

Automatic Generation of Algorithms and Tactics

Frank Theiß and Volker Sorge

Fachbereich Informatik, Universität des Saarlandes, Germany,
{lime|sorge}@ags.uni-sb.de <http://www.ags.uni-sb.de/~{lime|sorge}>

Abstract. Algorithms are powerful tools in well known fields of mathematics. Their use within Mechanized Reasoning Systems (MRS) is highly desirable, as algorithms are usually the most efficient way to solve the problem they are designed for, i.e. using algorithms to deal with polynoms makes it easier to find proofs involving polynoms.

To make an algorithm available to MRS' without affecting proof correctness, it is necessary to provide not only the result of the algorithm's computation, but also additional knowledge for proof extraction. Most of the available Computer Algebra Systems (CAS), e.g. Maple, do not provide this knowledge, so there often is a need for (re-)implementation of the algorithms used by MRS, or at least of algorithms to check a CAS' result.

In this contribution we propose a development tool for tactics in Ω MEGA that supports hierarchical and recursive design and show how it can be used to generate algorithms for Ω MEGA including proof extraction. We point out similarities and differences between common sense algorithms and MRS inference rules and show how to bring the two together.

1 Introduction

Within the last decades, the computer became a more and more important tool for mathematicians. Its computational speed increased, software became more sophisticated, and nowadays most mathematical work is done computer aided. The machine help comes from mainly two academic fields: Mechanized reasoning and mechanized symbolic computation, both having different traditions and methods. Real world formal problems however usually are a mixture of both, making it difficult to solve them automatically: Symbolic computation is used in Computer Algebra Systems (CAS for short), which use highly efficient algorithms, but their power is limited to a special range of applications, whereas Mechanized Reasoning Systems (MRS) are more generally applicable, but are beaten in performance by CAS when it comes to symbolic computation. An integration of both, i.e. a MRS that can use the computational power of a CAS would offer both: general applicability, but also high efficiency whenever partial problems can be solved algorithmically.

As the aims of the mechanized reasoning approach and the computer algebra approach are different, the systems that resulted from both are different as well. CAS provide libraries of algorithms that compute in an efficient way, but only supply a computation's result, i.e. CAS behave like blackboxes and a check for correctness is difficult, whereas a MRS' will provide a result along with its justification, so it is checkable for correctness, which a MRS is able to do. So proper integration of MRS and CAS requires the CAS' computation to be traceable, which unfortunately often means reimplementing instead of using an existing system. However as the algorithms used by a CAS should be proven correct, this is less a theoretical challenge - the algorithms' correctness proofs have to be coded straight into the MRS' representation - but the enormous number of commonly used algorithms requires a lot of coding work. Even considering that from the reasoning point of view the fully traceable algebra system that is needed to check a proof's correctness is not necessarily as sophisticated as the system that brought up the results used in the proof

- e.g. a factorization of a polynom can be proven correct by multiplying the factors which is easier than finding the polynom's correct factors - there still remains a lot of effort to be spent on implementing traceable algorithms.

This contribution's focus will be how traceable algorithms can be implemented within a MRS environment and how this effort can be eased. To do so, we will first have a look onto how a MRS's respectively CAS's computation works and at which point an integration of both would be feasible. Dismantling both systems' working methods reveals that both's computation can be broken up into computational steps which actually are the application of inference rules in the MRS's case respectively the application of algorithms in the CAS's case. As we chose these as our starting point for an integration, the following sections we will examine how both can be generally described and we will point out similarities and differences between them.

1.1 Common sense algorithms

Generally an algorithm is a procedure to solve a problem from a certain class in a systematical way, which is frankly a loose definition. Unfortunately it is hard to get a closer grip on the term, as there are countless examples of algorithms from all fields of mathematics: Algorithms to find the Greatest Common Divisor of two numbers, to multiply large natural numbers, to find solutions to equational systems, to find minimal spanning trees in graphs, to find ways through a maze, to bring polynoms into a normal form and so on. Until the thirties of the 20th century the concept of an algorithms was used in quite an intuitive way, but then Gödel, Church, Turing and others started to work on the theory of computable functions, which led to the theses by Church and Turing that every effectively computable function can be denoted in terms of recursive functions (Church) respectively can be computed on a Turing machine (Turing); both theses are provably equivalent. As a function being effectively computable means that there is an algorithm denoting how to perform this computation, Church's recursive functions and the Turing machine can be considered an appropriate way to describe what an algorithm actually is.

Nowadays the ideas of Church and Turing find their practical sequel within hardware design and programming language, where particularly studies of the latter reveal how algorithms can be conveniently denoted and implemented. In fact, the most common form to denote algorithms is using some sort of programming language or so called pseudo code, i.e. fragments of programming languages. Although there is a wide variety of programming languages used each laying a focus on different aspects of computation, some basic elements are common to all programming languages.

In the following we will give an example of an algorithm to determine the Greatest Common Divisor of two natural numbers (GCD). It is quite a simple algorithm, but will serve well as an example.

```
function GCD(a, b)
  begin
  r := a mod b;
  if (r = 0) then b else GCD(b, r);
  end
```

The syntactic elements found here are very basic, but are fundamental to almost every programming language. First we have iteration, i.e. a sequence of command statements that are sequentially executed. This together with additional control structures, here the conditional statement `if ... then`, makes up the control structure of the execution of this program. Second we can see previously defined functions being reused, in this example it is the modulo function, the equality function

in the conditional statement and the GCD function itself being called recursively. The importance of composition and recursion is also reflected by the appearance of algorithms if they are commercially delivered: they are usually delivered in form of libraries, which are collections of algorithms, and usually these libraries themselves are incrementally designed making extensive use of composition and recursion. Third we have variables, which is very common in different appearances throughout the whole field of mathematics and computer science.

1.2 MRS inference rules

Inference rules are part of a logical calculus. Let \mathcal{F} be the set of well formed formulas of the calculus' logic, then formally an inference rule \mathcal{I} formally is a n -ary relation over \mathcal{F}^n . Inference rules can be denoted as follows:

$$\frac{f_{m+1} \dots f_n}{f_1 \dots f_m} \mathcal{I}$$

where $f_1 \dots f_m$ denote premises of inference rule \mathcal{I} and $f_{m+1} \dots f_n$ denote its conclusions. The meaning of \mathcal{I} being part of a calculus is that if there is a proof for $f_1 \dots f_m$ being true, $f_{m+1} \dots f_n$ can be deduced by application of \mathcal{I} :

$$f_1 \dots f_m \vdash^{\mathcal{I}} f_{m+1} \dots f_n$$

So finding a proof for a theorem t means to deduce it by applying a sequence of inference rules whose premises are either axioms of the calculus or can be deduced through inference rules themselves:

$$\mathcal{F}_0 \vdash^{\mathcal{I}_1} \mathcal{F}_1 \mathcal{I}_2 \vdash \dots \vdash^{\mathcal{I}_n} \mathcal{F}_n$$

where \mathcal{F}_0 is the set of the calculus' axioms and $t \in \mathcal{F}_n$.

The specification of an inference rule's conclusions and premises is usually denoted by formulae or schemata of formulae, i.e. formulae containing free variables. E.g. the rule *AndElimination* can be denoted like this:

$$\frac{A}{A \wedge B} \textit{AndElimination}$$

where A and B are free variables that are instantiated every time the rule is applied.

Technically the implementation of inference rules has to fulfill further requirements to be applicable in real world purposes. As the set of a calculus' well formed formulas is not explicitly known, it occurs most often that an inference rule's arguments are only partially known, i.e. we possibly are given a goal to prove and an inference rule whose conclusion schema matches this goal. So in order to apply this inference rule to our goal, we have to construct the premises according to the schemata of its premises and insert them into our proof plan. For example given the formula $P(a)$ to prove and two applicable inference rules *ForallElimination* and *AndElimination*

$$\frac{P(A)}{\forall x.P(x)} \textit{ForallElimination}$$

$$\frac{A}{A \wedge B} \textit{AndElimination}$$

we easily can construct the premise of *ForallElimination*, because there are no unknown variables contained. On the other hand we cannot use *AndElimination* here: while A has to be instantiated $P(a)$, we cannot construct B . *AndElimination* however could be applied easily, if only the premise was known, but not the conclusion; all variables, here A and B , would be constructable from what we got. *ForallElimination* could not be applied under these conditions, variable A cannot be derived from the premise.

Generally we can apply an inference rule to a set of so called patterns of existent and nonexistent proof lines. A pattern of proof lines for an inference rule that specifies one premise and one conclusion could be for example (**nonexistent existent**), which would state the conclusion being known while the premise is not. This denotation of patterns is obtained by replacing each argument of an inference rule according to its state by **existent** or **nonexistent**, conclusions before premises. In our example we will find that *AndElimination* can be applied to patterns (**existent existent**) and (**nonexistent existent**), but not to (**existent nonexistent**), while *ForallElimination* could be applied to (**existent existent**) and (**existent nonexistent**), but not to (**nonexistent existent**).

Sometimes however it is useful to increase an inference rule's applicability by providing additional parameters. If, for example, we apply *ForallElimination* to pattern (**nonexistent existent**), we can do so by providing the yet unconstructable variable A as an additional parameter.

As well as algorithms inference rules can be hierarchically designed, too. Starting from the basic inference rules defined by a calculus, more complex rules can be implemented as a sequential application of previously defined rules very similar to what we have seen in section 1.1. For example several applications of *AndElimination* can be combined to a more potent rule *AndElimination** to split conjunctions of more than two arguments. The implementation of such higher level inference rules may be more efficient than a sequential application of the basic rules it is assembled from, but its computation should be expandable, i.e. on demand it is possible to reconstruct the computation by breaking it up into its very basic steps, the calculus' basic inference rules.

1.3 Bringing both together

Bringing together both principles, the power and efficiency of a CAS when it comes to algebraic problems and the generality and checkability of results offered by a MRS is the main focus of this paper. Due to the vast variety of algorithms used in a mathematicians everyday work, we will not concentrate on special algorithms, but will have a rather general look on how an integration can be done.

Our approach to integrate algorithms and inference rules is to embed the first into the latter. For example the GCD algorithm introduced above could be encapsulated into an inference rule as follows:

$$\frac{\Phi(c)}{\Phi(GCD(a,b))} GCD$$

where c is what the above algorithm returns on input a and b , and Φ is a formula with an occurrence of $GCD(a,b)$ respectively c . Obviously the application of this inference rule is restricted to patterns **existent existent** and **nonexistent existent** due to the fact that the GCD algorithms can only determine c from a and b , but not vice versa.

So far the GCD algorithm is used as a black box that is given the input a and b and that will eventually return the result c . To incorporate the algorithm's

result into a fully expandable and checkable proof plan however it is necessary to make its computation traceable, i.e. the GCD algorithm should return not only the computation's result but also its justification. The justification actually should be a sequence of inference rule applications.

Starting from the correctness proof of the algorithm, which is left to the reader, we will face several problems.

2 TACO, a development tool for Ω_{MEGA} tactics

TACO is a development tool to implement tactics for Ω_{MEGA} . It provides a graphical user interface to view and edit tactics and their expansion and automatically generates executable code from these specifications that can be used from within Ω_{MEGA} . The aim of TACO is to provide an environment to implement fully expandable tactics with little more effort than sketching them on paper and thus to reduce development and debugging time considerably. It also supports incremental design of tactic libraries as it is easy to combine previously defined tactics to more complex ones and offers control structures to implement traceable algorithms.

2.1 Tactics

To design tactics in Ω_{MEGA} we first have to find a way to properly describe them. TACO provides a Graphical User Interface to do so. The elements to be found there are:

variables the variables used in the tactic's description

theory constants all symbols used in the tactic that are constants in their theory

arguments additional parameters needed to apply a tactic, e.g. the term to be inserted when applying a forall elimination

patterns the patterns to which the tactic should be applicable

theory the theory in which the tactic is defined

premises the tactic's premises. Premises are proof line schemata and consist of a name and a formula

conclusions the tactics conclusions. Conclusions are described the same way the premises are.

constraints the constraints are either equations or pieces of LISP code. A tactic cannot be applied unless all constraints evaluate to true

2.2 Algorithms

3 Example

4 Conclusion