Automatic Synthesis of Decision Procedures: a Case Study of Ground and Linear Arithmetic*

Predrag Janičić
Faculty of Mathematics, University of Belgrade
Studentski trg 16, 11000 Belgrade, Serbia and Montenegro
email: janicic@matf.bg.ac.yu

Alan Bundy
School of Informatics, University of Edinburgh
Appleton Tower, Crichton St, Edinburgh EH8 9LE, UK
email: A.Bundy@ed.ac.uk

Abstract

In this paper we address the problem of automatic synthesis of decision procedures. We evaluate our ideas on ground arithmetic and Fourier/Motzkin decision procedure for linear arithmetic, but the approach can be applied to other domains as well. The approach is well-suited to the proof-planning paradigm. The synthesis mechanism consists of several stages and sub-mechanisms. Some of these steps are performed automatically, while in some steps human assistance is necessary. Our system (ADEPTUS), which we present in this paper, synthesized a decision procedure for ground arithmetic completely automatically and it used some specific method generators in generating a decision procedure for linear arithmetic. We believe that this approach can lead to automated assistance in constructing decision procedures and to more reliable implementations of decision procedures.

1 Introduction

Decision procedures are often vital in theorem proving [2, 11]. In order to have decision procedures usable in a theorem prover it is often necessary to have them implemented not only efficiently, but also flexibly. Also, it is often very important to have decision procedures for new, user-defined theories. The implementation should also be such that it can be verified in some formal way. For all these reasons, it would be fruitful if we can automate the process (or, at least,

*The first author was supported by EPSRC grant GR/R52954/01. The second author was supported in part by EPSRC grant GR/S01771.

all its routine steps) of synthesizing and implementing decision procedures. It would also help avoiding human mistakes in implementing decision procedures. In addition, since many steps in different decision procedures can be described via rewriting, object level proofs can often be also relatively easily derived from an application of a decision procedure.

Due to their importance in software and hardware verification and, therefore, in different schemes for combining and augmenting decision procedures, in this paper we evaluate our techniques on ground arithmetic and linear arithmetic. As it is interesting and non-trivial, we focus on ideas from the Fourier/Motzkin procedure for linear arithmetic [13].

In this paper we follow ideas from Bundy's paper on proof plans for normalisations [5]. As discussed in [5], most steps of many decision procedures can be described via sets of rewrite rules. This observation prepares the way for automatic generation of new decision procedures (given the necessary rewrite rules), which is vital for user defined theories. Bundy's programme (slightly modified) and the synthesis process we present in this paper can be decomposed into several subproblems:

- Make a method which is capable of doing the following: given two syntactical classes and some rewrite rules, select (if it is possible) a subset of rewrite rules which is sufficient to transform (or *normalise*) any member of the first syntactical class (described by Backus-Naur form (bnf) or, equivalently, by a context-free grammar) into a member of the second syntactical class. Given the input syntactical class, the kind of a method and its parameters, the output syntactical class can be automatically generated in a number of special cases, i.e., for all kinds of methods discussed in this paper.[1] An algorithm which can generate such a method (on the basis of the given input class, available rewrite rules, the kind of the method, and the method's parameters) we call a *method generator*.
- If the available set of rewrite rules is not sufficient for performing the transformation from one syntactical class to another, then the method generator has to report about missing rules; in a planned advanced version (which is not part of the work presented in this paper), it has to speculate the remaining necessary rules and/or to redefine/relax the output class;
- there will be different kinds of normalisation methods, e.g., one for removing some function symbol, one for stratification, one for thinning etc. (see further text and [5] for explanation of these terms.); for each of them, there is a method generator.
- methods should be designed in such a way that their soundness, completeness, and termination are guaranteed; the same holds for compound methods;

[1] In general, given the input bnf and a set of rewrite rules, the output language is not always context-free (i.e. it cannot be described via a bnf). Moreover, it is undecidable whether such output language is context-free. The general problem of determining the output bnf (when it exists), given the input bnf and a set of rewrite rules is the subject of our current research.

- given all necessary methods, it should be possible to combine them (automatically) into a compound method or, sometimes, into a decision procedure for some theory;
- since some transformations (required for some decision procedures) are very complex, building some methods (or some method generators) may require some human interaction and assistance.

We have implemented several method generators: generators for remove methods, stratify methods, thin methods, absorb methods and left-assoc methods and several special-purpose method generators. These generators take a given BACKUS-NAUR form, transform it into another one, and build a method which uses some available rewrite rules such that each input formula (which belongs to the first BNF) will be transformed into a formula which belongs to the second BNF. On the set of all these generators, we can perform a (heuristically guided) search for a sequence of methods which goes from the starting BNF to a trivial BNF (consisting of only $\top$ and $\bot$). If the final syntactical class is equal to $\{\bot, \top\}$, then the whole of the sequence yields a decision procedure for the underlying theory (under some assumptions about available rewrite rules). If such a method can be built, soundness, termination, and completeness can be easily proved (see §3.1).

The system consisting of the implemented method generators and search engines we call ADEPTUS (coming from *Assembly of DEcision Procedures via TransmUtation and Synthesis*[2]).

Our first target theory was ground arithmetic over rationals. We had available all the necessary rewrite rules. ADEPTUS synthesized the decision procedures for this theory completely automatically in around 3 seconds of CPU time (see more details in §6).

While a decision procedure for ground arithmetic can be also obtained by exhaustive application of all rewrite rules, our system gives a procedure which uses their subsets in stages and gives structured proofs (easily understandable to a human).

Our second target theory was (quantified) linear arithmetic. For this theory we had to implement three more method generators: one for adjusting the innermost quantifier, one for isolating a variable, and one for removing a variable (*cross-multiply and add* step). The search is performed in three stages and all three of them give sequences of methods. Combined, these sequences (the second one in a loop) give a decision procedure for linear arithmetic. ADEPTUS automatically performed these three stages and synthesized a decision procedure for linear arithmetic in around 5 seconds of CPU time (see more details in §9). For some conditional rewrite rules, this decision procedure itself can be used to prove that their conditions cover all possible cases. We prove that both of the generated procedures are sound, complete and terminating.

[2]Also, Adeptus (Lat.) is "one with the alchemical knowledge to turn base metals into gold".

For some theories this approach gives not only automatically generated decision procedures, but also a higher-level understanding of syntactical transformations within the underlying theory. We believe that this approach can be helpful in both easier implementation of decision procedures and their deeper understanding. The approach can also be used just to construct normalisation procedures.

Overview of the paper: In §2 we give some basic background and notation information. In §3 we introduce the notion of normalizers and discuss their automatic generation. In §4 we present several generic method generators and in §5 a search engine which uses them. In §6 we report on how we used that search engine to synthesize a decision procedure for ground arithmetic. In §7 we present several special-purpose method generators. In §8 we describe a compound search engine and in §9 we report on how we used it to synthesize a decision procedure for linear arithmetic. In §10 we present some additional examples. In §11 we discuss some possibilities and limitations in automatic synthesis of decision procedures. In §12 we discuss related work. In §13 we discuss future work, and in §14 we draw final conclusions.

2 Background and Notation

2.1 Decidable theories and decision procedures

A theory $\mathcal{T}$ is decidable if there is an algorithm (which we call a *decision procedure*) such that for an input $\mathcal{T}$-sentence f, it returns *yes* if and only if $\mathcal{T} \vdash f$ (and returns *no* otherwise).

2.2 Presburger arithmetic

Presburger Natural Arithmetic is (roughly speaking) a first-order theory built up from the constant 0, variables, binary function symbol $+$, unary function symbol s and binary predicate symbols $<$, $>$, $=$, $\neq$, $\leq$, $\geq$. Note that nx (where n is a numeral) can also be considered as in Presburger Natural Arithmetic: nx is treated as $x+\cdots+x$, where x appears n times. We denote Presburger Natural Arithmetic as PNA. By analogy we define Presburger arithmetic over integers (Presburger Integer Arithmetic, denoted PIA) and over rationals (Presburger Rational Arithmetic, denoted PRA); these two theories have the additional unary function symbol $-$. PRA is sometimes referred to as linear arithmetic [2].

There are several decision procedures for PIA, including Cooper's procedure [6]. The Fourier/Motzkin procedure [13] is a decision procedure for PRA. Sometimes, it is referred to as Hodes' procedure [9]. This procedure has very important uses and it has been rediscovered a number of times. In theorem proving, because its worst case complexity is lower than of Cooper's procedure, this procedure is often used for the universally quantified fragment of PIA, as it is sound (although incomplete) for it [2].

2.3 Backus-Naur form

We can define a set of PIA (PNA, PRA) formulae by a context-free grammar or in Backus-Naur Form (BNF).

Example 1 *Using a* BNF *we can define* PRA *formulae in the following way:*

$$\begin{array}{rcl} f & := & af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f|(\exists var : sort)f|(\forall var : sort)f \\ af & := & \top|\bot|t = t|t < t|t > t|t \leq t|t \geq t|t \neq t \\ t & := & var|rc|rc \cdot t| - t|t + t \\ var & := & x_1|x_2|x_3|\ldots \\ sort & := & rational \end{array} \tag{1}$$

where f denotes the syntactical class of formulae, af the class of atomic formulae, t denotes the class of terms, var variables, and rc denotes rational constants.

In representing some infinite syntactical classes (for instance, the classes rc and var in the above example) sometimes, for convenience, we use some additional device and some meta-level conditions.

We will assume that each BNF definition has attached its *top class* (or *start class*). While a BNF definition corresponds to a context-free grammar, this class corresponds to the start symbol in formal grammars. The language of a BNF is a set of all expressions that can be derived from the top class. Normally, it is a class that corresponds to the set of formulae (the class f in the above example).

2.4 Rewrite rules

In the rest of the paper we will assume that during the method generation there are available some unconditional or conditional rewrite rules. Unconditional rewrite rules are of the form:

$$RuleName: \quad l \longrightarrow r \; .$$

Conditional rewrite rules are of the form:

$$RuleName: \quad l \longrightarrow r \quad if \quad p_1, p_2, \ldots, p_n,$$

where p_1, p_2, ..., p_n are literals.[3] These rewrite rules correspond to some underlying (background) theory $\mathcal{T}$ and the conditions may rely on some theory-specific properties. For instance, we can have the rewrite rule (corresponding to PRA):

$$n_1x + n_2x \longrightarrow nx \quad if \quad n = n_1 + n_2.$$

In the above rule, the condition $n = n_1 + n_2$ is not of a syntactical nature and hence, in general, we will have to use some semantic knowledge, i.e., we will

[3]Note that this form can be considered as a normal form of conditional rewrite rules with arbitrary propositional structure in their conditions.

have to use rewriting modulo the underlying theory. However, these conditions are, usually, very simple and they usually involve only ground literals.

For a rule

$$RuleName: \quad l \longrightarrow r \quad if \quad p_1, p_2, \ldots, p_n,$$

we say that it is *sound* with respect to theory $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula Φ and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ if $\mathcal{T}$, $p_1\varphi$, $p_2\varphi$, $\ldots$, $p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$ (where $\mapsto$ denotes substitution), and we say that it is *complete* with respect to theory $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ only if $\mathcal{T}, p_1\varphi, p_2\varphi, \ldots, p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$. We say that a rewrite rule is *two-way* rule w.r.t. $\mathcal{T}$ if it is sound and complete w.r.t. $\mathcal{T}$.

2.5 Syntactical relations

We introduce two syntactical relations: *ec* (from *element of class*) and *aec* (*abstract element of class*). $ec(b, e, c)$ holds if and only if a syntactical element e is an element of a class c in the BNF definition b. For instance, if we denote by b_1 the definition (1) (from Example 1, p5), then it holds that

$ec(b_1, (\exists x : rational)0 = x, f)$

and

$ec(b_1, (0 = x) \wedge (1 = y), f \wedge f)$

The relation *aec* is defined by analogy, but its arguments can also include some abstract components. For example, it holds:

$aec(b_1, af \wedge (1 = y), f \wedge f)$.

2.6 Proof planning and methods

Proof-planning is a technique for guiding the search for a proof in automated theorem proving. To prove a conjecture, within a proof-planning system, a method constructs the proof plan and this plan is then used to guide the construction of the proof itself [4, 3]. These plans are made up of tactics, which represent common patterns of reasoning. A tactic's preconditions and effects are specified by a method, i.e., a method is a specification of a tactic. A method has several slots: a name, input, preconditions, transformation, output, postconditions and the name of the attached tactic.[4]

Definition 1 *A* method *describes the preconditions for the use of a tactic, the transformation that corresponds to it and the postconditions (conditions that hold for the conjecture after the application of the tactic). These conditions and*

[4]In our implementation of the system presented in this paper, we have not implemented tactics yet. So, our procedures produce meta-level proof plans, not the object level proofs. However, for most of the methods, implementing tactics would not be difficult, as most of the required tactics are based on applying given rewrite rules. For some steps (like *cross multiply and add* for a decision procedure for linear arithmetic, see §7.4), tactics would be more involved. Also, for providing tactics and object level proofs, it would be suitable to incorporate ADEPTUS into some more general system with proof engine, able for running tactics on (e.g., IsaPlanner built on top of Isabelle).

transformation are syntactic properties of the logical expressions manipulated by the tactic and are expressed in a meta-logic.

A method cannot be applied if its preconditions are not met. Also, with the transformation performed and the output computed, the postconditions are checked and the method application fails if they fail.

2.7 Target language

The whole of ADEPTUS is implemented in PROLOG as a stand-alone system.[5] All generated methods are built in the spirit of the proof planning paradigm (and implemented in PROLOG).

2.8 Standardizing apart

For simplicity, in the rest of the paper and in all described normalisation methods and decision procedures, we assume that, in formulae being transformed, variables are standardized apart, that is — there are no two quantifiers with the same variable symbol in one formula.

3 Normalisation Methods

We will describe a number of steps of different decision procedures for ground arithmetic and for linear arithmetic in a purely syntactical way and in terms of rewriting (in the spirit of Bundy's proof plans for normalisations [5]). For instance, one of the steps in different decision procedures is elimination of equivalence and that step can be described as an exhaustive application of the following rewrite rule:

$f_1 \Leftrightarrow f_2 \quad \longrightarrow \quad (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$

This rule links a pair of classes of formulae, which can be represented in Backus-Naur Form (BNF) as shown in the following example.

Example 2 *Each formula belonging to the following class (af denotes the class of atomic formulae):*

$$f \quad := \quad af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f|(\exists var : sort)f|(\forall var : sort)f$$

can be transformed by using the rewrite rule

$f_1 \Leftrightarrow f_2 \quad \longrightarrow \quad (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$

and the resulting formula belongs to the following class f:

$$f \quad := \quad af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|(\exists var : sort)f|(\forall var : sort)f \ .$$

This way, a description of the effect of a certain normalisation can lead to an automated synthesis of that normalisation. Namely, given a set of rewrite rules and BNF form of the input class of formulae, it is possible to automatically generate certain normalization steps or sequences of normalization steps (and

[5]The presented system is available on-line from `www.matf.bg.ac.yu/~janicic`.

the corresponding output classes). In the above example, it can be easily shown that any formula belonging to the first class can be transformed to a formula belonging to the second class (by exhaustive application of the given rewrite rule). As we will see, for some theories the whole decision procedure can be generated on these bases (with a trivial output class, consisting of only two formulae — $\top$ and $\bot$).

Definition 2 *A* Method generator *is a procedure with the input consisting of*

- *a* BNF *form b of the input expressions;*
- *a set of rewrite rules R;*
- *a kind t of the required method (e.g.,* remove*,* stratify*, etc.)*

that generates a method $\mathcal{M}$ and a BNF *form b' (of the output expressions). The method $\mathcal{M}$ has a kind t and, by using exhaustive application of rules from R, it transforms any expression belonging to b to an expression belonging to b'.*

We have implemented method generators for several kinds of methods: remove, stratify, thin, absorb and left-assoc.

Method generators for most normalizers (normalisation methods) work, in a sense, in a uniform way: each of them takes the input BNF set (and given parameters), searches for "problematic" BNF entries (for instance, BNF entries with occurrences of the symbol that should be eliminated — $f := f \Leftrightarrow f$ in Example 2) and constructs the target output BNF set; after that, a common algorithm for searching over the set of available rewrite rules is invoked and it checks if all "problematic" entries can be rewritten in such a way that any input formula, when rewritten, falls in the target output top class. In addition, this search mechanism attaches rewrite rules to particular entries (because selected rewrite rules cannot be applied arbitrarily, as will be illustrated in Example 8, (p15)). When we are discussing the properties of some method generators, we consider both their properties and the properties of the generated methods.

3.1 Properties of Method Generators and of Generated Methods

In this part of the paper we (preliminarily) discuss properties (termination, soundness and completeness) of method generators and of generated methods (i.e., normalization methods). We also briefly discuss properties of compound methods (of methods built from several generated methods).

3.1.1 Properties of method generators

For each method generator we present in this paper it is easy to show that it is terminating. Method generators are not complete, i.e., they are not guaranteed to produce the required methods — in some circumstances they fail, as they cannot find the necessary rewrite rules. Moreover, we don't claim completeness

of all the method generators in a narrower sense — that they produce required methods whenever it is possible. On the other hand, we can consider soundness of the method generators, i.e., the issue whether, if a method is generated, then it meets the given conditions. Soundness of method generators defined in this way can actually be considered as a set of properties (soundness, completeness, termination) of the generated methods.

3.1.2 Properties of the generated methods

A normalisation method links two syntactical sets — these two sets should be equivalent modulo the underlying theory $\mathcal{T}$. In purely syntactical terms, each formula f_1 that belongs to the top class of the input BNF set should be transformed (in a finite number of steps) into a formula f_2 that belongs to the top class of the output BNF set. Moreover, taking the properties of the underlying theory $\mathcal{T}$, it should hold that $\mathcal{T} \vdash f_1$ if (and only if) $\mathcal{T} \vdash f_2$. If the "if" condition holds, then the method is sound (w.r.t. $\mathcal{T}$). If the "only if" condition holds then the method is complete (w.r.t. $\mathcal{T}$). If these properties are fulfilled, then the method is terminating, sound and complete.

Termination. For each generated method it must be shown that it is terminating (by considering properties of the rewrite rules used[6]). For some sorts of methods, their termination is guaranteed by the way they are generated.

Soundness. We distinguish soundness of a method w.r.t. syntactical restrictions and soundness of a method w.r.t. the underlying theory $\mathcal{T}$. If a method transforms one formula into another one, then it is ensured by the method's postconditions that the second one does meet the required syntactical restrictions (given by the method specification), so the method is sound w.r.t. syntactical restrictions. In addition, all available rewrite rules (all of them correspond to the underlying theory $\mathcal{T}$) are assumed to be sound. Thus, since a method is (usually) based on exhaustive application of some (normally sound) rewrite rules, it is trivially sound w.r.t. $\mathcal{T}$.

Completeness. We distinguish completeness of a method w.r.t. syntactical restrictions and completeness of a method w.r.t. the underlying theory $\mathcal{T}$. It is not *a priori* guaranteed that a generated method can transform any input formula (which meets the preconditions) into some other formula (that belongs to the output class), i.e., it is not guaranteed that the method is complete w.r.t. syntactical restrictions. Namely, a method maybe uses some conditional rewrite rules (which cannot be applied to all input formulae). If a method uses only unconditional rewrite rules or if a method uses conditional rewrite rules which cover all possible cases, then it can transform any input formula into a formula belonging to the output class. Completeness of a method w.r.t. $\mathcal{T}$ also relies on the completeness of the rewrite rules used. If a method can transform any input formula into a formula belonging to the output class and if all the rewrite rules it uses are complete, then the method is complete w.r.t. $\mathcal{T}$. Note

[6] Note that these sets of rewrite rules are not always confluent. Moreover, for certain tasks, such as, for instance, transforming a formula into disjunctive normal form, there is no confluent and terminating rewrite system [16].

that both of these conditions are necessary: for instance, we can build a method for PIA by using rewrite rules for PRA; this method will be sound (w.r.t. PIA) (for universally quantified fragment) and it could transform any input formula, but it will still be incomplete w.r.t. PIA (see Example 3). However, such methods can still be very useful.

Example 3 *The formula* $(\forall x)(1 \geq x \ \vee \ x \geq 2)$ *is a theorem of* PIA*, but is not a theorem of* PRA *(i.e., it is valid if* x *ranges over integers, but is not if* x *ranges over rationals). The Fourier/Motzkin decision procedure for* PRA *would transform* $(\forall x)(1 \geq x \vee x \geq 2)$ *in the following way:*

$\neg(\exists x)\neg(1 \geq x \ \vee \ x \geq 2)$
$\neg(\exists x)(1 < x \ \wedge \ x < 2)$
$\neg(1 < 2)$
$1 \geq 2$
$\bot$

Thus, it is shown that the given formula is not a theorem of PRA*. This example shows that Fourier/Motzkin procedure is not complete for* PIA*.*

3.1.3 Properties of compound methods

Given a set of generated methods for some underlying theory $\mathcal{T}$, they can be combined (by a human, or — as we will see — automatically) into a compound method (for that theory). Compound methods (in this context) can use primitive methods in a sequence or in a loop (but not conditional branching). The preconditions of a compound method are the preconditions of the first (primitive) method used, and the postconditions are the postconditions of the last (primitive) method used.[7]

Termination. If a compound method is based on using a sequence of terminating methods, then it is (trivially) terminating. If it includes a loop, then some deeper argument is required.

Soundness. Since it relies on the soundness of the used primitive methods, every compound method is also sound (both w.r.t. syntactical restrictions and w.r.t. the underlying theory $\mathcal{T}$). Meeting the syntactical restrictions of the compound method is also ensured by its postconditions.

Completeness. If all the used methods are complete and if the compound method is terminating, then it is (trivially) complete. More precisely, if a compound method (*i*) is terminating; (*ii*) uses only (primitive) methods which never fail (i.e., the methods which transform any input formula to a formula belonging to the output class) and which use only complete rewrite rules, then that compound method is complete (w.r.t. $\mathcal{T}$).

Based on the above considerations, we can make a crucial observation: if a compound method for some theory $\mathcal{T}$ has an input BNF set corresponding to the whole of $\mathcal{T}$ and a trivial output BNF set consisting only of $\top$ and $\bot$,

[7] This way of constructing the preconditions and postconditions of a compound method is not adequate in general (for instance, when one has conditional branching) but suffices for the examples we were working on.

then it is a decision procedure for $\mathcal{T}$ if it is terminating, sound (w.r.t. $\mathcal{T}$), and complete (w.r.t. $\mathcal{T}$), while these soundness and completeness properties rely only on properties of the rewrite rules used. This way, we can, in some case[8], trivially get a proof that some (automatically generated) compound method is a decision procedure for some theory.

3.2 Simplification of BNFs

Each method generator first tries to simplify the input BNF. By "simplifying" we mean elimination of non-recursive classes. Namely, if a class (different from the top class) is not recursive (i.e., it does not occur in its BNF entries) then it can be eliminated by replacing all its derivations in all positions where elements of this class occur.

Example 4 *We can eliminate the class af from the following definition*

$$\begin{array}{rcl} f & := & af|f \vee f|f \wedge f \\ af & := & t = t|t < t \end{array}$$

and get

$$f \quad := \quad t = t|t < t|f \vee f|f \wedge f$$

The need for this step is illustrated in Example 5.

Example 5 *Consider the following* BNF *classes:*

$$\begin{array}{rcl} f & := & af|\neg af|f \vee f|f \wedge f \\ af & := & \top|\bot|t = t|t < t \end{array} \tag{2}$$

Let us suppose that there are available the following rewrite rules:

`rm_neg_less`:	$\neg(t_1 < t_2)$	$\longrightarrow$	$t_2 < t_1 \vee t_1 = t_2$
`rm_neg_eq`:	$\neg(t_1 = t_2)$	$\longrightarrow$	$(t_1 < t_2) \vee (t_2 < t_1)$
`rm_top`:	$\neg\top$	$\longrightarrow$	$\bot$
`rm_bottom`:	$\neg\bot$	$\longrightarrow$	$\top$

The above rules are sufficient to eliminate $\neg$ from any formula belonging to the class f from (2), but it cannot be done within the class f (i.e., $\neg$ cannot be removed from the BNF *definition of the class f without taking into account other classes, as in removing the symbol $\Rightarrow$). However, if we first simplify (as described above) the given* BNF *definition, we eliminate the class af and we get the following* BNF *definition:*

$$f \quad := \quad \top|\bot|t = t|t < t|\neg\top|\neg\bot|\neg t = t|\neg t < t|f \vee f|f \wedge f$$

Now we can eliminate the symbol $\neg$ from the class f.

3.3 Search for the Necessary Rewrite Rules

Several method generators use the same algorithm for searching for the necessary rewrite rules. This algorithm is given in Fig. 1.

---

[8] If we generate and use some methods which do something more than rewriting, the situation can be more complicated (see §9 on linear arithmetic).

The algorithm tries to find rewrite rules that rewrite the given BNF entry in such a way that it belongs to the given class of the given definition or to some of its predecessor classes. Thus, if b_0 is an input and b an output BNF definition, if $aec(b_0, e, class)$ and if the BNF entry e is rewritten to e' by some rewrite rule, then it has to hold that $aec(b, e', class')$, where $class'$ is either $class$, or $class'$ has $class$ as its entry, according to the given BNF (so any expression of $class$ also belongs to $class'$) . The need for condition about predecessor clauses is illustrated in Example 6.

Example 6 *Consider the following* BNF *classes:*

$f' \quad := \quad f | f' \vee f'$

$f \quad := \quad f \wedge f | t = t | t < t | \neg(t = t) | \neg(t < t)$

Let us suppose that there are available the following rewrite rules:

`rm_neg_less`*:* $\neg(t_1 < t_2) \quad \longrightarrow \quad t_2 < t_1 \vee t_1 = t_2$

`rm_neg_eq`*:* $\neg(t_1 = t_2) \quad \longrightarrow \quad (t_1 < t_2) \vee (t_2 < t_1)$

The above rules are sufficient to eliminate $\neg$ *from any formula belonging to the class* f'*, but it cannot be done within the class* f *because the construction* $(t < t) \vee (t < t)$ *does not belong to the class* f*. However, this construction belongs to the class* f' *which has* f *as its entry (i.e., any expression of* f *also belongs to* f'*, according to the given* BNF*).*

Thus, when we remove a symbol from a BNF *class c, we have to check whether the obtained construction belongs to c or to some other class from which a construction c can be derived. This example illustrates the purpose of checking this condition.*

The convention about considering two or more subexpressions in the same class to be different objects is necessary in situations like the following. Let us suppose that there is a rewrite rule

$R: \quad F \Leftrightarrow F \quad \longrightarrow \quad \top$

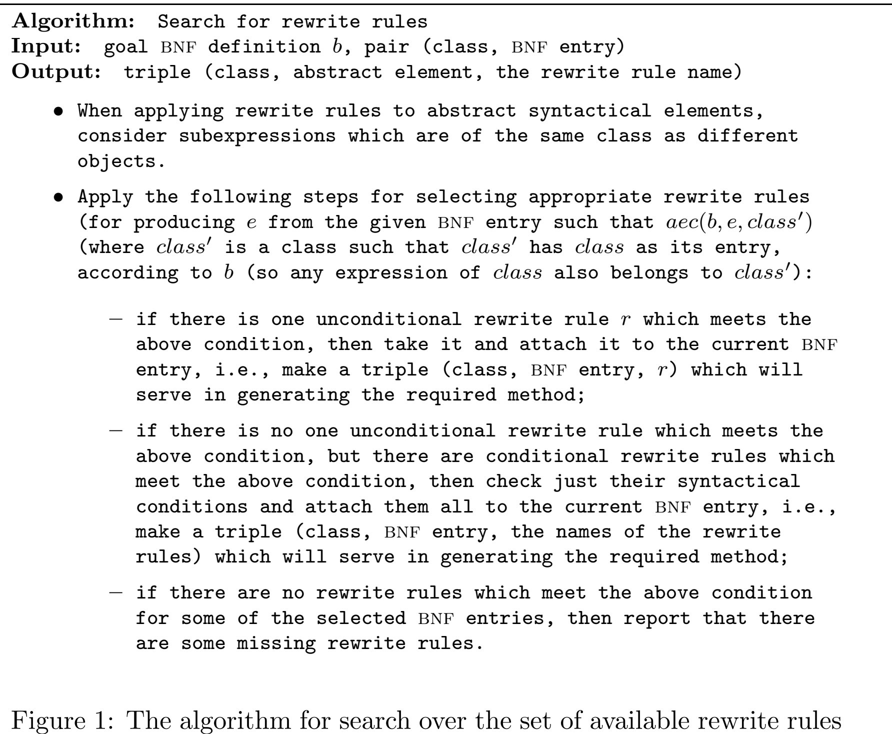
and that we want to rewrite the abstract syntactical expression $f \Leftrightarrow f$; two occurrences of f generally do not represent the same formula, so we must not apply the above rewrite rule (i.e., we consider that it is not applicable; in other words, we consider $f \Leftrightarrow f$ as $f_1 \Leftrightarrow f_2$).

Concerning the step with "checking syntactical conditions", note that some of the rewrite rules that the algorithm browses can have conditions. Some of those conditions may be of a syntactical nature, while some can be of a semantical nature, i.e., such conditions should be checked modulo a background theory. The above algorithm (and the whole process of generating methods) works in a purely syntactical manner and does not reason with respect to any background theory. This sort of condition can be checked in the final stage, when the completeness of the generated methods is examined (see p28). See Example 7 that illustrates these restrictions.

Example 7 *Let us suppose that the class* `f` *of ground formulae is given in the following way:*

$f \quad := \quad f \wedge f | n = n | n < n | true | false$

where `n` *denotes the class of numbers. If the following rewrite rules are available*



Algorithm: `Search for rewrite rules`
Input: `goal` BNF `definition` b, `pair (class,` BNF `entry)`
Output: `triple (class, abstract element, the rewrite rule name)`

- When applying rewrite rules to abstract syntactical elements, consider subexpressions which are of the same class as different objects.
- Apply the following steps for selecting appropriate rewrite rules (for producing e from the given BNF entry such that $aec(b, e, class')$ (where $class'$ is a class such that $class'$ has $class$ as its entry, according to b (so any expression of $class$ also belongs to $class'$):
  - if there is one unconditional rewrite rule r which meets the above condition, then take it and attach it to the current BNF entry, i.e., make a triple (class, BNF entry, r) which will serve in generating the required method;
  - if there is no one unconditional rewrite rule which meets the above condition, but there are conditional rewrite rules which meet the above condition, then check just their syntactical conditions and attach them all to the current BNF entry, i.e., make a triple (class, BNF entry, the names of the rewrite rules) which will serve in generating the required method;
  - if there are no rewrite rules which meet the above condition for some of the selected BNF entries, then report that there are some missing rewrite rules.

Figure 1: The algorithm for search over the set of available rewrite rules

`rm_ls1`: $n_1 < n_2 \quad \longrightarrow \quad true \quad$ *if* $number(n_1)$, $number(n_2)$, $n_1 < n_2$
`rm_ls2`: $n_1 < n_2 \quad \longrightarrow \quad false \quad$ *if* $number(n_1)$, $number(n_2)$, $\neg n_1 < n_2$
then one can eliminate the symbol $<$ *from any formula from the class* `f`*. The algorithm 1 would find and take the above two rules when trying to build a method that eliminates* $<$*. However, it would only check the conditions* $number(n_1)$, $number(n_2)$ *and not the conditions* $n_1 < n_2$ *and* $\neg n_1 < n_2$*. Namely, the conditions* $number(n_1)$, $number(n_2)$ *are of syntactical nature and it suffices to check that the entity of the class n from the above* BNF *meets the syntactical restrictions given by the predicate number (see p5). On the other hand, during the method generation, concrete expressions that will be the subject of rewriting cannot be known in advance, so it is not possible to check semantical conditions, conditions modulo a background theory (such as* $n_1 < n_2$*) that are sensible only for concrete elements.*

In Example 7, for any two numbers n_1 and n_2, it holds that $n_1 < n_2$ or it holds that $\neg n_1 < n_2$. So, the rules `rm_ls1` and `rm_ls2` cover all possible cases, i.e., they have the coverage property, and, therefore, each expression of the form $n < n$ can be rewritten (to $true$ or $false$). However, one has to be careful with the coverage property. Consider, for instance, the following rules:
`r1`: $a \quad \longrightarrow \quad b \quad$ if $f(c)$
`r2`: $a \quad \longrightarrow \quad b \quad$ if $\neg f(c)$

Although it always holds that $f(c) \vee \neg f(c)$, it might be the case that neither $f(c)$ nor $\neg f(c)$ can be proved, and, hence, neither `r1` nor `r2` can be applied.

It is not difficult to see that the given algorithm for search over the set of available rewrite rules is complete, i.e., if there are necessary rewrite rules for transforming expression from one BNF into an expression of the second BNF, then the algorithm finds those rules.

3.4 General Form of the Normalisation Methods

We have two groups of method generators: generic method generators and special-purpose method generators. Special-purpose method generators are designed for use in generating some special kinds of compound methods (say quantifier elimination procedures). Each normalization method has the general form given in Fig. 2.

name: *methodname*;

input: f;

preconditions: $ec(b, f, top\ class)$ (where b is the input BNF definition);

transformation: exhaustive application of the set of rewrite rules (but only applying to positions that correspond to the attached syntactical classes); it transforms f to f';

output: f';

postconditions: $ec(b', f', top\ class)$ (where b' is the output BNF definition).

Figure 2: General form of normalisation methods

4 Generic Method Generators

In this section we describe several kinds of normalisation methods and procedures which can automatically generate them. Most of these methods are based on ideas from [5]. However, there are some differences in specification of some of them. In addition, we haven't used and implemented some kinds of methods from [5] (*reorder*, *collect*, *isolate*), but we introduced one new kind of method — *absorb*. Descriptions of method generators are somewhat simplified.

4.1 *Remove* Method Generator

Remove is a normalization method used to eliminate a certain function symbol, predicate symbol, logical connective, or a quantifier from a formula. The method uses sets of appropriate rewrite rules and applies them exhaustively to the current formula until no occurrences of the specific symbol remain.

For instance, as shown in Example 2 (p7), the BNF definition (1) can be transformed to the corresponding BNF definition without the symbol $\Leftrightarrow$.

Example 8 *Consider the following class:*

$f \quad := \quad h(a)|h(b)|g_1(a)|g_2(b)$

where a and b are some (recursive) classes and suppose there are available the following rewrite rules:

$R_1: \quad h(x) \quad \longrightarrow \quad g_1(x)$

$R_2: \quad h(x) \quad \longrightarrow \quad g_2(x)$

The above rules are sufficient for eliminating the symbol h and for transforming the above class into the class:

$f \quad := \quad g_1(a)|g_2(b)$

However, it cannot be reached by arbitrary use of exhaustive applications of the given rewrite rules. Indeed, the term $h(a)$ belongs to the class f and it can be rewritten to $g_2(a)$ by R_2, but $g_2(a)$ does not belong to the class f. Instead, the rule R_1 should have been used and it would give $g_1(a)$, which does belong to the class f. The lesson is that we have to take care about which rule we use in which situation (i.e., for each construction of syntactical classes). This information has also to be built into the remove method that we want to construct (but similar consideration holds for other kinds of methods as well).

The remove method generator is given in Fig. 3.[9]

The given remove method generator relies on the correctness of the algorithm for search over the set of available rewrite rules (given in Fig. 1) and on the correctness of the construction of the output BNF (within the algorithm itself). This construction is simple and it is not difficult to prove that it succeeds if (and only if) there is an entry involving the given symbol, while the output BNF has fewer classes that involve it. Termination of a generated remove method is guaranteed, as each rewrite applied reduces the number of occurrences of the given symbol. If all available rewrite rules are sound, the generated method is also sound. However, recall that we still cannot claim that each generated method is complete (w.r.t. its underlying theory). Namely, for some steps there might be some conditional rewrite rules used that do not cover all possible cases or there might be some rewrite rules used that are not complete w.r.t. this underlying theory (see 3.1). For instance, if all required rules are found and all of them are unconditional (and complete w.r.t. the underlying theory), then the generated method can transform any input formula and it is also complete w.r.t. the underlying theory.

When checking rewrite rules during the generation of the list of rewrite rules, note that the full check on conditions is not (and cannot be) performed (because the generation mechanism works in a purely syntactical way; see Example 7, p12). That is why the remove method constructed in this way may fail (i.e.,

[9]While generating remove methods, we could consider only rewrite rules such that their left hand side include and that their right hand side does not include the given symbol (the symbol that should be removed). Instead, however, we use the general method for searching for appropriate rewrite rules (given in Fig. 1) with the same result. We believe that gains from generality in this context outweigh the losses in efficiency. Anyway, since the generated output BNF set does not include the critical symbol, it is obvious that all selected rules must be such that their right hand side does not include the given symbol (so recursively defined symbols obviously cannot be eliminated this way).

Algorithm: *Remove* `method generator`
Input: BNF `definition` b`, the symbol to be deleted, the set of available rewrite rules`
Output: `remove method, or report on the missing rewrite rules`

- `Generate the output` BNF `set in the following way: take the input` BNF `definition, find one class that has entries involving the given symbol (let us call it the` *`target class`*`) delete all` BNF `entries (of that class) that include the given symbol (the symbol to be removed);`
- `make a set of all deleted` BNF `entries from the input` BNF `set (with the target class), i.e., all` BNF `entries (of the target class) that include the symbol to be removed;`
- `for each such pair ((target class,` BNF `entry)) call the algorithm for searching for rewrite rules (the algorithm given in Fig. 1). If it succeeds`
  - `then construct a method based on exhaustive application of selected rewrite rules;`
  - `else report on the missing rules.`

Figure 3: The algorithm for generating remove methods

its postconditions may fail) even if its preconditions are fulfilled. However, during the generation process, all relevant rewrite rules are collected and if their conditions cover all possible cases (i.e., if they meet the coverage property), the generated method never fails. This holds for other kinds of methods as well.

If the given algorithm succeeds, the selected set of the rewrite rules is terminating (so is the constructed method). Indeed, each application of these rewrite rules decreases the number of occurrences of the given symbol in the current formula. Since this number is non-negative, this process will stop after a finite number of steps.

Example 9 *For our example 8, the algorithm would work as follows: the set of pairs is* $\{(f, h(a)), (f, h(b))\}$ *and the goal* BNF *is:* $f \quad := \quad g_1(a)|g_2(b)$

When the search algorithm is called for the pair $(f, h(a))$*, it returns* $(f, h(a), R_1)$*, when it is called for* $(f, h(b))$*, it returns* $(f, h(b), R_2)$*. These two triples are used for guiding rewriting in the method generator.*

Example 10 *Given the following* BNF *definition:*

$$\begin{array}{rcl} f & := & af|\ af|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f|(\exists var : sort)f|(\forall var : sort)f \\ af & := & \top|\bot|t = t|t < t \\ t & := & var|rc|rc \cdot t| - t|t + t \\ var & := & x_1|x_2|x_3|\ldots \\ sort & := & rational \end{array}$$

the algorithm from Fig. 3 (implemented in PROLOG*) generates automatically the remove method (given in Fig. 4) which removes the symbol* $\neg$ *from the class* f *(*`sclass` *defines* BNF *classes (defined in pure syntactical terms), and* `inf_class`

defines infinite classes (see §2.3); two `element_of_sclass` *predicates define preconditions and postconditions of the method).*

```
method(rm_neg,la,F,FF) :-
   element_of_sclass(
      [sclass(f, [f#f, f\f, f=>f, f<=>f, ~false, ~true, ~t<t, ~t=t, false,
                 true, t<t, t=t, var:rational##f, var:rational\\f]),
       sclass(t, [var, re, -t, t+t, re*t]),
       sclass(re, [rc, re+re, re*re, -re]),
       sclass(var, []),
       sclass(rc, []),
       inf_sclass(var, _G965, [variable(_G965)]),
       inf_sclass(rc, _G977, [number(_G977)])],f,F),
   rewrite_wrt_bnf(la,
      [sclass(f, [f#f, f\f, f=>f, f<=>f, ~false, ~true, ~t<t, ~t=t, false,
                 true, t<t, t=t, var:rational##f, var:rational\\f]),
       sclass(t, [var, re, -t, t+t, re*t]), sclass(re, [rc, re+re, re*re, -re]),
       sclass(var, []),
       sclass(rc, []),
       inf_sclass(var, _G965, [variable(_G965)]),
       inf_sclass(rc, _G977, [number(_G977)])],
      [[f, ~false, rm_bottom],
       [f, ~true, rm_top],
       [f, ~t<t, rm_neg_less],
       [f, ~t=t, rm_neg_eq]],F,FF),
   element_of_sclass(
      [sclass(f, [f#f, f\f, f=>f, f<=>f, false, true, t<t, t=t,
                 var:rational##f, var:rational\\f]),
       sclass(t, [var, re, -t, t+t, re*t]),
       sclass(re, [rc, re+re, re*re, -re]),
       sclass(var, []),
       sclass(rc, []),
       inf_sclass(var, _G965, [variable(_G965)]),
       inf_sclass(rc, _G977, [number(_G977)])],f,FF).
```

Figure 4: Example of generated remove method (see Example 10)

The method rewrites the input formula by using `rewrite_wrt_bnf`*, which does rewriting with respect to the given* BNF*. For instance,* `rewrite_wrt_bnf` *rewrites subexpressions which are of the class* `f` *(w.r.t. given* BNF*) and of the form* `~false` *by using the rewrite rule* `rm_bottom`*.*

Note that if some elements of the set of pairs cannot be deleted, then a required remove method cannot be constructed. In that case, the algorithm reports such pairs, so the user could try to provide missing rewrite rules. For instance, if, in the above example, the rule `rm_top` was not available, the generation mechanism would issue a report given in Fig. 5 (`~` stands for ¬). It reports on the entry for which it failed to find the necessary rewrite rules (and on the other entries it still has not transformed).

As said before, we don't claim that the given remove method generator is

```
Failed to find a rewrite rule for: [f, ~true]

Cannot synthesize required method.
Hint: try to provide required rewrite rules for the following syntactical
elements: [[f, ~true], [f, ~t<t], [f, ~t=t]]
```

Figure 5: Output of remove method generator when it fails

complete (i.e., that it can construct the required remove method whenever it is possible). However, this algorithm is powerful enough for the case study presented in this paper.

4.2 *Stratify* Methods Generator

Stratify is a normalization method used to stratify one syntactical class into two syntactical classes containing some predicate or function symbols, logical connectives or quantifiers.

Example 11 *A stratify method for moving disjunctions beneath conjunctions can be constructed if the following rewrite rules are available:*

`st_conj_disj1`: $f_1 \wedge (f_2 \vee f_3) \longrightarrow (f_1 \wedge f_2) \vee (f_1 \wedge f_3)$

`st_conj_disj2`: $(f_2 \vee f_3) \wedge f_1 \longrightarrow (f_2 \wedge f_1) \vee (f_3 \wedge f_1)$

Given the above rewrite rules, a stratify method can be constructed such that it can transform each formula of the class

$f \quad := \quad af|f \vee f|f \wedge f$

into a formula of the (new) class f:

$f \quad := \quad f_1|f \vee f$

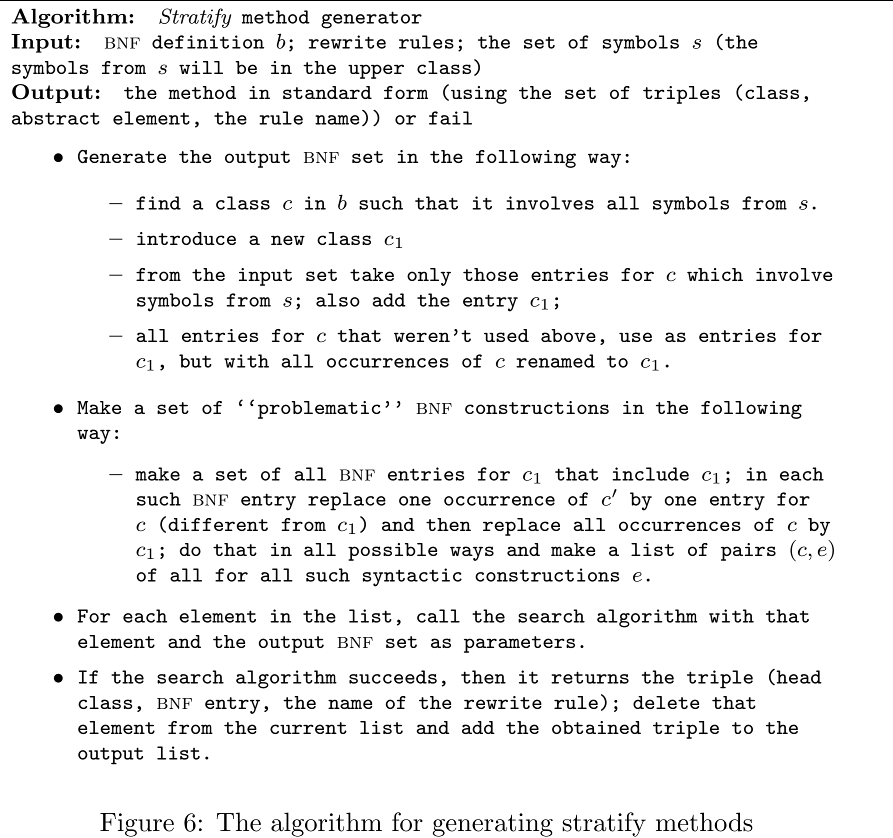
$f_1 \quad := \quad af|f_1 \wedge f_1$

Stratification methods separate one class into two classes, and in this example the symbol $\vee$ is a symbol which stays in the "upper" class (from which the new one is derived) and is a parameter for the method generator.

The algorithm for constructing stratify methods is given in Fig. 6.

The given stratify method generator correctly constructs possible output BNFs, i.e., it constructs an output BNF such that one input class is separated into two classes, one involving symbols from the given set, and the other involving (only) the remaining symbol from the input class. Termination of the generated stratify method is guaranteed, as each rewrite applied reduces the number of the given symbols occurring "below" other symbols (in the expression being rewritten). Soundness and completeness features of generated stratify methods are similar to generated remove methods (see also 3.1).

Example 12 *Stratify methods can also serve for reordering elements within an expression. For instance, a stratify method can use the following rule (together with other necessary rules):*



Algorithm: *Stratify* method generator
Input: BNF definition b; rewrite rules; the set of symbols s (the symbols from s will be in the upper class)
Output: the method in standard form (using the set of triples (class, abstract element, the rule name)) or fail

- Generate the output BNF set in the following way:
  - find a class c in b such that it involves all symbols from s.
  - introduce a new class c_1
  - from the input set take only those entries for c which involve symbols from s; also add the entry c_1;
  - all entries for c that weren't used above, use as entries for c_1, but with all occurrences of c renamed to c_1.
- Make a set of ''problematic'' BNF constructions in the following way:
  - make a set of all BNF entries for c_1 that include c_1; in each such BNF entry replace one occurrence of c' by one entry for c (different from c_1) and then replace all occurrences of c by c_1; do that in all possible ways and make a list of pairs (c, e) of all for all such syntactic constructions e.
- For each element in the list, call the search algorithm with that element and the output BNF set as parameters.
- If the search algorithm succeeds, then it returns the triple (head class, BNF entry, the name of the rewrite rule); delete that element from the current list and add the obtained triple to the output list.

Figure 6: The algorithm for generating stratify methods

`reorder_plus8`: $c_0 \cdot x_0 + c \cdot v \quad \longrightarrow \quad c \cdot v + c_0 \cdot x_0$ *if* v *is a variable different from* x_0

in order to stratify x_0 *and rearrange a polynomial in such a way that its summands involving* x_0 *will be at the end of the expression.*

4.3 *Thin* Method Generator

Thin is a normalization method that eliminates multiple occurrences of a unary logical connective or a unary function symbol. For instance, we can use the following rewrite rule:

$\neg\neg f \quad \longrightarrow \quad f$

in order to transform each formula belonging to

$f \quad := \quad af|\neg f$

to a formula belonging to

$f \quad := \quad af|\neg af$

The algorithm for constructing thin methods is given in Fig. 7.

Algorithm: *Thinning* `method generator`
Input: BNF `definition` b`; rewrite rules; the symbol` F `to be ''thinned''`
Output: `the method in standard form (using the set of triples (class, abstract element, the rule name)) or fail`

- `Select a` BNF `class` c `such that it has the entry` $F(c)$ `and that it does not have any other entry that involves` c
- `Generate the output` BNF `set in the following way: delete the entry` $F(c)$`, and for each other entry` e `add the entry` $F(e)$
- `Check if there is a rewrite rule` R `of the form:`
  $F(F(x)) \quad \longrightarrow \quad x$
  `If there are such rewrite rule, then form the triple (c, F(F(c)), R) (it is of the form (head class,` BNF `entry, the name of the rewrite rule)) and use it for building the method in a standard form; otherwise fail.`

Figure 7: Generator for thinning methods

The given thin method generator correctly constructs output BNF: it construct an output BNF by eliminating an entry $F(c)$ and replacing it by (or keeping) an entry $F(e)$ (as denoted in the algorithm). If the method generator succeeds, the selected rewrite rule is obviously terminating (each application of the rule decreases the number of occurrences of symbol F by 2), and so is the constructed method. Soundness and completeness (w.r.t. the underlying theory) rely on the properties of the rewrite rules available/used (see also 3.1).

4.4 *Absorb* Method Generator

Absorb is a normalization method that can eliminate some recursion rules. For instance, we can use the following rewrite rule:

`rm_mult` $c_1 \cdot c_2 \longrightarrow \quad c_3 \quad \text{if} \quad c_3 = c_1 \cdot c_2$

in order to transform each term belonging to

$t \quad := \quad t \cdot rc | rc$ (where rc denotes rational constants) to a term belonging to

$t \quad := \quad rc$

The algorithm for constructing absorb methods is given in Fig. 8.

Example 13 *Consider the following class:*

$t \quad := \quad t \cdot rc | rc | var$

Using the notation from the given algorithm, $A = \{t \cdot rc\}$*,* $B = \{rc\}$*,* $C = \{rc, var\}$*. According to the algorithm, the new definition of the class t will be:*

$t \quad := \quad var \cdot rc | rc | var$

while the necessary rewrite rules are of the form:

$$rc \cdot rc \quad \longrightarrow \quad rc$$
$$(t \cdot rc) \cdot rc \quad \longrightarrow \quad t \cdot rc$$

Algorithm: *Absorb* method generator
Input: BNF definition b; rewrite rules; a function symbol of connective F.
Output: the method in standard form (using the set of triples (class, abstract element, the rule name)) or fail

- Select a BNF class c such that all its entries are either of the form $F(c, c_i)$, $F(c_i, c)$ (for some function symbol or a connective F; c_i are not equal to c) or does not have c. Let: A be the set of all $F(c, c_i)$ and $F(c_i, c)$ entries (for that class); B the set of all c_i, C the set of all entries apart from those in A.
- Generate the output BNF set such that its set of entries for c is equal to

$$C \cup \bigcup_{e \in A, g \in C \setminus B} \{e[c \mapsto g]\}$$

- Check if there are the following rewrite rules:
  - for all $c_i \in B$, $c_j \in B$:
    $F(c_i, c_j) \longrightarrow c_k$
    such that $c_k \in B$.
  - if the set $C \setminus B$ is nonempty, for all $c_i \in B$, $c_j \in B$:
    $F(F(c, c_i), c_j) \longrightarrow F(c, c_k)$
    such that $F(c, c_i), F(c, c_j), F(c, c_k) \in A$ and $c_k \in B$
  - if the set $C \setminus B$ is nonempty, for all $c_i \in B$, $c_j \in B$:
    $F(c_j, F(c_i, c)) \longrightarrow F(c_k, c)$
    such that $F(c_i, c), F(c_j, c), F(c_k, c) \in A$ and $c_k \in B$

  If there are such rewrite rules, then form the rewrite triples and use them for building the method in a standard form; otherwise fail.

Figure 8: Generator for absorb methods

The given absorb method generator constructs an output BNF with fewer recursive rules (than the input BNF). Termination of the generated absorb method is guaranteed, as each application of the selected rewrite rules decreases (by 1) the number of occurrences of the symbol F (as denoted in the algorithm). Soundness and completeness features of generated absorb methods are similar to the generated remove and stratify methods, i.e., they rely on the properties of the selected rewrite rules (see also 3.1).

4.5 $Left - assoc$ Method Generator

Left-assoc is one of the normalization methods for reorganising within a class. If a syntactical class contains only one function symbol or a connective and if that symbol is both binary and associative, then members of this class can be put into left associative form using the method *left-assoc*. For instance, we will need the left association of addition and the left association of conjunction.

Example 14 *We can transform the following class*

$f \quad := \quad f \wedge f|\top|\bot$

into the following class

$f \quad := \quad f \wedge \top|f \wedge \bot|\top|\bot$

by the use of the following rewrite rule:

`left_assoc_conj`*:* $f_1 \wedge (f_2 \wedge f_3) \quad \longrightarrow \quad (f_1 \wedge f_2) \wedge f_3$

The algorithm for constructing *left-assoc* methods is given in Fig. 9.

Algorithm: *Left-assoc* method generator
Input: BNF definition b; rewrite rules; the symbol F to be processed
Output: the method in standard form (using the set of triples (class, abstract element, the rule name)) or fail

- Select a BNF class c such that it has the entry $F(c, c)$
- Generate the output BNF set in the following way: delete the entry $F(c, c)$, and for each other entry e add the entry $F(c, e)$
- Check if there is a rewrite rule R of the form:
  $F(x, F(y, z)) \quad \longrightarrow \quad F(F(x, y), z)$
  If there are such rewrite rule, then form the triple (c, F(c,F(y,z)), R) (it is of the form (head class, BNF entry, the name of the rewrite rule)) and use it for building the method in a standard form; otherwise fail.

Figure 9: Generator for *left-assoc* methods

The given *left-assoc* method generator simply constructs an output BNF with a symbol F (as denoted in the algorithm) put in left associative form. If the method generator succeeds, the constructed method is terminating (as the selected rewrite rule is terminating). Soundness and completeness w.r.t. the underlying theory rely on the properties of the rewrite rules available/used (see also 3.1).

5 Search Engine for Synthesizing Decision Procedure

Given some method generators, a BNF description of a theory $\mathcal{T}$ and a set of corresponding rewrite rules, a user can go step by step and try to combine different generated methods. Moreover, an automatic search for compound methods or for a decision procedure for $\mathcal{T}$ can be performed. The goal of the search process is to generate a sequence of methods such that:

- the output BNF class of the non-final method is the input BNF class of the next method in the sequence;

- the output BNF class of the last method in the sequence is a goal BNF, for instance, the trivial BNF — consisting of only two entries ($\top$ and $\bot$ for the top BNF class).

We will define the search procedure, which is based on a size of BNF and on a heuristic which controls it. The search procedure tries to find a sequence of methods that consists of subsequences, such that each of them is of length less than or equal to a given value M and such that each of them decreases the measure of the current BNF. The motivation for this heuristic is as follows: we (normally) look for a sequence of methods that reduces the (appropriately defined) size of the input BNF. Within that sequence, it might be the case that some steps do not decrease the size of the current BNF. However, as the whole of the required procedure decreases the size of BNF, there must be *subsequences* such that their last element has lower BNF size than the first one (i.e., each of these subsequences decreases the size of the current BNF). The maximal allowed length of these subsequences is controlled by the value M. So, in the generated procedure there might be some BNF size increasing steps, but the whole of the procedure would be built from subsequences of length less or equal to M, and each of them will be BNF size decreasing. It is not likely that $M = 1$ is a good choice (i.e., usually not all steps decrease BNF size). The value $M = 3$ can be good.

The size of BNF definition is a heuristic measure and we define it as follows:

- the size of a BNF definition is equal to the sum of sizes of all its entries;
- the size of the entry $c := c'$ is defined as follows:
  - each symbol c in c' adds 100 to the size of c;
  - each symbol c'' (where c'' is some other class than BNF) adds 10 to the size of c;
  - each other symbol adds 1 to the size of c.

As said, this is a heuristic measure and it can be defined in some other way. Defined this way, it forces the search engine to try to get rid of recursive classes,[10] and then of the classes which involve some other classes. The trivial, goal BNF class (consisting of only $f := \top|\bot$) has the size 2.

Example 15 *The size of the following* BNF *definition (corresponding to ground arithmetic):*

$$\begin{array}{rcl} f & := & af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f \\ af & := & \top|\bot|t = t|t < t|t > t|t \leq t|t \geq t|t \neq t \\ t & := & rc| - t|t \cdot t|t + t \end{array}$$

is 1556.

The size of BNF *definition 1 is 2233.*

The search algorithm (in a somewhat simplified form) is given in Fig. 10.

[10] Note that, within the current scope of our system, we don't deal with mutual recursion (that can be problematic in many aspects).

```
Algorithm: Search engine for compound methods
Input:
          starting BNF definition b,
          goal BNF definition b',
          maximal length a sequence M,
          maximal length of a current subsequence m,
          goal BNF size S for a current subsequence,
          prefix sequence of methods p,
          rewrite rules,
          method generators
Output:
          sequence of generated methods (the input BNF class of the first method is
b, the output BNF class of the last method is b')

  • if m = 0, then return with failure;

  • for all possible and successful applications of all available method
    generators (with the input BNF for methods equal to b)

       – let nb be equal to the output BNF of the newly generated method;

       – let q be p concatenated by the newly generated method;

       – if nb is equal to b' then report success and return the current
         sequence of methods (q);

       – let S' is the size of nb; if S' < S

            * then call recursively this algorithm with starting BNF definition
              nb, goal BNF definition b', numbers M and M, q, rewrite rules,
              and method generators as input parameters; if it succeeds, then
              attach its resulting sequence of methods at the end of q;

            * else call recursively this algorithm with starting BNF definition
              nb, goal BNF definition b', numbers M and m - 1, q, rewrite
              rules, and method generators as input parameters; if it succeeds,
              then attach its resulting sequence of methods at the end of q;

  • return and report no more solutions.
```

Figure 10: The algorithm for search over the set of available method generators

The search algorithm is initially invoked with the starting BNF definition which corresponds to the whole of the theory $\mathcal{T}$, the trivial goal BNF (consisting only of $\top$ and $\bot$), a value M (M is a chosen value, say, 3), the size of the starting BNF, the empty sequence of methods, and with the set of available (or chosen) rewrite rules and method generators. The given algorithm can be modified in such a way that it stops when it finds the first sequence of methods which meets the given conditions. Also, the algorithm can be modified in such a way that instead of a specific target BNF b', the input parameter is some condition on the goal BNF definition (e.g., "target BNF does not have variables").

The ordering of method generators is not important for termination and correctness of the given search algorithm, but it is important for its efficiency. We used the following ordering: `remove`, `thin`, `absorb`, `stratify`, `left_assoc`. This ordering is based on the nature of these methods (and how they change the BNF size) and on different tests. When generating all possible methods,

all acceptable parameters for method generators have to be generated too. All possible parameters are generated from the list of all connectives, function and predicate symbols used in the given BNF. For generators that take sets of symbols, these sets are made as subsets of all symbols.[11]

Given a finite number of method generators and a finite number of rewrite rules, at each step a finite number of methods can be generated (there is also a finite number of possible parameters). Thus, since the algorithm produces subsequences (of maximal length M) of decreasing sizes (natural numbers) of corresponding BNF definitions, the given algorithm obviously terminates. If method generators can generate all methods necessary for building the required compound method, then the given algorithm can build one such compound method (for M large enough). If we iterate the given algorithm (for $M = 1, 2, 3, \ldots$), then it will eventually build the required compound method (if it is possible to build it using the method generators), so that the iterated algorithm is also *complete*. However, we can also use it only with particular values for M (then the procedure is not complete, but it obviously gives much better results if it used only for an appropriate value for M).

6 Ground Arithmetic

We ran the algorithm given in Fig. 10 (p24), on the BNF definition given in Example 15 (p23), with $M = 3$, with the described method generators, and with rewrite rules available in Appendix A. We set the goal BNF definition to be the trivial one ($f := \top|\bot$), thus aiming at synthesizing a decision procedures for ground arithmetic.

The search algorithm took 3.22 seconds of *cpu* time[12] and during the search there were 48 methods successfully generated (while there are 22 of them in the final sequence). The search algorithm produced the sequence of methods `DP_GA` with the following "overview" (`#` denotes conjunction and `\` denotes disjunction; the symbol `(*)` denotes the end of a BNF size decreasing sequence):

```
1 remove <=>; Target class: f;
   Rules:[[f, f<=>f, rm_equiv]];
   Input size: 1556 Output size : 1345 (*)
2 remove =>; Target class: f;
   Rules:[[f, f=>f, rm_impl]];
   Input size: 1345 Output size : 1144 (*)
3 remove leq; Target class: f;
   Rules:[[f, leq(t, t), rm_leq]];
   Input size: 1144 Output size : 1123 (*)
4 remove geq; Target class: f;
   Rules:[[f, geq(t, t), rm_qeq]];
```

[11] One of the specific method generators renames the innermost variable of the input formula to x_0. This distinguished variable name is also used as a possible parameter for generated methods.

[12] The system is implemented in SWI Prolog; experiments were run on a 256Mb PC 2.4Ghz.

```
   Input size: 1123 Output size : 1102 (*)
5 remove neq; Target class: f;
   Rules:[[f, neq(t, t), rm_neq]];
   Input size: 1102 Output size : 1081 (*)
6 remove >; Target class: f;
   Rules:[[f, t>t, rm_gr]];
   Input size: 1081 Output size : 1060 (*)
7 remove -; Target class: t;
   Rules:[[t, -t, rm_minus]];
   Input size: 1060 Output size : 959 (*)
8 stratify [#, \]; Target class: f;
   Rules:[[f, ~(f#f), st_neg_conj], [f, ~(f\f), st_neg_disj]];
   Input size: 959 Output size : 969
9 thin ~; Target class: f1;
   Rules:[[f1, ~(~f1), thin_neg]];
   Input size: 969 Output size : 906 (*)
10 remove ~; Target class: f;
   Rules:[[f, ~false, rm_bottom], [f, ~true, rm_top],
          [f, ~t<t, rm_neg_less], [f, ~t=t, rm_neg_eq]];
   Input size: 916 Output size : 858 (*)
11 stratify [\]; Target class: f;
   Rules:[[f, f#f\f, st_conj_disj1], [f, (f\f)#f, st_conj_disj2]];
   Input size: 858 Output size : 868
12 stratify [+]; Target class: t;
   Rules:[[t, t*(t+t), st_mult_plus1], [t, (t+t)*t, st_mult_plus2]];
   Input size: 868 Output size : 878
13 left_assoc \; Target class: f;
   Rules:[[f, f\f, left_assoc_disj]];
   Input size: 878 Output size : 788 (*)
14 left_assoc +; Target class: t;
   Rules:[[t, t+t, left_assoc_plus]];
   Input size: 788 Output size : 698 (*)
15 left_assoc *; Target class: t1;
   Rules:[[t1, t1*t1, left_assoc_mult]];
   Input size: 698 Output size : 608 (*)
16 absorbe *; Target class: t1;
   Rules:[[t1, rc*rc, rm_mult]];
   Input size: 608 Output size : 487 (*)
17 absorbe +; Target class: t;
   Rules:[[t, rc+rc, reduce_plus]];
   Input size: 497 Output size : 366 (*)
18 remove <; Target class: f1;
   Rules:[[f1, rc<rc, rm_ls1], [f1, rc<rc, rm_ls2]];
   Input size: 376 Output size : 345 (*)
19 remove =; Target class: f1;
   Rules:[[f1, rc=rc, rm_eq1], [f1, rc=rc, rm_eq2]];
   Input size: 345 Output size : 324 (*)
20 left_assoc #; Target class: f1;
   Rules:[[f1, f1#f1, left_assoc_conj]];
   Input size: 324 Output size : 327
21 remove #; Target class: f1;
   Rules:[[f1, f1#false, reduce_bool3], [f1, f1#true, reduce_bool1]];
   Input size: 327 Output size : 206 (*)
22 remove \; Target class: f;
   Rules:[[f, f\false, reduce_bool4], [f, f\true, reduce_bool2]];
   Input size: 123 Output size : 2
```

In the presented overview we give the kind, parameters, target class, the triples with selected rewrite rules, and input and output size. Note that we show the input BNF size *after* and not *before* simplification (so the output size is not necessarily equal to the input size of the next one.

In Fig. 11 we illustrate work of the given procedure DP_GA on one example — we give a sequence of formulae produced by the methods for the input formula $\neg(7 \leq 5) \Rightarrow \neg(2 \cdot (1+3) \geq 3)$.

```
 1 ~leq(7, 5)=> ~geq(2*(1+3), 3)
 2 ~(~leq(7, 5))\ ~geq(2*(1+3), 3)
 3 ~(~(7<5\7=5))\ ~geq(2*(1+3), 3)
 4 ~(~(7<5\7=5))\ ~(3<2*(1+3)\2*(1+3)=3)
 5 ~(~(7<5\7=5))\ ~(3<2*(1+3)\2*(1+3)=3)
 6 ~(~(7<5\7=5))\ ~(3<2*(1+3)\2*(1+3)=3)
 7 ~(~(7<5\7=5))\ ~(3<2*(1+3)\2*(1+3)=3)
 8 (~(~7<5)\ ~(~7=5))\ ~3<2*(1+3)# ~2*(1+3)=3
 9 (7<5\7=5)\ ~3<2*(1+3)# ~2*(1+3)=3
10 (7<5\7=5)\(2*(1+3)<3\3=2*(1+3))#2*(1+3)<3\3<2*(1+3)
11 (7<5\7=5)\((2*(1+3)<3#2*(1+3)<3)\3=2*(1+3)#2*(1+3)<3)\
    (2*(1+3)<3#3<2*(1+3))\3=2*(1+3)#3<2*(1+3)
12 (7<5\7=5)\((2*1+2*3<3#2*1+2*3<3)\3=2*1+2*3#2*1+2*3<3)\
    (2*1+2*3<3#3<2*1+2*3)\3=2*1+2*3#3<2*1+2*3
13 ((((7<5\7=5)\2*1+2*3<3#2*1+2*3<3)\3=2*1+2*3#2*1+2*3<3)\
    2*1+2*3<3#3<2*1+2*3)\3=2*1+2*3#3<2*1+2*3
14 ((((7<5\7=5)\2*1+2*3<3#2*1+2*3<3)\3=2*1+2*3#2*1+2*3<3)\
    2*1+2*3<3#3<2*1+2*3)\3=2*1+2*3#3<2*1+2*3
15 ((((7<5\7=5)\2*1+2*3<3#2*1+2*3<3)\3=2*1+2*3#2*1+2*3<3)\
    2*1+2*3<3#3<2*1+2*3)\3=2*1+2*3#3<2*1+2*3
16 ((((7<5\7=5)\2+6<3#2+6<3)\3=2+6#2+6<3)\2+6<3#3<2+6)\3=2+6#3<2+6
17 ((((7<5\7=5)\8<3#8<3)\3=8#8<3)\8<3#3<8)\3=8#3<8
18 ((((false\7=5)\false#false)\3=8#false)\false#true)\3=8#true
19 ((((false\false)\false#false)\false#false)\false#true)\false#true
20 ((((false\false)\false#false)\false#false)\false#true)\false#true
21 ((((false\false)\false)\false)\false)\false
22 false
```

Figure 11: Sequence of formulae produced by the methods in a decision procedure for ground arithmetic (DP_GA)

The procedure DP_GA is sound and terminating (as all generated methods are sound and terminating and there is no loop). Note, however, that we don't claim that it is complete as there are some conditional rewrite rules used. For instance, in step 16 of the procedure DP_GA, the conditional rule rm_mult is used, but it is still not shown that its condition covers all possible cases, which the user can show by proving:

$$(\forall c_1 : rational)(\forall c_2 : rational)(\exists c_3 : rational)(c_3 = c_1 \cdot c_2)$$

In step 17 of the procedure DP_GA, the conditional rule reduce_plus is used,

but it is still not shown that its condition covers all possible cases, which the user can show by proving:

$$(\forall c_1 : rational)(\forall c_2 : rational)(\exists c_3 : rational)(c_3 = c_1 + c_2)$$

It is very easy to prove that the above conjectures are theorems of PRA. Moreover, the second one can be proved by the decision procedure `DP_LA` for linear arithmetic (which we also automatically generated and we report on that in the subsequent text).

As another example, in step 19, the conditional rewrite rules `rm_eq1` and `rm_eq2` are used, but it is not shown that their conditions cover all possible cases, which the user can show by proving that for any two rational numbers c_1 and c_2, either $c_1 = c_2$ or $\neg(c_1 = c_2)$ can be proved as a theorem of linear arithmetic (i.e., there always holds one of them, and one can decide which). Notice that this is not equivalent to a conjecture:

$$(\forall c_1 : rational)(\forall c_2 : rational)(c_1 = c_2 \vee \neg(c_1 = c_2))$$

but is rather a metha-theorem of linear arithmetic. However, for all these conjectures (concerning the coverage property) the user can easily ensure they are valid.

This leads us to conclude that the procedure `DP_GA` is correct.

Theorem 1 *The procedure* `DP_GA` *for ground arithmetic is terminating, sound and complete, i.e., it is a decision procedure for ground arithmetic.*

Proof sketch. Since all methods in `DP_GA` use only sound and complete rewrite rules, all of them are sound and complete, and hence, the whole of the procedure is sound and complete. In each method, either unconditional rules are used or conditional rules that cover all possible cases. Thus, all methods always succeed. As discussed in the previous text, each of the used methods is terminating. There are no loops, so termination of individual methods is sufficient for termination of `DP_GA`. All in all, the procedure `DP_GA` terminates, it transforms an arbitrary input (ground arithmetic) formula Φ into the resulting formula $\top$ or $\bot$, while the resulting formula is $\top$ if and only if Φ is a theorem in ground arithmetic.

7 Special-Purpose Method Generators

In this section we describe four special-purpose method generators. The first one of them (adjusting the innermost quantifier) can be used for a quantifier elimination procedure for any theory, while the remaining three are specific for linear arithmetic. Note, however, that it is essential to have these generators (despite they are theory-specific): they can be used in an automatic search process and generate the required methods with the given preconditions (which are not known in advance).

7.1 Method Generator for Adjusting the Innermost Quantifier

This generator generates a method which transforms a formula in prenex normal form (and with variables) in the following way: if its innermost quantifier is existential, then keep it unchanged; if its innermost quantifier is universal, then rewrite the formula

$$(Qx_1)(Qx_2)\dots(Qx_n)(\forall x)f$$

to a formula

$$(Qx_1)(Qx_2)\dots(Qx_n)\neg(\exists x)\neg f$$

by using the following rewrite rule:

`rm_univ`: $(\forall x)f \quad \longrightarrow \quad \neg(\exists x)\neg f$

The motive of this method is to deal only with elimination of existential quantifiers.

This method is generated in such a way that it also always (both when the innermost quantifier is universal and when the innermost quantifier is existential) renames the innermost variable to x_0, assuming that there was no a variable x_0 already (if there is, then that variable will be renamed). That way it is always easily known what variable is the innermost one.

7.2 *One-side* Method Generator

This generator generates (if the input BNF admits that) a method which transforms all (linear arithmetic) literals (with given predicate symbols) in such a way that each of them has 0 as its second argument. For instance, if we take symbols $<$, $>$, $\leq$, $\neq$, $\geq$, $=$ as parameters, then after applying that generated method each literal (that had one of these predicate symbols) will have one of the following forms: $t < 0$, $t > 0$, $t \leq 0$, $t \neq 0$, $t \geq 0$, $t = 0$.

Example 16 *The following rules*

`one_side1`*:* $t_1 = t_2 \quad \longrightarrow \quad t_1 + (-1)\cdot t_2 = 0 \quad$ *if* $\quad t_2 \neq 0$

`one_side2`*:* $t_1 < t_2 \quad \longrightarrow \quad t_1 + (-1)\cdot t_2 < 0 \quad$ *if* $\quad t_2 \neq 0$

are sufficient to transform all literals to one-side *form if they have only predicate symbols* $=$ *and* $<$*.*

7.3 Method Generator for Isolating a Variable

This generator generates (if the input BNF admits that) a method which isolates a distinguished variable x_0 in all (linear arithmetic) literals (that distinguished name is normally introduced by the method that adjust the innermost quantifier). After applying that method, each of the literals either does not involve x_0 or has one of the forms: $\nu x_0 = \gamma$, $x_0 = \gamma$, $\lambda x_0 < \alpha$, $x_0 < \alpha$ (where x_0 is the innermost quantifier). This method will use such rewrite rules as:

`isolate2`: $t + x_0 = 0 \quad \longrightarrow \quad x_0 = -t$

`isolate3`: $t + c\cdot x_0 < 0 \quad \longrightarrow \quad c\cdot x_0 < -t$

7.4 Method Generator for Removing a Variable

The, so-called, *cross multiply and add* step is an essential step of the Fourier/Motzkin style procedure. It is applied in situations where the current formula is in prenex normal form (while $\neg$ can precede the innermost quantifier if it is an existential one), its quantifier free part is in disjunctive normal form (or negation of disjunctive normal form) and in each disjunct, each of the literals either does not involve x_0 or has one of the forms: $\nu x_0 = \gamma$, $x_0 = \gamma$, $\lambda x_0 < \alpha$, $x_0 < \alpha$ (where x_0 is the innermost quantifier). It consists of the following steps for each disjunct:

- if there is an equality of the form $\nu x_0 = \gamma$ where $\nu \neq 0$ (or $x_0 = \gamma$ for $\nu = 1$), then rewrite each atomic formulae of the form $\nu_i x_0 = \gamma_i$ to $\nu_i \gamma = \nu \gamma_i$, rewrite each atomic formulae of the form $\lambda_i x_0 < \alpha_i$ to $\lambda_i \gamma < \nu \alpha_i$ (if $\nu > 0$) and to $\lambda_i \gamma > \nu \alpha_i$ (if $\nu < 0$), and then delete the literal $\nu x_0 = \gamma$; that is, rewrite

  $$\nu x_0 = \gamma \wedge \bigwedge_k \nu_k x_0 = \gamma_k \wedge \bigwedge_i \lambda_i x_0 < \alpha_i \wedge \bigwedge_l \phi_l$$

  (where ϕ_l are literals not involving x_0) to (if $\nu > 0$):

  $$\bigwedge_k \nu_k \gamma = \nu \gamma_k \wedge \bigwedge_i \lambda_i \gamma < \nu \alpha_i \wedge \bigwedge_l \phi_l$$

  or to (if $\nu < 0$):

  $$\bigwedge_k \nu_k \gamma = \nu \gamma_k \wedge \bigwedge_i \lambda_i \gamma > \nu \alpha_i \wedge \bigwedge_l \phi_l$$

- if there is no equality of the form $\nu x_0 = \gamma$, then for each pair of literals $\lambda_i x_0 < \alpha_i$, $\mu_j x_0 < \beta_j$ (where $\lambda_i > 0$ and $\mu_j < 0$) add the new literal $\lambda_i \beta_j > \mu_j \alpha_i$, and then delete all literals of the form $\lambda_i x_0 < \alpha_i$, $\mu_j x_0 < \beta_j$; that is, rewrite

  $$\bigwedge_i \lambda_i x_0 < \alpha_i \wedge \bigwedge_j \mu_j x_0 < \beta_j \wedge \bigwedge_l \phi_l$$

  (where $\lambda_i > 0$, $\mu_j < 0$, and ϕ_l are literals not involving x_0) to

  $$\bigwedge_{i,j} \lambda_i \beta_j > \mu_j \alpha_i \wedge \bigwedge_l \phi_l$$

After performing this step, the variable x_0 does not occur in the current formula and so the corresponding quantifier can be deleted.

It is not very difficult to prove that this transformation is both sound and complete for PRA [9].

We have implemented the generator which takes a given BNF and tries to build a method which performs the *cross multiply and add* step to formulae belonging to that BNF.

8 Compound Search Engine for Synthesising Decision Procedures

The search for a decision procedure based on quantifier elimination is performed in three stages:

- the first stage is reaching the BNF for which the method for adjusting the innermost quantifier (changing the innermost universal quantifier into an existential one) is applicable (i.e., reaching a formula in prenex normal form that has quantifiers);
- the second stage continues while the variable elimination method is applicable (in each loop one variable is being eliminated); note that the output BNF of this stage has to be a subset of its input BNF;
- the third stage starts with the output BNF of the first stage, but with all entries involving variables and quantifiers deleted (it is the BNF of the current formula after the loop described as the second stage); its goal BNF definition is the trivial one (i.e., consisting only of $\top$ and $\bot$).

For each of these stages we use the algorithm given in Fig. 10 (p24) (modified in order to accept not only a fixed goal BNF definition, but also weaker specified BNF).

In this search procedure we use all method generators with priority given to the special-purpose method generators.

This compound search is aimed at synthesizing decision procedures based on quantifier elimination. For other underlying techniques or domains, new compound search mechanisms should be defined.

9 Linear Arithmetic

We used the approach described in the previous section on the BNF definition given in Example 1 (p5), with $M = 3$ for the first and the third stage, with $M = 5$ for the second stage[13], with all described method generators, and with the rewrite rules given in Appendix A.

The search algorithm took 5.4 seconds of *cpu* time and during the search there were 89 methods successfully generated (while there are 51 of them in the final sequence). The search algorithm produced a sequence of methods, i.e., a procedure `DP_LA`, with the "overview" given in Appendix B (p41) (dashed lines divides three stages; `##` denotes universal quantifier and `\\` denotes existential quantifier).

In Appendix C, we illustrate the work of the generated procedure `DP_LA` on one example — we give a sequence of formulae produced by the methods for the input formula $(\exists x : rational)(x > 0)$.

[13] For lower values of M the system failed to generate the required procedure.

Theorem 2 *The procedure* `DP_LA` *for linear arithmetic (given in Appendix B (p41) is terminating, sound and complete, i.e., it is a decision procedure for linear arithmetic.*

Proof sketch. Each of individual methods used in the generated procedure `DP_LA` is terminating. Since each loop eliminates one variable and since there are a finite number of variables in the input formula, the loop terminates. Hence, the procedure `DP_LA` is terminating.

Since all methods in `DP_LA` use only sound rewrite rules, all of them are sound, and hence, the whole of the procedure is sound.

The completeness relies on completeness of the rewrite rules used, but also on coverage property for the methods that use conditional rewrite rules. It is easy to see (similarly as in the generated procedure for `DP_LA`) that all required coverage properties are fulfilled. (moreover, some of the coverage properties can be proved by the generated procedure itself, which is, of course, acceptable, as we know that the procedure is sound). Therefore, in each method, either unconditional rules are used or conditional rules that cover all possible cases. Thus, all methods always succeed and all methods are complete. Hence, the procedure `DP_LA` is complete (see also subsection 3.1).

All in all, the procedure `DP_LA` terminates, it transforms an arbitrary input (linear arithmetic) formula Φ into the resulting formula $\top$ or $\bot$, while the resulting formula is $\top$ if and only if Φ is a theorem in linear arithmetic.

We don't claim that the generated procedure `DP_LA` is the shortest or the most efficient one. However, we doubt that a decision procedure for linear arithmetic can be described correctly in some much shorter way. It also shows that it might be non-trivial for a human programmer to implement this procedure without flaws and bugs (even with provided the code for the key step: *cross multiply and add*). Namely, we believe that any implementation (at least, any understandable implementation) of Fourier/Motzkin's procedure has around 50 steps, and, despite the fact that the step *cross multiply and add* is the most complex one, the probable flaws are rather in correctly combining the remaining steps.

10 Additional Examples

The proposed approach can be used for extensions of linear arithmetic with user defined facts.

Example 17 *A "positive difference" function $p(x,y)$ is defined in the following way:*

$$p(x,y) \stackrel{def}{=} \begin{cases} x-y, & \text{if } x \geq y \\ 0, & \text{if } x < y \end{cases} \quad (3)$$

By using the conditional rewrite rules:

$$\begin{array}{lcll} p(x,y) & \longrightarrow & x-y & \text{if } x \geq y \\ p(x,y) & \longrightarrow & 0 & \text{if } x < y \end{array} \quad (4)$$

we can synthesize a decision procedure for linear arithmetic augmented by the function p.

The system presented in this paper can be used not only for producing decision procedures, but also different normalisation procedures (and help in implementing routine procedures, but procedures still very much subject to making implementation flaws). The following example illustrates one such normalisation for the theory of lists.

Example 18 *Let us consider the following* BNF*:*

$$\begin{array}{rcl} t & := & nil|var|append(t,t)|rev(t)|qrev(t,t) \\ var & := & x_1|x_2|x_3|\ldots \end{array} \qquad (5)$$

and the set of rewrite rules:

$$\begin{array}{rcl} qrev(x,y) & \longrightarrow & append(y,rev(x)) \\ rev(append(x,y)) & \longrightarrow & append(rev(y),rev(x)) \\ rev(rev(x)) & \longrightarrow & x \\ append(x,append(y,z)) & \longrightarrow & append(append(x,y),z) \end{array} \qquad (6)$$

Our system can be used to produce the following normal form:

$$\begin{array}{rcl} t & := & nil|var|rev(nil)|rev(var)|append(t,nil)|append(t,var)| \\ & & append(t,rev(nil))|append(t,rev(nil)) \\ var & := & x_1|x_2|x_3|\ldots \end{array} \qquad (7)$$

The normalisation consists of the following steps:
1. remove qrev
2. stratify append
3. thin rev
4. left-assoc append

The approach can be also used for constructing procedures for computing derivations, differentials and other similar operations over elementary functions.

In a similar manner, we believe that it will be possible to construct a cut-elimination procedure for Gentzen's sequent calculus [8] or, at least, fragments (e.g., propositional fragment) of such a procedure (that is a nice example of plausible normalisation). That application of our system is the subject of our current investigation.

11 Realm of the Approach and Further Automation

In §5 and §7 we presented a range of method generators. In each of them, we use a method kind, input BNF and some set of rewrite rules, and used them

to generate a required method (with some output BNF). However, it would be fruitful if we could start with an input BNF and look at BNFs (and also methods) which could be obtained by subsets of the available rewrite rules. Thus, it is interesting to consider whether for a given BNF and a set of (terminating) rewrite rules we can always compute the output BNF. This general problem is the subject of our current research and there are techniques that cover many cases. However, the answer for the general case is negative, since, in a general case, the resulting set of expressions (obtained from the initial set of expressions by exhaustive rewriting) is not necessarily definable by a BNFs (i.e., by a context-free grammar).

Given a terminating set of unconditional rewrite rules R, let $R(x)$ denote the result of exhaustive application of rules from R to the expression x.

Example 19 *Let T be a* BNF *class defined in the following way:*

$$T \quad := \quad f(a, f(c, b)) | f(f(a, T), f(c, b))$$

Let R be a set consisting of the following rewrite rules:

$$\begin{aligned} r_1: &\quad f(x, f(y, z)) \longrightarrow f(f(x, y), z) \\ r_2: &\quad f(f(x, c), b) \longrightarrow f(f(x, b), c). \end{aligned}$$

The above set of rewrite rules is, obviously, terminating. Let L' be the set of normal forms of instances of T under the exhaustive application of the set R. L' is not context-free.

If t is derived from T, then t was generated by applying the second production rule m times ($m \geq 0$) and then the first production rule once. It can be easily proved that t has $3m + 2$ occurrences of the symbol f and $m + 1$ occurrences of each of symbols a, b and c. Also, all occurrences of the symbols a precede all occurrences of symbols b and c.

The exhaustive application of r_1 transforms t into t', while t' is in left-associative normal form (while all occurrences of the symbols a still precede all occurrences of the symbols b and c). So, the term t' is of the form:

$$\underbrace{f(f(f(\ldots(f}_{3m+2}\underbrace{(a, a), a), \ldots, a)}_{m+1}, \underbrace{c), b), c), b), \ldots, c), b)}_{2m+2}$$

The exhaustive application of r_2 "moves" all occurrences of the symbol b leftwards, "through" occurrences of the symbol c and the term $\hat{t}$ is of the form:

$$\underbrace{f(f(f(\ldots(f}_{3m+2}\underbrace{(a, a), a), \ldots, a)}_{m+1}, \underbrace{b), b), \ldots), b)}_{m+1}, \underbrace{c), c) \ldots, c)}_{m+1}$$

However, the set of such terms is not context-free (which can be easily proved by the pumping lemma *for context-free languages [7]).*

The above example shows that given a BNF and a terminating set of rewrite rules, the resulting (normalised) language might not be definable by a BNF.

Hence, since the output is not always definable by a BNF, there is no algorithm which always construct the resulting BNF.

Even if there is an algorithm that (given a BNF and a terminating set of rewrite rules) constructs an output BNF *whenever it is possible*, it would still not ensure further automation of our programme. Indeed, even if we have such an algorithm it might still not be of operational use. For instance, if want to synthesize a decision procedure, we would generate a sequence of BNFs looking for a trivial one (consisting of only $\top$ and $\bot$) and, so, we would need to check whether two BNFs give the same language, but that problem is undecidable in general. Therefore, it is likely that we cannot have a complete procedure for synthesizing decision procedures. On the other hand, extensions of our system along the discussed lines, might improve our current heuristic solutions and hence are the subject of our future research.

As said above, there is no system that can always produce the output BNF (since at some situations there is no one). So, the proposed system cannot be extended by some new additional method generators in order to become complete. Of course, the realm of the approach extends as the number of method generators extends. The current set of method generators defines the current realm of the proposed system. The system is heuristic, so it is difficult to make a formal characterisation of its realm. It is likely and expected that it can synthesize procedures that are based on rewriting and that consist of methods of the available kinds.

In synthesizing decision procedures, the approach does not distinguish if the theory is a combination of some simpler theories or not. Thus, the problem of combining decision procedures (for combination of decidable theories) is not addressed: if there are synthesized decision procedures for component theories, these cannot be combined. A decision procedure for a combination theory can be synthesized only if it as a whole can be described in terms of normalisation methods.

12 Related Work

Our approach is based on ideas from [5] and apart from that strong link, as we are aware of, it can be considered basically original.

The work presented is related to Knuth-Bendix completion procedure [12] and its variants in a sense that it performs automatic construction of decision procedures. However, there are significant differences. While the completion procedure produces a confluent and terminating set of rewrite rules, and hence a way how to reach a normal form, it does not give a description of the whole of the normal forms. In contrast, our system does not necessarily produce a decision procedure (or a normalisation procedure) whenever the completion procedure does, but when it produces a decision procedure, it also provides the finite description of the output (normalised) language (something that the completion procedure does not give). In addition, the completion procedure produces procedures that are based on exhaustive applications of rewrite rules,

while our system produces procedures that are based successive rule sets and, can generate a procedure for conjunctive normal form, which cannot be done by a single rule set. Also, a procedure divided into a number of steps is more easy understandable to a human then a single step procedure. We believe that it would be worthwhile to combine our work with the Knuth-Bendix completion procedure in the following way: the completion procedure can be used to find a confluent and terminating rule set and then use ADEPTUS to describe the normal form it produces; on that basis one can build methods with before and after BNFs as the preconditions and post-conditions of a new method describing the rewrite rule set.

As it addresses the automatic construction of decision procedures our work is also related to work presented in [1]. That work is aimed at deriving decision procedures using superposition-based inference system for clausal equational logic. The approach is an alternative for congruence closure algorithm and for the Knuth-Bendix completion procedure. It can handle not only pure equality, but also some other axiomatic equational theories such as theories of lists, arrays and extensional arrays.

Our work is related to [10] which performs automatic learning of proof methods. The system LEARNΩMATIC described in [10] learns proof methods (e.g. decision procedures) from proof traces obtained by brute force application of available primitive methods. LEARNΩMATIC can help in building efficient implementation of proof procedures, however does not give simple opportunities for proving termination or completeness of learnt methods. On the other hand, within the system presented in this paper, the correctness of synthesized procedures is obtained automatically because they are composed only of the application of sound rules.

Automatic generation of decision procedures (especially for user-defined theories) can be very fruitful for using decision procedures in theorem proving. For instance, we believe that the system presented in this paper fit nicely as an additional module to the system presented in [11].

13 Future Work

For future work we are planning the following lines of research:

- we will be looking for other suitable domains for our techniques (for instance, we will look at other quantifier elimination procedures and at automation of constructing *solvers* required in Shostak's schemes for using decision procedures [15, 14]);
- we will try to further automate our system (along the lines discussed in §11);
- we will try to extend the set of our method generators and search engines and will try to further improve their efficiency;

- we will implement mechanisms which generate not only necessary methods, but also the corresponding tactics;
- we will try to automate the process of proving completeness (i.e., whether conditions in the rewrite rules used cover all possible cases); we will try to do it whenever possible by using the "self-reflection" principle mentioned in §9.
- we will apply the proposed approach to other problems (e.g., cut-elimination in sequent calculus);
- we will combine our system with Knuth-Bendix completion procedure (as discussed in §12).
- the realm of tactics is the theory object level. Methods are meta-level representation of tactics. Their object level is a meta-level of the theory we deal with. Method generators are meta-level representation of methods. They also have preconditions, postconditions, and transformations (while one of the side-effects is a generated method). They can be seen as meta-methods. Their object level is a meta-meta-level of the theory we deal with. Our search engine works with method generators and its object level is the method generators level, or a meta-meta-meta-level of the theory we deal with. We will try to automatically generate some method generators (similarly as methods). Our goal would be to consider all these layers in a uniform way.

14 Conclusions

In this paper we presented a system (ADEPTUS) for synthesising decision procedures. It is based on ideas from [5].

ADEPTUS consists of several method generators and mechanisms for searching over them and combining them. We have implemented the system and used it for generating decision procedures for ground arithmetic and for linear arithmetic. These procedures ensure correct implementation which is not quite easy for a human programmer to achieve (as these procedures consist of dozens of steps). We believe that our approach can be used in other domains as well and can lead to automation of some routine steps in different types of programming tasks. Moreover, the presented approach is such that it provides a framework for easy proving of termination, soundness and completeness of generated procedures. Also, the approach can give a deeper insight into the nature of some decision procedures.

References

[1] Armando, A., S. Ranise, and M. Rusinowitch: 2001, 'Uniform Derivation of Decision Procedures by Superposition'. In: *Computer Science Logic,*

15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings, Vol. 2142 of *LNCS*. pp. 513–527.

[2] Boyer, R. S. and J. S. Moore: 1988, 'Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic'. In: J. E. Hayes, J. Richards, and D. Michie (eds.): *Machine Intelligence 11*. pp. 83–124.

[3] Bundy, A.: 1988, 'The Use of Explicit Plans to Guide Inductive Proofs'. In: R. Lusk and R. Overbeek (eds.): *9th Conference on Automated Deduction*. pp. 111–120. Longer version available from Edinburgh as DAI Research Paper No. 349.

[4] Bundy, A.: 1991a, 'A Science of Reasoning'. In: J.-L. Lassez and G. Plotkin (eds.): *Computational Logic: Essays in Honor of Alan Robinson*. pp. 178–198. Also available from Edinburgh as DAI Research Paper 445.

[5] Bundy, A.: 1991b, 'The Use of Proof Plans for Normalization'. In: R. S. Boyer (ed.): *Essays in Honor of Woody Bledsoe*. pp. 149–166. Also available from Edinburgh as DAI Research Paper No. 513.

[6] Cooper, D. C.: 1972, 'Theorem Proving in Arithmetic Without Multiplication'. In: B. Meltzer and D. Michie (eds.): *Machine Intelligence 7*. pp. 91–99.

[7] Davis, M., R. Sigal, and E. Weyuker: 1994, *Computability, Complexity, and Languages (Fundamentals of Theoretical Computer Science)*. Morgan Kaufmann/Academic Press.

[8] Gentzen, G.: 1935, 'Untersuchungen über das logische Schliessen, I, II'. *Mathematische Zeitschrift* **39**, 176–210, 405–431. English translation in "The Collected Papers of Gerhard Gentzen", North-Holland Publ.Co, 1969.

[9] Hodes, L.: 1971, 'Solving Problems by Formula Manipulation in Logic and Linear Inequalities'. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. Imperial College, London, England.

[10] Jamnik, M., M. Kerber, M. Pollet, and C. Benzmuller: 2002, 'Automatic Learning of Proof Methods in Proof Planning'. *Submitted to* Journal of Artificial Intelligence. Also available as Technical Report CSRP-02-5, School of Computer Science, University of Birmingham.

[11] Janičić, P. and A. Bundy: 2002, 'A General Setting for the Flexible Combining and Augmenting Decision Procedures'. *Journal of Automated Reasoning* **28**(3), 257–305.

[12] Knuth, D. E. and P. B. Bendix: 1970, 'Simple word problems in universal algebra'. In: J. Leech (ed.): *Computational problems in abstract algebra*. Pergamon Press, pp. 263–297.

[13] Lassez, J.-L. and M. Maher: 1992, 'On Fourier's algorithm for linear arithmetic constraints'. *Journal of Automated Reasoning* **9**, 373–379.

[14] Shostak, R. E.: 1979, 'A practical decision procedure for arithmetic with function symbols'. *JACM* **26**(2), 351–360.

[15] Shostak, R. E.: 1984, 'Deciding combinations of theories'. *Journal of the ACM* **31**(1), 1–12. Also: *Proceedings of the 6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pp. 209–222. Springer-Verlag, June 1982.

[16] Socher-Ambosius, R.: 1991, 'Boolean algebra admits no convergent rewriting system'. In: *Proceedings of the 4th International Conference on rewriting techniques and applications*, Vol. 488 of *LNCS*.

A Appendix: List of Rewrite Rules

```
rewrite_rule(la, 'rm_univ', (V:Sort##F), ~(V:Sort\\(~F)),
    [quantifier_free(F)], two_way).
rewrite_rule(la, 'rm_redundant', (V:_Sort\\F), F,
    [not(occurs(V,F))], two_way).

rewrite_rule(la, 'rm_impl', (F1=>F2), (~F1\F2), [], two_way).
rewrite_rule(la, 'rm_equiv', (F1<=>F2), (F1=>F2)#(F2=>F1), [], two_way).
rewrite_rule(la, 'rm_gr', (T1>T2), (T2<T1), [], two_way).
rewrite_rule(la, 'rm_leq', leq(T1,T2), (T1<T2)\(T1=T2), [], two_way).
rewrite_rule(la, 'rm_qeq', geq(T1,T2), (T2<T1)\(T1=T2), [], two_way).
rewrite_rule(la, 'rm_neq', neq(T1,T2), (T1<T2)\(T2<T1), [], two_way).
rewrite_rule(la, 'rm_neg_eq', ~(T1=T2), (T1<T2)\(T2<T1), [], two_way).
rewrite_rule(la, 'rm_neg_less', ~(T1<T2), (T2<T1)\(T1=T2), [], two_way).
rewrite_rule(la, 'rm_top', ~(true), false, [], two_way).
rewrite_rule(la, 'rm_bottom', ~(false), true, [], two_way).
rewrite_rule(la, 'rm_minus', -T, (-1)*T, [], two_way).
rewrite_rule(la, 'rm_eq1', C1=C2, true,
    [number(C1),number(C2),(C1=C2)], two_way).
rewrite_rule(la, 'rm_eq2', C1=C2, false,
    [number(C1),number(C2),not(C1=C2)], two_way).
rewrite_rule(la, 'rm_ls1', C1<C2, true,
    [number(C1),number(C2),(C1<C2)], two_way).
rewrite_rule(la, 'rm_ls2', C1<C2, false,
    [number(C1),number(C2),not(C1<C2)], two_way).
rewrite_rule(la, 'rm_mult', C1*C2, C3,
    [number(C1),number(C2),(C3 is C1*C2)], two_way).

rewrite_rule(la, 'st_neg_univ', (~(V##F)), (V\\(~F)),
    [], two_way).
rewrite_rule(la, 'st_neg_exi', (~(V\\F)), (V##(~F)),
    [], two_way).
rewrite_rule(la, 'st_conj_univ1', (F1#(V##F2)), (V##(F1#F2)),
    [], two_way).
rewrite_rule(la, 'st_conj_univ2', ((V##F2)#F1), (V##(F2#F1)),
    [], two_way).
rewrite_rule(la, 'st_conj_exi1', (F1#(V\\F2)), (V\\(F1#F2)),
    [], two_way).
```

```
rewrite_rule(la, 'st_conj_exi2', ((V\\F2)#F1), (V\\(F2#F1)),
    [], two_way).
rewrite_rule(la, 'st_disj_univ1', (F1\(V##F2)), (V##(F1\F2)),
    [], two_way).
rewrite_rule(la, 'st_disj_univ2', ((V##F2)\F1), (V##(F2\F1)),
    [], two_way).
rewrite_rule(la, 'st_disj_exi1', (F1\(V\\F2)), (V\\(F1\F2)),
    [], two_way).
rewrite_rule(la, 'st_disj_exi2', ((V\\F2)\F1), (V\\(F2\F1)),
    [], two_way).
rewrite_rule(la, 'st_neg_conj', (~(F1#F2)), ((~F1)\ (~F2)),
    [], two_way).
rewrite_rule(la, 'st_neg_disj', (~(F1\ F2)), ((~F1)# (~F2)),
    [], two_way).
rewrite_rule(la, 'st_conj_disj1', (F1#(F2\F3)), (F1#F2)\ (F1#F3),
    [], two_way).
rewrite_rule(la, 'st_conj_disj2', ((F2\F3)#F1), (F2#F1)\ (F3#F1),
    [], two_way).
rewrite_rule(la, 'st_mult_plus1', (T1*(T2+T3)), (T1*T2)+(T1*T3),
    [], two_way).
rewrite_rule(la, 'st_mult_plus2', ((T2+T3)*T1), (T2*T1)+(T3*T1),
    [], two_way).

rewrite_rule(la, 'thin_neg', (~(~(F))), F, [], two_way).

rewrite_rule(la, 'left_assoc_conj', (F1#(F2#F3)), ((F1#F2)#F3),
    [], two_way).
rewrite_rule(la, 'left_assoc_disj', (F1\(F2\F3)), ((F1\F2)\F3),
    [], two_way).
rewrite_rule(la, 'left_assoc_plus', (T1+(T2+T3)), ((T1+T2)+T3),
    [], two_way).
rewrite_rule(la, 'left_assoc_mult', (T1*(T2*T3)), ((T1*T2)*T3),
    [], two_way).

rewrite_rule(la, 'reduce_plus', (T1+T2), T3,
    [number(T1),number(T2),T3 is T1+T2 ], two_way).

rewrite_rule(la, 'absorbe_mult', (C1*(C2*T)), C3*T,
    [number(C1),number(C2),C3 is C1*C2 ], two_way).

rewrite_rule(la, 'reorder_plus1', (C*V+T1),(T1+C*V),
    [variable(V),number(T1)], two_way).
rewrite_rule(la, 'reorder_plus2', (V+T1),(T1+V),
    [variable(V),number(T1)], two_way).
rewrite_rule(la, 'reorder_plus3', ((T+(C*V))+T1), ((T+T1)+(C*V)),
    [variable(V),number(T1)], two_way).
rewrite_rule(la, 'reorder_plus4', ((T+(V))+T1), ((T+T1)+(V)),
    [variable(V),number(T1)], two_way).
rewrite_rule(la, 'reorder_plus5', (x0+V),(V+x0),
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus6', (x0+C*V), (C*V+x0),
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus7', (C*x0+V),(V+C*x0),
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus8', (C0*x0+C*V), (C*V+C0*x0),
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus9', (T+x0)+V, (T+V)+x0,
```

```
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus10', (T+x0)+C*V, (T+C*V)+x0,
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus11', (T+C*x0)+V, (T+V)+C*x0,
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus12', (T+C0*x0)+C*V, (T+C*V)+C0*x0,
    [variable(V),not(V=x0)], two_way).
rewrite_rule(la, 'reorder_plus13', (T+x0)+x0, T+C*x0,
    [number(C), C is 2], two_way).
rewrite_rule(la, 'reorder_plus14', (T+x0)+C*x0, T+C1*x0,
    [number(C1),C1 is C+1], two_way).
rewrite_rule(la, 'reorder_plus15', (T+C*x0)+x0, T+C1*x0,
    [number(C1),C1 is C+1], two_way).
rewrite_rule(la, 'reorder_plus16', (T+C0*x0)+C1*x0, T+C*x0,
    [number(C1),C is C0+C1], two_way).

rewrite_rule(la, 'reduce_bool1', (F1#true), (F1), [], two_way).
rewrite_rule(la, 'reduce_bool2', (_F1\true), (true), [], two_way).
rewrite_rule(la, 'reduce_bool3', (_F#false), false, [], two_way).
rewrite_rule(la, 'reduce_bool4', (F1\false), (F1), [], two_way).

rewrite_rule(la, 'collect1', x0+x0, C*x0,
    [C is 2], two_way).
rewrite_rule(la, 'collect2', x0+C*x0, C1*x0,
    [number(C1),C1 is C+1], two_way).
rewrite_rule(la, 'collect3', C*x0+x0, C1*x0,
    [number(C1),C1 is C+1], two_way).
rewrite_rule(la, 'collect4', C0*x0+C1*x0, C*x0,
    [number(C1),C is C0+C1], two_way).

rewrite_rule(la, 'one_side1', (T1=T2), (T1+(-1)*T2=0),
    [not(T2=0)], two_way).
rewrite_rule(la, 'one_side2', (T1<T2), (T1+(-1)*T2<0),
    [not(T2=0)], two_way).

rewrite_rule(la, 'isolate1', T+C*x0=0, C*x0= (-(T)),
    [number(C)], two_way).
rewrite_rule(la, 'isolate2', T+x0=0, x0=(-(T)),
    [], two_way).
rewrite_rule(la, 'isolate3', T+C*x0<0, C*x0<(-(T)),
    [number(C)], two_way).
rewrite_rule(la, 'isolate4', T+x0<0, x0<(-(T)),
    [], two_way).
```

B Appendix: Generated Procedure for Linear Arithmetic

```
1 remove <=>; Target class: f;
   Rules:[[f, f<=>f, rm_equiv]];
   Input size: 2233 Output size : 2004 (*)
2 remove =>; Target class: f;
   Rules:[[f, f=>f, rm_impl]];
   Input size: 2004 Output size : 1803 (*)
3 remove leq; Target class: f;
   Rules:[[f, leq(t, t), rm_leq]];
```

```
   Input size: 1803 Output size : 1782 (*)
4 remove geq; Target class: f;
   Rules:[[f, geq(t, t), rm_qeq]];
   Input size: 1782 Output size : 1761 (*)
5 remove neq; Target class: f;
   Rules:[[f, neq(t, t), rm_neq]];
   Input size: 1761 Output size : 1740 (*)
6 remove >; Target class: f;
   Rules:[[f, t>t, rm_gr]];
   Input size: 1740 Output size : 1719 (*)
7 remove -; Target class: t;
   Rules:[[t, -t, rm_minus]];
   Input size: 1719 Output size : 1618 (*)
8 remove -; Target class: re;
   Rules:[[re, -re, rm_minus]];
   Input size: 1618 Output size : 1517 (*)
9 stratify [##, \\]; Target class: f;
   Rules:[[f, f#var:rational##f, st_conj_univ1],
          [f, f#var:rational\\f, st_conj_exi1],
          [f, var:rational##f#f, st_conj_univ2],
          [f, var:rational\\f#f, st_conj_exi2],
          [f, f\var:rational##f, st_disj_univ1],
          [f, f\var:rational\\f, st_disj_exi1],
          [f, var:rational##f\f, st_disj_univ2],
          [f, var:rational\\f\f, st_disj_exi2],
          [f, ~var:rational##f, st_neg_univ],
          [f, ~var:rational\\f, st_neg_exi]];
   Input size: 1517 Output size : 1527
---------------------------------------------------------------------------
10 adjust_innermost x0; Target class: f;
   Rules:[[f, x0:rational##f1, rm_univ]];
   Input size: 1527 Output size : 1547
11 stratify [#, \]; Target class: f1;
   Rules:[[f1, ~(f1#f1), st_neg_conj], [f1, ~(f1\f1), st_neg_disj]];
   Input size: 1553 Output size : 1557
12 thin ~; Target class: f11;
   Rules:[[f11, ~(~f11), thin_neg]];
   Input size: 1557 Output size : 1494 (*)
13 remove ~; Target class: f1;
   Rules:[[f1, ~false, rm_bottom], [f1, ~true, rm_top],
          [f1, ~t<t, rm_neg_less], [f1, ~t=t, rm_neg_eq]];
   Input size: 1504 Output size : 1446 (*)
14 one_side [0, [<, >, leq, neq, geq, =]]; Target class: f1;
   Rules:[[f1, t<t, one_side2], [f1, t=t, one_side1]];
   Input size: 1446 Output size : 1428 (*)
15 stratify [\]; Target class: f1;
   Rules:[[f1, f1#f1\f1, st_conj_disj1],
          [f1, (f1\f1)#f1, st_conj_disj2]];
   Input size: 1428 Output size : 1438
16 stratify [+]; Target class: t;
   Rules:[[t, re*(t+t), st_mult_plus1]];
   Input size: 1438 Output size : 1448
17 stratify [+]; Target class: re;
   Rules:[[re, re*(re+re), st_mult_plus1],
          [re, (re+re)*re, st_mult_plus2]];
   Input size: 1448 Output size : 1458
18 left_assoc \; Target class: f1;
```

```
   Rules:[[f1, f1\f1, left_assoc_disj]];
   Input size: 1458 Output size : 1368 (*)
19 left_assoc #; Target class: f11;
   Rules:[[f11, f11#f11, left_assoc_conj]];
   Input size: 1368 Output size : 1597
20 stratify [<, =]; Target class: f11;
   Rules:[[f11, (f11#t<0)#false, reduce_bool3],
          [f11, (f11#t<0)#true, reduce_bool1],
          [f11, (f11#t=0)#false, reduce_bool3],
          [f11, (f11#t=0)#true, reduce_bool1],
          [f11, t<0#false, reduce_bool3],
          [f11, t<0#true, reduce_bool1],
          [f11, t=0#false, reduce_bool3],
          [f11, t=0#true, reduce_bool1]];
   Input size: 1597 Output size : 1607
21 remove #; Target class: f111;
   Rules:[[f111, f111#false, reduce_bool3],
          [f111, f111#true, reduce_bool1]];
   Input size: 1607 Output size : 1393
22 left_assoc +; Target class: re;
   Rules:[[re, re+re, left_assoc_plus]];
   Input size: 1403 Output size : 1303 (*)
23 left_assoc +; Target class: t;
   Rules:[[t, t+t, left_assoc_plus]];
   Input size: 1303 Output size : 1213 (*)
24 left_assoc *; Target class: re1;
   Rules:[[re1, re1*re1, left_assoc_mult]];
   Input size: 1213 Output size : 1123 (*)
25 absorbe *; Target class: re1;
   Rules:[[re1, rc*rc, rm_mult]];
   Input size: 1123 Output size : 1002 (*)
26 absorbe +; Target class: re;
   Rules:[[re, rc+rc, reduce_plus]];
   Input size: 1012 Output size : 881 (*)
27 absorbe *; Target class: t1;
   Rules:[[t1, rc*(rc*t1), absorbe_mult], [t1, rc*rc, rm_mult]];
   Input size: 891 Output size : 1241
28 stratify [x0]; Target class: t;
   Rules:[[t, x0+rc, reorder_plus2],
          [t, x0+var, reorder_plus5],
          [t, x0+rc*var, reorder_plus6],
          [t, rc*x0+rc, reorder_plus1],
          [t, rc*x0+var, reorder_plus7],
          [t, rc*x0+rc*var, reorder_plus8],
          [t, t+x0+rc, reorder_plus4],
          [t, t+x0+var, reorder_plus9],
          [t, t+x0+rc*var, reorder_plus10],
          [t, t+rc*x0+rc, reorder_plus3],
          [t, t+rc*x0+var, reorder_plus11],
          [t, t+rc*x0+rc*var, reorder_plus12]];
   Input size: 803 Output size : 1251
29 absorbe +; Target class: t;
   Rules:[[t, t+x0+x0, reorder_plus13],
          [t, t+rc*x0+x0, reorder_plus15],
          [t, x0+x0, collect1],
          [t, rc*x0+x0, collect3],
          [t, t+x0+rc*x0, reorder_plus14],
```

```
       [t, t+rc*x0+rc*x0, reorder_plus16],
       [t, x0+rc*x0, collect2],
       [t, rc*x0+rc*x0, collect4]];
   Input size: 1251 Output size : 2045
30 isolate [[x0, rc*x0], [<, >, leq, neq, geq, =]]; Target class: f11;
   Rules:[[f11, f11#t1+x0<0, isolate4],
       [f11, f11#t1+rc*x0<0, isolate3],
       [f11, f11#t1+x0=0, isolate2],
       [f11, f11#t1+rc*x0=0, isolate1],
       [f11, t1+x0<0, isolate4],
       [f11, t1+rc*x0<0, isolate3],
       [f11, t1+x0=0, isolate2],
       [f11, t1+rc*x0=0, isolate1]];
   Input size: 1071 Output size : 2037
31 eliminate [[x0, rc*x0], [<, >, leq, neq, geq, =]]; Target class: f;
   Rules:[[f, x0:_G18107\\_G18104, rm_redundant]];
   Input size: 2037 Output size : 1127
---------------------------------------------------------------------
32 stratify [#, \]; Target class: f1;
   Rules:[[f1, ~(f1#f1), st_neg_conj], [f1, ~(f1\f1), st_neg_disj]];
   Input size: 1312 Output size : 1312 (*)
33 thin ~; Target class: f11;
   Rules:[[f11, ~(~f11), thin_neg]];
   Input size: 1312 Output size : 1249 (*)
34 remove ~; Target class: f1;
   Rules:[[f1, ~false, rm_bottom], [f1, ~true, rm_top],
       [f1, ~t<t, rm_neg_less], [f1, ~t=t, rm_neg_eq]];
   Input size: 1259 Output size : 1201 (*)
35 stratify [\]; Target class: f1;
   Rules:[[f1, f1#f1\f1, st_conj_disj1],
       [f1, (f1\f1)#f1, st_conj_disj2]];
   Input size: 1201 Output size : 1211
36 stratify [+]; Target class: t;
   Rules:[[t, re*(t+t), st_mult_plus1]];
   Input size: 1211 Output size : 1221
37 left_assoc \; Target class: f1;
   Rules:[[f1, f1\f1, left_assoc_disj]];
   Input size: 1221 Output size : 1131 (*)
38 stratify [+]; Target class: re;
   Rules:[[re, re*(re+re), st_mult_plus1],
       [re, (re+re)*re, st_mult_plus2]];
   Input size: 1131 Output size : 1141
39 left_assoc +; Target class: re;
   Rules:[[re, re+re, left_assoc_plus]];
   Input size: 1141 Output size : 1051 (*)
40 left_assoc +; Target class: t;
   Rules:[[t, t+t, left_assoc_plus]];
   Input size: 1051 Output size : 961 (*)
41 left_assoc *; Target class: re1;
   Rules:[[re1, re1*re1, left_assoc_mult]];
   Input size: 961 Output size : 871 (*)
42 absorbe *; Target class: re1;
   Rules:[[re1, rc*rc, rm_mult]];
   Input size: 871 Output size : 750 (*)
43 absorbe +; Target class: re;
   Rules:[[re, rc+rc, reduce_plus]];
   Input size: 760 Output size : 629 (*)
```

```
44 absorbe *; Target class: t1;
   Rules:[[t1, rc*rc, rm_mult]];
   Input size: 639 Output size : 508 (*)
45 absorbe +; Target class: t;
   Rules:[[t, rc+rc, reduce_plus]];
   Input size: 518 Output size : 387 (*)
46 remove <; Target class: f11;
   Rules:[[f11, rc<rc, rm_ls1], [f11, rc<rc, rm_ls2]];
   Input size: 397 Output size : 366 (*)
47 remove =; Target class: f11;
   Rules:[[f11, rc=rc, rm_eq1], [f11, rc=rc, rm_eq2]];
   Input size: 366 Output size : 345 (*)
48 left_assoc #; Target class: f11;
   Rules:[[f11, f11#f11, left_assoc_conj]];
   Input size: 345 Output size : 348
49 remove #; Target class: f11;
   Rules:[[f11, f11#false, reduce_bool3],
          [f11, f11#true, reduce_bool1]];
   Input size: 348 Output size : 227 (*)
50 remove \; Target class: f1;
   Rules:[[f1, f1\false, reduce_bool4],
          [f1, f1\true, reduce_bool2]];
   Input size: 144 Output size : 6 (*)
51 remove ~; Target class: f;
   Rules:[[f, ~false, rm_bottom], [f, ~true, rm_top]];
   Input size: 23 Output size : 2
```

C Appendix: Example of a Linear Arithmetic Solved Formula

```
1 x:rational##(x>0)
2 x:rational##(x>0)
3 x:rational##(x>0)
4 x:rational##(x>0)
5 x:rational##(x>0)
6 x:rational##(0<x)
7 x:rational##(0<x)
8 x:rational##(0<x)
9 x:rational##(0<x)
10 ~x0:rational\\(~0<x0)
11 ~x0:rational\\(~0<x0)
12 ~x0:rational\\(~0<x0)
13 ~x0:rational\\(x0<0\0=x0)
14 ~x0:rational\\(x0<0\0+-1*x0=0)
15 ~x0:rational\\(x0<0\0+-1*x0=0)
16 ~x0:rational\\(x0<0\0+-1*x0=0)
17 ~x0:rational\\(x0<0\0+-1*x0=0)
18 ~x0:rational\\(x0<0\0+-1*x0=0)
19 ~x0:rational\\(x0<0\0+-1*x0=0)
20 ~x0:rational\\(x0<0\0+-1*x0=0)
21 ~x0:rational\\(x0<0\0+-1*x0=0)
22 ~x0:rational\\(x0<0\0+-1*x0=0)
23 ~x0:rational\\(x0<0\0+-1*x0=0)
24 ~x0:rational\\(x0<0\0+-1*x0=0)
25 ~x0:rational\\(x0<0\0+-1*x0=0)
```

```
26 ~x0:rational\\(x0<0\0+-1*x0=0)
27 ~x0:rational\\(x0<0\0+-1*x0=0)
28 ~x0:rational\\(x0<0\0+-1*x0=0)
29 ~x0:rational\\(x0<0\0+-1*x0=0)
30 ~x0:rational\\(x0<0\-1*x0= -0)
31 ~(0*1<1*1\-1*0=-1*0)
32 ~0*1<1*1# ~-1*0=-1*0
33 ~0*1<1*1# ~-1*0=-1*0
34 (1*1<0*1\0*1=1*1)#-1*0<-1*0\-1*0<-1*0
35 ((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    (1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
36 ((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    (1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
37 (((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
38 (((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
39 (((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
40 (((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
41 (((1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0)\
    1*1<0*1#-1*0<-1*0)\0*1=1*1#-1*0<-1*0
42 (((1<0#0<0)\0=1#0<0)\1<0#0<0)\0=1#0<0
43 (((1<0#0<0)\0=1#0<0)\1<0#0<0)\0=1#0<0
44 (((1<0#0<0)\0=1#0<0)\1<0#0<0)\0=1#0<0
45 (((1<0#0<0)\0=1#0<0)\1<0#0<0)\0=1#0<0
46 (((false#false)\0=1#false)\false#false)\0=1#false
47 (((false#false)\false#false)\false#false)\false#false
48 (((false#false)\false#false)\false#false)\false#false
49 ((false\false)\false)\false
50 false
51 false
```