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

Predrag Janičić[1] and Alan Bundy[2]

[1] Faculty of Mathematics, University of Belgrade
Studentski trg 16, 11000 Belgrade, Serbia — email: janicic@matf.bg.ac.yu
[2] School of Informatics, University of Edinburgh
Appleton Tower, Crichton St, Edinburgh EH8 9LE, UK — email: A.Bundy@ed.ac.uk

Abstract. We address the problem of automatic synthesis of decision procedures. We evaluate our ideas on ground arithmetic and 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. 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, in only a few seconds of CPU time. 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, 6]. 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. In addition, it is very important to have decision procedures for new, user-defined theories. The implementation of decision procedures should be such that it can be verified in some formal way. For all these reasons, it would be fruitful if the process (or, at least, all its routine steps) of synthesizing and implementing decision procedures can be automated. It would help avoiding human mistakes in implementing decision procedures. Since many steps in different decision procedures can be described via rewriting, object level proofs could often be also relatively easily derived.

In this paper we follow ideas from the programme on proof plans for normalisations from [4]. As discussed there, many steps of many decision procedures can be described via sets of rewrite rules. This observation prepares the way for automatic generation of decision procedures (given the necessary rewrite rules), which is vital for user defined theories. Following and extending the ideas from [4], we have developed a system ADEPTUS (coming from *Assembly of DEcision Procedures via TransmUtation and Synthesis*[1]) capable of automatically synthe-

* First author supported by EPSRC grant GR/R52954/01 and Serbian Ministry of Science grant 144030. Second author supported in part by EPSRC grant GR/S01771.

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

sizing normalisation procedures and decision procedures.[2] All the methods that ADEPTUS generates are built in the spirit of the proof planning paradigm (and are implemented in PROLOG). For some theories, the approach presented 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 providing an easier implementation of decision procedures and gaining a deeper understanding of them.

In this paper we evaluate our techniques on ground arithmetic and linear arithmetic (over rationals). ADEPTUS synthesized the decision procedures for ground arithmetic in around 3 seconds, and a decision procedure for (quantified) linear arithmetic in around 5 seconds of CPU time.

2 Preliminaries

Decision procedure. 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).

Ground and linear arithmetic. Ground arithmetic is a fragment of arithmetic that does not involve variables. Linear arithmetic is a fragment of arithmetic that involves only addition (nx is treated as $x + \cdots + x$, where x appears n times). For both these theories, we assume that variables can range over rational numbers. The Fourier/Motzkin procedure [8] is one of the decision procedures for linear arithmetic.

Backus-Naur form. For describing syntactical classes, we use Backus-Naur form — BNF (equivalent to context-free grammars). We will assume that each BNF definition has attached its *top class*. The language of a BNF is a set of all expressions that can be derived from the top class. For representing some infinite syntactical classes, for convenience, we use some meta-level conditions. We define the relation *ec* (*element of class*) as follows: $ec(b, e, c)$ holds iff e is an element of the class c in the BNF definition b.

Rewrite rules. Unconditional rewrite rules are of the form: $RuleName : l \longrightarrow r$. Conditional rewrite rules are of the form: $RuleName : l \longrightarrow r \ if \ p_1, p_2, \ldots, p_n$, where p_1, p_2, ..., p_n are literals. These rewrite rules may rely on some properties specific to the underlying theory $\mathcal{T}$. The rules like $n_1x + n_2x \longrightarrow nx \ \ if \ \ n = n_1 + n_2$. (corresponding to linear arithmetic), we will have to use modulo the underlying theory. For a rule $RuleName : \ l \longrightarrow r \ if \ p_1, p_2, \ldots, p_n$, we say that it is *sound* w.r.t. theory $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula Φ and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ if $\mathcal{T}$, $p_1\varphi$, $p_2\varphi$, ..., $p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$, and we say that it is *complete* w.r.t. theory $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ only if $\mathcal{T}, p_1\varphi, p_2\varphi, \ldots, p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$.

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

[2] ADEPTUS is implemented in PROLOG as a stand-alone system. The code and the longer version of this paper are available from `www.matf.bg.ac.yu/~janicic`.

proof-planning system, a method constructs the proof plan and this plan is then used to guide the construction of the proof itself [3]. These plans are made up of tactics, which represent common patterns of reasoning. 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. 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.[3]

3 Proposed Programme

Our programme (slightly modified from the first version [4]) for automated synthesis of different kinds of normalisation methods and normalisation procedures can be decomposed into several parts:

- 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 any member of the first syntactical class into a member of the second syntactical class. The output syntactical class and the corresponding method can be automatically generated in a number of special cases. An algorithm which can generate such a method we call a *method generator*.
- There will be different kinds of methods, e.g., one for removing some function symbol, one for stratification, one for thinning etc. (see further text and [4] for explanation of these terms); for each of them, there is a method generator.
- Given several generated methods, it should be possible to combine them (automatically) into a compound method or, sometimes, into a decision procedure for some theory;
- Methods (and compound methods) should be designed in such a way that their soundness, completeness, and termination can be easily proved;
- Since some transformations (required for some procedures) are very complex, building methods may require human interaction and assistance.

Following the above programme, we implemented our system ADEPTUS capable of generating code for real-world decision procedures. We have implemented several method generators. They take a given BNF, transform it into another one, and build a method that can transform any formula that belongs to the first BNF into a formula that 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,

[3] Alteratively, instead of (active) postconditions, methods can have effects — conditions that are guaranteed to be true when the method succeeds.

and completeness can be easily proved. Apart from these method generators, we use also special-purpose method generators. For simplicity, in the rest of the paper we assume that, in formulae being transformed, variables are standardized apart, i.e., there are no two quantifiers with the same variable symbol.

4 Method Generators and Generated Methods

Normalisation Method Generators. Normalisation methods are methods based on exhaustive application of rewrite rules. Each normalization method has the following general form:

name: *methodname*;
input: f;
preconditions: $ec(b, f, top\ class)$ (where b is the input BNF definition);
transformation: transforms f to f' by exhaustive application of the set of rewrite rules (applying to positions that correspond to the attached syntactical classes);
output: f';
postconditions: $ec(b', f', top\ class)$ (where b' is the output BNF definition).

Example 1. Each formula belonging to the following class:
$f := af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f|(\exists var : sort)f|(\forall var : sort)f$
can be transformed by using the rewrite rule $f_1 \Leftrightarrow f_2 \longrightarrow (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$ and the resulting formula belongs to the following class f:
$f := af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|(\exists var : sort)f|(\forall var : sort)f$.

We have implemented generators for several kinds of methods:

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 1, the given BNF definition can be transformed to the corresponding BNF definition without the symbol $\Leftrightarrow$.

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. For instance, 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`: $(f_2 \vee f_3) \wedge f_1 \longrightarrow (f_2 \wedge f_1) \vee (f_3 \wedge f_1)$. Stratify methods can also serve for reordering elements within an expression, e.g., a stratify method can be used to *stratify* variable x and rearrange a polynomial in such a way that its summands involving x will be at the end of the expression.

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 rule $\neg\neg f \longrightarrow f$ in order to transform each formula belonging to $f := af|\neg f$ to a formula belonging to $f := af|\neg af$.

Absorb is a normalization method that can eliminate some recursion rules. For instance, we can use the rule `rm_mult`: $c_1 \cdot c_2 \longrightarrow c_3 \quad \text{if} \quad c_3 = c_1 \cdot c_2$ in order to transform each term belonging to $t := t \cdot rc | rc$ (where rc denotes rational constants) to a term belonging to $t := rc$.

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. For instance, we will need the left association of addition and the left association of conjunction.

A *normalisation method generator* is a procedure with the following input: *(i)* a BNF form b of the input expressions; *(ii)* a set of rewrite rules R; *(iii)* a kind t of the required method (e.g., *remove*). It generates a method $\mathcal{M}$ and a BNF form b' (of the output expressions).[4] By exhaustively applying the rules from R, $\mathcal{M}$ transforms any expression belonging to b to an expression belonging to b'.

Example 2. Consider the following class: $f := h(a)|h(b)|g_1(a)|g_2(b)$ where a and b are some classes and suppose there are the following rewrite rules available: $R_1 : h(x) \longrightarrow g_1(x)$, $R_2 : h(x) \longrightarrow g_2(x)$. These rules are sufficient for eliminating the symbol h and for transforming the above class into the class: $f := g_1(a)|g_2(b)$. However, it cannot be reached by arbitrary use of exhaustive applications of the given rewrite rules, but R_1 should be applied only to $h(a)$, and R_2 only to $h(b)$. The lesson is that we have to take care about which rule we use for each construction of syntactical classes. This (or analogous) information has to be built into the method we want to construct.

Method generators for most normalisation methods work, in a sense, in a uniform way: each of them first tries to eliminate non-recursive classes in the input BNF, then searches for "problematic" BNF entries and constructs the target output BNF set; then, a generic 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. Also, this search mechanism attaches rewrite rules to particular entries. If there are no required rewrite rules, a method generator reports it, so the user could try to provide missing rules (in a planned, advanced version, which is not part of the work presented in this paper, the method generator would speculate the remaining necessary rules and/or try to redefine/relax the output class).

Special-Purpose Method Generators. The first one of the following special-purpose generators 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 (although 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).

[4] In our system, the tactics are not implemented yet. So, our procedures produce meta-level proof plans, not the object level proofs.

Method Generator for Adjusting the Innermost Quantifier. This generator generates a method which transforms a formula in prenex normal form in the following way: if its innermost quantifier is existential, then keep it unchanged; if its innermost quantifier is universal, then rewrite the formula $(Qx_1)(Qx_2)\ldots(Qx_n)(\forall x)f$ to a formula $(Qx_1)(Qx_2)\ldots(Qx_n)\neg(\exists x)\neg f$ by using the following rewrite rule: `rm_univ`: $(\forall x)f \longrightarrow \neg(\exists x)\neg f$. The motive of this method is to deal only with elimination of existential quantifiers.

One-side Method Generator. This generator generates (if the input BNF admits that) a method which transforms all literals in such a way that each of them has 0 as its second argument. For instance, for symbols $<$, $>$, $\leq$, $\neq$, $\geq$, $=$ as parameters, after applying that generated method each literal will have one of the following forms: $t < 0$, $t > 0$, $t \leq 0$, $t \neq 0$, $t \geq 0$, $t = 0$.

Method Generator for Isolating a Variable. This generator generates (if the input BNF admits that) a method that isolates a distinguished variable x in all literals. After applying that method, each of the literals either does not involve x or has one of the forms: $\nu x = \gamma$, $x = \gamma$, $\lambda x < \alpha$, $x < \alpha$ (where ν, γ, λ, α have no occurrences of x).

Method Generator for Removing a Variable. The *cross multiply and add* step is the essential step of the Fourier/Motzkin's procedure [8]. It is applied for elimination of x from $\exists x F(x)$, where F is in disjunctive normal form and each of its literals either does not involve x or has one of the forms: $\nu x = \gamma$, $x = \gamma$, $\lambda x < \alpha$, $x < \alpha$ (where ν, γ, λ, α have no occurrences of x). After performing this step, x does not occur in the current formula and so the corresponding quantifier can be deleted.

Properties of Generated Methods. A normalisation method links two syntactical sets. 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. In semantical terms, 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, and if the "only if" condition holds then the method is complete (w.r.t. $\mathcal{T}$).

Termination. For each generated method it must be shown that it is terminating (by considering properties of the rewrite rules used[5]). 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 re-

[5] 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 [9].

quired syntactical restrictions (given by the method specification), so the method is sound w.r.t. syntactical restrictions.[6]
- 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 w.r.t. the underlying theory $\mathcal{T}$:
- It is not *a priori* guaranteed that a 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 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. $\mathcal{T}$ 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. $\mathcal{T}$.

5 Search Engine for Synthesizing Compound Methods

Given 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. Also, an automatic search for compound methods or a decision procedure for $\mathcal{T}$ can be performed. The goal of this process is to generate a sequence of methods such that: *(i)* the output BNF class of the non-final method is the input BNF class of the next method in the sequence; *(ii)* the output BNF class of the last method in the sequence is a goal BNF, for instance, a trivial BNF — consisting of entries $\top$ and $\bot$ for the top class. This sequence can have more methods that are instances of the same kind of methods, or even the very same method more than once. Our search procedure in each step invokes all available method generators, with all possible parameters (depending on the underlying language). 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 fixed value M, and such that the final BNFs of the subsequences are of strictly decreasing size. So, in the generated procedure there might be some BNF size increasing steps, but the whole of the procedure would be decreasing. The size of BNF definition is a heuristic measure and we define it to be the sum of sizes of all its entries; the size of the entry $c := c'$ is defined as follows: *(i)* each symbol c in c' adds 100; *(ii)* each symbol c'' in c' (where c'' is some other class) adds

[6] Conditional rules are the reason for using active postconditions in methods (instead of passive *effects*). Namely, in some situations no rewrite rule is applicable, but the processed formula is not yet in normal form, so the postcondition must be tested.

10; *(iii)* each other symbol in c' adds 1. Defined this way, the measure forces the engine to try to get rid of recursive classes and then of the classes that involve some other classes. The trivial, goal BNF class (consisting of only $f := \top|\bot$) has the size 2. If the current sequence cannot be continued, the search engine backtracks and tries to find alternatives.

Example 3. The size of the following BNF definition (for ground arithmetic):

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

is 1556. The size of BNF for the full linear arithmetic is 2233.

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 terminates. If method generators can generate all methods necessary for building the required compound method, then (thanks to backtracking) 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, so this iterated algorithm is *complete*. However, we can also use it only with particular values for M (then the procedure is not complete, but it gives better results if it used only for an appropriate value for M).

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 (based on empirical tests): `remove`, `thin`, `absorb`, `stratify`, `left_assoc`.

For theories for which normalisation methods cannot build a decision procedure, we use special-purpose method generators and the basic search engine in a more complex way. The search for a decision procedure based on quantifier elimination is performed in three stages, by the following *compound search engine*:

- the first stage is reaching a BNF for which the method for adjusting the innermost quantifier is applicable;
- the second stage continues while the variable elimination method is applicable (in each loop one variable is being eliminated); 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 basic search engine and we use all method generators with priority given to the special-purpose method generators.

Properties of Compound Methods. A set of generated methods for some underlying theory $\mathcal{T}$ can be combined (by a human, or 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 method, and the postconditions are the postconditions of the last method used.[7]

Termination. If a compound method is a sequence of terminating methods, then it is (trivially) terminating. If it has a loop, a 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 $\mathcal{T}$, a trivial output BNF set consisting only of $\top$ and $\bot$, and if it is terminating, sound, and complete (w.r.t. $\mathcal{T}$)[8], then it is a decision procedure for $\mathcal{T}$. This way, we can, in some cases, trivially get a proof that some (automatically generated) compound method is a decision procedure for some theory.

6 Evaluation

We ran the basic search engine, on the BNF definition given in Example 3, with $M = 3$, with the described method generators, and with 59 relevant rewrite rules available. 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 2.91 seconds of *cpu* time[9] 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": *remove* $\Leftrightarrow$, *remove* $\Rightarrow$, *remove* $\leq$, *remove* $\geq$, *remove* $\neq$, *remove* $>$, *remove* $-$, *stratify* $[\wedge, \vee]$, *thin* $\neg$, *stratify* $[\vee]$, *stratify* $[+]$, *left_assoc* $\vee$, *left_assoc* $+$, *left_assoc* $*$, *absorbe* $*$, *absorbe* $+$, *remove* $<$, *remove* $=$, *left_assoc* $\wedge$, *remove* $\wedge$, *remove* $\vee$.

[7] This way of constructing the preconditions and postconditions of a compound method is not adequate in general but suffices for the examples we were working on (recall that in compound methods that our system generates, the output BNF class of a method is always the input BNF class of the next method in the sequence).

[8] Soundness and completeness properties rely on properties of the rewrite rules used.

[9] The system is implemented in SWI Prolog and tested on a 512Mb PC 2.4Ghz.

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. The procedure `DP_GA` is sound and terminating, as all generated methods are sound and terminating and there is no loop. We still don't claim that it is complete as there are some conditional rewrite rules used. For instance, in the step *absorbe* + of `DP_GA`, the conditional rule `reduce_plus`: $t_1 + t_2 \Rightarrow t_3$, if $t_3 = t_1 + t_2$ is used, but it is still not shown that its condition covers all possible cases. The user can show this by proving: $(\forall c_1 : rational)(\forall c_2 : rational)(\exists c_3 : rational)(c_3 = c_1 + c_2)$. It is easy to prove that such conjectures are theorems of linear arithmetic. Moreover, some of them 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). All this leads us to conclude that the procedure `DP_GA` is correct.

We applied the compound search engine on the BNF description of the full linear arithmetic, with $M = 3$ for the first and the third stage, with $M = 5$ for the second stage[10], with all the described methods, and with 71 relevant rewrite rules available. The search algorithm took 4.80 seconds of *cpu* time and during the search there were 89 methods successfully generated, while there are 51 of them in the final sequence, yielding a decision procedure `DP_LA`.

Theorem 2 *The procedure* `DP_LA` *for linear arithmetic 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 not only on the completeness of the rewrite rules used, but also on the coverage property for the methods that use conditional rewrite rules. It is easy to see (similarly as for `DP_GA`) 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. 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 a much shorter way. This suggests that it is non-trivial for a human programmer to implement this procedure without flaws and bugs, even when provided with the code for the key step (*cross multiply and*

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

add), because the most probable flaws are rather in correctly combining all the remaining steps.

7 Related Work

Our approach is based on ideas from [4] and apart from that strong link, as we are aware of, it can be considered basically original.

The work presented here is related to the Knuth-Bendix completion procedure [7] 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 normal form. 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 procedure, it also provides the finite description of the output (normalised) language. In addition, the completion procedure produces procedures that are based on exhaustive applications of rewrite rules, while our system produces procedures that use subsets of rewrite rules in stages and gives structured proofs (easily understandable to a human). For instance, our system can generate a procedure for constructing conjunctive normal form, which cannot be done by a single rule set. 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 ADEPTUS can be used to describe the normal form it produces.

Our work is also related to work aimed at deriving decision procedures using superposition-based inference system for clausal equational logic [1]. That approach is an alternative to the congruence closure algorithm and to the Knuth-Bendix completion procedure. It does not use rewrite rules in a structured way, so it is not suitable for dealing with large sets of rules (as for linear arithmetic). Our work is also related to work that performs automatic learning of proof methods [5]. The system LEARNΩMATIC learns proof methods (including decision procedures) from proof traces obtained by brute force application of available primitive methods. This approach (unlike ours does not give opportunities for simple proofs of termination or completeness of learnt methods.

8 Realm of the Approach and Further Automation

In the presented method generators, we took 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 methods that can be obtained by subsets of the available rewrite rules. It is interesting to consider whether for a given BNF and a set of (terminating) rewrite rules we can always compute the output BNF. The answer for the general case is negative, since, in a general case, the resulting set of expressions is not necessarily definable by a BNFs. 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* (this is subject of our current research), it would still not ensure further automation of our programme in general case. Namely, 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, 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, we believe that our system can work well in many practical situations. Our system is heuristic, and its realm is determined by the set of method generators available (so it is difficult to make a formal characterisation of the realm).

In synthesizing decision procedures, the approach does not distinguish if the theory is a combination of some 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.

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

- we will be looking for other challenging domains for our techniques;
- 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 mentioned "self-reflection" principle;
- we will try to combine our system with Knuth-Bendix completion procedure.

9 Conclusions

We presented a system (ADEPTUS) for synthesising decision procedures. It is based on ideas from [4]. ADEPTUS consists of several method generators and mechanisms for searching over them and combining them. We have implemented the system and used it for automatically generating decision procedures (in PROLOG) for ground arithmetic and for linear arithmetic. These implementations are correct (and the user gets help in proving correctness, completeness and termination), which is not quite easy for a human programmer to achieve. 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. 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: 'Uniform Derivation of Decision Procedures by Superposition'. *CSL 2001*, Vol. 2142 of *LNCS*, Springer, 2001.
2. Boyer, R. S. and J. S. Moore: 'Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic'. *Machine Intelligence 11*. 1988,
3. Bundy, A.: 'The Use of Explicit Plans to Guide Inductive Proofs'. In: R. Lusk and R. Overbeek (eds.): *9th Conference on Automated Deduction*. 1988.
4. Bundy, A.: 'The Use of Proof Plans for Normalization'. In: R. S. Boyer (ed.): *Essays in Honor of Woody Bledsoe*, 1991.
5. Jamnik, M., M. Kerber, M. Pollet, and C. Benzmuller: 'Automatic Learning of Proof Methods in Proof Planning'. CSRP-02-5, University of Birmingham, 2002.
6. Janičić, P. and A. Bundy: 'A General Setting for the Flexible Combining and Augmenting Decision Procedures'. *Journal of Automated Reasoning* **28**(3), 2002.
7. Knuth, D. E. and P. B. Bendix: 'Simple word problems in universal algebra'. In: J. Leech (ed.): *Computational problems in abstract algebra*. Pergamon Press, 1970.
8. Lassez, J.-L. and M. Maher: 'On Fourier's algorithm for linear arithmetic constraints'. *Journal of Automated Reasoning* **9**, 373–379, 1992.
9. Socher-Ambosius, R.: 'Boolean algebra admits no convergent rewriting system'. *4th Conference on Rewriting Techniques and Applications*, Vol. 488 of *LNCS*, 1991.