Non-trivial Symbolic Computations in Proof Planning

Volker Sorge

Universität des Saarlandes Fachbereich Informatik
D-66041 Saarbrücken Germany
sorge@ags.uni-sb.de

Abstract. We discuss a pragmatic approach to integrate computer algebra into proof planning. It is based on the idea to separate computation and verification and can thereby exploit the fact that many elaborate symbolic computations are trivially checked. In proof planning the separation is realized by using a powerful computer algebra system during the planning process to do non-trivial symbolic computations. Results of these computations are checked during the refinement of a proof plan to a calculus level proof using a small, self-implemented, system that gives us protocol information on its calculation. This protocol can be easily expanded into a checkable low-level calculus proof ensuring the correctness of the computation. We demonstrate our approach with the concrete implementation in the ΩMEGA system.

1 Introduction

In recent years there have been many attempts at combining computer algebra systems (CAS) and deduction systems (DS). Either for the purpose of enhancing the computational power of the DS [17, 18, 3] or in order to strengthen the reasoning capabilities of a CAS [1, 4]. For the former integration there exist basically three approaches: (1) to fully trust the CAS, (2) to use the CAS as an oracle and to try to reconstruct the proof in the DS with purely logical inferences, and (3) to generate protocol output during a CAS calculation and to use this protocol to verify the computation. Following approach (1) one cannot guarantee the correctness of the proof in the DS any longer. While the correctness is no issue in approach (2) it foregoes the efficiency of a CAS and to replay the computation with purely logical reasoning might still impose a hard task to the DS. (3) is a compromise, where one can employ the computational strength of a CAS and additionally gains important hints easing the reconstruction and checking of the computation.

We have, indeed, successfully experimented with idea (3) by implementing a prototypical CAS ($\mu\mathcal{CAS}$) that is a small library of simple polynomial algorithms which give us protocol information on their computations [18, 23]. This protocol information is used to derive abstract proof plans that can be transformed into proofs of the ΩMEGA system [16]. Exploiting ΩMEGA's ability of step-by-step expansions of proof plans into natural deduction (ND) calculus proofs [13], the

computations can be machine-checked on a fine-grained calculus level. While this way of integrating a computer algebra system into ΩMEGA solves the correctness issue, it has the drawback that apart from $\mu\mathcal{CAS}$ there does not exist a full grown CAS that provides us with the necessary protocol output on its calculations.

In this paper we present a pragmatic approach to work around this problem in proof planning. It is based on the idea presented in [17] that many hard symbolic computations are easy to check. As example for this thesis one may consider symbolic integration, a surely non-trivial computation, whose verification is to simply differentiate the result of the integration algorithm. We exploit this fact within the proof planning component of ΩMEGA: Results of non-trivial symbolic computations are used during the proof planning process. The verification of these calculations is postponed until a complete proof plan is refined to a low level calculus proof and it is arranged in a way, such that we can use the trivial direction of the verification. This is achieved by using MAPLE [22] for computations during the planning process and $\mu\mathcal{CAS}$ to aid the verification. The technique we present here is already successfully applied in the ΩMEGA system in the domains of limit theorems and optimization problems. For detailed report on proof planning limit theorems see [20] and for some earlier work on optimization problems see [18]. In this paper we will only be concerned with the details of these problems that involve the application of computer algebra.

The paper is organized as follows: We first give a general overview on proof planning and its special features in ΩMEGA in the next section. In Sec. 3 we present the general architecture of the integration of CAS into ΩMEGA and elaborate how the systems MAPLE and $\mu\mathcal{CAS}$ are used within the proof planning component of ΩMEGA in order to achieve our goal of separating non-trivial computation and easy verification. In Sec. 4 we tackle the problem of how to deal with different normal forms of different systems that can arise in our context. In Sec. 5 we illustrate our ideas with two examples from the domains of limit theorems and optimization problems. We finally discuss advantages and drawbacks of the presented approach in Sec. 6 before concluding in Sec. 7.

2 Proof Planning in ΩMEGA

The purpose of the ΩMEGA system is to provide assistance in the process of deriving, presenting and checking proofs in mathematics. It provides several means to prove theorems by applying tactics, by using external reasoners — such as Automated Theorem Provers or CAS — or by automatically planning proofs with the help of a proof planner. ΩMEGA's basic logic is a variant of natural deduction (ND) calculus [13] based on a higher order λ-calculus [8]. As ΩMEGA neither makes correctness assumptions about the incorporated external reasoners nor about tactics or planning methods a user specifies, it requires that proofs are finally checked in the basic calculus to ensure correctness. Thus, although proofs can be both constructed (either interactively or automatically) and represented on more abstract levels, ΩMEGA accepts a proof to be valid,

only, if it can be successfully expanded into a proof object whose derivations are in the basic ND calculus that can be machine checked.

ΩMEGA's proof planner is designed to automatically plan proofs on an abstract level by constructing a sequence of methods that represents an abstract derivation for a given theorem from a set assumptions. Methods are (partial) specifications of tactics [5] known from tactical theorem proving [15]. In ΩMEGA planning methods are declaratively represented in frame-like structures consisting of four major slots (two instances of planning methods are given in Sec. 5):
Premises and Conclusions are sets of sequents where the method serves to logically derive the conclusions from the premises. Single sequents can be annotated in STRIPS-style with $\oplus$ or $\ominus$ (cf. [10]) specifying whether a sequent is added to or removed from the planning state by the method. In particular, a conclusion annotated with $\ominus$ is a goal that is closed by the method and a premise annotated with $\oplus$ represents a new open subgoal that will have to be proven after the application of the method.
Proof Schema is a schematic specification of the sub-proof the method abbreviates. The schemas of premise and conclusion sequents are used to be matched with actual formulas in a proof in order to determine the methods applicability. Moreover the proof schema can contain lines that are introduced into the proof only when the method is refined by expansion.
Application Conditions are constraints on the particular instantiations of the parameters of a method that have been introduced by matching. A method is applicable with an instantiation $\mathcal{I}$ of parameters if and only if for $\mathcal{I}$ none of the application-conditions evaluates to *false*. Thus, application condition can either fail or return some values which can be bound to additional method parameters. This offers a platform to execute arbitrary functions, e.g., calls to external reasoners.

Once a proof plan has been found in ΩMEGA, it has to be refined by expanding it to a ND calculus-level proof to gain a proof object which can be checked in order to ensure its validity. This expansion is a recursive process, i.e., the expansion of a planning method or an abstract tactic yields a subproof which can again contain abstract proof steps that can be expanded further. The expansion is generally not carried out immediately but postponed until a full proof plan has been found. Yet, as long as a proof step has still an *abstract justification*, i.e., a justification that does not belong to the set of basic ND calculus rules, it is considered *planned*. ΩMEGA's main data structure for storing proofs and proof plans — the proof plan data structure, $\mathcal{PDS}$ [7] — offers facilities to execute this expansion as well as to contract expanded subproofs to shorter, abstract proofs again. The expansion of abstract proof steps is triggered either manually by the user or automatically by the proof checker and executed automatically by ΩMEGA.

3 Integration of CAS

In this section we first present the general architecture for the integration of CAS into ΩMEGA. For a more detailed introduction see also [18, 23]. We then present

our new approach to integrating symbolic computations and there verification into proof planning in ΩMEGA.

3.1 Architecture

The integration of computer algebra into ΩMEGA is accomplished by the SAPPER system [23] which can be seen as a generic interface for connecting one or several computer algebra systems (see Figure 1). An incorporated CAS, like MAPLE [22] or $\mu\mathcal{CAS}$ [23], is treated as a slave to ΩMEGA which means that only the latter can call the former and not vice versa. From the technical point of view, ΩMEGA and the CAS are independent processes while the interface is a process providing a bridge for communication. Its role is to automate the broadcasting of messages by transforming output of one system into data that can be processed by the other[1]. The maintenance of processes and passing of messages is managed by the MATHWEB [11] environment ΩMEGA is embedded into.

The role of SAPPER in the integration has two distinct aspects: Firstly, arbitrary CAS' can be easily used as black box systems for term rewriting (similar to approaches of [4, 3]) and SAPPER works as a simple bridge between the planner and the CAS. Secondly, SAPPER also offers means to use a CAS as a proof planner. That is, if the CAS can provide additional information on its computations, this information is recorded by SAPPER and translated into a sequence of tactics that can eventually verify the computation. Since there does not exist a state of the art system that provides this information, we use our own $\mu\mathcal{CAS}$ system, which basically is a collection of simple algorithms for arithmetic simplification and polynomial manipulations which include a plan generating mode (cf. [18]).

The two tasks of a CAS, rewriting and plan generation, are mirrored in the interface (cf. Fig. 1) that basically can be divided into two major parts; the *translator* and the *plan generator*. The former performs syntax translations between ΩMEGA and a CAS in both directions while the latter only transforms protocol output of $\mu\mathcal{CAS}$ to ΩMEGA proof plans. Figure 1 also depicts the different uses of the two CAS involved: while MAPLE is connected as a black box system, only, $\mu\mathcal{CAS}$ can be used both as black box and as plan generator.

While the translation part is very common, the plan generator is the actual specialty of SAPPER. It provides the machinery for the proof plan extraction from the specialized algorithms in $\mu\mathcal{CAS}$. These are equipped with a *proof plan generating mode* that returns information on single steps of the computation within the algorithm. The output produced by the execution of a particular algorithm is recorded by the plan generator which converts it, according to additional information on the proof, into a proof plan. In order to produce meaningful information $\mu\mathcal{CAS}$ needs to have a certain knowledge on the proof methods and tactics available to ΩMEGA in its knowledge base. Thus references to logical objects (methods, tactics, theorems, or definitions) of the knowledge base are compiled a priori into the algebraic algorithms in order to document their calculations. SAPPER's plan generator uses produced protocol output to lookup

[1] This is an adaptation of the general approach on combining systems in [9].

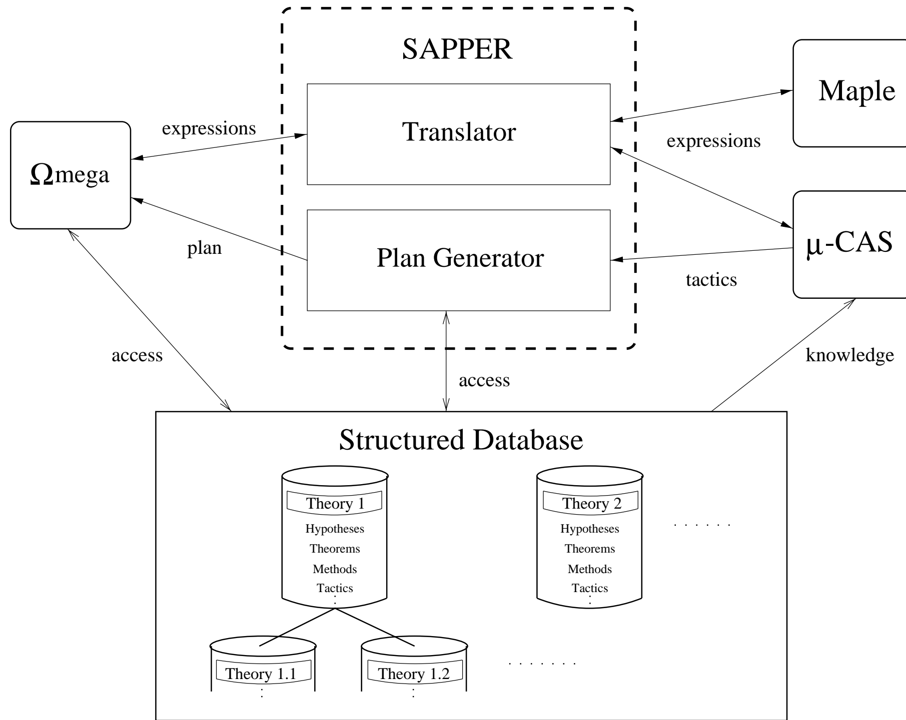

Fig. 1. Interface between ΩMEGA and computer algebra systems

tactics and theorems of an ΩMEGA theory (cf. Figure 1) in order to assemble a valid proof plan. How algorithms in $\mu\mathcal{CAS}$ have to be expanded and how the plan generation is performed is illustrated in Sec. 3.3.

3.2 Integration into Proof Planning

When integrating computer algebra into proof planning we have to keep in mind that all plans have to be expandable to ND calculus proofs. However, using a system whose computations are checkable, like $\mu\mathcal{CAS}$, restricts us to the use of its rather simple algorithms which might not always be sufficient for the task at hand. What we really would like, is to combine the computational power of a CAS like MAPLE to perform non-trivial computations with the verification strength of $\mu\mathcal{CAS}$, i.e., simple arithmetic.

Therefore, we try to exploit as much as possible the fact that many difficult symbolic computations are easy to verify. This is folklore in mathematics and has already been elaborated in [17]; the most prominent example for this is certainly symbolic integration which is still a hard task for many CAS. Results of symbolic integration algorithms are, however, easily checked, since it involves to differentiate the result and compare it with the original function, only. Other

examples are computation of roots of functions or factorization of polynomials, which involve non-trivial algorithms, but the verification of the results only involves straight-forward arithmetic.

The separation of computation and verification can be easily achieved within proof planning: During the planning process the applicability of a method is solely determined by matching and checking the application-conditions. As mentioned earlier, the latter can be used to execute arbitrary functions, therefore we can also implement conditions that call MAPLE and in case useful results are returned, bind these to some method parameters. During the planning process we are not concerned with the verification of the computation, and postpone it until the method is actually expanded. This is done by stating a rewriting step that is justified by the application of CAS within the proof schema of the method, preferably in those lines that are introduced during the expansion of the method.

Thus, we design our planning methods in a way that MAPLE is called in one of the application conditions to perform the difficult computations during the planning process. The proof schema then contains the appropriate proof steps that enable the application of $\mu\mathcal{CAS}$ to verify MAPLE's computation during the refinement of a proof plan, in the easier direction.

3.3 Verification of Computations

To implement a plan generating mode is a simple task for simple CAS algorithms. Figure 2 shows the simplified version of the recursive addition algorithm in $\mu\mathcal{CAS}$'s arithmetic simplification module in a lisp-like pseudo-code. $\mu\mathcal{CAS}$ works with polynomial-like representation of given terms together with a lexicographic monomial ordering. Thus the simplification algorithm of $\mu\mathcal{CAS}$ always transforms terms into lexicographically ordered polynomials and in case of rational functions into a fraction of two polynomials.

The algorithm in Fig. 2 is basically a case split where the first two cases ensure that the arguments are in some normalized form. The last three cases then take care of correctly adding the two argument polynomials. The predicates $=_{lex}$, $<_{lex}$, and $>_{lex}$ correspond to comparison of monomials according to the lexicographical term-ordering in $\mu\mathcal{CAS}$. The protocol output of the `tactic` function corresponds to names of tactics that are known to ΩMEGA and from which a proof plan can be assembled. This proof plan can be inserted into the given proof and further expanded using ΩMEGA's tactic expansion mechanism. Note, that such a proof plan only serves to verify the correctness of the result of one run of the algorithm and cannot be used to verify the correctness of the algorithm as a whole.

Still, the verification of a single computation is not totally trivial, since it involves to have the appropriate algorithm available in $\mu\mathcal{CAS}$, to extend this algorithm with a proof plan generating mode, and maybe to write some of the tactics for ΩMEGA that correspond to those in the algorithm. However, as we are mainly concerned with verifications involving arithmetic, the latter task should diminish over time, since many of the tactics involved can be reused.

```
(addition (a+c b+d)
          (cond ((not (monomial a))
                  (addition (addition (simplify a) c) b+d))
                ((not (monomial b))
                 (addition a+c (addition (simplify b) d)))
                ((a =_lex b)
                 (tactic "mono-add")
                 ((add a b) + (addition c d)))
                ((a >_lex b)
                 (tactic "pop-first")
                 (a + (addition c b+d)))
                ((a <_lex b)
                 (tactic "pop-second")
                 (b + (addition a+c d)))))
```

Fig. 2. Polynomial addition in $\mu\mathcal{CAS}$.

$\mu\mathcal{CAS}$ could also be viewed as a collection of elaborate tactics for ΩMEGA. However, it is implemented as an independent system (in Common Lisp) and can be used like a regular CAS, even though with very limited power, only.

4 Dealing with Normalforms

When using MAPLE for a computation within some method and $\mu\mathcal{CAS}$ to verify MAPLE's result we might have problems identifying the term resulting from $\mu\mathcal{CAS}$'s computation with the original term MAPLE was applied to. Thus, we have to take care of the problem of distinct normal forms of the systems involved during the expansion of a computation [2].

Let Φ_0 be the original term in the proof, while Φ_{MAPLE} denotes the term which results from applying MAPLE to Φ_0, and let $\Phi_{\mu\mathcal{CAS}}$ be the term returned by $\mu\mathcal{CAS}$ applied to Φ_{MAPLE}. Furthermore, let $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$ be the sequence of tactics computed by $\mu\mathcal{CAS}$, whose application to Φ_{MAPLE} yields the proof plan (1)[3]

$$\Phi_{\text{MAPLE}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \ldots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}}. \tag{1}$$

We then have three cases to consider:

(a) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ coincide,
(b) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ are distinct, however Φ_0 occurs at some point during the expansion, and
(c) Φ_0 and $\Phi_{\mu\mathcal{CAS}}$ are distinct, and Φ_0 does not occur during the expansion.

[2] The form of MAPLE's result may vary even for equivalent arithmetic expressions (in two different runs of MAPLE), depending on the form of the input. For instance, MAPLE's simplification of $x+2z+y-z$ yields $x+z+y$, while the same computation with input $x+y+2z-z$ would yield $x+y+z$ in a different run.

[3] For the sake of clarity, we omit any context the terms Φ_j might be embedded in, i.e., we view the proof plan as rewriting steps of a sub-term of some arbitrary formula.

Case (a) is trivial. Case (b) means that we have some $1 \leq i \leq n$, such that

$$\Phi_{\text{MAPLE}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \dots \xrightarrow{\mathcal{T}_i} \Phi_0 \xrightarrow{\mathcal{T}_{i+1}} \dots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}}. \quad (2)$$

This problem can be easily solved by successively applying the single tactics and checking after each application whether the resulting term is already equivalent to Φ_0. In this case the proof can be concluded directly. The remainder of the tactic sequence, i.e., $(\mathcal{T}_{i+1}, \dots, \mathcal{T}_n)$ in (2), is discarded.

Case (c) is less trivial since the produced tactics are not sufficient to fully justify the computation and thus we are left with a new proof problem, namely to derive the equality of $\Phi_{\mu\mathcal{CAS}}$ and Φ_0. However, at this point we can make use of the lexicographic term ordering of $\mu\mathcal{CAS}$: if $\Phi_{\mu\mathcal{CAS}}$ and Φ_0 really constitute the same arithmetic expression, applying $\mu\mathcal{CAS}$ simplification algorithm to Φ_0 will yield $\Phi_{\mu\mathcal{CAS}}$. Note, that this step might not only include trivial reordering of a sum but can contain more sophisticated arithmetic. The execution of the simplification algorithm will then return a sequence of tactics $(\mathcal{S}_1, \dots, \mathcal{S}_m)$ such that we get:

$$\Phi_{\text{MAPLE}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \dots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}} \xleftarrow{\mathcal{S}_m} \dots \xleftarrow{\mathcal{S}_1} \Phi_0 \quad (3)$$

In praxis, we deal with this problem slightly different, since in ΩMEGA's tactic expansion mechanism calls to $\mu\mathcal{CAS}$ have to be carried out explicitly by expanding the accordant justification, and not implicitly during an expansion itself. Thus, we introduce a new subproof for the equality of $\Phi_{\mu\mathcal{CAS}}$ and Φ_0:

$$\begin{array}{ll} \Phi_{\mu\mathcal{CAS}} = \Phi_{\mu\mathcal{CAS}} & \quad (=\text{Ref}) \\ \Phi_{\mu\mathcal{CAS}} = \Phi_0 & \quad (\texttt{CAS}) \end{array}$$

The equation of the second line serves then to apply a =Subst rule to finish the original expansion, resulting in proof plan (4).

$$\Phi_{\text{MAPLE}} \xrightarrow{\mathcal{T}_1} \Phi' \xrightarrow{\mathcal{T}_2} \dots \xrightarrow{\mathcal{T}_n} \Phi_{\mu\mathcal{CAS}} \xrightarrow{=\text{Subst}} \Phi_0. \quad (4)$$

In order to completely verify the computation the justification (`CAS`) above must be expanded as well. This results in the second call to $\mu\mathcal{CAS}$, yielding a proof plan equivalent to the right hand side of (3).

5 Examples

We illustrate our approach with two examples of proof planning methods that are used in different domains. The first is the `Complex-Estimate` method that is needed for proof planning limit theorems; the second is the `Polynomial-Root` methods which is employed for solving optimization problems.

The use of proof planning to solve optimization problems is advancement of work already reported on in [18]. However, in [18] we were restricted to the use of $\mu\mathcal{CAS}$ and therefore could only tackle problems involving polynomials of degree at most two, due to lack of sophisticated algorithms for computing roots

Method: `Complex-Estimate`		
Premises	L_1, $\oplus L_2$,$\oplus L_3$, $\oplus L_4$	
Appl. Cond.	$\sigma \longleftarrow GetSubst(a, b)$ $k, l \longleftarrow CAS_split(a_\sigma, b)$	
Conclusions	$\ominus L_7$	
Proof Schema	$(L_1)\ \Delta \vdash \lvert a\rvert < e_1$	
	$(L_2)\ \Delta \vdash \lvert k\rvert < \mathbf{M}$	(OPEN)
	$(L_3)\ \Delta \vdash \lvert a_\sigma\rvert < \epsilon/2 * \mathbf{M}$	(OPEN)
	$(L_4)\ \Delta \vdash \lvert l\rvert < \epsilon/2$	(OPEN)
	$(L_5)\quad \vdash b = b$	(=Ref)
	$(L_6)\quad \vdash b = k * a_\sigma + l$	(`CAS` L_5)
	$(L_7)\ \Delta \vdash \lvert b\rvert < \epsilon$	(`fix` $L_6 L_1 L_2 L_3 L_4$)

Fig. 3. The `Complex-Estimate` Method

in $\mu\mathcal{CAS}$. We are currently experimenting with a set of 30 different problems where most of them can be proof planned automatically.

Proof planning of limit theorems is extensively described in [20]. Besides the application of computer algebra it also involves the use of a constraint solver in order to fully automate the planning process. However, we are not concerned with these details here since we will elaborate for neither example how the complete proof plans are found or look like but rather concentrate on the application of those methods involving computer algebra. In ΩMEGA we can currently automatically plan around 20 theorems from the limit domain [19] including theorems such as LIM+, LIM*, and LIM/.

5.1 The `Complex-Estimate` Method

Figure 3 depicts the planning method `Complex-Estimate` whose purpose is to estimate the magnitude of the absolute value of a complex term by estimating its simpler factors. The method formalizes the derivation of the conclusion L_7 from the premises L_1, L_2, L_3, and L_4. The annotations indicate that the lines L_1 and L_7 have to be present in the current planning state for the method to be applicable, whereas L_2, L_3, and L_4 will be introduced as new open goals.

Thus, `Complex-Estimate` is handled by the planner as follows: If an open line in the planning state matches L_7 and if an assumption matching L_1 is available, `Complex-Estimate`'s parameters a, b, e_1, ϵ are instantiated. Then the two application conditions are evaluated. The first, $GetSubst(a, b)$, computes a substitution in order to match sub-terms of the expressions a and b. If such an substitution exists, it is returned and bound to the parameter σ and the next application condition, $CAS_split(a_\sigma, b)$, is evaluated. CAS_split calls MAPLE to factorize b. This factorization is done using MAPLE's functions `quo` and `rem` which both employ the Euclidean algorithm where `quo` returns the quotient of a polynomial division (corresponding to k in our method) and `rem` returns the remainder of that division (i.e., l in the method). The two functions are called in the following way: `quo(`b`,`a_σ`,`a'`)` and `rem(`b`,`a_σ`,`a'`)`, where a_σ denotes the term

resulting from the application of the substitution σ computed by *GetSubst* to a. a' is a sub-term of a_σ that functions as the reference variable for the polynomial division in `quo` and `rem` respectively. Instead of using both MAPLE functions we could have supplied the call of `quo` with an additional formal parameter to store the remainder of the division automatically. However, using `rem` frees us from the requirement to choose the formal parameter distinct from any term occurring in both b and a_σ.

If the two application-conditions are successfully evaluated, the method is introduced into the partial plan. The goal L_7 is removed as planning goal, and the new goals L_2, L_3, and L_4 are introduced instead.

When the method is expanded later on, the complete subproof given in the *proof schema* is introduced, i.e., the sequents L_5 and L_6 are newly added to the proof and the justification of L_7 is updated. It is then justified by (`fix` $L_6 L_1 L_2 L_3 L_4$), where `fix` is an abstract justification itself that will be expanded during further refinement of the proof plan, as well. Lines L_5 and L_6 serve to certify the correctness of the values computed by MAPLE by making this computation explicit. Here the equality of b and $k * a_\sigma + l$ is derived from the reflexivity of equality which is an axiom of the underlying calculus. The justification of line L_6 indicates both that the step has been introduced by the application of a CAS and that its expansion can be realized by using $\mu\mathcal{CAS}$ in plan generating mode.

L_5 and L_6 indicate that for the verification of MAPLE's computation it is necessary to certify that $k * a_\sigma + l$ can successfully be transformed into b. To perform this goal, we must use basic arithmetic, only, instead of the considerably harder polynomial division performed by MAPLE. Thus, we can use $\mu\mathcal{CAS}$' simplification component which employs, among others, the algorithm of Fig. 2. For a concrete instance $\mu\mathcal{CAS}$ would return a sequence of tactics indicating single computational steps that have been performed inside the computer algebra algorithm. This proof plan is then inserted into the proof and further expanded to show the correctness of the computation.

As a concrete example we consider the application of `Complex-Estimate` in the proof of LIM+. LIM+ informally states that the limit of the sum of two functions is the sum of their limits and is formalized by:

$$\forall f. \forall g. \forall l_1. \forall l_2. \lim_{x \to a} f(x) = l_1 \wedge \lim_{x \to a} g(x) = l_2 \to \lim_{x \to a} (f(x) + g(x)) = l_1 + l_2.$$

During the proof planning process for LIM+, the assumption $\Delta \vdash |f(X_1) - l_1| < E_1$ is available at some point. Then the goal $\Delta \vdash |f(x) + (g(x) - (l_1 + l_2))| < \epsilon$ can be removed by applying the method `Complex-Estimate`. After matching the premise and conclusion lines the method variables a and b are instantiated by $(g(X_1) - l_2)$ and $f(x) + (g(x) - (l_1 + l_2))$, respectively. When the *application-conditions* of `Complex-Estimate` are evaluated, *GetSubst* returns $[x/X_1]$ and *CAS_split* calls MAPLE to compute the instantiations k and l. The actual function calls passed to MAPLE are

`quo(`$f(x)+g(x)-(l_1+l_2)$`,`$f(x)-l_1$`,`$f(x)$`);` `rem(`$f(x)+g(x)-(l_1+l_2)$`,`$f(x)-l_1$`,`$f(x)$`);`

returning 1 and $(g(x) - l_2)$ as results for k and l, respectively. Because `quo` and `rem` are algorithms dealing with polynomials, only, functional notation is abolished in SAPPER's translator when MAPLE is called; e.g., $f(x)$ would become f_x.

The goal $\Delta \vdash |f(x) + g(x) - (l_1 + l_2)| < \epsilon$ is then replaced by the new goals[4]
1. $\Delta \vdash |1| < \mathbf{M}$, 2. $\Delta \vdash |f(x) - l_1| < \epsilon/2 * \mathbf{M}$, 3. $\Delta \vdash |g(x) - l_2| < \epsilon/2$.

After a complete proof plan has been found for the LIM+ theorem, the plan can be refined to a calculus level proof. We elaborate this expansion for the `Complex-Estimate` method, focusing on the steps of the expansion dealing with the verification of MAPLE's computation:

$$\begin{array}{rcll} f(x) + (g(x) - (l_1 + l_2)) & = & f(x) + (g(x) - (l_1 + l_2)) & (\text{=Ref}) \\ f(x) + (g(x) - (l_1 + l_2)) & = & ((1 * (g(x) - l_2)) + (f(x) - l_1)) & (\texttt{CAS}) \end{array}$$

where the second line is justified by the application of a CAS. In order to obtain a pure ND-level proof this line needs to be further expanded. However, since during the application of `Complex-Estimate` the values for l and k were computed by MAPLE, we do not have any additional information for an expansion. To justify the computation in more detail we use an algorithm within our $\mu\mathcal{CAS}$ system in plan generation mode that produces a trace output giving more detailed information on single computational steps. Instead of simulating the complex algorithm for polynomial division within $\mu\mathcal{CAS}$, we simply use an algorithm that simplifies the term on the right-hand side of the equation. Thus, $\mu\mathcal{CAS}$ verifies the result of MAPLE's computation with the help of a much simpler algorithm. The yielded proof plan consists of a sequence of tactics indicating single computational steps of the algorithm. Within the $\mathcal{PDS}$, the single step can be expanded to a plan with higher granularity. The newly introduced proof steps are:

$$\begin{array}{rcll} f(x) + ((g(x) - l_1) - l_2) & = & f(x) + ((g(x) - l_1) - l_2) & (\text{=Ref}) \\ f(x) + ((g(x) - l_1) - l_2) & = & f(x) + (g(x) - (l_1 + l_2)) & (\texttt{CAS}) \\ & & & \\ f(x) + (g(x) - (l_1 + l_2)) & = & f(x) + (g(x) - (l_1 + l_2)) & (\text{=Ref}) \\ f(x) + (g(x) - (l_1 + l_2)) & = & f(x) + ((g(x) - l_1) - l_2) & (\text{=Subst}) \\ f(x) + (g(x) - (l_1 + l_2)) & = & ((g(x) - l_1) + (f(x) - l_2)) & (\text{Pop-Second}) \\ f(x) + (g(x) - (l_1 + l_2)) & = & ((1 * (g(x) - l_1)) + (f(x) - l_2)) & (\text{Mult-1-Left}) \end{array}$$

The lower four lines correspond to a step-by-step version of the computation as one might do it by hand. We can also observe a conflict of normalforms here, as described in the preceding section. $\mu\mathcal{CAS}$ simplification algorithm only yields $f(x) + ((g(x) - l_1) - l_2)$ as a result and thus the upper two lines had to be introduced in the proof in order to justifiy the equality substitution (=Subst). The new (`CAS`) justification is expanded with another call to $\mu\mathcal{CAS}$. However, we want to focus on the expansion of the original proof plan, i.e., of the lower four lines. So far the expansion of the original `CAS` justification has been exclusively done by $\mu\mathcal{CAS}$ proof plan generation mode. At this stage $\mu\mathcal{CAS}$ cannot provide any more details about the computation and the subsequent expansion of the next hierarchic level can be achieved without further use of a CAS. Let us, for instance, take a look at the expansion of the `Pop-Second` tactic which basically describes the reordering within a sum:

[4] Note, that $\mathbf{M}$ is a meta-variable, i.e., a place holder for instantiations of an existentially quantified variable, that will be instantiated later in the course of planning LIM+. It's actual value is, however, irrelevant to our example.

$$\begin{array}{llll} \ldots & = & f(x)+((g(x)-l_1)-l_2) & (\text{=Subst}) \\ \ldots & = & (f(x)+(g(x)-l_1))-l_2 & (\text{Associativity}_{+-}) \\ \ldots & = & ((g(x)-l_1)+f(x))-l_2 & (\text{Commutativity}_{+}) \\ \ldots & = & ((g(x)-l_1)+(f(x)-l_2)) & (\text{Associativity}_{+-}) \end{array}$$

Here the tactics named Associativity$_{+-}$ and Commutativity$_+$ correspond to the application of the obvious theorems as a rewrite rule. Now the little subproof introduced when expanding (Pop-Second) is already on the level of application of basic laws of arithmetic. These tactics can, however, be expanded even further. Expanding, for example, the (Commutativity$_+$) justification will yield:

$$\begin{array}{lr} \forall a.\forall b.\, a+b=b+a & (\text{Theorem}) \\ \forall b.\, (g(x)-l_1)+b=b+(g(x)-l_1) & (\forall E \quad (g(x)-l_1)) \\ (g(x)-l_1)+f(x)=f(x)+(g(x)-l_1) & (\forall E \quad f(x)) \\ & \\ \ldots=(f(x)+(g(x)-l_1))-l_2 & (\text{Associativity}_{+-}) \\ \ldots=((g(x)-l_1)+f(x))-l_2 & (\text{=Subst}) \end{array}$$

This last expansion step details the application of commutativity of addition as rewrite step by deriving the right instance from the theorem of commutativity. If the expansion is carried out up to this level of detail for the whole proof plan $\mu\mathcal{CAS}$ computed to justify the computation, we can proof check for correctness.[5] Provided the proof checker approves and we have correct proofs for all the applied theorems in our database, we have successfully verified the particular computation which guarantees us the correctness of the overall proof.

5.2 The `Polynomial-Root` Method

In general optimization problems are of the form $Opt_<(f, I)$ or $Opt_>(f, I)$, where f is cost function consisting of a rational polynomial an some denomination, that either has to be minimized or maximized within the rational interval I. Thus the goal is to show that the polynomial part of f has a total maximum or minimum within I. Generally f is not directly given but has to be computed from a set of given cost-functions of economic processes and prices for resources. These manipulations are done by methods using $\mu\mathcal{CAS}$ computations, whereas the calculation of extrema and subsequently of roots to justify total minimum or maximum properties are performed by methods using MAPLE and are verified by $\mu\mathcal{CAS}$.

One of these latter methods is `Polynomial-Root` shown in Fig. 4 whose purpose is to confirm the existence of a root of a polynomial. It therefore has one conclusion only and no premises, that is, when the method is applied, it closes an open line, but does not introduce any new goals into the proof plan. The application conditions are handled as follows: if p is actually a polynomial

---

[5] Actually, we have not yet expanded the subproof to ND calculus level. Since equality is a concept defined by Leibniz-equality in ΩMEGA, =Subst is a tactic which can be further expanded. However, we omitted the rather tedious details of this expansion here.

Method: `Polynomial-Root`	
Premises	
Appl. Cond.	$IsPolynomial(p)$ $a \longleftarrow CAS_root(p, x)$ $\sigma \longleftarrow$[x/a]
Conclusions	$\ominus L_3$
Proof Schema	$(L_1)\quad \vdash 0 = 0$ (=Ref) $(L_2)\ \Delta \vdash p_\sigma = 0$ (`CAS` L_1) $(L_3)\ \Delta \vdash \exists x. p = 0$ ($\exists I$ L_2)

Fig. 4. The `Polynomial-Root` Method

the *CAS_root* function calls MAPLE's function `roots` to compute the rational roots of p. In case MAPLE's computation is successful the method is applied and the first of the returned values is bound to the method parameter a (whereas possible additional roots are kept for backtracking purposes). The value of a is then used to construct an appropriate substitution for p which is bound to the parameter σ. Because `Polynomial-Root` does not introduce any new lines during the planning process all the instantiations are needed for the expansion of the method.

We observe the behavior of `Polynomial-Root` with a concrete example, but only focus on the application and expansion of the method and do not elaborate on the actual optimization problem and how it is solved. Suppose we apply `Polynomial-Root` to an open line of the form: $\exists x. (x^3 + x^2 - 5x - 2) = 0$ the method variable p is then bound to $x^3 + x^2 - 5x - 2$ and MAPLE is called with `roots(`$x^3 + x^2 - 5x - 2$`,`x`)`. MAPLE returns 2 as the only rational root of p (together with its multiplicity which we can ignore) which is bound to x and finally the substitution σ is constructed as $[x/2]$.

When expanding the method the lines L_1 and L_2 are properly substituted and introduced into the proof, as shown on the right. When expanding the `CAS` justification in the second line $\mu\mathcal{CAS}$ is again used to perform the step-by-step low-level verification of the result. The expansion employs tactics that represent single computational steps on rational numbers. We omitt, however, the details of this expansion since it works analogously to the one presented in Sec. 5.1.

0	$=$	0	(=Ref)
$((2^3 + (2^2 - 5 * 2)) - 2)$	$=$	0	(`CAS`)
$\exists x. ((x^3 + (x^2 - 5x)) - 2)$	$=$	0	($\exists I$)

6 Discussion

From our current experience, the presented approach is well suited for symbolic computations whose verification are relatively trivial, e.g., where only simple arithmetic needs to be employed. However, the method is not feasible for computations where the verification is as costly or even more complicated as the computation itself. At least in the latter case it might be more practicable to specify the computation as a $\mu\mathcal{CAS}$ algorithm immediately. Computations where

the verification will be definitely non-trivial are those involving certain uniqueness properties of the result. For instance, when employing MAPLE to compute all roots of a function, it will be a hard task to verify that there exist no more roots than those actually computed. See also [17] for further discussion of this point.

Although we presented our ideas in this paper in the context of proof planning, we strongly believe that the approach could also work in tactical (interactive) theorem proving. One necessary prerequisite will be the existence of a proof object such that a tactic employing elaborate computation can be verified with the help of the simple algorithm and the attached tactics. For instance, one can imagine a tactic such as `Complex-Estimate` in TPS[2] which would be verified after application with the help of $\mu\mathcal{CAS}$ after successful proof construction.

For systems not maintaining explicit proof objects, such as HOL [14] or PVS [21] the approach of [17] will be best suited. However, this approach directly implements the verification algorithms as tactics in the HOL system as direct correspondences to MAPLE's computation. Using a separate system for the verification, i.e., the generation of more or less detailed inference chains justifying computations, keeps the DS clean from special algorithms, methods and tactics employed for this task, only. Moreover, both SAPPER and $\mu\mathcal{CAS}$ are generic enough to easily adapt them for connection with other systems, whereas tactics can generally not that easily be exported.

7 Conclusion

We have presented an approach for integrating symbolic computations into logical derivations without foregoing the formal correctness requirements for proofs in deduction systems. The main idea underlying the approach is to employ the computational power of full grown CAS, such as MAPLE, during proof planning and thereby ease the derivation of certain witness terms. After a proof has been successfully planned we use a little self-tailored CAS, $\mu\mathcal{CAS}$, during the refinement phase of the plan to an actual natural deduction proof in order to verify the correctness of the original computation. This is based on the idea that for some computations the verification of a result is much simpler than the computation itself, and, indeed, for our examples it suffices to use arithmetic.

Although we hinted at how this method could also be successfully employed outside the proof planning context, it is certainly not the *ultima ratio* as a paradigm for integrating deduction and computer algebra. However, since there are currently no CAS available that are either provably correct or give us enough information on their computations in order to derive proof plans which in turn could be used to justify the correctness of results, we have chosen this pragmatic approach in the ΩMEGA system.

We demonstrated the presented technique with two example from the domain of limit theorems and optimization problems. We are currently investigating how CAS such as GAP [12] and MAGMA [6] can be employed to proof plan theorems in group theory.

References

[1] A. Adams, H. Gottliebsen, S. Linton, and U. Martin. VSDITLU: a verifiable symbolic definite integral table look-up. In *Proc. of CADE–16*, *LNAI* 1632, p. 112–126. Springer, 1999.

[2] P. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A Theorem Proving System for Classical Type Theory. *J. of Autom. Reasoning*, 16(3):321–353, 1996.

[3] C. Ballarin, K. Homann, and J. Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In *Proc. of ISSAC'95*, p. 150–157. ACM Press, 1995.

[4] A. Bauer, E. Clarke, and X. Zhao. Analytica: an Experiment in Combining Theorem Proving and Symbolic Computation. *J. of Autom. Reasoning*, 21(3):295–325, 1998.

[5] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In *Proc. of CADE–9*, *LNCS* 310. Springer, 1988.

[6] J. Cannon and C. Playoust. *Algebraic Programming with Magma.* Springer, 1998.

[7] L. Cheikhrouhou and V. Sorge. $\mathcal{PDS}$ — A Three-Dimensional Data Structure for Proof Plans. In *Proc. of ACIDCA'2000*, 2000.

[8] A. Church. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[9] D. Clément, F. Montagnac, and V. Prunet. Integrated Software Components: a Paradigm for Control Integration. In *Proc. of the Europ. Symp. on Software Development Environments*, *LNCS* 509. Springer, 1991.

[10] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[11] A. Franke, S. Hess, Ch. Jung, M. Kohlhase, and V. Sorge. Agent-Oriented Integration of Distributed Mathematical Services. *J. of Universal Computer Science*, 5(3):156–187, 1999. Special issue on Integration of Deduction System.

[12] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4*, 1998. http://www-gap.dcs.st-and.ac.uk/ gap.

[13] G. Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[14] M. Gordon and T. Melham. *Introduction to HOL.* Cambridge Univ. Press, 1993.

[15] M. Gordon, R. Milner, and Ch. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, *LNCS* 78. Springer, 1979.

[16] The ΩMEGA Group. ΩMega: Towards a Mathematical Assistant. In *Proc. of CADE–14*, *LNAI* 1249, p. 252–255. Springer, 1997.

[17] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *J. of Autom. Reasoning*, 21(3):279–294, 1998.

[18] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra Into Proof Planning. *J. of Autom. Reasoning*, 21(3):327–355, 1998.

[19] E. Melis. The "Limit" Domain. In *Proc. of the Fourth International Conference on Artificial Intelligence in Planning Systems*, p. 199–206, 1998.

[20] E. Melis and V. Sorge. Specialized External Reasoners in Proof Planning. Seki Report SR-00-01, Computer Science Department, Universität des Saarlandes, 2000.

[21] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Computer-Aided Verification, CAV '96*, *LNCS* 1102, p. 411–414. Springer, 1996.

[22] D. Redfern. *The Maple Handbook: Maple V Release 5.* Springer, 1998.

[23] V. Sorge. Integration eines Computeralgebrasystems in eine logische Beweisumgebung. Master's thesis, Universität des Saarlandes, November 1996.