety of situations.

\(^1\) It must be noted that the simplified version of CCR presented in this paper is not guaranteed to terminate. However it can be refined to ensure termination. The refined calculus as well as its soundness and termination proofs are available in Armando and Ranise (2003).
4 Maple’s Symbolic Evaluation as CCR

The interplay between Maple’s evaluation process, the property reasoner, and the solver, including linear programming and — more interestingly — general solving can be recast in the CCR framework. This is shown in the rest of the paper.

4.1 Evaluation as Rewriting

The first step is to model Maple’s symbolic evaluation as rewriting. The key idea is to regard evaluation functions as rewrite rules. This is adequate because most evaluation functions perform local transformations on terms and evaluate the subterms recursively. More intricate computations are performed by the evaluator, for instance, when a factoriser is called. In this framework, that amounts to calling a further reasoning specialist, and is beyond the goal of the present analysis.

Evaluation functions test conditions, either by recursively calling the evaluator or by invoking the assume facility. Both cases are readily modelled in CCR by the rules \((c\text{-rew})\) and \((c\text{-entails})\) respectively.

4.2 Property Reasoner as Reasoning Specialist

The constraint store is a finite set of judgements of the form \(t : p\) where \(t\) is an object term and \(p\) is an property term. We abbreviate \(\{t_1 : p_1, \ldots, t_n : p_n\}\) with \(\vec{t} : \vec{p}\). We assume that \(\text{cs-init}(C)\) holds if and only if \(C\) is the empty set of judgements and that \(\text{cs-unsat}(C)\) holds if and only if either

1. \((u : \bot) \in C\) or
2. \((u : p_0) \in C\) and \(-\{u : p\} \in C\) for some term \(u\) and properties \(p_0 \preceq p\).

The application of \(\text{additionally}(t, p)\) on constraint store \(C\) leads to constraint store \(C'\) if and only if \(\{t : p\} \vdash C \xrightarrow{\text{cs-simp}} C'\). The command \(\text{assume}\) is similar, but requires \(\text{cs-init}(C)\). The invocation of \(\text{is}(t, p)\) in context \(C\) corresponds to \(\{-(t : p)\} \vdash C \xrightarrow{\text{cs-simp}} C'\) and \(\text{cs-unsat}(C')\), \(P \vdash C \xrightarrow{\text{cs-simp}} C'\) is modelled by the following inference rules:

\[
\frac{}{(\text{assume}) \quad P \vdash C \xrightarrow{\text{cs-simp}} P \cup C}
\]
Let \( p \) and the elements of \( \bar{p} \) be real intervals and let \( p_0 \) be the real interval comprising the set of values of \( u \) in the region defined by \( \bar{t} : \bar{p} \), then
\[
(\text{lin-progr}) \quad P :: \{- (u : p)\} \cup (\bar{t} : \bar{p}) \xrightarrow{\text{cs-simp}} \{u : p_0, \neg (u : p)\} \cup (\bar{t} : \bar{p})
\]

General solving is modelled by the following inference rule:
\[
(\text{solve}) \quad (\bar{t} : \bar{p}) :: u \xrightarrow{\text{solve}} v[\bar{t}] \quad \bar{v} [\bar{p}] \xrightarrow{\text{prop-eval}} p_0 \quad P :: \{- (u : p)\} \cup (\bar{t} : \bar{p}) \xrightarrow{\text{cs-simp}} \{u : p_0, \neg (u : p)\} \cup (\bar{t} : \bar{p})
\]

The premise \((\bar{t} : \bar{p}) :: u \xrightarrow{\text{solve}} v[\bar{t}]\) models the invocation to the solver that determines an object term \( v[\bar{t}] \) equivalent to \( u \) but built out of the terms occurring in the context. The premise \( \bar{v} [\bar{p}] \xrightarrow{\text{prop-eval}} p_0\) models the activity of simplifying the property term \( \bar{v}[\bar{p}] \) by invoking the corresponding property functions. Note that this rule models the behaviour of the solver in Case 3 correctly in the sense that the solver fails.

**Example 8 (Property Reasoning)** Let us consider the problem of determining whether \( \sin(2 \cdot x + y) : [0, \infty) \) in a context \( C = \{x : [0, \frac{\pi}{2}], x + y : [0, \frac{\pi}{2}]\} \) (cf. Example 2). This can be done by showing that \( C :: \sin(2 \cdot x + y) : [0, \infty) \xrightarrow{\text{ccf}} \text{true} \). By applying (ext-entails) this reduces to finding a \( T_e \)-unsatisfiable context \( C' \) such that
\[
\{\neg(s : [0, \infty))\} :: C \xrightarrow{\text{cs-simp}} C',
\]
where \( s \) abbreviates \( \sin(2 \cdot x + y) \). By (assume) and transitivity, this amounts to finding a \( T_e \)-unsatisfiable \( C' \) such that
\[
\{\neg(s : [0, \infty))\} :: \{\neg(s : [0, \infty))\} \cup C \xrightarrow{\text{cs-simp}} C'.
\]

This done by a single application of (solve):
\[
C :: s \xrightarrow{\text{solve}} \sin(x + (x + y)) \quad \sin([0, \frac{\pi}{2}] + [0, \frac{\pi}{2}]) \xrightarrow{\text{prop-eval}} [0, 1] \quad \{\neg(s : [0, \infty))\} :: \{\neg(s : [0, \infty))\} \cup C \xrightarrow{\text{cs-simp}} \{s : [0, 1], \neg(s : [0, \infty))\} \cup C
\]
whose effect is to set \( C' = \{s : [0, 1], \neg(s : [0, \infty))\} \cup C \) which is trivially found \( T_e \)-unsatisfiable by \text{cs-unsat} (cf. Clause 2 of the definition of \text{cs-unsat}).

Note that the evaluator and the property reasoner operate on the same set of expressions. Hence \( \Sigma_e = \Sigma_j \) and \( \Pi_e = \Pi_j \). The theory \( T_e \) consists of the facts known to the property reasoner and the solver; \( T_j \) may contain additional, user-provided facts.

17
4.3 Improving the Solver: Augmentation

Our reconstruction of the solver behaves correctly on the problem from Example 3 by failing. A closer inspection of the problem reveals that it contains a linear sub-problem, namely that of deriving \(0 \leq 2 \cdot x + y \leq \pi\) from \(x \geq 0, y \geq 0\) and \(x + y \leq \frac{\pi}{2}\). The difficulty lies in identifying the subgoal \(2 \cdot x + y : [0, \pi]\) from the goal \(\sin(2 \cdot x + y) \geq 0\). This can be achieved by means of the augmentation rule.

The simplification of property terms performed by property functions can be recast as augmentation. Indeed a property function for \(f\) can be encoded by a lemma stating properties of \(f\). For instance, if \(p, p_1, \ldots, p_n\) are properties and \(\bar{f}(p_1, \ldots, p_n) = p\), then this fact can be encoded by the formula:

\[
X_1 : p_1, \ldots, X_n : p_n \implies f(X_1, \ldots, X_n) : p
\]  

(6)

If \(f\) is uninterpreted for the reasoning specialist, then augmentation can extend the current context with the conclusions of appropriate instances of (6) thereby enabling the reasoning specialist to conclude without resorting to (solve). By adding the augmentation rule, the reasoning power of the assume facility can be extended considerably, as illustrated by the following example. Formulae encoding the property functions of Table 2 are given in Table 4.

Example 9 (Property Reasoning via Augmentation) Let \(R\) be the list of facts in Table 4 and let us consider again the problem analysed in Example 3, i.e. that of determining whether \(\sin(2 \cdot x + y) : [0, \infty)\) in context \(C = \{x : [0, \infty), y : [0, \infty), x + y : (-\infty, \frac{\pi}{2})\}\). Similarly to Example 8, by applying (contains), (assume) and transitivity, the problem boils down to determining a \(T_c\)-unsatisfiable context \(C'\) such that

\[
\{\neg(s : [0, \infty))\} :: \{\neg(s : [0, \infty))\} \cup C \quad \xrightarrow{\text{cs-simp}} \quad C'
\]

where \(s\) abbreviates \(\sin(2 \cdot x + y)\). A single application of (augment) does the job, by first selecting the following instance from the facts available in \(R\)

\[
2 \cdot x + y : [0, \pi] \implies \sin(2 \cdot x + y) : [0, 1]
\]

(7)

and then by adding its conclusion to \(C\) thereby yielding \(C' = \{s : [0, 1], \neg(s : [0, \infty))\} \cup C\) which is trivially \(T_c\)-unsatisfiable. The premise \(2 \cdot x + y : [0, \pi]\) is readily found to be a consequence of \(C\) by (lin-progr).

As suggested by the above example a simple strategy for the selection and instantiation of the rules in \(R\) amounts to selecting a rule \((Q \implies c) \in \bar{R}\) and a substitution \(\sigma\) such that \(\sigma\) leads to a context which is immediately found \(T_c\)-unsatisfiable by cs-unsat. In the example above this heuristics readily
identifies (7) as candidate rule instance. Notice however that this heuristics alone does not ensure termination as the same augmentation step may be attempted over and over again in the attempt of relieving the condition. This happens, for instance, whenever the condition of the rule is not $T_e$-entailed by the context. A simple way to avoid this form of divergence amounts to disabling the application of the augmentation rule while trying to establish the conditions of the rules.

CCR does not support the specification of proof strategies that would allow, for instance, to precisely characterise and reason about the heuristics outlined above. We believe that this could be done by adapting the meta-languages developed for analogous purposes in tactic-based theorem provers, see e.g. (Paulson, 1979), and in rewriting frameworks, see e.g., (Borovanský et al., 1997). But this clearly goes beyond the scope of the present paper.
4.4 Beyond Property Functions

An important observation is that not all the properties of functions can be expressed by property functions. For instance, the fact

$$ Y - X : [0, \infty) \implies f(Y) - f(X) : [0, \infty) $$

expressing the monotonicity of \( f \) is not of the form (6) and therefore it cannot be expressed as a property function. This is not surprising as monotonicity of a function cannot be expressed as a property of its input argument.

Augmentation does not suffer from the above limitation. This is best illustrated by an example.

**Example 10 (Beyond Property Functions)** Let \( R \) contain the following facts:

\[
X : [0, \infty) \wedge Y - X : [0, \infty) \implies \sqrt{Y} - \sqrt{X} : [0, \infty) \quad (8)
\]
\[
X : [-\frac{\pi}{2}, \frac{\pi}{2}] \wedge Y : [-\frac{\pi}{2}, \frac{\pi}{2}] \wedge Y - X : [0, \infty) \implies \sin Y - \sin X : [0, \infty) \quad (9)
\]

and consider the problem of determining whether \( \sqrt{\sin y} - \sqrt{\sin x} : [0, \infty) \) holds in a context \( C = \{ x : [0, \frac{\pi}{2}], y : [0, \frac{\pi}{2}], y - x : [0, \infty) \} \). This can be done by showing that \( C :: \sqrt{\sin y} - \sqrt{\sin x} : [0, \infty) \xrightarrow{\text{ccl}} \text{true} \). By applying (ctxt-entails) this reduces to showing that \( \{ -(\sqrt{\sin y} - \sqrt{\sin x}) : [0, \infty) \} :: C \xrightarrow{\text{cs-imp}} C' \) where \( \text{cs-unsat}(C') \). First — using (9) — \( C \) is augmented with \( \sin y - \sin x : [0, \infty) \), then — using (8) — with \( \sqrt{\sin y} - \sqrt{\sin x} : [0, \infty) \). Premises of the lemmas are deduced from the context \( C \); in particular \( \sin x : [0, \infty) \), the first premise of (8), is derived from \( x : [0, \frac{\pi}{2}] \).

5 Conclusion

Maple’s symbolic evaluation process results from the combination of specialised reasoning modules: the evaluator, the property reasoner, a solver for linear programming problems, and a general solver. An attractive feature of the resulting simplifier is that certain properties of arbitrary functions can be encoded via property functions and be exploited by the property reasoner. This is a very useful feature as it makes the simplifier user-programmable and hence extensible. However, properties functions allow to express a restricted class of properties and this limits the potential of the technique.

In this paper we have shown that an extended form of rewriting developed in

20
the context of Automated Reasoning, CCR, can be used to give a neat account of Maple’s evaluation process. This is achieved by regarding evaluation as rewriting and by modelling property reasoning as a special form of contextual reasoning that exploits knowledge declaratively encoded in logical formulae. Our reconstruction of Maple’s evaluation process as CCR gives new, important insights: (i) a bug in the actual implementation of Maple’s simplifier has been identified; (ii) an extension to the evaluation process that enables the use of a considerably wider class of properties than those expressible by means of property functions has been put forward. These results provide further evidence of the good potential of the cross-fertilisation between Computer Algebra and Automated Reasoning and we believe that further efforts in this direction would be highly rewarding.

We conclude by mentioning that many other works identify limitations of and propose improvements over existing simplification mechanisms. For instance, Stoutemyer (1991) points out that the semantics of symbols can be unclear in these systems. Also, Corless and Jeffrey (1992) discuss possible forms of user interaction suitable to inform the user about additional contextual information that may be required to continue a simplification or that has been assumed by the simplifier of its own accord, as well as enabling the user to input such information. Maple’s assume facility — via the assume command — allows a priori assumptions only. The assume facility also addresses the problem that contextual information needs to be passed around inside the system. Various other approaches have been presented to this end, notably dynamic evaluation in the domain of algebraic numbers (Duval, 1994), and guarded expressions (Dolzmann and Sturm, 1997). A symbolic computation can be seen as a tree where nodes are branching points, and branches denote assumptions. Dynamic evaluation maintains all paths simultaneously. This is implemented either by computing in parallel or by back-tracking if assumptions on the current path are found inconsistent (Broadbery et al., 1995). The latter strongly resembles CCR with a depth-first search strategy and where the reasoning specialist performs greatest common divisor computations on polynomials that represent algebraic numbers. Dynamic evaluation does not employ the augmentation rule. This is not surprising since it works over a domain where new function symbols cannot be introduced. Guarded expressions maintain contextual information directly in the expressions. Thus, there is no immediate connection to our reconstruction of Maple’s simplifier.

More in general, it is worth pointing out that our work differs from these in that it aims at a reconstruction of a simplifier of a state-of-the-art computer algebra system and in doing so it identifies a serious bug as well as important weaknesses; furthermore a way to fix the bug is proposed together with improvements that, if implemented in Maple, could lead to a much stronger tool. Finally, our work has been done by borrowing ideas developed in the context of automated deduction and therefore it contributes to the cross-fertilisations
of the two areas.

Acknowledgements

We are grateful to Waterloo Maple Inc., and in particular to Jacques Carette, for their cooperation and the provision of Maple source code. The work has been funded in part by the European Commission via the Research Training Network “CALCULEMUS: Systems for Integrated Computation and Deduction” (HPRN-CT-2000-00102).

References