Integrating Decision Procedures in Reflective Rewriting-Based Theorem Provers [⋆]

Manuel Clavel, Miguel Palomino, and Juan Santa-Cruz

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{clavel,miguelpt,juansc}@sip.ucm.es

Abstract

We propose a design for the integration of decision procedures in reflective rewriting-based equational theorem provers. Rewriting-based equational theorem provers use term rewriting as their basic proof engine; they are particularly well suited for proving properties of equational specifications. A reflective rewriting-based theorem prover is itself an executable equational specification, which has reflective access to the rewriting engine responsible of its execution to efficiently accomplish rewriting-based equational proofs. This reflective access means that the built-in rewriting engine can be called with different rewriting commands. This opens up the possibility of interpolating calls to appropriate decision procedures in the midst of a rewriting-based proof step —typically to solve a condition in the application of a conditional equation. To illustrate our proposal and show its feasibility, we explain how our reflective design ideas can be used to integrate a decision procedure for Presburger arithmetic in the ITP tool, a reflective rewriting-based equational theorem prover, written entirely in Maude, for proving properties of Maude equational specification.

Key words: Reflection, rewriting, theorem proving, strategies.

1 Introduction

Many authors have stressed the importance of integrating decision procedures, that is, algorithms that for particular theories can automatically decide whether a given formula is valid or not, in the proof engines of (semi-)automated theorem provers. As Boyer and Moore wrote [3]: "It is generally agreed that when practical theorem provers are finally available they will contain both heuristic components and many decision procedures." Decision

⋆ Research supported by Spanish MCYT Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000.

procedures are indeed at the core of many industrial-strength verification systems such as ACL2 [16], PVS [22], STeP [18], or Z/Eves [23]. The crucial role of decision procedures motivates the development of ICS [10], an efficient decision procedure for a fragment of first-order logic, that can be used as a standalone application and may also be included as a library in any application that requires embedded deduction.

Rewriting-based theorem provers [12,19,11,15] use term rewriting [1] as their basic proof engine, and they are particularly useful for proving properties of equational specifications. As is well known, when a finite equational specification is Church-Rosser and terminating, term rewriting can be used as an efficient procedure for deciding equalities between terms: two terms are provably equal if and only if their canonical forms, which are computed by using the equations as simplifications rules, are syntactically identical. There are, however, many practical equational specifications that contain function symbols that are not equationally defined. Consider, for example, those specifications that use first-order arithmetic: typically, they do not contain the equational definitions of the arithmetic function symbols and relations. This omission poses no problem when *evaluating* functional expressions not containing indeterminate values, since practical executable specification languages [7,12,9] provide internal links to built-in implementations of the arithmetic functions and relations. The problem arises, however, when *proving* properties.[1] In general, these proofs require solving arithmetic formulas containing indeterminate values. In those situations, the built-in implementations of the arithmetic functions and relations are useless, and the lack of equations explicitly defining them prevents us from applying term rewriting.[2] In some cases, however, we can overcome the difficulty in a general, non-ad hoc form by calling appropriate decision procedures. This is the case, for example, when the formula to be solved falls within the class of Presburger linear arithmetic formulas.

In summary, decision procedures are also important for practical rewriting-based theorem provers, and they must be integrated with their basic rewriting engines in such a way that calls to the appropriate decision procedure can be interpolated in the midst of rewriting-based proof steps. The design and implementation of the RRL [15] reflects indeed the relevance of decision procedures in rewriting-based theorem provers, and the experiments reported in [14] show that "the use of the procedure for Presburger arithmetic has made the proofs compact and relatively easier to automate and understand in contrast to proofs generated without using Presburger arithmetic." In this paper we propose a novel *reflective design* for the integration of decision procedures in rewriting-based equational theorem provers. Although our proposal can

[1] The inequality function symbol `_=/=_`, that is provided as a built-in function in [7,12], gives rise to similar complications, as discussed in detail in [13, Sec.2.1.1].

[2] Notice also that, as pointed out in [13], "in fact, there is no set of equations that can allow the automatic verification of *all* properties of integer expressions which contain indeterminate values [...]; in other words, first order arithmetic is 'undecidable' [20]."

be formulated in a general form using the axiomatic definitions of reflective logics and reflective programming languages [5], we rather illustrate it here by explaining the integration of a decision procedure for Presburger arithmetic in the ITP tool.

The ITP tool [4] is an experimental rewriting-based theorem prover for proving properties of equational specifications; it accepts equational specifications presented as functional modules of the Maude system [7]. The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called membership equational logic [21]. Thus functional modules support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort. In addition, operators can be declared with any combination of the following equational attributes: associativity, commutativity, idempotency, and identity.[3] In the presence of equational attributes, equational simplification using the other equations in the module does not take place at the purely syntactic level, but is understood *modulo* those equational attributes. The current Maude implementation can execute syntactic rewriting with typical speeds from half a million to several million rewrites per second. Similarly, associative and associative-commutative equational rewriting with term patterns used in practice can be performed at the typical rate of several hundred thousand rewrites per second.

A key feature of the ITP is its reflective design. The tool is written entirely in Maude and is in fact an executable specification of the formal inference system that it implements. Maude supports reflective computations through a predefined module called `META-LEVEL`, which includes different built-in functions providing direct access to the Maude rewriting engine itself. The ITP tool extends the module `META-LEVEL` with equationally defined functions defining the effect of the different proof commands, and uses the built-in functions provided by `META-LEVEL` to efficiently accomplish rewriting-based equational proofs.[4] This direct access to the Maude rewriting engine can justify in itself the reflective design of the ITP. But the reflective capabilities of Maude provide also the possibility of defining *rewriting commands* different from the Maude's default rewriting command, and of applying them to specific terms in the midst of a rewriting computation. There is in fact great freedom for defining different rewriting commands inside Maude, with ease and competitive performance, since they will typically use the built-in functions in the module `META-LEVEL` as the basic components of their definitions [5,6]. In particular, it is possible to define a rewriting command that calls the appropriate decision procedures, in the midst of a rewriting-based proof step, to solve, for

[3] The one restriction is that the idempotency attribute cannot be used together with the associativity attribute, since the combination of these two attributes is not currently supported by the Maude implementation.

[4] A typical metalevel computation only pays the cost (linear in the size of the term) of changing the representation from the metalevel to the object level and back only at the beginning and at the end of the computation [7].

example, a condition in a conditional equation. This capability is the base for the integration of decision procedures in the ITP basic rewriting proof engine.

Organisation

The paper is organised as follows. In Section 2 we introduce the ITP tool, its default rewriting command (which is directly based on Maude's default rewriting command) and its limitations. In Section 3 we describe the implementation of a different, more granular rewriting command. In Section 4 we explain how a decision procedure for Presburger arithmetic, written in Maude, can be integrated with the rewriting command introduced in Section 3. This extended rewriting command makes appropriate calls to the decision procedure when the condition of a conditional equation falls within the class of Presburger linear arithmetic formulas. Finally, in Section 5, we conclude with a description of the current state of the ITP tool, and of our plans for further extending the tool with other decision procedures.

2 The ITP tool and its default rewriting command

The ITP tool [4] is an experimental interactive theorem prover, written in Maude, for proving properties of Maude [6,7] functional modules, which are equational theories in membership equational logic [21]. The fact that membership equational logic is a reflective logic [8], and that Maude efficiently supports reflective membership equational logic computations is systematically exploited in the tool. Maude supports reflective computations through its predefined `META-LEVEL` module. In this module, Maude functional modules can be metarepresented as terms of a certain sort, which can then be efficiently manipulated and transformed by appropriate built-in functions. In particular, the module `META-LEVEL` includes a built-in function `metaReduce` that can be used to reduce a term in a functional module to canonical form, and a built-in function `metaXmatch` that can be used to try to match two terms in a functional module. The function `metaReduce` takes the metarepresentations of a module M and a term t, and it returns the metarepresentation of the fully reduced form of t, using the equations in M, together with the metarepresentation of its corresponding sort. The reduction strategy used by `metaReduce` coincides with that of the `reduce` command in Maude. The function `metaXmatch` takes the metarepresentations of a module M and two terms, t_1 and t_2 (and four more arguments that we can safely ignore here), and tries to match t_1 with any subterm of t_2 in the module M. If successful, it returns the representations of a substitution σ and a context C such that $C[\sigma(t_1)] \equiv t_2$, where we use $\equiv$ to denote equality *modulo* the equational attributes declared in the module for the operators involved; otherwise, the result is `noMatch` A full description of the module `META-LEVEL` can be found in [7, Chapter 10].

The ITP tool extends the module META-LEVEL with sorts for representing both the ITP goals and the ITP database of modules about which those goals have to be proved. In addition, it contains equationally defined functions specifying the effects of the different proof commands, like commands for carrying out inductive proofs, proofs based on case analysis, or proofs by rewriting. The basic ITP proof command is the rwr command that rewrites both sides of an equality to canonical form, using the equations contained in the module associated to the goal as simplification rules. As expected, the function that implements the rwr command directly calls the built-in function metaReduce to efficiently accomplish its task. And this, of course, may be sufficient when the equations in the module associated to the goal are Church-Rosser and terminating, and all the functions declared in that module are equally defined. There are, however, many practical equational specifications that do not satisfy such conditions. Consider, for example, the following specification in Maude of a sorting algorithm for lists of integers; the specification of the length function is included here for the sake of the example below.

```
fmod INS-SORT is
  protecting INT .
  sorts List .
  op nil : -> List [ctor] .
  op _:_ : Int List -> List [ctor] .
  op ins : Int List -> List .
  op sort : List -> List .
  op sorted : List -> Bool .
  op length : List -> Int .
  vars N M : Int . var L : List .
  --- ins
  eq ins(N, nil) = N : nil .
  ceq ins(N, M : L) = N : M : L
      if N <= M = true .
  ceq ins(N, M : L) = M : ins(N, L)
      if N > M = true .
  --- sort
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
  --- length
  eq length(nil) = 0 .
  eq length(N : L) = 1 + length(L) .
endfm
```

The module INS-SORT imports, through its protecting declaration, the predefined module INT that defines the integers with the expected arithmetic functions and relations; as usual, the latter are defined as Boolean functions. The module INT does not contain, however, the equational specification of the arithmetic functions but rather it provides internal links to built-in implementations of those functions. The module INS-SORT also imports, by default, the predefined module BOOL that defines the Boolean values. In this situation, the following property can easily be proved using Skolemization and term rewriting:

$$\forall\{N\}(\texttt{sort}(N\texttt{:nil}) = N\texttt{:nil}) \tag{1}$$

since, for N* a *new* (Skolem) constant of sort Int, the canonical form of sort(N*:nil) is syntactically identical to N*:nil. Consider, however, the following property about INS-SORT:

$$\forall\{N, M\}(\texttt{length}(\texttt{ins}(N, M\texttt{:nil})) = \texttt{2} \tag{2}$$

Again, by Skolemization, it will be sufficient to prove that

$$\texttt{length}(\texttt{ins}(\texttt{N*}, \texttt{M*:nil})) = \texttt{2} \tag{3}$$

for N* and M* new (Skolem) constants of sort Int. Given the conditional specification of ins, term rewriting will be useless at this point since no equations can be applied, and it is necessary to split the goal, using a Boolean-case analysis principle, into two subgoals that become associated to two different extensions of INS-SORT: one in which we add to INS-SORT the equation N* <= M* = true, and another in which the equation that we add is N* <= M* = false. In the first case, the proof is easily completed by applying term rewriting. However, in the second case, term rewriting will still remain useless since, in order to apply the second equation that specifies ins, we must prove first that N* > M* = true. But this cannot be proved simply by term rewriting, even when the module associated to this subgoal contains the equation N* <= M* = false, since INS-SORT does not contain any equation specifying >. Of course, for this particular case, the problem has two easy solutions: either we add to INS-SORT the equation N > M = true if N <= M = false, or we replace the second equation specifying ins by the conditional equation ins(N, M : L) = M : ins(N, L) if N <= M = false. But both solutions are manifestly ad-hoc.[5] A more general solution consists in integrating the appropriate decision procedure in the function implementing the rwr command. And this can be easily accomplished in a reflective rewriting-based theorem prover, as we show in the next two sections.

[5] Solutions of this sort can be found, however, in the literature. In the otherwise excellent textbook [13], the authors propose an extension of the OBJ's built-in representation of the integers with an equality predicate and with some equations that are useful for manipulating inequalities. In particular, these equations are useful as lemmas in the correctness proofs given in the book. But the authors warn the reader that they are not strong enough to allow all properties of integers to be proved by reduction. In general, they say, if a property of the integers is needed for a correctness proof, then an appropriate equation will need to be added as a lemma for the proof.

3 A different, non-default rewriting command for the ITP

In Section 2 we explained the implementation of the ITP default rewriting command `rwr`: how it uses the built-in function `metaReduce`, provided in the module `META-LEVEL`, to efficiently accomplish its task; and how its applicability is limited by the fact that `metaReduce` reduces terms in a module to canonical form using exclusively Maude's `reduce` command. In this section we show how the built-in function `metaXmatch`, also provided in the module `META-LEVEL`, allows us to implement, with ease and efficiency, a different, more granular rewriting command, `nrwr`, which does not call Maude's `reduce` command and includes, in particular, the implementation of the process of solving conditions when a conditional equation is applied to a term. In Section 4 we then explain how the implementation of `nrwr` can be easily extended to a command `xrwr` in order to integrate decision procedures in the rewriting process, to solve, in particular, linear arithmetic conditions in the application of a conditional equation. A detailed explanation of how to define in Maude, using its reflective capabilities, different rewriting commands can be found in [7, Chapter 10].

Like all ITP commands, the `nrwr` rewriting command is implemented in the ITP tool by equationally defining a function in an extension of the module `META-LEVEL`. We call *red* the function that implements the `nrwr` command. Like `metaReduce`, *red* takes the metarepresentations of a module and a term as arguments, and returns the metarepresentation of the reduced term; also as `metaReduce`, equations are applied in *red* only from left to right. We use the symbol $\triangleq$ for definitional equality.

$$\mathit{red}(M, t) \triangleq \mathit{redAux}(M, t, \mathit{getEqs}(M))$$

The function *getEqs* extracts from the metarepresentation of module M its set of equations; we omit here its definition. The behaviour of the auxiliary function *redAux* is simple. For each equation in the module it tries to match its left-hand side with any subterm of the term being reduced: if there is no match, it discards the equation; otherwise it reduces the term accordingly (after solving the condition if it is a conditional equation) and restarts the process again from the beginning. Its implementation is immediate using the built-in function `metaXmatch`.

$$\mathit{redAux}(M, t, \emptyset) \triangleq t$$

$$\mathit{redAux}(M, t, \{l = r\} \cup \mathit{Eq}) \triangleq \begin{cases} \mathit{red}(M, C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \\ \mathit{redAux}(M, t, \mathit{Eq}) & \text{otherwise} \end{cases}$$

$$redAux(M, t, \{l = r \textbf{ if } Cond\} \cup Eq) \triangleq \begin{cases} red(M, C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \text{ and} \\ & \quad solveCond(M, \sigma(Cond)) \\ redAux(M, t, Eq) & \text{otherwise} \end{cases}$$

The function *solveCond* tries to solve the equations in the condition by just reducing both sides as much as possible using *red*.

$$solveCond(M, \emptyset) \triangleq true$$

$$solveCond(M, \{l = r\} \cup Eq) \triangleq \begin{cases} solveCond(M, Eq) & \text{if } red(M, l) \equiv red(M, r) \\ false & \text{otherwise} \end{cases}$$

Note that, as for the case of `metaReduce`, the function *red* assumes that the equations in the module are Church-Rosser and terminating, and therefore that they can be applied in arbitrary order and to arbitrary redexes. Of course, lack of confluence would result in incompleteness (but not unsoundness) of the `nrwr` command; and, if the equations were not terminating then `nrwr` would not terminate in some cases either.

4 A rewriting command with integrated decision procedures for the ITP

We explain in this section the implementation in the ITP tool of the `xrwr` command that extends `nrwr` by integrating a decision procedure for Presburger arithmetic in the process of reducing a term. The technique used in the implementation of `xrwr` also applies to the implementation of similar commands that integrate other decision procedures.[6] The general technique is based on the fact that any decision procedure is a computable function, and, therefore, by the metatheorem of Bergstra and Tucker [2], it can be equationally specified by a finite set of Church-Rosser and terminating equations. To implement in the ITP tool a rewriting command that integrates a certain decision procedure in the rewriting process, all we have to do is to modify the function `nrwr` in such a way that the function implementing the given decision procedure is called at the appropriate times on the appropriate expressions. The function *redPlus* below is an example of this; but before introducing this function we briefly present the decision procedure that is thus integrated in the rewriting process.

[6] In fact, the current implementation of the `xrwr` command in the ITP tool integrates, using the technique described here, a decision procedure introduced in [25] for an extension of quantifier-free Presburger arithmetic that permits arbitrary uninterpreted function symbols; this theory includes many of the formulas that one tends to encounter in program verification.

In [24], Shostak describes a decision procedure for quantifier-free Presburger arithmetic. Presburger expressions are those that can be built up from integers, integer variables, and addition. (Arbitrary multiplication is not allowed, but it is convenient to use multiplication by constants as an abbreviation for repeated addition.) Linear inequalities are constructed by combining Presburger expressions with the usual arithmetic relations ($\leq$, $<$, $\geq$, $>$, $=$) and the propositional logic connectives. The procedure to check validity of a formula φ consists in expanding its negation into disjunctive normal form and expressing each disjunction as a conjunction of linear inequalities of the form $A \leq B$. Then, φ is valid if and only if its negation is not satisfiable, which is checked by looking for a solution in integers for each of the disjunctions with the help of an integer programming algorithm. This is an NP-complete problem and there is a certain tradeoff between using complete algorithms (in the sense of always terminating with a correct solution) or efficient but incomplete ones. Our design decision has been to use an efficient and terminating algorithm introduced in [24] that, however, yields no result in some uncommon situations.

The `xrwr` command is implemented in the ITP tool by an equationally defined function *redPlus* that, following the general technique described above, simply modifies the function *red* by introducing a new layer that corresponds to the decision procedure. It first checks whether the term is amenable to being dealt with (if it is a linear inequality in our case) and depending on the answer it calls a different auxiliary function.

$$\mathit{redPlus}(M,t) \triangleq \begin{cases} \mathit{redPlusAux1}(M,t) & \text{if } \mathit{isLinIneq?}(t) \\ \mathit{redPlusAux2}(M,t,\mathit{getEqs}(M)) & \text{otherwise} \end{cases}$$

The function *isLinIneq?* decides if a term corresponds to the representation of a linear inequality. To understand the definition of *redPlusAux1* below think of the term t as being, for example, the Boolean expression `N* <= M*`. We start by checking if the expression represented by t holds in M. For that, we use a function *getLinIneqs* that refines *getEqs* with the help of *isLinIneq?* and extracts from the metarepresentation of a module only those equations that involve linear inequalities. Then, if we let φ stand for the formula $\mathit{getLinIneqs}(M) \rightarrow t = \mathit{true}$, we make use of the function *isSatisfiable?*, that implements the decision procedure for quantifier-free Presburger arithmetic, to check the satisfiability of the negation of φ. If this negation is not satisfiable then φ is valid and t is provably equal to *true* in M, and thus can be reduced to *true*. If not, we try to see if it is provably equal to *false* by checking the satisfiability of the negation of $\mathit{getLinIneqs}(M) \rightarrow t = \mathit{false}$. Of course, because of the incompleteness of the decision procedure used to check satisfiability, it could actually be the case that t is provably equal to *true* or *false*, but we cannot decide it. In this uncommon situation, we call the *redPlusAux2* function.

$$\mathit{redPlusAux1}(M,t) \triangleq$$
$$\begin{cases} \mathit{true} & \text{if not}(\mathit{isSatisfiable?}(\neg(\mathit{getLinIneqs}(M) \rightarrow t = \mathit{true}))) \\ \mathit{false} & \text{if not}(\mathit{isSatisfiable?}(\neg(\mathit{getLinIneqs}(M) \rightarrow t = \mathit{false}))) \\ \mathit{redPlusAux2}(M,t,\mathit{getEqs}(M)) & \text{otherwise} \end{cases}$$

The function *redPlusAux2* is called on terms that are not linear inequalities, or that are linear inequalities but the (incomplete) decision procedure has not succeeded in deciding them. Its definition is completely analogous to *redAux*, except for the fact that we use now *redPlus* and *solveCondPlus* instead of *red* and *solveCond*.

$$\mathit{redPlusAux2}(M,t,\emptyset) \triangleq t$$
$$\mathit{redPlusAux2}(M,t,\{l=r\}\cup \mathit{Eq}) \triangleq \begin{cases} \mathit{redPlus}(M,C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \\ \mathit{redPlusAux2}(M,t,\mathit{Eq}) & \text{otherwise} \end{cases}$$

$$\mathit{redPlusAux2}(M,t,\{l=r \textbf{ if } \mathit{Cond}\}\cup \mathit{Eq}) \triangleq$$
$$\begin{cases} \mathit{redPlus}(M,C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \text{ and } \mathit{solveCondPlus}(M,\sigma(\mathit{Cond})) \\ \mathit{redPlusAux2}(M,t,\mathit{Eq}) & \text{otherwise} \end{cases}$$

Finally, *solveCondPlus* deviates from *solveCond* in that, for equations whose terms are Presburger expressions, it also calls the decision procedure to check their validity. The function *isPresExp?* recognises those terms that represent Presburger expressions; we also omit here its definition.

$$\mathit{solveCondPlus}(M,\emptyset) \triangleq \mathit{true}$$
$$\mathit{solveCondPlus}(M,\{l=r\}\cup \mathit{Eq}) \triangleq$$
$$\begin{cases} \mathit{solveCondPlus}(M,\mathit{Eq}) & \text{if } (\mathit{isPresExp?}(l) \text{ and } \mathit{isPresExp?}(r) \text{ and} \\ & \quad \mathit{isSatisfiable?}(l=r)) \text{ or} \\ & \quad \mathit{redPlus}(M,l) \equiv \mathit{redPlus}(M,r) \\ \mathit{false} & \text{otherwise} \end{cases}$$

We end this section with a high-level description of how the rewriting command implemented by the function *redPlus* solves the difficulty we faced in proving (3) in Section 2. Let us denote by `INS-SORT+` the module `INS-SORT` extended with the equation `eq N* <= M* = false`. Recall that the problem was that the additional information provided by this equation could not be used to reduce the term `ins(N*,M*:nil)`. Now, when we apply *redPlus* to the metarepresentations of `INS-SORT+` and `ins(N*,M*:nil)`, since `ins(N*,M*:nil)` is not a linear inequality, the function *redPlusAux2* is called.

Eventually, the equation `ins(N,M:L)` = `M:ins(N,L) if N > M` = `true` will be tried, and the function *solveCondPlus* will then be invoked to discharge the condition `N* > M*` = `true`. At this point, since `true` is not a Presburger expression, *redPlus* is invoked to reduce both `N* > M*` and `true`. Here is when the decision procedure comes into action: *redPlus* detects that `N* > M*` is a linear inequality and calls *redPlusAux1*. This function extracts the linear inequalities in `INS-SORT+` and uses the decision procedure to check if they imply `N* > M* = true`. Since this is actually the case, *redPlusAux1* reduces `N* > M*` to `true`. At this point, the condition `N* > M* = true` is discharged and `ins(N*,M*:nil)` is reduced to `M*:ins(N*:nil)`. The function *redPlus* will then eventually reduce `length(M*:ins(N*:nil))` to 2, as required, using the equations in `INS-SORT+` as simplification rules.

5 Conclusions and future work

In this work we have shown how the reflective design of a rewriting-based theorem prover like the ITP can be easily extended with decision procedures. The smoothness with which we have been able to integrate in the ITP tool an extension of the decision procedure described in Section 4 (which can be applied to the more general case of quantifier-free Presburger arithmetic with uninterpreted function symbols [25]) is encouraging. In this latest version of the ITP we have proved, as basic examples, the correctness of three different sorting functions, specifying respectively the insertion, merge, and quicksort algorithms. The correctness proofs for the merge and quicksort algorithms proceed by induction on the length of the list and make heavy use of the integrated decision procedure.

As a follow-up of this work we plan to add other decision procedures to our tool. As we have already pointed out, their integration will follow exactly the same steps as those described here and thus should pose no special difficulty. An important task ahead is to combine these decision procedures to be able to tackle expressions that involve diverse semantic constructs that belong, not just to one, but to several of them. In this regard, the recent generalisation of the Nelson-Oppen combination method to order-sorted theories [26] and the recent studies clarifying the relation between Nelson-Oppen's and Shostak's procedures [17] seem particularly relevant. Finally, we also plan to compare in more detail our reflective design for the integration of decision procedures in the ITP with the non-reflective one used in RRL. This comparison should be possible and interesting since both tools pursue similar goals and share a similar logical foundation.

Acknowledgements

We would like to thank José Meseguer for many interesting discussions on the subject of reflective rewriting-based theorem provers. We also thank José

Meseguer, Peter Ölveczky, and the anonymous referees for their detailed and helpful comments on earlier versions of this paper.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, August 1999.

[2] J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification. In J. W. de Bakker and J. van Leeuwen, editors, *Automata Languages and Programming, Seventh Colloquium*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[3] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study for arithmetics. *Machine Intelligence*, 11:83–124, 1988.

[4] M. Clavel. The ITP tool's home page. March 2004. The site contains the latest version of the ITP tool, the documentation currently available, and several examples of proof scripts of different complexity. http://geminis.sip.ucm.es/~clavel/itp.

[5] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2002.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.1). http://maude.cs.uiuc.edu, 2004.

[8] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In F. Gadducci and U. Montanari, editors, *Proc. Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 63–78. Elsevier, 2002.

[9] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing.* World Scientific, 1998.

[10] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification: 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[11] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors,

Conditional Term Rewriting Systems, 1st International Workshop Orsay, France, July 8–10, 1987, Proceedings, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.

[12] J. Goguen, A. Stevens, H. Hilberdink, and K. Hobley. 2OBJ: A metalogical framework theorem prover based on equational logic. *Philosophical Transactions of the Royal Society of London*, 339:69–86, 1992.

[13] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* The MIT Press, 1996.

[14] D. Kapur and X. Nie. Reasoning about numbers in Tecton. In Z. W. Ras and M. Zemankova, editors, *Proc. of 8th International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, volume 869 of *Lecture Notes in Computer Science*, pages 57–70. Springer-Verlag, 1994.

[15] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[16] M. Kauffmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):202–213, April 1997.

[17] S. Krstić and S. Conchon. Canonization for disjoint union of theories. In F. Baader, editor, *Automated Deduction — CADE-19: 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 – August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 197–211. Springer-Verlag, 2003.

[18] Z. Manna and the STeP group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV 96)*, volume 1102 of *lncs*, pages 415–418. Springer-Verlag, July/August 1996.

[19] U. Martin and J. M. Wing, editors. *First International Workshop on Larch.* Springer-Verlag, 1992.

[20] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, 2nd edition edition, 1979.

[21] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.

[23] M. Saaltink. The Z/EVES system. In *ZUM'97: The Formal Specification Notation, 10th International Conference on Z Users*, volume 1212 of *lncs*, pages 72–85. Springer-Verlag, April 1997.

[24] R. E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24:529–543, 1977.

[25] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26:351–360, 1979.

[26] C. Tinelli and C. Zarba. Combining decision procedures for theories in sorted logics. Technical Report 04-01, Department of Computer Science, The University of Iowa, Feb. 2004.