On the White Box Integration of Computer Algebra Algorithms into a Deduction System

Diplomarbeit
von
Frank Theiß

verfaßt nach einem Thema von Dr. Christoph Benzmüller
unter Betreuung von Dr. Volker Sorge und Martin Pollet
am Fachbereich Informatik der Universität des Saarlandes

30. Oktober 2005

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter Verwendung keiner anderen Hilfsmittel als der angegebenen verfaßt zu haben.

Danksagungen

Ich möchte diese Arbeit meinen Eltern widmen, die den Abschluß meines Studiums nicht mehr erlebt haben.

Mein besonderer Dank gilt Martin Pollet, Volker Sorge und Christoph Benzmüller für die große Unterstützung bei der Erstellung dieser Arbeit und für ihre unendliche Geduld. Insbesondere bei Martin Pollet bedanke ich mich für die Unterstützung beim Korrekturlesen und auch für viele lange und aufschlußreiche Diskussionen.

Bei Professor Jörg Siekmann bedanke ich mich für die Möglichkeit, meine Diplomarbeit in seiner Gruppe Deduktionssysteme zu erstellen, und dafür, daß er meine Begeisterung für das Gebiet der Künstlichen Intelligenz geweckt hat.

Ich möchte mich auch bei den Mitarbeitern, ehemaligen Mitarbeitern und Studenten der AG Deduktionssysteme für ihre Hilfe bei zahlreichen kleineren und größeren Problemen bedanken.

Desweiteren möchte ich mich mich bei Georg Weisweiler für die regelmäßige und nachdrückliche Nachfrage nach dem Stand meines Studiums bedanken.

Contents

Chapter 1

Introduction

Mathematics is one of the oldest sciences, it is as old as human culture and can be found in almost every aspect of everyday life. Although its pure foundations are abstract, mathematics finds its application in the description of real world problems. In fact real world problems today are not what they used to be 3000 years ago, and while the world has changed, the appearance of mathematics has as well.

From the very beginnings, which might have been e.g. calculating with stones, each stone representing a cow [30], the appearance of mathematics grew more and more complex and abstract. During the centuries sciences as geometry, astronomy and physics got into the focus of mathematical interest and drove mathematics itself to the development of a wide variety of theories, concepts, representations and techniques. While mathematics turned out to be an appropriate tool to handle various scientific problems, every new challenge fertilised the development of mathematics itself, and this is what characterised the role of mathematics for a long time: a science that founded its existence upon its application within other sciences.

However, the sheer plentitude of emerging mathematical ideas as well as these ideas becoming more and more abstract created a strong need to find general ways to properly describe mathematical problems. This has led not only to an upcoming formalism in mathematics, but also initiated a development which had much deeper consequences, namely the negotiation of mathematics with itself.

In the 17th century Locke's 'Nihil est in intellectu, quod non prius fuerit in sensu' (nothing is in the mind, that has not been in the senses before) was countered by Leibniz' 'Nisi intellectus ipse' (except for the mind itself) [48]. Although Leibniz was referring to the human mind in a philosophical sense, these words also illustrates the change of attitude towards doing mathematics starting in these times. Now mathematics was no longer only a tool to challenge problems from other scientific domains, but there was also a consciousness for the pure innate nature of mathematics decoupled from its application. Leibniz started to work not only on the application of mathematics, but was also researching the principles of mathematical reasoning and tried to model methods of reasoning by application of a calculus, and his call 'Calculemus!' (Let us compute!) [47] was the first step towards the idea of reasoning by means of computation.

Two centuries later the work of Boole [11] established logics as a field of mathematics. Boole approached logic in a new way, reducing it to simple algebra and thus contributed to the dream of modelling human thought by means of computation. He also saw the relationship between his work and the foundations of mathematics, in his opinion 'it is not the essence of

mathematics to be conversant with the idea of numbers and quantities', which characterised the detachment of mathematics from its application in other sciences, paving the way to modern mathematics in its pure and abstract appearance. At this time his ideas were not only ancestors of the science of artificial intelligence, but also had deep consequences on contemporary mathematics and especially its formalisation. The 19th century saw the idea of logicism come up, i.e. the thesis that all mathematics is reducible to logic, the first one to fully develop this thesis was Frege with the 'Begriffsschrift' [28] at the end of this century. Frege's ideas were of great influence to Russell, who started to put them into practice in his Principia Mathematica together with Whitehead [79], and laid the foundations to the work of Hilbert, whose aim was the development of a 'proof theory', i.e. an approach to directly check the consistency of mathematics. He wanted mathematics to be formulated on a solid and complete logical foundation by showing that

- all of mathematics follows from a correctly-chosen finite system of axioms and
- that some such axiom system can be shown to be consistent.

In 1920 he proposed explicitly a research project (in metamathematics, as it was then termed) that became known as Hilbert's program to pursue this goal. Although this was shown to be impossible later by Gödel [31], his ideas are still fruitful to the field of automated theorem proving and led to an axiomatic approach to negotiate with mathematics, which still characterises modern mathematics.

During the history of mathematics, not only the theoretical basis of mathematics had to cover a long distance from its beginnings to its nowadays appearance, but so had also its tools. The first mathematical tools, like the Roman abacus, were engineered to quickly perform computations on numbers, and in fact this is roughly the task their offspring stayed dedicated to for the next some thousand years. However, apart from mechanical calculating machines and later the electronic pocket calculator, which became standard helpers for mathematicians and all people who had to perform numeric calculations, the 20th century saw the rise of the electronic computer. For the first time there was a tool to mechanise any processing of any data, and it quickly became not only an important helper in everyday life, but also an inabdicable tool for mathematicians. The computer made it possible to perform voluminous computations within seconds, to handle large amounts of data and also to reduce the rate of human errors by a reasonable degree. Nowadays the electronic computer is a standard tool for applied mathematics, and the software used for applied mathematics purposes is descended from a comparatively long tradition.

Beneath its benefits for applied mathematics however, the electronic computer also offered a basis to mechanise the pure and abstract approach to mathematics propagated by Hilbert. Now it was possible to automatically handle data that explicitely represents mathematical knowledge and it was possible to implement logic calculi and apply them to this data. One of the first programming languages was LISP, developed by McCarthy [51], is inspired by the λ-calculus described by Church [20]. In fact the computer turned out to be such a promising tool to implement the so far theoretical ideas of various logicians that it not only lead to the birth of the science of Artificial Intelligence, but even led to a literal euphoria about the idea of constructing a thinking machine. Although this dream of a machine challenging human intelligence is still way ahead of the state of the art today, the science of artificial intelligence quickly took an important place among the various fields of computer science. Nowadays artificial intelligence is an accredited science that developed its own traditions and

techniques, and although the dream of modelling human thought still seems to be far away, artificial intelligence nevertheless succeeded in various parts of the challenge and brought about systems for various purposes.

Apart from software that is designed for special purposes, the art of doing general mathematics supported by an electronic computer brought about mainly two schools: the school of computer algebra, whose aim is to efficiently perform symbolic computations on a computer, and the school of deduction systems, whose aim is to derive mathematical proofs from available knowledge. Both of them have their specific targets, so the results obtained by these both fields differ heavily concerning the techniques they use as well as in the objectives they succeed to conquer. In the following I will give a quick overview over both schools.

1.1 Computer Algebra Systems

In the beginning of the history of the electronic computer, the focus of its application was on the performance of voluminous numerical computations. With the development of programming languages, it became furthermore possible to process symbolic expressions. The development of data structures and algorithms to do so laid the foundation of what became known as computer algebra. First Computer Algebra Systems (CAS for short) originated from collections of such algorithms.

Computer Algebra Systems were developed for a variety of purposes. There are general purpose systems that can be used in various applications [36, 80, 18], but also specialised systems whose application is restricted to special purposes like differential equations or number theory [10, 12, 35]. Modern systems like MAPLE [18] provide furthermore elaborated facilities for manipulation and inspections of expressions, e.g. formatted formulae and graphical output of function graphs. The focus of Computer Algebra Systems is in general on providing efficient data structures and algorithms for symbolic computation.

In spite of elaborated facilities, however, Computer Algebra Systems are still limited to an algorithmic processing of a formatted input to produce a formatted output in a straightforward manner. Unlike deduction systems they are not able to show a 'creative' behaviour, i.e. they are unable to master any problem other than those that can be solved by the algorithms of their library.

A further drawback of Computer Algebra Systems is that a formal justification of their computations is in general not provided. Thus the correctness of their computations is depending on the correctness of the system's implementation, and the formal theoretical foundation of their algorithms has certainly been considered in their implementation, but is not explicitly available. This threatens the reliability of Computer Algebra Systems severely, not only because bugs in the implementation are hard to detect, but furthermore because specific properties of mathematical objects and their axiomatisation may influence the outcome of algebraic computation and may not be respected in an appropriate manner. This is of special importance when it comes to mathematical objects that are hard to define in an unambiguous way and the outcome of a computation is heavily depending on the axiomatisation it is based upon, e.g. as it is pointed out for the meaning of infinity by Beeson and Wiedijk [7].

Recent systems attempt to cure this drawback by introducing elaborated type systems [23, 29], such that special properties of mathematical objects are respected, or adding facilities to produce a protocol of the executed computation [29], but there are still gaps to fill to achieve a formal justification of such computations and their results.

1.2 Deduction Systems

Unlike Computer Algebra Systems, Deduction Systems operate on an explicit representation of their formal mathematical foundation in terms of a logic calculus. Thus any inference performed by these systems provides not only its result, but also a proof of its correctness based upon a formal foundation in terms of axioms and inference steps that have come to application to establish this result. This approach is close to Hilbert's idea of well founded mathematics, and it cures the drawbacks of computer algebra, as the correctness of a proof formalised within a calculus is easy to check. Furthermore any computation is theoretically reducible to theorem proving, thus the applicability of a Deduction System is not limited by a library of algorithms, but any statement that can be expressed in the system's calculus can be formalised and processed.

In practice, however, the limits are of course those of computational power that is available, which is usually much less than required to solve most problems by uninformed search. The main challenge of automated deduction from the very first beginning was to master a search space that grows in general exponentially to the length of the resulting proof. The very first program to implement mechanised reasoning, an algorithm programmed by M. Davis, implementing a procedure for the first order theory of addition in the arithmetic of integers, was performing poorly, as Davis stated, due to fact, that the underlying procedure had a worse than exponential complexity [24]. Nevertheless it could prove that the sum of two even numbers is even. In the following, two schools emerged that tried to master this problems by different means. The first tried to simulate the process by which a person might seek proofs ("simulate people"), employing heuristic procedures to guide the proof search. The first system of this school was the 'Logic Theory Machine' by Newell, Shaw and Simon [61]. The second school tried to find machine oriented proof techniques to master the complexity of the task. Representatives of this school are H. Wang [77] and A. Robinson [67]. There are arguments for both points of view, and in 1961 M. Minsky had the early insight that "it seems clear that a program to solve real mathematical problems will have to combine the mathematical sophistication of Wang with the heuristic sophistication of Newell, Shaw and Simon" [58].

Today there is a wide variety of systems for automated theorem proving in first order logic [52, 78, 37], for higher order logic [3, 8] and logical frameworks and general purpose theorem proving environments [70, 64, 75] combining different proof techniques within a single environment. There are systems that employ strongly machine-oriented techniques for automated theorem proving [68], and systems that use domain specific knowledge on a high level of abstraction or provide facilities for interactive proof development [70, 64]. Within the mixed-initiative systems, computer algebra ranges among the techniques that are to be integrated for proof development in various systems [4, 2, 43].

As for Computer Algebra Systems, modern deduction systems provide sophisticated facilities for proof manipulation and inspection within a graphical user interface [69].

1.3 Integration of Both Systems

The integration of a Computer Algebra System and a Deduction System could offer a way to handle the obvious disadvantages of both systems: to make the computational speed and precision of a Computer Algebra System available to a Deduction System would bring the

latter much closer to practical use, and to catch the bugs that are likely to threaten the reliability of Computer Algebra System by using a Deduction System to check its results would increase its fitness for safety critical purposes. An integration of both however has to fulfil some requirements, neither should the guaranteed correctness of the Deduction System be threatened, nor should weaknesses of the Deduction System affect the operability too much.

The first choice to be made is the kind of architecture that integrates both systems. Homann and Calmet propose several different kinds of architectures to do so [38], in which either the algorithms of the symbolic calculator are available to the theorem prover, the knowledge and deductional capabilities of the theorem prover are available to the symbolic calculator, e.g. to verify properties of objects, or a combination of both comes to application. In this work the CAS is intended to be a support system for the Deduction System, i.e. the algorithms of the CAS can be called by the Deduction System.

Still an integration of both systems is possible in several ways, as pointed out by Barendregt and Cohen [5]. First the Computer Algebra System can be used as a trusted system, i.e. the Computer Algebra System executes parts of the computation that are incorporated into the Deduction System's proof without being previously checked, corresponding to the *believing* approach. This is of course no approach to cure the natively lacking reliability of a Computer Algebra System, and it also threatens the correctness of the Deduction System. Nevertheless this is a very simple way to integrate both systems and will, supposed the Computer Algebra System is thoroughly implemented and well tested, lead to a useful result. There are implementations of this kind for several theorem provers, e.g. Isabelle and PVS [39, 2]. The CAS is MAPLE in both cases. For PVS [2] the occurring problem of possible unreliable results is explicitly pointed out, but a certain level of deficiency is accepted as a tradeoff for the utility of the system.

A second possibility is to use a Computer Algebra System as an oracle, corresponding to the *skeptical* approach according to Barendregt and Cohen [5]. This means that the result of a computation is provided to the Deduction System, so that this results marks a goal point for the Deduction System's search and therefore considerably restricts its search space. The success of such an approach is mainly determined by the Deduction System's ability to close the gaps and to actually justify the Computer Algebra System's results. This may reduce the applicability of the concept, but it never threatens the correctness of the resulting proof. In practice however, this possibility is not feasible, because even very simple calculations may require a proof of a length that makes them range among the hardest proofs ever found by totally automated theorem provers without domain-specific knowledge [43].

Finally there is the possibility to implement a Computer Algebra System that produce a trace of its computation that can be interpreted by the Deduction System. This means that, supposed the Computation's trace can be remodelled by the Deduction System in terms of inference steps of its underlying logic, the full algorithmic power of the Computer Algebra System is available to the Deduction System's proof search. Furthermore this approach does not threaten the correctness, as not the possibly buggy Computer Algebra System's computation is incorporated into the proof, but only its logical reconstruction, which is easy to check for correctness. The disadvantage of this approach is that available Computer Algebra Systems usually do not produce a task that is understood by a Deduction System, in fact there has to be a close correspondence between the computational steps that make up the atoms of the trace and the inference rules that can be applied by the Deduction System to reconstruct the computation. This means that this approach requires to implement a new

Computer Algebra System, and when doing so it is difficult to reach the high standard of systems like MAPLE that are developed by well established and well experienced teams. As this solution requires a very close integration of deduction and computer algebra in which the CAS can be seen as a subsystem of the theorem prover, it is close to the *autarkic* approach in Barendregt and Cohen's classification [5].

Fortunately there is a further possibility to pursue the task of making a Computer Algebra System's power available to a Deduction System, which actual is to combine the integration of a commercially available Computer Algebra System as an oracle and to attempt to close the gap between preconditions and the oracle's result by means of a traceable Computer Algebra System. In practical use this requires the traceable Computer Algebra System not to be as powerful as the commercial one, as there are many mathematical problems that are complicated to solve, but the result is easy to test for correctness by less sophisticated algorithms. An example for this is the division of polynomials, which is complicated in comparison to checking the result for correctness. This check requires the Computer Algebra System only to be able to multiply polynomials, which is easier.

The choice to use an established Computer Algebra System that is able to solve a wide variety of problems as an oracle to find the solutions of these problems and to justify these results by a less powerful but therefore formalised and traceable Computer Algebra System offers both advantages: this way it is possible to incorporate many results of an untrusted Computer Algebra System and doing so neither to threat the resulting proof's correctness nor having to spend the effort to completely reimplement such a full range system. As this combination of both approaches promises to considerably widen the range of problems that can be solved at a sensible effort to be spent, this approach is pursued in the ΩMEGA system. The systems in use in the ΩMEGA prover are Maple in the role of the untrusted system, while the results are established by the prototypical traceable Computer Algebra Systems μCAS by Sorge [71] and MASS. The MASS system and its integration within ΩMEGA is subject of this work.

1.4 Contribution of this Work

The first part of this work is dedicated to the implementation of MASS and its integration in ΩMEGA, described in chapter 3. MASS is technically similar to Sorge's μCAS. An increased robustness and applicability of the system has been achieved by redesigning the algorithmic library. Furthermore the application of the simple but verified MASS system in cooperation with the powerful commercial CAS MAPLE have been evaluated. A feasibility study of such a combination of MAPLE and μCAS has been described by Sorge [72]. The approach was now pursued using the much more elaborated MASS system and turned out to be a usable tool in a considerably number of experiments.

Second, TACO, a mathematical authoring tool, is described in chapter 4. As pointed out by Homann and Calmet [38], a common mathematical database is required for a proper cooperation of a deduction system and a CAS. TACO is a tool to comfortably maintain such a database for an integration of MASS and ΩMEGA.

During the development of MASS and its integration in ΩMEGA some 'wouldn't it be be easier if ...'-moments resulted in serious thinking about the interface to integrate computer algebra and deduction systems and related technical problems. The result was a data structure that could help solving some of the technical difficulties of such an integration. This data

structure is proposed in chapter 5.

Chapter 2

Introduction to Ωmega

The ΩMEGA system [70] is the deduction system in the scenario. The ΩMEGA proof development system is a mixed initiative environment for interactive and automatic proof development and is the core of several related research projects of the ΩMEGA research group. Its modular architecture provides a variety of facilities for proof development, proof checking and proof presentation. Its ultimate purpose is to support mathematicians in theorem proving in mainstream mathematics and mathematical education.

The ΩMEGA system supports the development of proofs in mathematical domains at a user-friendly level of abstraction, employing a central data structure and several complementary subsystems. While it has many characteristics in common with systems like NuPrl [22], CoQ [75], HOL [33] and PVS [63], it differs from these systems with respect to its focus on proof planning as introduced by Alan Bundy for induction theorem proving [14]. In that respect it is similar to the CLAM and λ-CLAM at Edinburgh [15]. Further features of the ΩMEGA system include facilities to access a number of different reasoning systems and to integrate their results into a single proof data structure, support for interactive proof development with facilities for proof inspection and guidance in proof development, and methods to develop proofs at a knowledge-based level.

2.1 Proof Data Structure

Proof construction in ΩMEGA is based on a higher order natural deduction (ND) variant of a sorted version of Church's simply typed λ-calculus [21], which is implemented in the *proof plan data structure* $\mathcal{PDS}$ [19]. The objects represented in the $\mathcal{PDS}$ are *proof lines*, also referred to as *proof nodes*. A proof line is of the form $L.\Delta \vdash F(\mathcal{J})$, where L is a unique label, $\Delta \vdash F$ a sequent denoting that the formula F can be derived from the set of hypotheses Δ, and $(\mathcal{J})$ is a *justification* expressing how the line was derived. In case the line has been introduced to the $\mathcal{PDS}$ but is not derived yet, i.e. it is a goal node, the value of this justification is *open*, those lines are in the following referred to as *open nodes* or *open lines*.

Starting from an open goal node and a set of closed support nodes, a proof or proof plan is developed step by step. In a *proof step* a proof node is derived from a (possibly empty) set of support nodes, and the derived node is attributed by a justification describing the proof step that was applied and the support nodes and possibly further parameters used in this step. The $\mathcal{PDS}$ allows to represent and develop proofs at various levels of granularity and abstraction, and accordingly there are different types of proof steps that can be applied

modify the $\mathcal{PDS}$:

- *inference rules* are part of the underlying ND calculus, and as such they are used for modification of the $\mathcal{PDS}$ at the lowest level of abstraction available in the ΩMEGA system, the calculus level. Proofs at calculus level can be checked for correctness by ΩMEGA's ND proof checker.

- *tactics* apply a sequence of proof steps within a single step. They allow the development and representation of proofs at a higher level of abstraction and support the user-friendly development of readable proofs. A tactic's applicability is defined by a set of *application patterns* or *outline patterns*

- *methods* are, like tactics, proof steps at a higher level of abstraction. Methods are represented in a declarative way and encode additional control information for an automated proof planner.

Tactics and methods can be used to integrate complex proof techniques into the $\mathcal{PDS}$, including e.g. calls to external systems. Proof steps at a higher level of abstraction, i.e. tactics and methods, can be *expanded*. Expansion is the mechanism to refine the granularity of representation of a proof step, any complex proof step can be refined down to its representation at calculus level, i.e. it can be transformed into a sequence of inference rules.

Tactics and methods are similar to LCF-style tactics [32] with respect to integrating sequences of proof steps within a single steps, however technique and philosophy are different. While at application time of a LCF-style tactic the actual sequence of inference steps is executed, application and expansion of tactics and methods are technically independent: The result of the application of a tactic or method is remodelled in terms of inference rules by the expansion mechanism, i.e. the correctness of such a proof step can be checked using the expansion mechanism. Unlike in the constructive approach of the LCF system, the application of a tactic or method does not guarantee the correctness of its results. This is important especially when integrating results of external systems into a proof: The possible incorrectness of external systems is prevented from threatening the proof's correctness by using the expansion mechanism to remodel its computation as a sequence of proof steps. An example is the integration of an external CAS described in chapter 3, where the CAS protocol is used to generate a checkable partial proof.

2.2 Ωmega's Architecture

ΩMEGA is a modular system for automated and interactive proof development and provides a number of independent modules to modify, inspect and check the proof being developed.

The $\mathcal{PDS}$ is modified interactively by the user, who is supported by the proof planner MULTI [55] and the suggestion mechanism ΩANTS [9]. MULTI is an automated proof planner using explicitly represented control knowledge to find high level proof plans, where traditional proof planning is enhanced by using mathematical knowledge and multiple proof strategies are applied to find proofs. ΩANTS is a suggestion mechanism to find a set of possible actions in a specific proof state. By ranking these actions heuristically and executing the best rated action, the ΩANTS mechanism can also be used in an automated mode.

For proof inspection and representation, ΩMEGA features some non-standard facilities. The $\mathcal{PDS}$ can be displayed in ΩMEGA's graphical user interface $\mathcal{LOUI}$ [69] in multiple cross-linked modalities: a graphical map of the proof tree, a linearised presentation of the proof nodes and a term browser. Moreover, natural language explanation of the proof is provided by the *P.rex* [26] system, which is interactive and adaptive.

As ΩMEGA's main focus is on knowledge-based proof planning, proof development is furthermore supported by a mathematical database. Currently a set of mathematical theories are available in ΩMEGA, further support is provided by the mathematical database MBASE [27].

After a proof has been developed, it can be checked by ΩMEGA's proof checker after expanding high level proofs to the underlying ND calculus.

2.3 External Systems

One of ΩMEGA's strengths is its ability to access external systems and to integrate their results. ΩMEGA provides interfaces to heterogeneous external systems such as computer algebra systems (CAS), higher- and first-order automated theorem provers (ATP), constraint solvers (CS) and model generators (MG). Their results are transformed and inserted as sub-proofs into the $\mathcal{PDS}$, thus they can be accessed seamlessly by ΩMEGA's inspection and proof checking facilities. Furthermore they can provide control knowledge for automated proof search.

Currently the ΩMEGA system employs the following subsystems:

- *CAS*s perform symbolic computation. ΩMEGA uses two types of systems: commercial CASs provide complex algebraic computations to compute hints to guide proof search and to normalise and simplify terms. Currently the systems MAPLE and GAP are used. Unfortunately these systems act like black boxes so that their result may threaten the correctness of the resulting proof. The second type of CASs are white box systems, i.e. systems that provide a trace of their computation that can be evaluated and transformed into a sub-proof in the $\mathcal{PDS}$. Both types of systems are accessed via the SAPPER interface [72]. The white box integration of computer algebra algorithms is addressed in this work.

- *ATPs* are employed to solve subgoals. ΩMEGA uses the first order systems BLIKSEM, EQP, OTTER, PROTEIN, SPASS and WALDMEISTER and the higher-order systems TPS and $\mathcal{LEO}$.

- *MGs* are used to provide witnesses for existentially quantified variables or counter-models that show that some subgoal is not a theorem. Among others, SATCHMO and SEM are currently used.

- *CSs* are employed to construct mathematical objects with theory-specific properties. ΩMEGA employs $\mathcal{CoSIE}$ [56], a constraint solver for inequalities and equations over the field of real numbers.

Obviously the integration of CAS results into ΩMEGA's proof development is of special interest to this work. As with other external systems the only requirement to integrate a CAS is the availability of a protocol of its computations that can be evaluated and transformed into a sub-proof to be integrated in the $\mathcal{PDS}$. However the more knowledge about a system

is available to ΩMEGA, the better it can be integrated into the process of proof development, especially with respect to automated proof search and suggestion mechanisms. As CASs are complex systems that are applicable for various purposes, providing detailed information on their capabilities for proof search guidance and suggestion mechanisms is crucial to the exploration of the power of these systems in proof development.

The implementation of ΩMEGA is based upon the KEIM [40, 60] library of algorithms and data structures for automated theorem proving. It is implemented in CLOS [42], which is an object oriented extension of the COMMON LISP [73] programming language.

Chapter 3

Traceable Polynomial Normalisation in MASS

3.1 Motivation

In this chapter I will introduce the MASS system, which is a prototype computer algebra system. The reason to implement the MASS system was the development of a simple but universally applicable algebra system that is fully traceable. Its functionality extends that of the μCAS system described by Sorge [71], and its integration into the ΩMEGA system is likewise based on the same interface, SAPPER.

The μCAS system was the first Computer Algebra System in the environment of the ΩMEGA system that was fully integrated and provided a trace of its computations along with the possibility to remodel these computations in ΩMEGA's proof data structure. As its purpose however was to evaluate the operability of a closer integration of Computer Algebra and deduction through the SAPPER interface, its abilities are restricted mainly to what is needed to solve the examples described by Sorge [71], i.e. to addition, multiplication and differentiation.

Thus the reason for the implementation of the MASS system was to overcome the obvious weaknesses of the μCAS system in order to obtain a more robust and generally applicable tool for automated theorem proving. To do so, the focus of the system is not, like in the μCAS system, the execution of specified computations over polynomials in normal form, but in first line the normalisation of polynomials in any representation, i.e. every computation by the MASS system starts by normalising its arguments. This improvement turned out to remove the biggest obstacle for a use in practical theorem proving, as in everyday mathematics polynomial expressions usually do not occur in a normalised form and thus the μCAS system was applicable to standardised problems only. MASS however is able to handle nonstandardised polynomials, too, and furthermore the normalisation of polynomials is already a useful tool in many situations, e.g. equality of two polynomials can be established by syntactic equality of their normal forms.

A further improvement is to allow occurrences of non-interpreted function symbols, i.e. function symbols that cannot be semantically evaluated by the Computer Algebra System. By allowing these non-interpreted functions, it is still not possible to solve problems if special semantic knowledge about these functions would be needed for a solution, but there are many cases where it is sufficient or at least helpful to rewrite the argument expressions of

such functions. For an example, MASS does not use yet any special knowledge of the function log, so the CAS is not able to verify the equation

$$\log(1) = 0,$$

but by manipulating polynomials it is possible to verify the equation

$$\log(x^2 - y^2) = \log((x + y)(x - y))$$

Finally the applicability of the MASS system was enhanced employing a further mode of use. Its predecessor, the μCAS system was in general used as the sole CAS to solve algebraic parts of the problems, and as such it was responsible for both finding the result of a computation and verifying it by providing the computation's trace that was translated into ΩMEGA's proof representation $\mathcal{PDS}$. A further approach is to use the traceable CAS in combination with the commercially available MAPLE system. A feasibility study using μCAS is described by Sorge [72]. As both the robustness and the applicability of μCAS are very limited, the feasibility has been shown, but only few theorems could be proved in practical use. This approach was now pursued using the new MASS system, and a considerable increase of the number of theorems that can be proved has been achieved.

The capabilities of MAPLE, of course, go far beyond the capabilities of the experimental MASS system, but MAPLE does not provide a formal justification of its computations. The solution here was to share the task between both systems: First the MAPLE system is employed to find the solution of a given problem, then this result is verified by making use of MASS' white box behaviour. As it is in many cases much more difficult to find the solution to a problem than checking this solution, this allows to use the power of the much more efficient MAPLE system without threatening the correctness of the resulting proof nor having to reimplement the respective algorithms in a white box CAS. An example is the factorisation of polynomials, which is a standard task for MAPLE and can be justified by the much simpler multiplication of the resulting factors by MASS.

The result of these improvements was a robust system with a considerably widened applicability whose computations can be translated into partial proofs in ΩMEGA's proof data structure and thus are fully automatically checkable. MASS is now available in the ΩMEGA environment and is used for various purposes that require verifiable algebraic support and was used e.g. in the experiments described in [53] and [57].

Unlike its predecessor μCAS, the development of MASS was backed by an authoring tool to maintain the common mathematical knowledge base of CAS and deduction system. The importance of this common knowledge base is pointed out by Homann and Calmet [38]. During the development of MASS, the authoring tool TACO, described in chapter 4, was employed.

MASS is a collection of algorithms for manipulation of polynomials. MASS offers a set of simple algorithms for polynomial manipulation such as polynomial normalisation, polynomial addition and subtraction, polynomial multiplication and to a limited extent equality testing. It can be accessed by the ΩMEGA system via the SAPPER interface [71]. Beneath the application of algorithms MASS's second main goal is to make these computations fully available to a deduction system, so MASS offers not only the result of a computation, but is also able to produce a trace of computational steps that were performed. This can be used for proof extraction, so any application of a MASS algorithm can be reconstructed in terms of inference

rules within a deduction system and its results can be fully incorporated within a proof plan without threatening the correctness of the proof.

MASS is actually implemented as a standalone system. Its computations use a proprietary data structure which is adapted to the needs of efficient polynomial manipulation, and its functionality does not rely on any external resources. However, as MASS's main benefit is the capability to make algorithmic computations reconstructible while its efficiency and computational power is weak compared to commercially available computer algebra systems as MAPLE, MASS's practical use has to be seen mainly as a supporting system for a deduction system. Furthermore proof extraction from a MASS trace requires a close interaction between both the computer algebra system and the deduction system, especially a common mathematical knowledge base available to both systems is necessary. Therefore the current implementation of MASS is accessible only through an interface to the ΩMEGA system: MASS can be accessed via the SAPPER interface, and all input and output is translated from respectively to POST syntax. The implementation of a dedicated front-end for the MASS system was omitted.

The MASS system was implemented in CLOS [42], which is an object oriented extension of the COMMON LISP [73] programming language. Furthermore the adaption of the SAPPER interface and the implementation of the translator to POST syntax makes use of the KEIM [40, 60] library of algorithms and data structures for automated theorem proving, and some of the functionalities of the ΩMEGA system.

3.2 Data Structure

All computations performed by MASS are based on a proprietary data structure. As the main purpose of the MASS system is the manipulation of polynomials and as it is intended to be used in close interaction with the ΩMEGA system, the focus of the design of this data structure was laid upon two main aims:

First the data structure should adequately represent specific structural features of polynomial terms, and the implementation of data access and data manipulation should be adjusted to the requirements of algorithms for polynomial manipulation, i.e. it should e.g. support an efficient sorting of sums.

Second a crucial point of the interaction between MASS and ΩMEGA is the translation of terms. The underlying data structure of ΩMEGA are terms in KEIM that are represented in POST syntax, so a translation of MASS terms into POST syntax and vice versa has to be provided. MASS terms represent polynomials in normal form, thus, while a translation of MASS objects to POST syntax is quite straight forward, a translation of POST terms to MASS objects is much more difficult and requires the use of MASS's algorithm library (see later for further explanation).

The result to fulfil these two aims is a data structure that models structural features of polynomial normal forms, but is also based on the use of POST primitives.

3.2.1 Structure

The focus of the MASS system is on simple manipulations of polynomials. Generally a polynomial is a sum of a product of powers of variables multiplied by a coefficient, i.e. a polynomial $\mathcal{P}$ can be represented as:

class	attributes	semantics
`mass+term`	a list of `mass+termatoms`	the sum of the elements of the list
`mass+termatom`	a `mass+monom` m and a numerical coefficient c	m multiplied by c
`mass+monom`	a list of `mass+monatoms`	the product of the elements of the list
`mass+monatom`	a function $func$ and a list of arguments $args$	depending on $func$

Figure 3.1: The MASS Data Structure's Object Classes

$$\mathcal{P} = \sum_{i=1}^{n} c_i \cdot \prod_{j=1}^{m} b_{i,j}^{e_{i,j}} \tag{3.1}$$

where c_i are coefficients of the monomials, which consist of the primitives $b_{i,j}$, i.e. constants or variables, and $e_{i,j}$ their appendant exponents.

The MASS data structure is an object oriented implementation of this representation of polynomials. The components of this notation are mirrored quite straight forward by object classes of MASS's data structure. A MASS term is composed from the following classes with their respective semantics:

Generally the classes `mass+term`, `mass+termatom` and `mass+monom` are used to denote a polynomial's structure, while the class `mass+monatom` is the equivalent of the powers of primitives in formula 3.1.

So the classes `mass+term`, `mass+termatom` and `mass+monom` are mainly employed to model computations involving commutativity, associativity and distributivity over addition and multiplication, which can be implemented e.g. by sorting of lists, while the class `mass+monatom` fulfils several purposes. The function of a `mass+monatom` is determined by its attribute $func$, which may have one the following values:

- `expt` denotes the simplest form of a power, where the base is a POST primitive. To avoid collisions in variable and constant naming, POST primitives are employed at this position without further modification. The applicability of commutativity, associativity and distributivity under addition and multiplication to these primitives is assumed as a precondition.

  The exponent may be any term in MASS syntax.

- `power` also denotes a power, but in this case the base may also be any MASS term. The distinction between `expt` and `power` is necessary, because term simplification is MASS's principal purpose, and the use of `power` instead of `expt` signals a potentially simplifiable object when involved in further computation.

- `unknown` denotes the application of a function on which no further knowledge is available to MASS, e.g. the application of a logarithm. If encountering any such function, MASS will encapsule the function in a `mass+monatom`, but will, if possible, simplify its arguments.

argument	**result**
`mass+term(term:[])`	0
`mass+term(term:[t])`	`rebuild(t)`
`mass+term(term:[t1 ... tn])`	`rebuild(t1)` + ... + `rebuild(tn)`
`mass+termatom(coeff:1 monom:m)`	`rebuild(m)`
`mass+termatom(coeff:c monom:m)`	if `rebuild(m)` = 1: c otherwise: $c *$ `rebuild(m)`
`mass+monom(monom:[])`	1
`mass+monom(monom:[t])`	`rebuild(t)`
`mass+monom(monom:[t1 ... tn])`	`rebuild(t1)` * ... * `rebuild(tn)`
`mass+monatom(func:expt args:[a1 a2])`	$a_1^{a_{r2}}$, where $a_{r2} =$ `rebuild(a2)`
`mass+monatom(func:power args:[a1 a2])`	$a_{r1}^{a_{r2}}$, where $a_{r1} =$ `rebuild(a1)` and $a_{r2} =$ `rebuild(a2)`
`mass+monatom(func:unknown args:[f a1 ... an])`	$(f_r a_{r1} ... a_{rn})$, where $f_r =$ `rebuild(f)`, $a_{r1} =$ `rebuild(a1)`, $\vdots$ $a_{rn} =$ `rebuild(an)`

Figure 3.2: Rebuilding a POST Term from a MASS Object.

It is to be noticed, that the representation of powers $b_{i,j}^{e_{i,j}}$ is somewhat more complex than denoted in equation 3.1, namely the $b_{i,j}$ is not necessarily a primitive but can be any term. This means that, actually through the implementation of the `term+monatom` class, the MASS data structure can be recursively nested, so that MASS has to be able to cope with polynomial structures at any position of a term.

Furthermore MASS terms can represent any POST term in a manner that will adequately handle polynomial structures occurring in the respective term, but will leave anything else untouched. In general the data structure will follow the paradigm of being useful without being a nuissance, which means that the data structure should support an efficient handling of polynomials whenever possible, but will not lead to critical behaviour of the MASS system, when trying to translate anything non-polynomial to MASS representation.

3.2.2 Translation

The translation of MASS objects to POST syntax is quite straight forward; it is implemented in the `rebuild` function. An application of `rebuild` to a MASS object will have the results given in figure 3.2.

To translate a POST term to a MASS object, however, requires considerably more effort. This is due to the fact, that the MASS system keeps the normal form described above as an invariant while operating on terms. So generating a MASS object, which is in normal

form, from a POST term, which is in general not, makes already up a considerable part of the simplification performed by the MASS system, and thus the translation of a POST term to a MASS object has to make extensive use of the MASS algorithm library. The translation is computed by the `read` function.

Technically the application of the `read` function is based on structural recursion: Whenever `read` encounters a POST primitive, i.e. a constant, a variable or a number, in which case we can assume that it is in normal form, `read` generates the equivalent MASS object.

Whenever encountering an application of a function to its arguments, `read` will make a distinction on whether or not there is mathematical knowledge available to the MASS system on how to handle this function.

If the function is known to the system, `read` will be recursively applied to the function's arguments, i.e. the result will be a MASS polynomial in normal form. Then `read` will interpret the function as a CAS command, e.g. if `read` is given the term $p_1 + p_2$ it will first process the arguments p_1 and p_2 and will then apply the appropriate algorithm, in this case the algorithm for polynomial addition, from the MASS algorithm library to the results of this argument preprocessing.

If the function is unknown to the MASS system, `read` will also process its arguments first, but then it will encapsulate the whole term in a `mass+monatom`, so that the term will be normalised as far as possible and correctly represented in spite of the use of unknown functions.

Considering the example translation of the term

$$(x + 1) * (x + 1 + \log(2 * (x + 1)))$$

to a MASS object, `read` will show the behaviour depicted in figure 3.3.

The result of this application of `read` is in normal form according to formula 3.1, and the MASS algorithm library was employed several times to perform operations on normal forms, like addition or multiplication of polynomials.

3.3 Algorithms

The power and efficiency of any Computer Algebra System, be it experimental or commercial, depends to a great portion on the algorithms that are implemented at its core; the greater the library of available algorithms is, the wider is the variety of problems that can be solved, and the more careful their implementation is, the less is the rate of errors and bugs the CAS will produce. Thus an implementation of a Computer Algebra System should be preceded by a careful consideration on what algorithms should be provided, how these algorithms should be realized and how they will fit in the overall system.

Unlike commercially available CAS, e.g. Maple, which are mostly very powerful all purpose engines and set a focus on broad applicability and high efficiency, the purpose the MASS system is designed for comprises a rather small area of operation and has very specific requirements. MASS is designed for an application in interaction with the ΩMEGA system, and its duty are typically simple manipulations of polynomials. The requirements set by this focus are correctness and reconstructible justifications of the performed computations rather than high efficency or a widespread applicability. Thus we can afford and have to restrict the algorithms provided by the MASS algorithm library to a small and well specified class of algorithms.

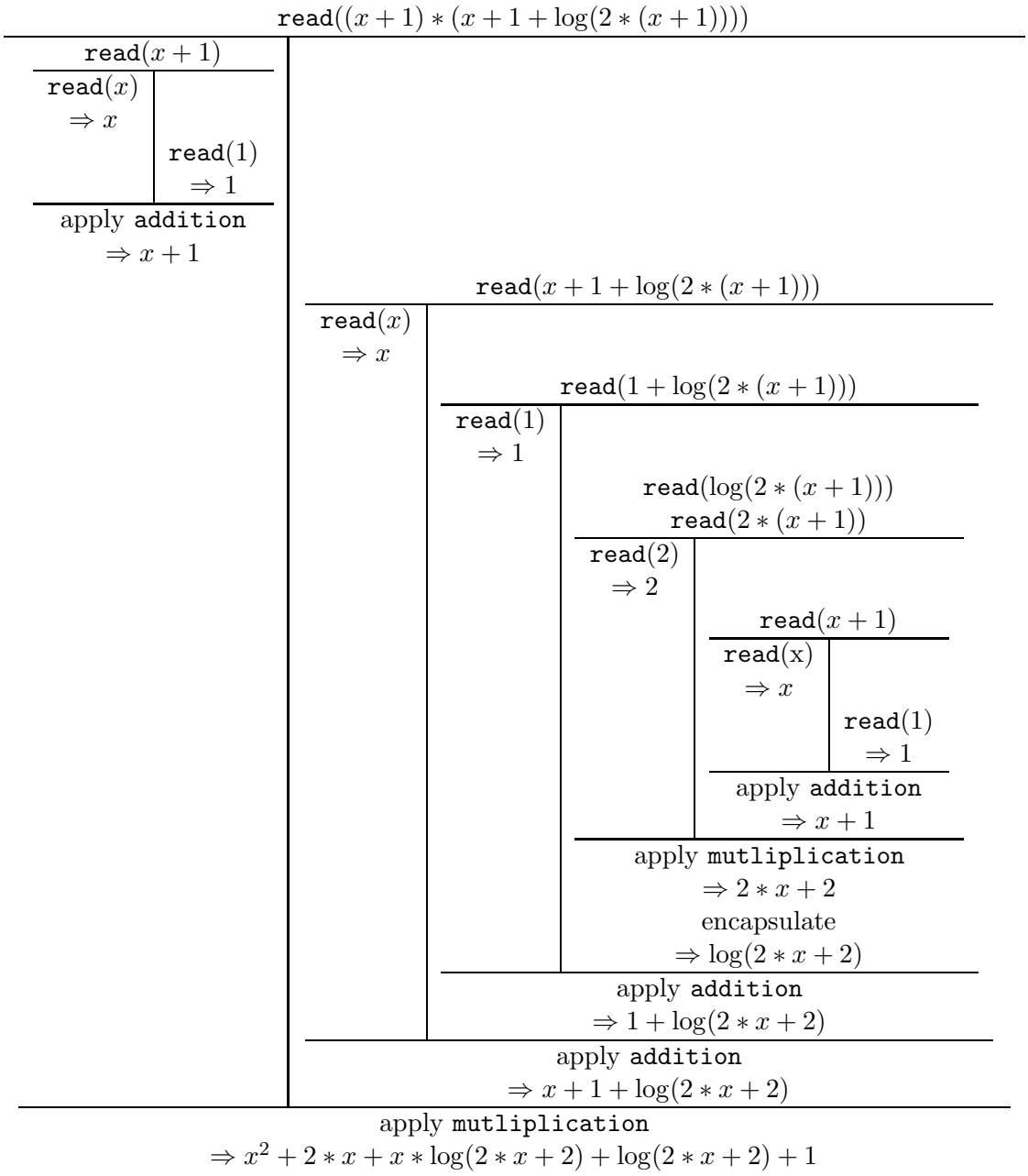


Figure 3.3: Building a MASS Object from a POST Term.

To provide a frictionless interaction with the deduction system, the interface SAPPER and its functionality has to be considered, too. A typical SAPPER call of the MASS system consists of a CAS command along with its arguments. The CAS command will be executed by the MASS system and the result will be returned to the interface. Additionally the SAPPER interface may set the integrated CAS into a so called verbose mode, which means that the CAS is forced to return not only a computation's result, but also a trace that can be used to reconstruct a logical justification of the outcome. The concept pursued by the MASS system is that of an abstract trace of the computation performed by the CAS. If set to verbose mode, the CAS will log each computational step it performs, actually the CAS will in each step output an abstract representation of what was computed in this step. This representation consists of an identifier of the computational step that was performed along with its parameters, if there are any. For an example think of the CAS sorting a sum, which usually can be justified by repeated application of the rules of associativity and commutativity. If sorting e.g. the term $((z+x)+y)$ to obtain $(x+(y+z))$, the CAS will continuously output the abstract identifiers of the step it is actually performing. In the simple example shown in figure 3.3 we see the abstract identifier of each step next to the term that is obtained by applying the respective computational step to its predecessor.

term	**CAS step**
$((z+x)+y)$	
$((x+z)+y)$	`comm+ [1]`
$(x+(z+y))$	`assoc+ []`
$(x+(y+z))$	`comm+ [2]`

Figure 3.4: A Term Manipulation and its Trace.

In this case the CAS trace is the list [`comm+ [1]`,`assoc+ []`,`comm+ [2]`], i.e. it is a list containing the abstract identifiers `comm+` and `assoc+` and their respective arguments denoting an application of the rules of associativity respectively commutativity under addition. Along with this identifier a single parameter is returned, actually the position within the term at which the rule was applied. The objects of the trace are collected by the SAPPER interface and, again step by step inserted into the deduction system's proof plan to fill the computational gap left by the CAS application. To do so the objects contained in the trace are successively translated into proof steps of the deduction system's calculus, in the current implementation of MASS into an applicable sequence of ΩMEGA tactics; actually each identifier represents a fixed computational step that can be modelled by the application of an tactic or a fixed sequence of tactics within the deduction system's proof plan. This mapping of identifiers to tactics is computed by the SAPPER interface, which thus is able to mirror the computation performed by the CAS in terms of proof steps of the Deduction System. For further explanation see also section 3.5.

Concerning the deduction system's proof plan we will encounter the following situation (without restriction of generality I will only consider a forward application of the CAS, backward application of the CAS is analogous). Say the proof plan contains a line L_{prep} that should be simplified using the MASS system, the formula in L_{prep} is $\Phi(a)$ where a is the subterm we want to be simplified by the CAS, and b is the result of this simplification. So a new proof line L_{conc} will be inserted into the proof plan:

L_{prep}	$\Phi(a)$	
L_{conc}	$\Phi(b)$	CAS L_{prep}

The justification of line L_{conc} is the application of the CAS on line L_{prep} with additional parameters $args*$, i.e. whether L_{conc} is a correct consequence in terms of the logical calculus underlying the deduction system depends of the CAS' implementation. However, from the abstract trace the MASS system supplies when in verbose mode, it is possible to reconstruct the computation at calculus level. To do so is the task of the SAPPER interface. The SAPPER interface translates each computational step in the trace together with its arguments, if there are any, into one tactic or a sequence of tactics of the deduction system. A sequential application of these tactics will now replace the abstract justification of line L_{conc} in the proof plan. This refinement of the proof's justifications is performed by the ΩMEGA proof line expansion, which is also used to justify the result of a complex tactic by a sequential application of simpler inference steps. Supposed the subterm a in our example was $((z + x) + y)$ and this term was transformed to $(x + (y + z))$ by the CAS, which also returned the above abstract trace, our example could be expanded to:

L_{prep}	$\Phi((z + x) + y)$	
L_1	$\Phi((x + z) + y)$	COMM+ L_{prep} pos|1
L_2	$\Phi(x + (z + y))$	ASSOC+ L_1 pos
L_{conc}	$\Phi(x + (y + z))$	COMM+ L_2 pos|2

where COMM+ and ASSOC+ are tactics of the deduction system that justify a term rewrite according to the rules of commutativity and associativity. Besides relating the abstract identifiers of the MASS trace to actual tactics, the SAPPER interface also adjusts the parameters of each computational step to the environment of the deduction system, in this example we can see that the relative position of the application of a computational step was transformed into an absolute one, in this case the position of the subterm that is passed to the CAS, here pos, has to be concatenated with the relative positions the CAS returns in its trace.

Our approach to integrate algorithms within a logical environment is to return a result that can be justified employing a step by step justification of the computation. Integrating an algorithm this way includes that the implementation in the MASS system requires to keep track of the logical foundations of the algorithm during the computation, and at each computational step we have to keep track of when and how they have to be applied. The need to involve the foundations of an algorithm into its implementation can lead to a much more complex and costly implementation than the coding of the bare algorithm. This typically applies for efficient algorithms that make use of implicit mathematical principles or whose correctness is based on consideration on their runtime behaviour (e.g. if an algorithm is proved to be correct using arguments like 'this is the greatest number to fulfil this predicate, because if there was a greater one, the algorithm would have returned it in a previous iteration', see section 3.7 for further discussion). In other words there are algorithms that are suiting our approach better than others.

Second, due to the interface we use and in favour of the standalone idea, the trace output is restricted to linearised term manipulation, i.e. the computation is expressed by a sequence of rewrite steps, each of which is applied to the result of its predecessor, and all proof line management is left to the interface respectively the deduction system. So we can keep the MASS

system's trace output independent from the data structure that is used for proof presentation and thus obtain a high portability when planning to use the MASS system in interaction with deduction systems other than ΩMEGA; the simplicity of the output trace makes an adaption to different systems or logical calculi easy (see also section 3.5).

Thus we can settle a standard on how to implement algorithms for the MASS algorithm library. The algorithms provided by this library are independent from the deduction system's logical calculus and the deduction system's mathematical knowledge base, but rely on set of computational steps that are known to the SAPPER interface and that can be modelled and justified in the deduction system's calculus (of course this set of computational step may have to be expanded when expanding the algorithm library, see also section 3.5). From the logical point of view the role of the algorithm is to provide the control of a computation, but no property of the algorithm will contribute to the justification of the result. Furthermore every algorithm returns its result along with a sequence of abstract computational steps that can be translated into a sequence of inference rules respectively tactics that reconstruct the computation in the context of the deduction system's calculus.

In general an algorithm can be denoted as a function f so that an application to a term x and possibly additional parameters $arg*$ will have the following result:

$$f(x, arg*) = y \times [\mathcal{R}_1 \dots \mathcal{R}_n]$$

where the translation of the trace can be integrated into a proof plan

L_{prep}	$\Phi(x)$	
L_{conc}	$\Phi(y)$	CAS L_{prep} $arg*$

such that

L_{prep}	$\Phi(x)$	
L_1	$\Phi(x_1)$	$\mathcal{R}'_1$
L_2	$\Phi(x_2)$	$\mathcal{R}'_2$
	$\vdots$	
L_{conc}	$\Phi(y)$	$\mathcal{R}'_n$

where $\mathcal{R}'_i$ are inference rules that result from SAPPER translating the abstract steps $\mathcal{R}_i$ and proof lines $L_1 \dots L_{conc}$ are justified by the application of the respective inference rule to its predecessor.

The core of MASS' library of algorithms is a simple mechanism for normalisation of polynomials which implemented this way. It is based upon a set of algorithms which implement basic arithmetic operations on polynomials, e.g. addition or multiplication of two polynomials, and return the result along with the according trace. Apart from these basic operations, MASS provides a mechanism to lexicographically reorder expressions, e.g. to rewrite the expression $c + a + b$ by $a + b + c$, which again supplies a trace of the necessary computational steps. The normal form of a polynomial is computed recursively: If e.g. the head symbol of the POST expression to be normalised is `plus`, its two arguments are recursively normalised, then MASS' algorithm for polynomial addition is invoked. The recursion is repeated until the arguments of the head symbol are primitive terms, i.e. variables, constants or numbers, which are by definition in normal form. The result of MASS' basic arithmetic operations is in normal form again. The mechanism for lexicographic reordering is used to establish equality

of subterms of the expression if required, e.g. to normalise $2{\cdot}a{\cdot}b + 3{\cdot}b{\cdot}a$, the subterm $b{\cdot}a$ is reordered, then the result $5{\cdot}a{\cdot}b$ can be computed by applying distributivity and adding the coefficients 2 and 3. Lexicographical reordering may furthermore be applied to establish syntactic equality to solve equations by reflexivity of the equality relation.

3.4 Architecture

This section describes the MASS system's components and their functionality as well as its interaction with the interface SAPPER. Note that although their is no proprietary front end to access MASS, the idea was to implement a quasi standalone system that can also be adapted to interact with deduction systems other than ΩMEGA. However ΩMEGA is the only system MASS is interacting with at the moment, so it will serve to demonstrate the deduction system's part of the architecture in the following.

The MASS system is integrated into the ΩMEGA system via the SAPPER interface, which is a generic interface and can serve as an interface to one or several CAS that are to be accessed by ΩMEGA. The purpose of the SAPPER interface is to provide the communication between both systems; the SAPPER interface allows this without requiring modification of neither the logic used in the ΩMEGA system nor the schemata of the algorithms implemented in the CAS. To achieve this the interface has to play the role of a translator for terms and commands that are passed from one system to another, and it also has to fulfil the task of proof plan manipulation that is needed to generate partial proof plans from the abstract traces the CAS outputs and to insert them into ΩMEGA's proof in the right place.

From the deduction system's point of view SAPPER provides an abstract representation of the CAS that are connected to it, and there are two ways to use these CAS: First they can be used as a *black box*, which means that SAPPER will call the CAS to typically simplify a term, and the CAS will return only the result of its computation, which is then passed to the ΩMEGA system. The second way is called *verbose mode* and causes the CAS to return also a trace of its computation, which will be translated by the interface into ΩMEGA proof plans and inserted to expand these steps in the proof whose justification is the application of the CAS. Here the duty of the SAPPER interface consists of the following tasks:

- passing POST objects to the CAS
- passing CAS objects back to ΩMEGA
- mapping of ΩMEGA function symbols to CAS commands
- proof plan generation from abstract computation traces

The two main components of the SAPPER interface to carry out these tasks are the translator and the tactic generator (see figure 3.5).

The translator ensures the translation of POST arguments that are passed to the CAS, and it invokes the appropriate CAS algorithm. The MASS system's purpose is the normalisation or simplification of a given term, thus to find the appropriate CAS algorithm to do so a mapping of possible head function symbols to CAS algorithms is employed.

The CAS, when called by the SAPPER interface, will execute the command it is given and, depending on whether it is called in verbose mode or not, will either only return the result of its computation or also output an abstract trace of the computational steps it performs

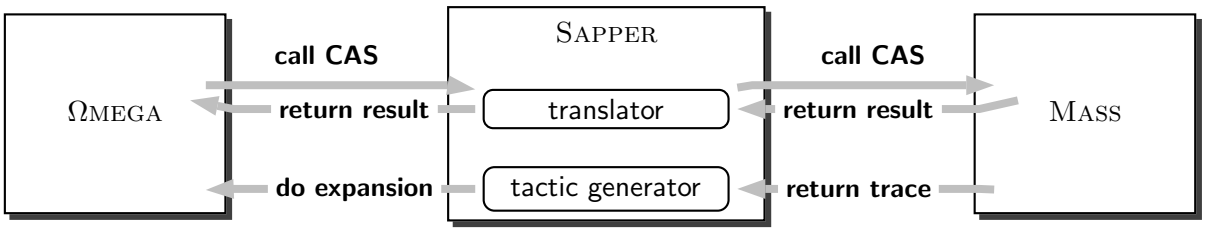


Figure 3.5: Functionality of the SAPPER Interface.

doing so, which are then collected by the SAPPER interface. Here the translator's duty is the retranslation of the result to POST syntax and the appropriate manipulation of ΩMEGA's proof plan. The tactic generator, if the CAS is called in verbose mode, collects the abstract computation trace steps returned by the CAS and eventually uses this trace to expand and justify the result of the CAS's computation within ΩMEGA's proof plan.

In the case of the MASS system this is realized by providing the module `reader` to translate POST terms to MASS objects and the module `rebuilder` to translate MASS objects to POST syntax, furthermore the SAPPER interface is able to access the MASS algorithm library. To simplify an POST function using MASS, the interface will map the function to the according algorithm from the MASS library, which is then applied to the translated arguments.

As already the creation of a MASS object from a POST term implies a simplification of this term where the MASS algorithm library is involved, the MASS system does not only have to provide a trace of the computation that is initiated by the command which is passed by the SAPPER interface, but it has to provide the trace of the argument's translation as well. Thus a computation performed by the MASS system consists of two cycles: in the first cycle (see figure 3.6) MASS will translate the arguments of a function. This is done by the `reader` module, which uses the MASS algorithm library to normalise its input terms. While doing so the algorithm library outputs a trace of its computation whenever needed. This trace is passed to the SAPPER interface where it is eventually used to expand and justify the term simplification performed so far.

The second cycle is the execution of the command that is passed by the SAPPER interface (see figure 3.7). Now the appropriate algorithm from the MASS library is invoked by the SAPPER interface and is applied to its already preprocessed arguments, which now are present as MASS objects in normal form. During this computation cycle the operation of the MASS algorithm library is analogous to the cycle of argument preprocessing, whenever an algorithm is applied it will output an abstract trace of the computation it performs.

Furthermore the result of the algorithm invoked by the SAPPER interface is passed to the MASS `rebuilder` and is retranslated to POST syntax.

Note that every computational step that is performed by an algorithm from the library is logged in the abstract trace, be it during the translation of the arguments or be it while executing a command, and that the whole trace is needed to reconstruct MASS's computation at calculus level

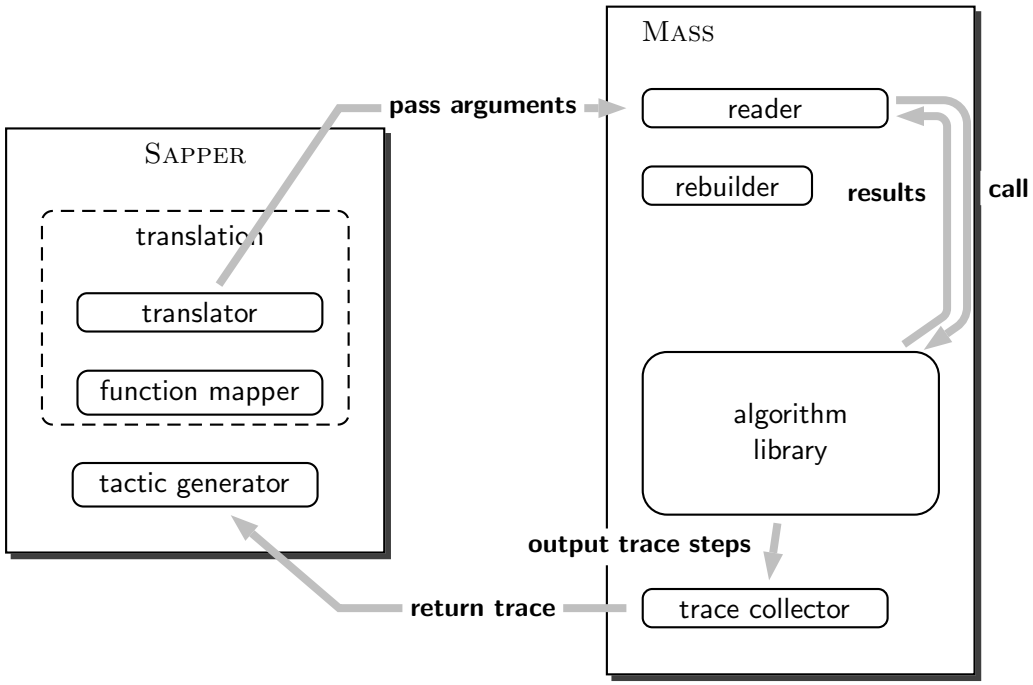


Figure 3.6: Translation of the Arguments of a MASS Command.

3.5 Integrating MASS in the SAPPER Interface

The SAPPER module is the interface that connects external Computer Algebra Systems to the ΩMEGA prover. To provide a frictionless interaction between CAS and theorem prover, the interaction has to follow clearly defined standards. While the concept of the interaction has already been described in section 3.4, this section will give a more detailed description of the actual functionality of the SAPPER interface.

The purpose of the SAPPER interface is first to translate requests from the ΩMEGA theorem prover and to pass them to the CAS. In general this requires to

- map the head function symbol of the term to be simplified to the appropriate CAS command, and to
- translate the argument terms of this function from the theorem prover's representation to a representation that is understood by the CAS.

Note that from the theorem provers point of view, a call to a CAS via the SAPPER interface has always the purpose to simplify respectively normalise a term, the kind of computation to be executed to do so depends on the structure of the term that is passed by the theorem prover. In the further proceeding it is assumed that the term can be simplified by the chosen CAS whenever the SAPPER interface can map the term's head function symbol to a corresponding CAS command, e.g. if the theorem prover passes the term `(plus t1 t2)` to the SAPPER interface, this is interpreted as a call to the CAS's implementation of addition, if there is one, applied to the arguments `t1` and `t2`, where these arguments have to be translated first.

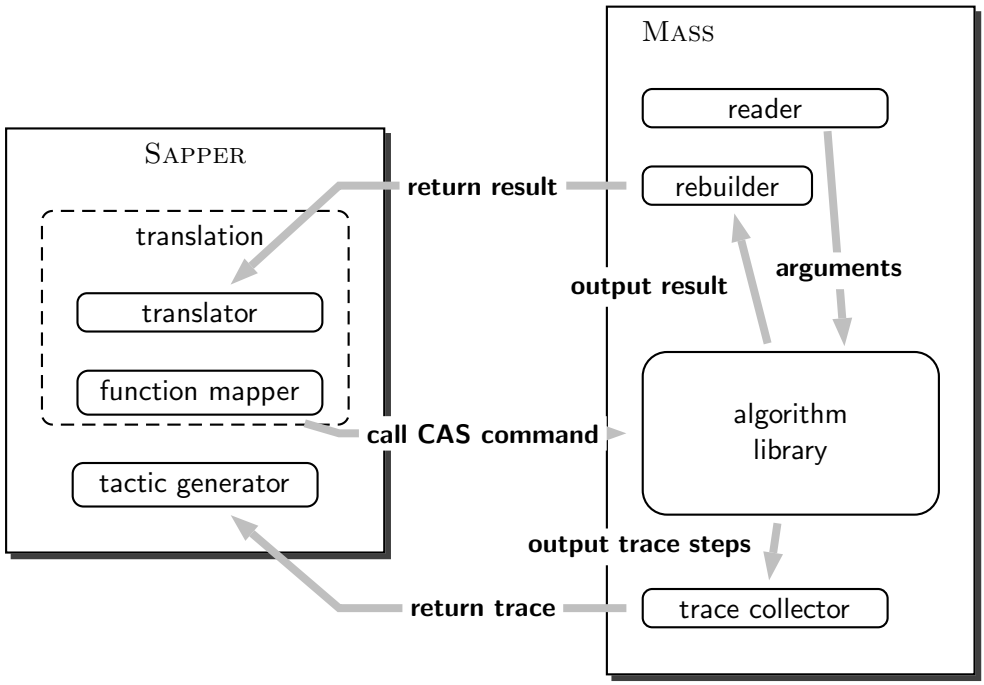


Figure 3.7: Execution of a MASS Command.

This should in general cause the CAS to start its computation and to return the resulting term and a trace of its computation. Now the SAPPER interface's duty is to retranslate the CAS's output to make it available to the theorem prover. Again we have two tasks to fulfil:

- to retranslate the CAS's result term to a representation that is understood by the theorem prover
- to translate the trace of the CAS to a sequence of inference rules that are represented in the theorem prover's knowledge base, and to apply these inferences in the theorem prover's proof plan. This is only required when a full expansion of the CAS's computation to an explicit sequence of inference rules is intended.

Apart from translation, the SAPPER interface takes care of the proper integration of the CAS's computation into the proof plan, i.e. the CAS's result is not only translated to the theorem prover's representation, but the SAPPER interface furthermore adds a new proof line to represent this result within the proof. When expanding a CAS computation, the SAPPER interfaces again takes care of the administration of proof lines, e.g. to apply inference rules in the right order and to establish the correct dependencies between proof lines.

To make a CAS available via the SAPPER interface requires an abstract specification of the CAS along with the algorithms it provides and the functions that translate POST syntax into the CAS's proprietary data structure and vice versa. This specification provides the name of the CAS and a mapping of POST functions to function names used for the CAS algorithm and the translation functions that have to be applied in the respective case. For MASS, this specification looks like this:

```
(ca~defsystem :MASS
 (help "The MASS system")
  (translations
   (plus  (mass~plus  (:mass-term :mass-term2 :mass-term1)))
   (times (mass~times (:mass-term :mass-term2 :mass-term1)))
   (minus (mass~minus (:mass-term :mass-term2 :mass-term1)))
   (power (mass~power (:mass-term :mass-term2 :mass-term1)))
   (div   (mass~div   (:mass-term :mass-term2 :mass-term1)))
   (sqrt  (mass~sqrt  (:mass-term :mass-term1)))
   (mod   (mass~mod   (:mass-term :mass-term2 :mass-term1)))
   (=     (mass~equal (:mass-term :mass-term2 :mass-term1))))
 (call eval))
```

If a POST expression as e.g. the head function symbol `plus`, the corresponding MASS algorithm is looked up from the table. The MASS function which implements the normalisation of an expression with head function `plus` is `mass~plus`. This function is called after its arguments have been translated. The indicators `mass-term`, `mass-term1` and `mass-term2` are used to select the proper method of the translation functions. For MASS, the following translation function methods are defined:

```
(defmethod ca~build-object
   (term (object (eql :mass-term1)) (system (eql :mass)))
  (ca~output-method '((:forward . init-mass-f)
                      (:backward . init-mass-b)))
  (mass~read term (pos~list-position '(1))))

(defmethod ca~build-object
   (term (object (eql :mass-term2)) (system (eql :mass)))
  (mass~read term (pos~list-position '(2))))

(defmethod ca~build-object
   (term (object (eql :mass-term3)) (system (eql :mass)))
  (mass~read term (pos~list-position '(3))))
```

Note that these function fulfil, apart from translation, two further purposes: First an initial method is required for correct integration of MASS's result into the proof plan. This method is inserted into the trace of the computation by

```
  (ca~output-method '((:forward . init-mass-f)
                      (:backward . init-mass-b)))
```

Second, further information that is required for remodelling the computation, in this case the position of the subterm to be translated, are passed to the translation function. As the normalisation algorithm of MASS is already used in translation of the arguments, the relative position of the argument has to be passed to MASS. Here the indicators `mass-term1` and `mass-term2` are used for subterms at the position of the first respectively second argument of an application.

The final function to be defined is the retranslation function:

```
(defmethod ca~rebuild-object
   (term (object (eql :mass-term)) (system (eql :mass)))
  (mass~rebuild term))
```

which completes the abstract specification of MASS as it is seen from ΩMEGA.

3.6 Application Examples

The MASS system can be applied in several modes in interactive proof development and automated proof planning. In all modes MASS is accessed via the SAPPER interface that provides different functionalities to make calls to a CAS from ΩMEGA.

3.6.1 As a Standalone System

In interactive mode, MASS can be called by the user to simplify an expression in a specified proof line at a specified term position. In this case, MASS operates as a standalone system and is called once to provide the result of the required computation, and a second time in verbose mode to provide the computation's trace that is used to generate a sub-proof during the expansion. The command in ΩMEGA to call an external CAS in this way is

```
CALL-CAS line pos system
```

where `line` is the proof node that is to be simplified, `pos` is the relative term position of the expression to be simplified within the formula of `line`, and `system` is the symbol of the CAS that is intended to be applied, in the following it will be `mass`.

An example is the interactive proof of the binomial formula $(a + b)(a - b) = a^2 - b^2$ in ΩMEGA. The problem can be stated in a single proof line:

Binom. $\vdash ((A+B)\cdot(A-B)) = ((A\hat{}2)-(B\hat{}2))$ (*Open*)

Now MASS can be called to verify this line

```
CALL-CAS binom () mass
```

and yields the new proof plan

L1. $\vdash ((A\hat{}2)+(-1\cdot(B\hat{}2))) = ((A\hat{}2)+(-1\cdot(B\hat{}2)))$ (=Ref)

Binom. $\vdash ((A+B)\cdot(A-B)) = ((A\hat{}2)-(B\hat{}2))$ (Cas L1)

In this case, the equation could be solved by MASS building the normal forms of the expressions at both sides of the equation. In case a lexicographical reordering of the resulting polynomials is required to establish syntactical equality, this is done by MASS, too. If MASS succeeds to establish syntactical equality, the proof line can be closed by applying *=Ref*, i.e. the reflexivity of the =-operator. The actual computation performed by MASS in this example is revealed when expanding the proof plan.

L1.	$\vdash ((A\hat{}2)+(-1\cdot(B\hat{}2))) = ((A\hat{}2)+(-1\cdot(B\hat{}2)))$		(=Ref)
L21.	$\vdash ((A\hat{}2)+(-1\cdot(B\hat{}2))) = ((A\hat{}2)-(B\hat{}2))$		(Plus2minus L1)
L20.	$\vdash ((A\hat{}2)+(0+(-1\cdot(B\hat{}2)))) = ((A\hat{}2)-(B\hat{}2))$		(0+Intro L21)
L19.	$\vdash ((A\hat{}2)+((0\cdot(A\cdot B))+(-1\cdot(B\hat{}2)))) = ((A\hat{}2)-(B\hat{}2))$		(0*Intro L20)
L18.	$\vdash ((A\hat{}2)+(((-1\cdot(A\cdot B))+(A\cdot B))+(-1\cdot(B\hat{}2)))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Plus L19)
L17.	$\vdash ((A\hat{}2)+(((-1\cdot(A\cdot B))+(B\cdot A))+(-1\cdot(B\hat{}2)))$ $((A\hat{}2)-(B\hat{}2))$	=	(C-Times L18)
L16.	$\vdash ((A\hat{}2)+((-1\cdot(A\cdot B))+((B\cdot A)+(-1\cdot(B\hat{}2)))))$ $((A\hat{}2)-(B\hat{}2))$	=	(A-Plus-Right L17)
L15.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(-1\cdot(B\hat{}2))))$ $((A\hat{}2)-(B\hat{}2))$	=	(A-Plus-Left L16)
L14.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(-1\cdot(B\hat{}(1+1)))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Plus L15)
L13.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(-1\cdot((B\hat{}1)\cdot(B\hat{}1)))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Power L14)
L12.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(-1\cdot((B\hat{}1)\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Power-1-Elim L13)
L11.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(-1\cdot(B\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Power-1-Elim L12)
L10.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+((B\cdot A)+(B\cdot(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Times L11)
L9.	$\vdash (((A\hat{}2)+(-1\cdot(A\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Cummulate-Left L10)
L8.	$\vdash (((A\hat{}2)+(A\cdot(-1\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Times L9)
L7.	$\vdash (((A\hat{}(1+1))+(A\cdot(-1\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Plus L8)
L6.	$\vdash ((((A\hat{}1)\cdot(A\hat{}1))+(A\cdot(-1\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Split-Power L7)
L5.	$\vdash ((((A\hat{}1)\cdot A)+(A\cdot(-1\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Power-1-Elim L6)
L4.	$\vdash (((A\cdot A)+(A\cdot(-1\cdot B)))+(B\cdot(A+(-1\cdot B))))$ $((A\hat{}2)-(B\hat{}2))$	=	(Power-1-Elim L5)
L3.	$\vdash ((A\cdot(A+(-1\cdot B)))+(B\cdot(A+(-1\cdot B)))) = ((A\hat{}2)-(B\hat{}2))$		(Cummulate-Left L4)
L2.	$\vdash (A+B)\cdot(A+(-1\cdot B))) = ((A\hat{}2)-(B\hat{}2))$		(Cummulate-Right L3)
Binom.	$\vdash (A+B)\cdot(A-B)) = ((A\hat{}2)-(B\hat{}2))$		(Plus2minus L2)

In this proof plan, every proof step corresponds to a computational step performed by MASS. The elementary proof steps are tactics that do a term rewrite step according to the laws of commutativity (e.g. *C-Times*), associativity (e.g. *A-Times*) and distributivity (e.g. *Cummulate-Left*) or make use of the special role of the numbers 0 and 1, (e.g. *0+Intro*) and the tactics *Simplify-Num* and *Expand-Num* to rewrite simple numerical expressions. Besides tactics that implement these basic laws of algebra, there are tactics that combine a sequence of such tactics within a single proof step. An example is the tactic *Split-Monomials-Plus*, used here to derive *L18* from *L19*. In this example, the expression $0\cdot A\cdot B$ is rewritten as $-1\cdot A\cdot B + A\cdot B$ in a single proof step in the following lines:

L19.	$\vdash ((A\hat{}2)+((0\cdot(A\cdot B))+(-1\cdot(B\hat{}2)))) = ((A\hat{}2)-(B\hat{}2))$		(0*Intro L20)
L18.	$\vdash ((A\hat{}2)+(((-1\cdot(A\cdot B))+(A\cdot B))+(-1\cdot(B\hat{}2)))) ((A\hat{}2)-(B\hat{}2))$	=	(Split-Monomials-Plus L19)

This is expanded to:

L19.	$\vdash ((A\hat{}2)+((0\cdot(A\cdot B))+(-1\cdot(B\hat{}2)))) = ((A\hat{}2)-(B\hat{}2))$		(0*Intro L20)
L24.	$\vdash ((A\hat{}2)+(((-1+1)\cdot(A\cdot B))+(-1\cdot(B\hat{}2)))) ((A\hat{}2)-(B\hat{}2))$	=	(Expand-Num L19)
L23.	$\vdash ((A\hat{}2)+(((-1\cdot(A\cdot B))+(1\cdot(A\cdot B)))+(-1\cdot(B\hat{}2)))) ((A\hat{}2)-(B\hat{}2))$	=	(Distribute-Right L24)
L18.	$\vdash ((A\hat{}2)+(((-1\cdot(A\cdot B))+(A\cdot B))+(-1\cdot(B\hat{}2)))) ((A\hat{}2)-(B\hat{}2))$	=	(1*E L23)

Two new proof lines have been inserted here, and the computations of *Split-Monomials-Plus* have been reduced to a simple numerical rewrite, an application of distributivity and the elimination of the neutral element of multiplication. Note that some of the syntactical peculiarities are encoded within the tactics and not within the CAS. While the expressions $A\cdot B$ and $1\cdot A\cdot B$ have the same representation in MASS's proprietary data structure, the re-translation into the $\mathcal{PDS}$ will drop the leading 1. Possible eliminations or introductions of the neutral element in this position are included in the implementation of the tactic. In this case the tactic *Split-Monomials-Plus* explicitly eliminates the neutral element if required.

After expanding these few special tactics, MASS' computation is completely remodelled within ΩMEGA's $\mathcal{PDS}$ in terms of simple algebraic laws. The resulting proofs can be further processed by ΩMEGA's facilities for proof representation and explanation or, after expansion to calculus level, proof checking.

3.6.2 In Combination with MAPLE

The standalone approach offers a helpful tool in interactive proof development for automated normalisation of algebraic expressions. In this case efficiency is not as critical as in automated proof search, and within its limitations, MASS is an easy-to-use module whose computations are seamlessly integrated into the ΩMEGA environment.

MASS however has its limits when it comes to nontrivial computations and issues of efficiency. This is a major drawback in automated proof search. MAPLE, on the contrary, is a versatile CAS that offers sufficient support to solve most algebraic problems occurring in real world proof challenges, and operates on efficient data structures and algorithms so that it can be reasonably used in automated proof planning. Like different further deduction systems, ΩMEGA provides facilities for interfacing to MAPLE and the system is successfully involved in automated proof planning for different domains. As MAPLE however acts as a black box system, integrating its computations into ΩMEGA's $\mathcal{PDS}$ without further verification threatens the correctness of the resulting proofs and makes it impossible to verify them using a proof checker.

The solution proposed in this work is a combined application of MAPLE and MASS. This allows to employ the power and efficiency of a full-grown CAS like MAPLE during proof planning, and to use the verification capabilities of MASS in ΩMEGA's expansion mechanism. This approach allowed to fill the gaps left by MAPLE in some experiments in the ΩMEGA

Method:*Solve-Equation*	
Premises	L_2
Application Conditions	$EqualWithMaple(\Phi, \Psi)$ $\Phi_* \leftarrow Normalize(\Phi)$
Conclusions	$\ominus L_1$
Declarative Content	$L_2.\quad \Delta \vdash \Phi_* = \Phi_* \qquad (=\text{Ref})$ $L_1.\quad \Delta \vdash \Phi = \Psi \qquad (\text{Cas } L_2)$

Figure 3.8: The *Solve-Equation* Method.

group, e.g. on the exploration of the domain of residue classes [53] and in the domain of limit theorems [54]. A feasability study of such an approach is described by Sorge [72].

On ΩMEGA's side, the knowledge required to control and coordinate the two CASs in proof planning is provided by methods. A simple example of such a method is *Solve-Equation* (see figure 3.8). An application of this method is very similar to the example described in section 3.6.1. The differences is first that the method provides the control for an application in automated proof planning, and second that in proof planning not MASS, but MAPLE is employed. In this method MAPLE is used to test whether two expressions Φ and Ψ are equal, and if so, the previously open proof line L_1 is closed and the justification delegates the expansion to MASS. This approach makes use of the greater efficiency of MAPLE during proof planning and uses MASS for verification only. If the method has come to application, e.g. to solve the example from 3.6.1, the expansion of the conclusion line has the same result as in a standalone application of MASS. Only different lexicographical orderings of the normal form produced by MAPLE may make a difference, but these are within a range that is easily handled by MASS's capabilities to reorder expressions. The *Solve-Equation* method is applied in examples from the domain of residue classes [53].

A more interesting example is the *Complex-Estimate* method (depicted in simplified form in figure 3.9), that comes to application in examples from the limit domain [54]. The purpose of this method is to estimate the magnitude of the absolute value of a complex term by estimating its simpler factors. The factorisation is done using MAPLE and its two functions `quo` and `rem` when the application condition $k, l \leftarrow CasSplit(a_\sigma, b)$ is evaluated. Here the polynomial division b/a_σ is computed, the resulting quotient (computed by the function `quo`) is bound to k and the remainder (computed by `rem`) is bound to l. Both of the MAPLE function employ the Euclidian algorithm to compute this polynomial division. If both of the computations yield the correct result, then the equation $b = k \cdot a_\sigma + l$ holds. As the verification of this equation can easily be done by normalisation and possibly lexicographical reordering of polynomials, MASS can employed here to verify the results of the MAPLE computations. This combination of MASS and MAPLE makes use of the fact, that some computations, while difficult to perform, are easy to verify given the result. In this case a polynomial division (which is beyond the capabilities of MASS, but is easy in MAPLE) can be verified using multiplication of polynomials (this verification cannot be supplied by MAPLE, but is easily computed by MASS). MAPLE is used here as an useful oracle, but it is not necessary to treat MAPLE as a trusted system.

An example where the *Complex-Estimate* method comes to application is the proof of LIM+, where it is proved that the limit of the sum of two functions is the sum of their limits.

Method:*Complex-Estimate*	
Premises	$L_1, \oplus L_2, \oplus L_3, \oplus L_4$
Application Conditions	$\sigma \leftarrow \mathit{GetSubst}(a, b)$ $k, l \leftarrow \mathit{CasSplit}(a_\sigma, b)$ $b_* \leftarrow \mathit{Normalize}(\Phi)$
Conclusions	$\ominus L_7$
Declarative Content	$L_1.\ \Delta \vdash \lvert a\rvert < e_1$ $L_2.\ \Delta \vdash \lvert k\rvert < M$ (*Open*) $L_3.\ \Delta \vdash \lvert a_\sigma\rvert \leq \epsilon/2 \cdot M$ (*Open*) $L_4.\ \Delta \vdash \lvert l\rvert \leq \epsilon/2$ (*Open*) $L_5.\ \vdash b_* = b_*$ (=Ref) $L_6.\ \vdash b = k \cdot a_\sigma + l$ (Cas L_5) $L_7.\ \Delta \vdash \lvert b\rvert < \epsilon$ (fix L_6, L_1, L_2, L_3, L_4)

Figure 3.9: The *Complex-Estimate* Method.

The problem is formalised in the following way:

Limit-F.	Limit-F	$\vdash \forall E_1 \exists D_1 \forall X_1 [(0 < E_1) \Rightarrow [(0 < D_1) \wedge [[(\lvert(X_1 - A)\rvert < D_1) \wedge (\lvert(X_1 - A)\rvert > 0)] \Rightarrow (\lvert(F(X_1) - \mathit{Limit}_1)\rvert < E_1)]]]$	(Hyp)
Limit-G.	Limit-G	$\vdash \forall E_2 \exists D_2 \forall X_2 [(0 < E_2) \Rightarrow [(0 < D_2) \wedge [[(\lvert(X_2 - A)\rvert < D_2) \wedge (\lvert(X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(X_2) - \mathit{Limit}_2)\rvert < E_2)]]]$	(Hyp)
Thm.	Limit-F, Limit-G	$\vdash \forall E \exists D \forall X [(0 < E) \Rightarrow [(0 < D) \wedge [[(\lvert(X - A)\rvert < D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (\mathit{Limit}_1 + \mathit{Limit}_2))\rvert < E)]]]$	(*Open*)

As the resulting proof in ΩMEGA is quite lengthy, the full proof is omitted here. It can be found in appendix A. At some time during proof planning, the *Complex-Estimate* method comes to application.

L32.	$\mathcal{H}_1$	$\vdash (\lvert(F(X) - \mathit{Limit}_1)\rvert < (E/2))$	(*Open*)
L31.	$\mathcal{H}_1$	$\vdash (M\_E_2 \leq (E/(2 \cdot M\_M)))$	(*Open*)
L30.	$\mathcal{H}_1$	$\vdash (\lvert 1\rvert \leq M\_M)$	(*Open*)
L29.	$\mathcal{H}_1$	$\vdash (\lvert(G(M\_X_2) - \mathit{Limit}_2)\rvert < M\_E_2)$	($\Rightarrow E$ L27,L26)
L28.	$\mathcal{H}_1$	$\vdash (\lvert(((F(X) + G(X)) - \mathit{Limit}_1) - \mathit{Limit}_2)\rvert < E)$	(Complex-Estimate L29,L30,L31,L32)

$\mathcal{H}_1$ = Limit-F, Limit-G, L4, L8, L19

In this situation proof line L_{29} has already been derived, and L_{28} has been closed by the method *Complex-Estimate*. From L_{29} and L_{28} a and b can be instantiated as $a = (G(M\_X_2) - \mathit{Limit}_2)$ and $b = (((F(X) + G(X)) - \mathit{Limit}_1) - \mathit{Limit}_2)$. With these instantiations, $\mathit{GetSubst}(a, b)$ evaluates to $\sigma = \{M\_X_2 \rightarrow X\}$, such that $a_\sigma = (G(X) - \mathit{Limit}_2)$. Furthermore MAPLE is called to compute the polynomial division, and $\mathit{CasSplit}(a_\sigma, b)$ yields $k = 1$ and $l = F(X) - \mathit{Limit}_1$. Finally the three new open goals L_{30}, L_{31} and L_{32} are inserted into the $\mathcal{PDS}$, and the process of proof planning is continued.

While the process of proof planning integrates MAPLE's results without further examination, the correctness of these computations are checked by MASS when the proof is expanded. In the actual example of LIM+, an expansion of L_{28} inserts a proof line to formally verify the result of the MAPLE computation.

L67.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$	=	(=Ref)
L68.	$\vdash (((F(X)+G(X))-Limit_1)-Limit_2)$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(Cas L67)

An expansion of L_{68} is analogous to the example in section 3.6.1 and results in a refined subproof:

L67.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$	=	(=Ref)
L91.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(G(X)+F(X))))$	=	(C-Plus L67)
L90.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((-1{\cdot}Limit_2)+(G(X)+((-1{\cdot}Limit_1)+F(X))))$	=	(Pop-Plus L91)
L89.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((-1{\cdot}Limit_2)+(G(X)+(F(X)+(-1{\cdot}Limit_1))))$	=	(C-Plus L90)
L88.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $(G(X)+((-1{\cdot}Limit_2)+(F(X)+(-1{\cdot}Limit_1))))$	=	(Pop-Plus L89)
L87.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((G(X)+(-1{\cdot}Limit_2))+(F(X)+(-1{\cdot}Limit_1)))$	=	(A-Plus-Left L88)
L86.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((G(X)+(-1{\cdot}Limit_2))+(F(X)-Limit_1))$	=	(Plus2minus L87)
L85.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((1{\cdot}(G(X)+(-1{\cdot}Limit_2)))+(F(X)-Limit_1))$	=	(1*I L86)
L84.	$\vdash ((-1{\cdot}Limit_2)+((-1{\cdot}Limit_1)+(F(X)+G(X))))$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(Plus2minus L85)
L83.	$\vdash (((-1{\cdot}Limit_1)+(F(X)+G(X)))+(-1{\cdot}Limit_2))$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(C-Plus L84)
L82.	$\vdash (((-1{\cdot}Limit_1)+(F(X)+G(X)))-Limit_2)$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(Plus2minus L83)
L81.	$\vdash (((F(X)+G(X))+(-1{\cdot}Limit_1))-Limit_2)$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(C-Plus L82)
L68.	$\vdash (((F(X)+G(X))-Limit_1)-Limit_2)$ $((1{\cdot}(G(X)-Limit_2))+(F(X)-Limit_1))$	=	(Plus2minus L81)

As for the previous examples, MASS's computation is now modelled within ΩMEGA's $\mathcal{PDS}$, and a expansion down to calculus level is possible without further participation of the CAS. Therewith the correctness of the computation can be verified by ΩMEGA's proof checker.

The approach of a combined application of MAPLE and MASS described here allowed (within its limits) to make use of the algebraic power of the commercial CAS MAPLE without having the proof's correctness threatened by possibly erroneous computations of the CAS and without the necessity to perform or remodel all computations that are involved within the Deduction System's formalism. Although this approach is limited to computations that are possibly difficult to perform, but easy to verify (like the verification of a polynomial division by means of a polynomial normalisation), it helped to fill the gaps left by MAPLE in the experiments in the domains of residue classes and limit proofs [53, 54].

3.7 Conclusion

The implementation of the MASS system had mainly two goals. The first intention was to develop a system that works in a similar style as Sorge's μCAS [43], but with an increased utility and a wider range of application. μCAS is able to add and multiply polynomials and

to compute their derivative, but has the major drawback that it can only handle polynomials in normal form. As polynomials hardly occur in normalised form in real world problems, the system's applicability was limited to a few example cases. In contrast to μCAS, the functionality of the MASS system is based on a simple recursive algorithm for polynomial normalisation. Further robustness was achieved by the allowance of non-interpreted functions, such that MASS is able to handle, if not to simplify, arbitrary expressions.

Equipped with this increased robustness and utility, the second goal could be attacked, the combination of the computational power of a commercial CAS like MAPLE to perform nontrivial computations with the verification strength of MASS in simple arithmetics. This approach was successfully evaluated in the context of exploration of the domain of residue classes and the limit domain [53, 54]. The employment of MASS in this context allowed a expansion of the resulting proofs down to the level of ΩMEGA's ND-calculus and therewith to verify the proof using ΩMEGA's proof checker. This was previously not possible due to MAPLE behaving as a black box system from ΩMEGA's point of view and due to the lack of suitable means to verify MAPLE's computations within ΩMEGA's formal environment.

While this combination of white box and black box was successfully integrated into the ΩMEGA system, the experiments described above revealed the difficulties of the approach pursued here, too. As Homann and Calmet pointed out [38], a true integration of CAS and Deduction System requires a common mathematical knowledge base. In the case of the integration of MASS in the ΩMEGA environment, this mathematical knowledge base is used to remodel MASS's computational steps as ΩMEGA tactics. It turned out that even the formalisation of the computations performed by a very simple system like the prototypical MASS required a considerable number of tactics. Furthermore the exact synchronisation of CAS steps and tactics is critical to the success of the integration of both systems, the more as some of tactics keep track of syntactical subtleties that are not necessarily obvious in MASS's proprietary data structure (e.g. $1{\cdot}a$ and a have the same representation in MASS, but are syntactically different in ΩMEGA). Thus the implementation of suitable and correct tactics in ΩMEGA was a nonneglegible part of the development. Furthermore an extension of a CAS that is integrated this way within an Deduction system requires not not only to extend the CAS' algorithmic libraries, but in parallel to maintain the common mathematical database. To ease the impact of this aspect, an authoring tool for tactics in ΩMEGA was developed, named TACO and described in chapter 4. TACO allows the generation of tactics from abstract specification, automatically producing code for term manipulation and headers of declarations in KEIM. To provide a comfortable environment for the development of tactics, TACO is equipped with a graphical user interface to provide a concise presentation as well as facilities for file management.

In the current implementation of the system the simplicity of the interface imposes further limitations on the applicability of the CAS. It maps the head function symbol of the term or subterm to be processed to a CAS algorithm. This is appropriate as MASS is currently applied as an engine for term simplification by means of polynomial normalisation. Furthermore the computations of the CAS as they are remodelled within the $\mathcal{PDS}$ are restricted to linear term rewritings, which again is suitable for currently implemented applications, as the computations required for polynomial normalisation can be described as a sequence of equivalence term rewritings. While suitable for the simple prototypical CAS MASS, more elaborate algorithms and algorithmic techniques are hard to implement this way. Efficient computational techniques like divide-and-conquer strategies, where results of auxiliary computations are reused in several places and therefore require cross-referencing within the CAS-generated subproof,

are hard to express in strictly linear term rewriting. Other algorithms, like the Gaussian algorithm to solve linear systems of equations, would be forced into a quite unnatural appearance when requiring all information available to the CAS to be encoded within a single proof line. Finally due to its simplicity, the interface can provide only little information to suggestion mechanisms and method heuristics. A desirable solution here would be an abstract description of algorithms in the style of method declarations in ΩMEGA. Unfortunately, an extension of the interface to handle method-like abstract outlines is a non-trivial task, as the matching of proof lines in possibly long proofs and abstract method declarations from a possibly large database is a technically advanced challenge. The suggestion mechanism ΩANTS [9] attacks a comparable challenge for methods in ΩMEGA. In chapter 5 a data structure is proposed which could offer the technical basis for this kind of enhancement of the interface. This data structure furthermore provides a blackboard architecture that allows the interface to act as a collector of constraints, comparable to $\mathcal{CoSIE}$ [56], i.e. a module to collect and evaluate relevant information during proof development and to suggest and invoke possibly applicable CAS algorithms for further algebraic processing.

A final fundamental limitation of the approach pursued in the integration of MASS and ΩMEGA is imposed by the control of the execution of algorithms being placed outside ΩMEGA's formalism. Therefore the very natural approach of human mathematicians to verify the correctness of an algorithm and properties of its results by an argumentation over the algorithm itself is impossible. This makes it difficult to verify algorithms like the following algorithm to determine the Greatest Common Divisor of two natural numbers:

```
fun GCD(a,b)
  m:=a mod b;
  if m=0 then return b else GCD(b,m);
```

In this case the property of the algorithm's result to be the greatest common divisor of two numbers is easy to verify by an argumentation over the algorithm, but an integration within the current implementation of the SAPPER interface would require to explicitly keep track of a whole bunch of side-conditions in each step of the computation every time the algorithm is applied. A thinkable solution here might be the implementation of a formalised machine that can both be used to execute algorithms during proof development and provides a basis for an argumentation over algorithms. While for a first evaluation a simple language providing a small set of formalised control structures and algebraic operations would probably do fine, the research on software verification, e.g. the formalisation of the Java Virtual Machine respectively parts of it in ISABELLE [66, 6, 62], could be a source of inspiration.

Chapter 4

Building a Mathematical Knowledge Base Using TACO

4.1 Motivation

As already discussed in the previous chapter, the applicability of a traceable CAS intended to interact with a deduction system depends heavily on the available common mathematical knowledge base. As for the CAS itself, this knowledge base should be easily extendable, and the focus of this work is rather on finding generic ways to do so than implementing a fixed system.

The particles this common mathematical knowledge base, or rather the part of it that is used for the interaction between MASS and the ΩMEGA system, is composed from are tactics. Tactics are the counterparts of computational steps in terms of the MASS system, and their purpose is to model the CAS's computation within the logical representation of the deduction system. Due to their central role in the interaction between the CAS and the deduction system the selection of tactics to be used in this context is of great influence on the overall outcome of the system, so the development of tactics to be used here should be carried out with great care. A sloppy selection and implementation of tactics may lead to a whole bunch of problems: The granularity of the CAS's output trace depends on the available set of computational steps and therewith on the available tactics, and the trace's granularity imposed by the tactics used here may lead to a reduced efficiency of the CAS and to an insufficient readability of the resulting proof plan, if it is too fine, or, if it is too coarse, to a disentanglement of the CAS's trace and its mirror in the deduction system's logical environment and thus may even threaten the correctness of the outcoming proof plan.

Moreover during the development of the MASS system the implementation of tactics turned out to make up a non negligible part of the effort spent on coding, in most cases the amount of code produced to implement the tactics to model a certain algorithm exceeded the code produced to implement the algorithm itself. This is due to the fact that first tactics are complex and flexible tools to manipulate a deduction system's proof plan, and in fact they have to be that flexible and complex to be able to model a CAS's trace without requiring a too fine granularity of this trace. Second the computation of an algorithm is a sequence of computational steps each of which has to be modelled by a tactic, so that the number of tactics that have to be implemented exceeds by far the number of algorithms. The same applies for a future extension of the algorithm library, as the implementation of a new algorithm usually

requires the implementation of several new tactics.

Thus, as the implementation of tactics turned out to be a major part of the development of a traceable CAS, it became necessary to find a comfortable way to implement and administrate a library of tactics. As it furthermore appeared that major parts of the implementation of a tactic, although the concept of tactics is very open, consists of schematic code, a possible solution is the use of an authoring tool for tactics in ΩMEGA.

The result of these deliberations was the realization of the mathematical authoring tool TACO. TACO is intended to provide a very user friendly way to implement and administer tactics under ΩMEGA without cutting the power of ΩMEGA's concept of tactics. The idea was to generate executable code from high level specifications of tactics, to keep furthermore the possibility to embed customised code within these specifications, and finally to provide the technique to compose high level tactics from simpler ones, which means that TACO allows the user to implement algorithms directly within a logical environment. Moreover, as the focus of the concept was strictly laid upon usability, TACO provides a graphical user interface and facilities for file handling and inspection.

The principle of TACO's functionality is that of a visual editing environment as it is common in many commercial systems like visual HTML editors, e.g. Macromedia's Dreamweaver [34], and visual programming environments, e.g. Borland's C++ Builder [41]. Similar to these systems TACO allows the user to work in a visual environment, which provides facilities for structured presentation of the subject of his development and reduces the amount of code that has to be handwritten by means of automatic code generation. Here TACO is responsible for the automatic generation of the basic skeleton of a tactic as well as the necessary LISP functions, which considerably reduces the rate of typing errors and increases the productivity during development by providing a concise representation and by freeing the user from that schematic part of coding that can easily be done mechanically but which can also be very tedious to a human programmer.

Beyond this the TACO environment provides a functionality that is special to the development of tactics in ΩMEGA. Tactics can be viewed as functions that produce new proof lines from a set of given proof lines and possibly further parameters. Technically these functions are implemented in native LISP code and referred to by the abstract specification of the tactic. Thus the functionality of a tactic is limited only by the capabilities of the LISP programming language itself, which is the basis for the flexibility of tactics in ΩMEGA. The full power of a programming language however is necessary only for comparatively small part of the computations which are performed during a tactic's application, while the rest of a tactic's implementation is in general concerned with the decomposition of terms to provide the suitable parameters for the computation and with composition of terms to encode the results of the computation in new proof lines. From the developers point of view it is much more comfortable to specify this task of term decomposition and term composition on an abstract logical level, as it is the case for ΩMEGA *methods*, and have these specification processed by a matching or unification algorithm. This proceeding is of course much less efficient at runtime than simply provide a native LISP function that e.g. selects subterms at a given position in a given proof line that is required as a parameter for further computations. Possibilities to apply a tactic as a proving step either in forward or in backward direction, each case may require additional functions.

In a development environment like TACO however it is therefore sensible to provide this facility of abstract logical specifications. Thus the core of the TACO system is a modified first order matching algorithm which is run only once when generating the code for a tactic

and which produces native LISP code that fulfils the purpose of term decomposition and composition as far as it is needed at application time of a tactic. To be noted here is that only one of the two terms that are to be matched, namely the term in the tactic specification, is known to the matching algorithm at runtime, and that the code that is produced reflects the testing and branching that is dependent on the so far unknown term. This way the execution of the matching algorithm is divided in two parts: First, one of the two terms to be matched is analysed, and executable LISP code is generated according to the result. This step is executed *once* when the code of a tactic is generated, as this happens only during the development of a new tactic, this step is not critical to the efficiency of the deduction system the resulting tactic is intended to be used with. In a second step, the code that has been produced is used to test and process terms in the context of actual proof development. This code is executed every time the tactic is applied in a deduction system and therefore it is time critical, but the computations performed by this code are much more efficient than running a full matching algorithm at a tactic's runtime. This way TACO achieves both goals, first allow the developer to specify tactic's on an abstract logical level, and second to use efficient straight forward LISP code to implement tactics.

4.2 Specification of Tactics

Tactics are tools to perform more or less complex manipulations of proofs in a logical environment. The concept of tactics was introduced in the Edinburgh LCF system by Gordon, Milner and Wadsworth [32] and was used to implement complex inference rules. The concept of LCF tactics was to schematically apply a sequence of inference rules and to provide the control for doing so, i.e. the concept of LCF tactics provides the control mechanisms of a programming language to examine a proof situation, to choose the appropriate inference rules and to apply them to the proof. The use of LCF tactics made it possible to handle proof situations where goals or subgoals could be solved in a schematic way, but at a higher level than that of inference rules of the underlying calculus. As the programming language elements of this concept only provides the control, while the proof manipulation itself is further on performed by calculus level inference rules, these tactics can be used as a tool to schematically find the solution to certain proof situations by means of a programming language without endangering the correctness of the proof.

The concept of tactics in the ΩMEGA system is a descendant of this idea with similarities and differences. The main goal of the concept of ΩMEGA tactics is still to make use of the advantages of a programming language to solve these parts of a goal that can be solved schematically, and to be able to justify the outcome of such a strategy by an application of a sequence of inference rules. The main difference to LCF style tactics however is that, for reasons of efficiency, the application of a tactic to a proof situation was separated from its justification at calculus level, i.e. when applying a tactic to a set of proof lines the deduction system will check whether its applicability constraints are met and will generate new proof lines if necessary, but the justification of the proof lines that are involved will remain abstract until a more detailed justification is needed. This means the application of inference rules is omitted during the application of a tactic and is postponed to the time when the expansion mechanism is used to generate are more detailed justification.

Thus the implementation of a tactic consists of two parts. The first part is the application of a tactic. Tactics are inference steps, which can be represented by a scheme of proof lines.

This scheme of proof lines includes at least one conclusion line, i.e. proof lines that are to be justified by an application of the tactic, and a number of premise lines (possibly zero), i.e. proof lines that necessary to derive the conclusion lines' justification. Additionally a tactic may have an arbitrary number of parameters, e.g. terms or positions of subterms. So in general a tactic T is characterised by an application scheme with conclusion lines $C_1 \ldots C_m$, premise lines $P_1 \ldots P_n$ and parameters $A_1 \ldots A_k$:

$$T(A_1 \ldots A_k) \quad \frac{P_1 \ldots P_n}{C_1 \ldots C_m}$$

At the time a tactic is applied not all of its arguments, i.e. in first place premises and conclusions, have to be instantiated by existent proof lines in the current proof situation. It is often sufficient if some of these proof lines are instantiated, while non-instantiated proof lines can be generated during the application of the tactic and are inserted into the proof plan afterwards. This is called partial argument instantiation or PAI for short.

Due to the capability of handling a PAI situation, the application part of a tactic may comprise not only one, but several specifications to apply a tactic, which are called application schemes. An application scheme defines, which of the lines of a tactic specification have to exist at application time and which not. Application schemes are defined by outline patterns, which are lists of the indicators `existent` and `nonexistent`. These declare whether a conclusion or premise has to exist already or whether it can be generated by the application scheme. For example a possible outline pattern for a tactic with one conclusion line and two premise lines is `(existent nonexistent nonexistent)`, which denotes that in this scheme is applicable if the conclusion line already exists in the proof plan, but the two premise lines do not. In case of an application of this tactic these two premise lines are generated and inserted in the proof plan. In general each application scheme of a tactic is separately implemented, where the implementation has to provide appropriate functionality to instantiate the specific missing lines.

In TACO a full specification of a tactic comprises, apart from proof line schemata, parameters and outline patterns, additional slots for further constraints, help facilities and theory symbols. A complete specification of the (simplified) tactic *Split-Monomials-Plus*, which has been described in section 3.6.1, has the following slots:

- *Variables*: This is the slot for variable declarations.

  ```
  phi z a
  ```

- *Theory Constants*: This slot is used to declare symbols that are defined in the mathematical theory. These symbols can be looked up from the proof environment.

  ```
  plus times div num
  ```

- *Parameters*: The tactic's parameters.

  ```
  (pos position)
  (x term)
  (y term)
  ```

- *Patterns*: Outline patterns of the situations the tactic is applicable in.

```
(nonexistent existent)
(existent nonexistent)
(existent existent)
```

- *Theory*: The mathematical theory the tactic belongs to.

```
real
```

- *Premises*: Proof line schemata for the tactic's premises.

```
(l1 (formula phi (times z a)  pos))
```

- *Conclusions*: Proof line schemata for the tactic's conclusions.

```
(l2 (formula phi
      (plus (times x a) (times y a))
      pos))
```

- *Constraints*: Further side-conditions of the tactic. These constraints have two purposes: they restrict the applicability of a tactic, but can also be used for variable instantiation. Brackets { and } are used to mark LISP code snippets, see section 4.3 for details.

```
{and (data~primitive-p ?x)
     (numberp (keim~name ?x))}
{and (data~primitive-p ?y)
     (numberp (keim~name ?y))}
{and (data~primitive-p ?z)
     (numberp (keim~name ?z))}
(z = {term~constant-create
    (+ (keim~name ?x) (keim~name ?y))
    ?num})
```

- *General Help*: Describes the purpose of a tactic.

```
Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y.
```

- *Argument Help*: Help for proof lines and parameters. The help strings are displayed in interactive proving, when the user is prompted to specify arguments for the application of a tactic.

```
(l2 "A Line containg x*a+y*a")
(l1 "A Line containing z*a")
(pos "The position of the term")
(x "The first coefficient")
(y "The second coefficient")
```

- *Expansion*: Defines the expansion of the tactic. It is a sequence of inference steps each of which has an identifier, an outline and possibly parameters. See section 4.4 for details.

```
(inference expand-num (l3 l1)({pos~add-end ?pos 1} x y))
(inference distribute-right (l2 l3) (pos))
```

4.3 Code Generation

Although the concept of tactics in ΩMEGA is open concerning the use of programming language elements, there is in most cases a part of straightforward schematic term composition, for example the replacement of a subterm in a given position of a proof line's formula by the result of some sort of calculation, required for the implementation of a tactic. This applies for the implementation of functions to instantiate non-existing proof lines of the application scheme as well as for the implementation of applicability predicates, whose purpose is among others to test proof lines for specific structural properties.

The task to perform here is to match the proof lines in question against the patterns of the tactic's specification. Thus, given a specification $C_1 \dots C_n, P_{n+1} \dots P_m$ of proof line patterns, the candidates $L_1 \dots L_m$ for a suitable argument instantiation (which have, with respect to the application schemes provided for this tactic, not necessarily to exist already in the current proof plan) have to be matched against these patterns $C_1 \dots C_n, P_{n+1} \dots P_m$. To be noted is that in this context symbols occurring within the proof line candidates' formulae are treated uniformly as constant, with no regard to them being constants or variables in the proof's context. The notion of free variables will be used in the following to denote free meta variables, that may occur only within the tactic's specification patterns $C_1 \dots C_n, P_{n+1} \dots P_m$. The solution to the pattern matching is a substitution θ such that for every $i \in \{1 \dots m\}$ the following holds:

- $L_i = \theta C_i$ for $i \in \{1 \dots n\}$ respectively $L_i = \theta P_i$ for $i \in \{n+1 \dots m\}$, if L_i is an existing proof line in the current proof plan, and
- θC_i for $i \in \{1 \dots n\}$ respectively θP_i for $i \in \{n+1 \dots m\}$ does not contain any free meta variables, if L_i is non-existing.

This means that the algorithm has to find an instantiation for the specification scheme's set of free meta variables such that neither conflicts arise from matching multiple occurrences against their respective occurrences in actual proof lines nor that this instantiation is incomplete, preventing one or more non-existing proof lines with uninstantiated meta variables from being synthesised. Thus the technique of pattern matching is used for two purposes: first to check whether a set of proof lines fits the specification of a tactic and therewith whether a tactic is applicable, and second to complete the argument instantiation of a PAI situation.

In the following I will describe an example for a tactic's specification and its application. Considering the tactic

$$\textit{Distributivity} \quad \frac{P_1.\Phi(A*B + A*C)}{C_1.\Phi(A*(B+C))}$$

a matching of the premise's specification pattern P against the actual proof line

$$L_1.\Phi(2 * a + 2 * b)$$

would yield the substitution

$$\theta = \{A \rightarrow 2, B \rightarrow a, C \rightarrow b\}$$

which, applied to the specification pattern of the tactic's conclusion C_1, will result in the newly generated proof line

$$L_2.\Phi(2 * (a + b))$$

In this case the two conditions stated above are met: First the application of θ to pattern P_1 equals proof line L_1, and second the result of an application of θ to pattern C_1 does not contain any free meta variables. Thus the new proof line L_2 can be inserted into the current proof plan and is justified by the inference step

$$\text{Distributivity} \quad \frac{L_1.\Phi(2 * (a + b))}{L_2.\Phi(2 * a + 2 * b)}$$

When however the same pattern P_1 is matched against the proof line

$$L_1.\Phi(2 * a + 3 * b)$$

no suitable substitution will be found, as the free meta variable A in pattern P_1 occurs twice and in the respective positions of L_3 the two obviously different symbols 2 and 3 can be found. Hence it is impossible to find a substitution to fulfil the condition $L_3 = \theta P_1$, so the tactic is not applicable here.

The type of matching used in the current implementation of TACO is, although operating on higher order terms, first order style matching, i.e. a substitution θ such that $T_i = \theta P_i$ or $T_i = \theta C_i$ denotes in the following a substitution such that both sides of the equation are syntactically equal modulo α-conversion. Thus the algorithm that is applied to find a suitable substitution is a variant of Robinson's algorithm for first order unification [67, 16]. It starts on a set of equations Γ of terms that are to be unified and with an empty substitution θ. The algorithm selects one equation from Γ and modifies Γ and θ according to the following set of rules depending on the structure of the selected equation.

In detail the following rules may come to application (note that the chosen equation is always removed from Γ):

- **Deletion**

  Any equations of the form

  $t = t$

  are removed from Γ, the substitution θ stays unchanged.

- **Elimination**

  If the chosen equation has the form

  $F = t$,

where F is a free variable and has no occurrences in t, then it is checked wether F occurs in the domain of θ. If it does not occur in θ, then this equation is removed from Γ, the substitution $\{F \to t\}$ is added to θ and applied to θ and each equation from Γ:

$\theta' = \{F \to t\} \cup [F \to t]\theta$

$\Gamma' = [F \to t]\Gamma$

If already $\{F \to t\} \in \theta$, then the equation is removed from Γ while θ stays unchanged.

If $\{F \to s\} \in \theta$ for some $s \neq t$, then fail.

- **Decomposition I**

  If the chosen equation has the form

  $s(a_1 \dots a_n) = t(b_1 \dots b_n)$,

  then the equation is removed from Γ, and equations of the subterms and their respective counterparts are added to Γ, while θ stays untouched:

  $\theta' = \theta$

  $\Gamma' = \Gamma \cup \{s = t, a_1 = b_1, \dots, a_n = b_n\}$

- **Decomposition II**

  If the chosen equation has the form

  $\lambda x.s = \lambda y.t$

  then the equation is removed from Γ, and equations of the abstractions' ranges are added to Γ, while θ stays untouched. To avoid clashes caused by naming of the bound variable, their occurrences in both subterms are substituted by the same new variable. This variable is to be treated as a constant in the following (i.e. in first place it should not occur in the domain of θ:

  $\theta' = \theta$

  $\Gamma' = \Gamma \cup \{[x \to v_{new}]s = [y \to v_{new}]t\}$

- **Failure**

  If the chosen equation does not fit any of these cases or substitution θ fails the occur check, then fail. This rule applies e.g. if trying to match two different constants or two different structures like an abstraction and an application.

These rules are applied until $\Gamma = \emptyset$, i.e. until all equations in Γ have been processed.

As the actual task of TACO is however not to match concrete terms and possibly find the according instantiations, but rather to produce executable LISP code that is able to fulfil this task on an arbitrary input of terms, some adaptions have to be made. The generation of code is based upon the fact that, given a set of patterns that are to be syntactically matched against concrete proof lines, the subterms that provide the actual instances of terms to be substituted for these patterns' free variables will occur in fixed term positions, furthermore the occurrence of conflicts concerning term instantiation can also be decided by checking the subterms in constant positions of the candidate expressions.

The core of TACO implements a two step procedure, whose first part analyses the tactic's specification, which is, except for some adaptions, rather similar to the unification algorithm

discussed above; the second part adapts the result of this analysis to the actual PAI-situation and generates the appropriate LISP code to test the respective proof lines for a possible failure of the unification algorithm (which is corresponding to the tactic not being applicable), to compute a correct substitution and to apply it in order to instantiate missing arguments.

To adapt the above unification algorithm, two main problems have to be tackled:

- during analysis of the tactic:

  Not all the terms that are to be unified are known at the time of code generation. This affects the algorithm as it is in particular unknown when the conditions of which reduction rule are met so that this rule would come to application. Furthermore the result of an application of say one of the Decomposition rules to an unknown term are also unknown. Therefore the analysis of the tactic's specification will rather provide a scheme for a unification with respect to those arguments of the algorithm that are known at code generation time.

- when adapting the result to a PAI:

  Depending on the application schemes that are to be handled, some of the terms may not be instantiated at all, and thus all computations the algorithm would perform on these terms become obsolete, which considerably affects the resulting substitution.

To tackle these difficulties requires some modifications of the algorithm. In particular some of the computational steps of the algorithm have to be performed virtually, which means that if an unknown term is object of a certain step of the algorithm, there may be several possibilities of the algorithm to branch. In this case these possible branchings along with their respective side-conditions are collected. The decision which branch of the algorithm comes to application is postponed to either the time of code generation or to the time the tactic is applied.

At code generation time the unification scheme is adapted to defined application patterns of the tactic, these patterns define which of the tactic's argument proof lines are already instantiated when the tactic is applied. Depending on this additional information some of the possible branches of the algorithm can be cut. At the time the tactic is applied, the generated code will provide the appropriate control flow statements to decide which branches of the algorithm are cut.

The actual modification to the above algorithm is first line to *collect* possible computational steps that affect the basic set of equations Γ and the substitution θ to be constructed rather than *applying* them. The collected steps are stored in a *Construction Graph* that will be explained in a more detailed discussion below. Apart from this the adapted unification algorithm is executed as before except that

- in an application of rule **Elimination** the resulting substitution is not applied to the set of equations Γ but is merely collected. Note that this does not necessarily result into a unique substitution for each variable of the domain of θ. Furthermore, as it is at analysis time not clear which of the affected variables will have to be treated as *constant* and which as *free variables*, as this depends on the proof lines that are instantiated before a tactic is applied. In general there will be two possible substitutions for an equation $a = b$, namely $\{a \to b\}$ and $\{b \to a\}$.

- the application of the **Decomposition I** and **II** may be applied to a term that is unknown at analysis time. In this case the decomposition is performed *virtually*, which means that new variables are introduced to handle the resulting subterms (e.g. the function term of an application), their origin is stored in the Construction Graph, and at the same time possible structural properties that are required to actually apply this rule are collected (e.g. when attempting to apply rule **Decomposition I**, the necessary structural property is that the respective term is an application).

- as at analysis time it is not always possible to distinct *free variables* from *constants*, it may be not clear whether rule **Elimination**, rule **Deletion** or rule **Failure** comes to application. Therefore the distinction between the **Elimination** case, the **Deletion** case and the **Failure** case is omitted, the relevant terms are collected, and possible clashes are resolved later. Note that for this reason a collected substitution $\{a \rightarrow b\}$ does not necessarily represent an actual substitution, but may be as well interpreted as an equation $a = b$ that determines failure or success of the unification algorithm (see below for details).

The result of an application of the modified algorithm is obviously not a unique substitution but rather a collection of relations between variables respectively structural properties of variables that make up the Construction Graph. In the next step this collection is adapted to a specific application scheme (which corresponds to a certain PAI situation and therefore determines the property of being a *free variable* or a *constant* for each variable in the graph). This means that unused relations are dropped (if they refer to variables that are not used in this particular PAI situation), and the useful ones are interpreted and assembled to an executable piece of LISP code. In this step the actual substitution θ is computed along with the required preconditions for the unification algorithm to be successful. Furthermore ambiguities are resolved that occur if there are several possible substitutions for one particular variable: If there are several substitutions for one free variable, then one of them is chosen to constitute the substitution (which is equivalent to an application of rule **Elimination**), while the others are considered subject to rule **Deletion** or **Failure** depending on the terms that are involved. Thus if the terms of these substitutions equal (syntactically) the one that is chosen for the construction, then the unmodified algorithm would have succeeded, if they do not, rule **Failure** would have come to application, i.e. the algorithm would have failed to find a suitable substitution.

The matching in TACO is done in three phases. In the first phase, the tactic's abstract specification is analysed and the construction graph is built up accordingly. In a second phase this graph is used to generate the code to match actual proof lines against these specification. For each application case, i.e. for each outline pattern, an applicability predicate and an own set of functions for argument instantiation is generated. The code generated in this second phase is executed in the third phase, when the tactic is applied in a running deduction system. Note that the potentially costly phases one and two have to be computed only once *before* an actual proof search. Thus the share of computations that have to be performed *in* an actual proof search are considerably reduced.

In the following I will give a detailed explanation of the data structures that are used to represent the Construction Graph and of the algorithm that is applied to build it from the specification of a tactic.

4.3.1 Construction Graph

In the first step the specifications of the tactic to be processed are examined for structural dependencies and requirements, i.e. in first line it is evaluated whether terms that were previously uninstantiated can be computed from any other given input or independently. The aim is to code these dependencies into a graph like structure that will be denoted the *construction graph*, whose nodes are (possibly automatically introduced) variables that are connected by so called *construction rules* and possibly annotated by *constraint rules* which represent structural properties that this variable has to fulfil. In this context all atoms of a tactic specification, namely proof line formulae, tactic parameters and symbols, are treated equally and are therefore uniformly denoted as terms in the following, furthermore every term is represented as a variable in the construction graph.

Construction Rules

Construction rules are used to construct terms from other terms. Their structure is that of an n-ary mapping, where the construction rule is characterised by its *base*, which is a (possibly empty) list of variables, and its *target*, which is a single variable that can be constructed using this rule. Furthermore the construction rule features a piece of LISP code that implements the actual description how to construct the *target* from the *base*. If we consider for example the term c which say occurs in the equation $c = a + b$, c can be constructed from the function symbol + and the list of the application's arguments $[a, b]$. TACO will now introduce a new variable for this argument list, say v_{new}, which will be treated separately, and the following construction rule is added to the construction graph:

```
construction_rule(target: c
                  base:   [+,v_new]
                  code:   'data~appl-create + v_new')
```

The code fragment `(data~appl-create + v_new)` contains the KEIM function to create a new application term whose function is + in this case and whose arguments are the terms contained in the list v_{new}.

When processing this example, supposed that + is a theory symbol, i.e. that it is already defined in ΩMEGA's mathematical database, two construction rules are added to the construction graph that is generated by TACO:

```
construction_rule(target: +
                  base:   []
                  code:   'env~lookup-object
                           :+
                           (pds~environment omega*current-proof-plan)')

construction_rule(target: v_new
                  base:   [a,b]
                  code:   'list a b')
```

Thus we can construct the term c if a and b are known by constructing v_{new} with the third rule. Then + can be looked up in ΩMEGA's proof environment via the second rule, and finally c can be constructed by using the code fragment of the first example rule.

Constraint Rules

Besides these rules to construct terms, TACO's construction graph contains also rules to test terms for specific structural properties, the *constraint rules*. These rules are annotations to a node respectively term in the construction graph and allow terms to be tested for e.g. the property of being an application, an abstraction or a list.

In the above example, TACO will add the following constraint rules to its construction graph:

```
constraint_rule(base: c
                code: 'data~application-p c')

constraint_rule(base: v_new
                code: 'listp v_new')

constraint_rule(base: v_new
                code: 'eql (list-length v_new) 2')
```

which tests c for being an application and v_{new} for being list of length two, and all three rules together test whether c is an application of arity two.

Graph Construction

The core of the process of construction graph generation is to decompose every useful term given in the tactic's specification. At this point the modified algorithm for syntactical unification comes to application.

As described above the algorithm starts on a set of equations, which are in this case extracted from the tactic's specification. For this purpose the useful parts of this specification are first the proof line specifications, where each specification of the form `(l_n t)`. This denotes a proof line labelled `l_n`, whose formula is `t`. When building the construction graph, `l_n` is treated as a variable that is bound to the node's formula, and the specification `(l_n t)` is interpreted as an equation of `l_n` and the specified term `t`. Thus the proof line's formula is always represented by one symbol, while the specification `t` will in most cases be a non-primitive term. For example in the tactic *Distributivity* introduced in section 4.2, these specification terms will be `a * (b + c)` and `a * b + a * c`. As TACO also provides the possibility to apply explicit constraints for a further specification of the tactic, this is a second source of equations for the algorithm's starting set. These constraints are either natively given in an equational form, in which case they are added to the starting set, or they are application of predicates (technically these are code fragments), which are directly translated to entities of constraint rules (see below).

To compute the respective *Construction Graph* from a set of equations, these equations are sequentially processed according to a set of rules that are described in the following. The equations are chosen one after the other and removed from the set, then, depending on the rule that comes to application, new nodes, i.e. new variables, and new edges, i.e. new construction rules or constraint rules are added to the *Construction Graph*.

Note that the distinction between free variables, constants and bound variables is omitted when talking about reduction rules to establish the *Construction Graph*. This is due to the fact that at the time of graph construction it is not clear which of these properties can be

established for which symbol, the matter will however be considered when generating code for an actual application scheme. Furthermore variables of a tactic's specification are meta variables and their namespace is disjunct from the namespace of the proof environment's variables.

In detail following rules may be applied to the initial set of equations Γ (as for the original algorithm for syntactic unification, the chosen equation is always removed from set Γ):

- **Elimination**

  If the chosen equation is of form

  $a = b$

  then the following construction rules are added to the *Construction Graph*:

```
construction_rule(target: b
                  base:   [a]
                  code:   'a')

construction_rule(target: a
                  base:   [b]
                  code:   'a')
```

  In this case neither new variables nor new constraint rules are introduced, and no new equations are added to Γ:

  $\Gamma' = \Gamma$

  The semantics is that both of the variables can be instantiated by their counterpart. To do so no further constraints are required.

- **Virtual Decomposition I**

  If the chosen equation is of form

  $a = f(b_1 \dots b_n)$

  then two new variables g_{new} and $args_{new}$ are introduced. Furthermore the following construction rules are added to the *Construction Graph*:

```
construction_rule(target: a
                  base:   [g_new, args_new]
                  code:   'data~application-create g_new args_new')

construction_rule(target: g_new
                  base:   [a]
                  code:   'data~appl-function a')

construction_rule(target: args_new
                  base:   [a]
                  code:   'data~appl-arguments a')
```

  The new constraint rule that is added to the *Construction Graph* are:

```
constraint_rule(base:   [a]
                code:   'data~appl-p a')
```

and finally two new equations are added to Γ:

$\Gamma' = \Gamma \cup \{g_{new} = f, args_{new} = [b_1 \dots b_n]\}$

The semantics is that an application can, given its function term and the list of its arguments, be constructed using the function `data~application-create`. If however the function is known, it can be decomposed into its function term and the list of its arguments, and to do so it has to be checked whether the term in question actually is an application term, which is done by the predicate `data~appl-p`.

- **Virtual Decomposition II**

  If the chosen equation is of form

  $a = \lambda x.b$

  then two new variables x_{new} and c_{new} are introduced. Furthermore the following construction rules are added to the *Construction Graph* (the code si simplifies for better readability):

```
construction_rule(target: a
                  base:   [x_new, b]
                  code:   'data~abstraction-create x_new b')

construction_rule(target: x_new
                  base:   []
                  code:   '... new variable ...')

construction_rule(target: c_new
                  base:   [x_new, a]
                  code:   '... replace x by x_new in abstr-range a ...')
```

  The new constraint rule that is added to the *Construction Graph* are:

```
constraint_rule(base:   [a]
                code:   'data~abstraction-p a')
```

  and finally two new equations are added to Γ:

  $\Gamma' = \Gamma \cup \{c_{new} = [x \to x_{new}]b\}$

  The semantics is that an abstraction can, given its range and domain, be constructed using the function `data~abstraction-create`. If the abstraction is known, we can extract its range. Note that, to avoid name clashes, the domain variable of the abstraction on either side of the equation is substituted by the new variable x_{new} for further processing of the terms in question. Technically this substitution differs at both sides of the equation: the substitution that is applied to a is hard coded in a construction rule and will therefore not be applied until the runtime of a tactic's application, the

substitution at the equation's left side however is applied to the meta term $\lambda x.b$ at code generation time and will therefore leave no obvious traces in the resulting code.

Here as well as in the **Virtual Decomposition I** rule the property of a actually being an abstraction can be checked using the predicate `data~abstr-p` when needed.

To complete the implementation of the algorithm for syntactical matching described above, the set of rules is extended by the **Introduction** rule:

- **Introduction**

  If the chosen equation is of form

  $a = b$

  where the expressions at both sides of the equation is a non-primitive term, then a new variable x_{new} is introduced. This rule adds neither construction rules nor constraint rules to the construction graph, and the equation is split into two new equations that are added to Γ:

  $\Gamma' = \Gamma \cup \{a = x_{new}, b = x_{new}\}$

  The effect is that a possible application of the **Decomposition I** or **II** rule of the former matching algorithm is reduced to an application of the **Virtual Decomposition I** or **II** rule of the actual implementation, as the resulting new equations will be subject to one of these rules.

Note that this rule operates on non-primitive meta terms, i.e. terms whose structure (at least at top level) is known at code generation time. As the purpose of this rule is the reduction of the former **Decomposition** rules to **Virtual Decomposition** rules, the effect is a shift from a decomposition at code generation time, which is performed at meta level, to a decomposition at runtime of a tactic's application.

Although this obviously affects the efficiency of the resulting code (as the decomposition has to be performed every time the tactic is applied), it nevertheless is advantageous concerning the maintainability of the system. This is due to the fact that the genuine matching algorithm is able to handle terms that are formed according to a quite simple syntactical scheme which allows a term to be either a primitive, i.e. a constant or variable, an application or an abstraction, while the implementation of a convenient environment for the specification of tactics requires a somewhat more elaborated syntax. As the focus of the Taco system is laid upon usability, beneath these possible structures the syntax of Taco provides further syntactical structures like lists or formulae that can be accessed at a given relative position (see below for a detailed description). As Taco is presently in a prototypical state and its syntax and the structures comprised within it are probably subject to further changes, the **Introduction** rule provides a convenient tool for a generic and uniform treatment of the decomposition of all those structures. The effect of this reduction of a meta level decomposition to a decomposition at runtime is that the implementation of new syntactical structures has only to be implemented for the runtime case, which usually can be done by the introduction of construction rules and constraint rules, while the meta level case is generically subsumed by the **Introduction** rule. This does not only cut the effort of an implementation of a new syntactical structure, but also provides a consistent behaviour of the system as it avoids differences in the result of such a decomposition between meta level case and runtime case, as they both are reduced to the same pieces of code.

Apart from the treatment of additional syntactical structures that will be discussed in the following, the **Introduction** rule completes the implementation of the matching algorithm. Obviously two of the former rules are abandoned, namely the **Deletion** rule and the **Failure** rule. The whereabout of both of these rules will be clarified in the following sections, where the actual generation of code for argument instantiation and applicability predicates is discussed.

First however I will complete the list of reduction rules for the generation of the construction graph. The following rules are concerned about additional syntactical structures provided by TACO.

- **List Decomposition**

  If the chosen equation is of form

  $a = [b_1, \ldots, b_n]$

  then the following construction rule is added to the *Construction Graph* for each element of the list:

```
construction_rule(target: b_i
                  base:   [a]
                  code:   'nth i a')
```

  furthermore a construction rule is added to construct the whole list from its elements:

```
construction_rule(target: a
                  base:   [b_1 ... b_n]
                  code:   'list b_1 ... b_n')
```

  The new constraint rules that are added to the *Construction Graph* are:

```
constraint_rule(base:   [a]
                code:   'listp a')

constraint_rule(base:   [a]
                code:   '= (list-length a) n')
```

  In the case of the **List Decomposition** rule the **Introduction** rule is additionally applied to each non-primitive element of the list, i.e. if an element of the list has a complex structure, a new variable will be introduced and the appropriate equation is added to Γ:

  $\Gamma' = \Gamma \cup \{x_{new} = b_n\}$

  This new variable x_{new} furthermore replaces all occurrences of b_n within the construction rules and constraint rules described above.

  The semantics is that a list can be decomposed into its elements if it is known, and that it can be composed from its elements, if these are known. Furthermore to decompose a list, it is necessary to check whether the term in question is a list of appropriate length. Note that lists are essential to deal with applications, as an application consists of a function term and a *list* of argument terms.

- **Number Elimination**

  If the chosen equation is of form

  $a = n$

  where n is an explicit number, then the following construction rule is added to the *Construction Graph*:

```
construction_rule(target: a
                  base:   []
                  code:   'data~constant-create n num')
```

  which means that a can be constructed by creating a number term of value n. Numbers are treated specially as they are literally defined constants whose value is determined by their symbol, which differs from the behaviour of other constants or variables.

- **Code Embedment**

  Besides the definition of syntactical patterns, term manipulation under TACO can also be performed by embedding native LISP code. Thus Γ may contain equations of the form

  $a =$ `piece-of-code`$(v_1 \ldots x_n)$

  where `piece-of-code` is a piece of native LISP code. This code can be straight forward converted to a construction rule:

```
construction_rule(target: a
                  base:   [v_1 ... v_n]
                  code:   'piece-of-code(v_1 ... v_n)')
```

  TACO provides furthermore the possibility to use meta variables within this piece of code, so that the base of such a construction rule is the set of meta variables contained in that code.

- **Formulae**

  Formulae are used to access subterms at a given position. They occur in the form

  $a =$ `formula`$(\Phi, x,$ `pos`$)$

  where Φ is the term, x is the subterm and `pos` is the relative position of x in Φ. As x may be a non-primitive term, it is replaced by a new variable x_{new}. This construct is used for term rewriting in given positions, i.e. constructing a from Φ, x and `pos` by replacing the subterm at position `pos` with x in Φ:

```
construction_rule(target: a
                  base:   [Phi x pos]
                  code:   'data~replace-at-position Phi pos x')
```

Furthermore x can be constructed by looking up the subterm in Φ at position `pos`:

```
construction_rule(target: x
                  base:   [Phi pos]
                  code:   'data~struct-at-position Phi pos x')
```

To find an instantiation for Φ, a can be used:

```
construction_rule(target: Phi
                  base:   [a]
                  code:   'a')
```

As the syntactical construct `formula`(Φx`pos`) denotes a term that is known except for position `pos`, TACO has to keep track of these definitions. Therefore definitions of formulae are collected and later tests for equality are evaluated with respect to the possibly differing subterm at position `pos`. The introduction of the new variable is recorded in Γ:

$\Gamma' = \Gamma \cup \{x_{new} = x\}$

This rule completes the list of possibly applicable construction rules. The computation of the construction graph is finished, if the set of equations Γ is empty. If there are equations remaining without an appropriate rule being applicable, the algorithm fails.

The construction graph however is only a rough skeleton of the term manipulation to be done. The generation of executable code from the set of construction rules and constraint rules, which depends heavily on the context of the actual application scheme of the tactic. The main task of adapting the information represented in the construction graph to an actual application scheme is to select appropriate construction rules and constraint rules and to apply them in the right place. How this is done is subject to the following sections.

4.3.2 Argument Instantiation

The instantiation of missing arguments in a PAI situation is the major purpose of the TACO algorithm. Regarding Robinson's algorithm for syntactic matching, the main differences to its implementation are in first place that the analysis are separated into two steps due to the lack of information at code generation time, as the actual terms to match are usually not known until runtime of a tactic's application, and in second place the separation between finding a suitable substitution for unification and its application to actually instantiate missing arguments is abandoned for reasons of efficiency. A further reason to do so is the dynamic nature of the context to which the information brought about by the matching has to be adapted to. However although the dynamics of the context, e.g. the existence of several application schemes and therewith the possibility of parts of the information gained by the matching algorithm becoming obsolete in some of these contexts, the data structure of the construction graph allows to represent this information in a sufficiently dynamic form to keep it adaptable to a changing context.

Once the construction graph is established, it is easy to extract the information needed to cope with a specified PAI situation. Note that the construction graph the procedure relies on

is the same for each argument in each PAI situation that is processed. The information that is extracted should result into the generation of executable code, and this code should provide the following functionality: code to instantiate missing arguments from information provided by a specified context and an applicability predicate to determine whether a unification is possible in this context at all. The functionality of instantiating a term from a given context is essential for the generation of instantiation functions as well as for the generation of an applicability predicate.

The basic fragments which the generated code is assembled from are the code fragments given by the construction graph's construction rules. The semantics of a construction rule is that the *target* symbol, say c, can be created by executing the rule's *code* fragment, and a precondition is that there are no uninstantiated symbols in the list given by the rule's *base*:

```
construction_rule(target: c
                  base:   [+,v_new]
                  code:   '(data~appl-create + v_new)')
```

Thus if all symbols from the *base* list are instantiated, the rule's *code* can be applied. The resulting code to instantiate c is assembled form both the code to instantiate its base variables and the code snippet in the rule:

```
(let* ((+ (... code to generate + ...))
       (v_new (...  code to generate v_new ...)))
 (data~appl-create + v_new))
```

In this example, an application is created from function symbol + and argument list v_{new}. The result is c, which is the return value of the function here.

If however there are uninstantiated symbols left in this list, we can attempt to recursively lookup construction rules to instantiate these missing *base* symbols. Thus the actual task to perform to assemble the code for argument instantiation is that of finding a directed acyclic graph (DAG) within the construction graph with the following properties:

- the argument symbol that is to be instantiated is a node in the DAG. In the following, this symbol is referred to as the *root* of the graph.
- all edges in the tree are given by valid construction rules. Construction rules are n-ary mappings of the *target* symbol and the set of supporting *base* symbols. If a construction rule is applied in the DAG, all of its base symbols are nodes in the DAG. Note that some construction rules, e.g. to look up theory symbols, have an empty base.
- every symbol node in the DAG may support the *base* of several construction rules, but it is the *target* of at most one construction rule.
- all nodes of the DAG that are not the target of a construction rule are already instantiated symbols; or technically it has to be ensured that all variables that are used by a code fragment have been assigned a value previously. Practically this means that the symbol in question is either given from the context of the PAI situation, i.e. it denotes the formula of a given proof line or one of the tactic's parameters.

The requirement that this graph should be translated to executable code is the reason why the graph has to be a DAG:

- The graph has to be free of *directed* circles, i.e. there is a path of construction rules that, pursued from their base to their target, connects a variable to itself. The consequence would be the attempt to recursively derive a variable from itself, which will not be successful here.

- *Undirected* cycles are allowed and, from a technical aspect, reasonable: in this situation a symbol supports the base of several construction rules and therefore has multiple occurrences in the resulting code. As it is however ensured that this symbol was properly defined previously, this does not affect the correctness of the resulting code.

The result is an algorithm that implements a depth first backward search over the construction graph. The arguments of this algorithm are first the symbol x that has to be instantiated and second the set $\mathcal{I}$ of variables that are already instantiated and a set $\mathcal{P}$ of variables that are pending, i.e. a construction to instantiate them has been found, but the search for an appropriate well founded DAG is not completed yet; when starting the procedure, $\mathcal{I}$ is initially the set of variables that occur in the actual context and are therefore already instantiated, $\mathcal{P}$ is empty.

The value of the call `instantiate(`x`, `$\mathcal{I}$`, `$\mathcal{P}$`)` is computed according to the following rules and either returns a DAG that meets the requirements stated above or fails, furthermore the procedure returns $\mathcal{I}'$, a modified version of $\mathcal{I}$ where all variables that have been instantiated during the process are added.

- If $x \in \mathcal{I}$ then terminate, as x is already instantiated. The DAG that is returned is the single node DAG whose root is x. No new variables have been instantiated, thus $\mathcal{I}' = \mathcal{I}$.

- If $x \notin \mathcal{I}$ and $x \in \mathcal{P}$ then fail. In this case there was an attempt to construct a DAG that contains a directed cycle.

- If $x \notin \mathcal{I}$ and $x \notin \mathcal{P}$, then lookup the list $\{c_1 \dots c_n\}$ of available construction rules whose target is x. Find the first rule c_i that can be successfully applied, which is determined by the following recursion:

  Let $\{b_1 \dots b_m\}$ the base of c_i. A rule c_i can be successfully applied if for none of its base symbols the procedure `instantiate(`b_j`, `$\mathcal{I}_j$`, `$\mathcal{P} \cup \{x\}$`)` fails. To construct b_j, the procedure is called with $\mathcal{I}_j = \mathcal{I}_{j-1}{}'$ for $j \in \{2 \dots m\}$ and $\mathcal{I}_1 = \mathcal{I}$, the set of variables that have already been instantiated before trying to instantiate x. This way, a double construction of variables that have already been instantiated after constructing $b_1 \dots b_{j-1}$ is avoided.

  If there is no such rule then fail, else return a DAG whose root is x, the sub-graphs at depth 1 are those returned by the procedure called to the base symbols of the successful construction rule, which furthermore represents the edges between root and sub-graphs. The set of variables that have been instantiated so far is $\mathcal{I}' = \mathcal{I}'_n \cup \{x\}$, i.e. the set of variables that have been instantiated after having processed the last base symbol of c_i plus x.

When such a DAG of symbols and code fragments is found, the only thing left to do is to translate the result from its DAG structure into a linear list of code fragments. The property that has to be preserved when doing so is the ordering given by the DAG: if a node is a child of another node, then its occurrence in the list should be placed before the parent's node.

- *Premises*:

```
(l1 (formula phi (times z a)  pos))
```

- *Conclusions*:

```
(l2 (formula phi
      (plus (times x a) (times y a))
      pos))
```

- *Constraints*:

```
{and (data~primitive-p ?x)
     (numberp (keim~name ?x))}
{and (data~primitive-p ?y)
     (numberp (keim~name ?y))}
{and (data~primitive-p ?z)
     (numberp (keim~name ?z))}
(z = {term~constant-create
    (+ (keim~name ?x) (keim~name ?y))
    ?num})
```

Figure 4.1: A Part of the Specification for Tactic *Split-Monomials-Plus*.

This is necessary because the DAG structure represents the dependencies between symbols: a symbol is instantiated by applying a specified code fragment to its child nodes, therefore these have to be instantiated previously. Regarding the linear ordering of the list however it is uncritical to place further code fragments between the instantiation code of a symbol and that of its children which it depends on. In the example tactic *Split-Monomials-Plus*, specified in figure 4.1, TACO finds the DAG depicted in figure 4.2 to instantiate proof line `l1` from `l2` and the parameters `x`, `y` and `pos` in the PAI situation that is given by outline pattern `(existent nonexistent)`.

The final step of code generation is the translation of the resulting list to executable LISP code. As the elements of the list already contain code fragments that are ready to use, the task to perform here is simply to pack them in linearised form into a valid `let*` statement. The result is depicted in figure 4.3.

Note that the result of this procedure is only the code to generate one specified argument of a tactic from a specified context. In general this is only a fraction of implementational work to specify a whole tactic, as a tactic usually contains several application schemes, each of which specifying another PAI situation. Furthermore each of these PAI situations may feature several uninstantiated arguments, and for each of these uninstantiated arguments there has to be an implementation to generate it from its specific context. Thus this procedure has to be repeated until the code for instantiation of all of these arguments is generated.

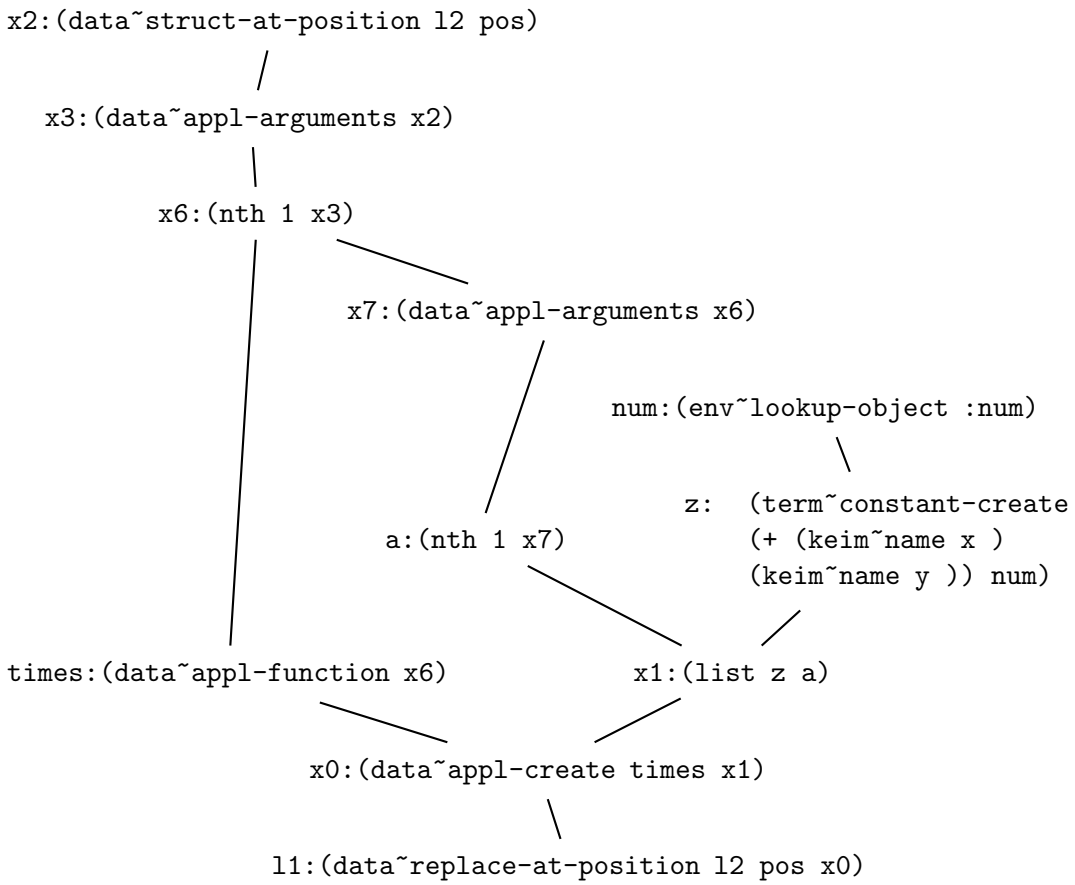


Figure 4.2: The Code DAG to instantiate `l1` in pattern `(existent nonexistent)` with parameters `x`,`y` and `pos`.

4.3.3 Applicability Predicates

Apart from the generation of code to instantiate previously uninstantiated arguments of an PAI situation, the code for the appropriate applicability predicate has to be generated. The purpose of an applicability predicate is to analyse a PAI situation to examine whether it fits the specification of a tactic and therewith whether a specified tactic can be applied in this situation.

From the technical point of view the code that is automatically generated by TACO has to fulfil two purposes: First it has to ensure that the arguments and parameters found at the time of a possible application of a tactic match the meta specification of a tactic defined in TACO. Second, as the code generated to instantiate missing arguments in a specific application scheme respectively PAI situation relies on properties of the processed objects which are guaranteed by the meta specification, it prevents the code of instantiation functions from crashing. In other words a PAI situation that is determined to fit a tactic's specification by an applicability predicate should cause no crashes when being processed by an instantiation function. An example is list decomposition: Applying the LISP function `nth` to a non-list object causes an error.

Critical points that have to be considered when implementing such an applicability test

```
(let*
 ((x2
   (data~struct-at-position l2 pos))
  (x3
   (data~appl-arguments x2))
  (x6
   (nth 1 x3))
  (x7
   (data~appl-arguments x6))
  (a
   (nth 1 x7))
  (num
   (env~lookup-object :num
    (pds~environment omega*current-proof-plan)))
  (z
   (term~constant-create
   (+ (keim~name x ) (keim~name y )) num))
  (x1
   (list z a))
  (times
   (data~appl-function x6))
  (x0
   (data~appl-create times x1)))
 (data~replace-at-position l2 pos x0)))
```

Figure 4.3: The Code DAG of figure 4.2 in linearised Form.

are mainly to ensure structural properties of the formulae in question and to ensure the equality of all occurrences of a symbol if it is used in several places of the specification, e.g. if within the specification of a tactic occurs the expression $a + a$, the applicability predicate's code should ensure that the structure of the respective object is that of an application with arity two, and that both of its arguments are equal.

The procedure of generating the code to implement an applicability predicate is based on the same construction graph that is used to generate code for argument instantiation. However *constraint rules*, while ignored in the previous section, come to application here. Now the construction graph has to be examined to find and handle the critical points in a tactic specification, i.e. structural properties of symbols and equalities of different occurrences of the same symbol. The necessary information can be extracted from the construction graph according to the following principles:

- **Structural properties** of symbols are explicitly given by *constraint rules*, i.e. if a symbol is related to one or more *constraint rules*, the execution of the code fragments of these rules determines whether or not this symbol meets the structural requirements of the tactic's meta specification. These *constraint rules* are generated, as already discussed, by the procedure that analyses the tactic's meta specification.
- **Equality of different occurrences** of a symbols can be examined by analysis of the

construction rules of the graph. If the symbol has several occurrences that are relevant in a specific PAI situation, then there are several *construction rules* that can be used to instantiate this symbol (note that a *construction rule* contained in the construction graph only becomes relevant to a PAI situation if it actually is applicable, i.e. if all symbols of its *base* can be instantiated). Thus to ensure the equality of all occurrences of the same symbol it has to be checked whether the execution of the code fragments of all relevant *construction rules* have the same result.

Note that the occurrence of such critical points is not limited to symbols that are instantiated at the time a tactic is applied, but may as well originate from symbols whose instantiation is done by the code that is generated by TACO. Thus to verify the applicability of a tactic in a specified proof situation, not only symbols which have obvious occurrences in the starting situation are examined, but also those symbols that can be instantiated by TACO's algorithm. All symbols that are reachable via a path of *construction rules* from the starting situation can be instantiated. For all those symbols it is examined whether they meet the requirements of the tactic's specification.

For this reason symbols are not only examined by the TACO procedure to generate applicability predicates, but may also be instantiated by this procedure. As the aim in this case is different from that when attempting to instantiate one specified symbol, which was the aim in the previous section about argument instantiation, the algorithm, although relying on the same principle and the same data structure of the construction graph, is different in this case, too. The actual differences now are first that the objective is to examine *all* symbols rather than instantiating *one* of them. The consequence is that the type of search that is applied to examine the construction graph is a breadth first search that is applied in forward direction, i.e. the task is to determine which new symbols can be instantiated from a given set of known symbols by forward application of a construction rule in the construction graph, while the instantiation of one special symbol rather suggests the use of a depth first backward search, i.e. it has to be determined whether there is a construction rule whose base is constructible and whose target is that special symbol. Second, unlike the code resulting from the procedure for argument instantiation, the code of an applicability predicate is intended to operate in a somewhat insecure area, i.e. no symbol has any guaranteed property like e.g. being an application. The consequence is that to prevent the code from crashing it is necessary to test symbols for the properties that are required for further processing, e.g. decomposition of non-primitive terms should be preceded by an examination of the property of being a non-primitive term of appropriate type. These checks will become obsolete in a later execution of code for argument instantiation, as we can assume that an applicability predicate has been applied prior to the actual tactic application, but the checks are required in the right place to produce applicability predicates that come up with a negative result rather than crashing in situations in which the tactic is not applicable

The consequence of these deliberations is an algorithm that implements a breadth first forward search over the construction graph while keeping track of the properties of affected symbols that have to be checked prior to further processing. The algorithm proceeds in two alternating cycles: first all *constraint rules* that are not depending on variables that have no instantiation yet are evaluated, then all applicable *construction rules* whose base variables have no pending *constraint rules* are used to instantiate new variables. If *constraint rules* are applicable that can be used to construct variables that already have an instantiation, both instantiations have to be equal. these two cycles are repeated until no more *constraint rules*

can be evaluated and no *construction rule* can be applied to instantiate new variables. To produce executable code, the result is processed similar to the instantiation of arguments. The results of a cycle to evaluate *constraint rules* are collected in an `and` statement, applicable *construction rules* of the next cycle are packed into a `let*` statement, whose range is the next cycle of *constraint rule* evaluation and which is the last argument of the previous `and` statement. Thus the resulting code has the form:

```
(and constraint cycle 1
     (let* construction cycle 1
      (and constraint cycle 2
           (let* construction cycle 2 ...))))
```

The applicability predicate to determine whether *Split-Monomials-Plus* is applicable in outline pattern `(nonexistent existent)` is thus computed as follows: In a first cycle *constraint rules* that do not need variable instantiations other than given by parameters and available proof lines are evaluated in an `and` statement:

```
(and
 (and (data~primitive-p x )
    (numberp (keim~name x )))
 (and (data~primitive-p y )
    (numberp (keim~name y )))
```

In this case, the two embedded code constraints from the specification are evaluated. `x` and `y` are parameters, so they are already instantiated at the time the applicability predicate is evaluated. The last argument of this `and` statement is a `let*` statement which is the result of the computation in following cycles. The first cycle of *construction* yields:

```
(let*
 ((x0
   (data~struct-at-position l1 pos))
  (num
   (env~lookup-object :num
    (pds~environment omega*current-proof-plan)))
  (times
   (env~lookup-object :times
    (pds~environment omega*current-proof-plan))))
```

The theory symbols are looked up here, and the subterm of `l1` at position `pos` is bound to `x0`. These instantiations are again used in the next *constraint* cycle:

```
(and
 (data~appl-p x0)
 (term~equal times
  (data~appl-function x0))
```

Here the structural property of `x0` being an application and having the function symbol `times` is assured. The next *construction* cycle yields:

```
(let*
 ((z
   (term~constant-create
    (+ (keim~name x ) (keim~name y )) num))
  (x1
   (data~appl-arguments x0)))
```

The variable `z` was instantiated using the code snippet from the tactic's constraints, and `x1` is instantiated as the list of arguments of the expression `x0`. This is used in the last *constraint* cycle:

```
(and
 (and (data~primitive-p z )
(numberp (keim~name z )))
 (listp x1)
 (=
  (list-length x1)
  2)
 (term~equal z
  (nth 0 x1))))))))
```

In this case the embedded code snippet to test whether `z` is a number is evaluated, and `x1` is tested to be a list of length 2. Furthermore two ways to instantiate `z` have been found, and both instantiation are tested to be equal. The first is to add the number values of parameters `x` and `y` and create a term from the result (as done in the last *construction* cycle), and the second is to analyse `l1` and lookup `z` according to the position it should occur in (here the first argument of the application `x0` found at position `pos` in `l1`).

4.3.4 An Example

The purpose of TACO is not only to generate executable code for term matching, but also to bring it to application. This means to produce system-specific declarations and function headers. In its current implementation, TACO is able to generate all code that is required to make a tactic ready for use in ΩMEGA. Furthermore the code to define an ΩMEGA command is generated, such that the tactic can be used from ΩMEGA's command line in interactive proof development. Further use of the abstract tactic definition developed in TACO is thinkable, e.g. the definition of agents within the ΩANTS mechanism [9] could be automated.

In its current implementation, TACO produces from the tactic specification of *Split-Monomials-Plus* (see figure 4.1) the following code:

First the tactic is declared along with its outline patterns and parameters. Outlines are mapped to the respective implementation of their application schemes.

```
(infer~deftactic split-monomials-plus
 (outline-mappings
  (((nonexistent existent)
    split-monomials-plus-1)
   ((existent nonexistent)
    split-monomials-plus-2)
```

```
   ((existent existent)
    split-monomials-plus-3)))
 (expansion-function taco=expand-split-monomials-plus)
 (parameter-types position term term)
 (help "Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y."))
```

A ΩMEGA command is defined to use the tactic from ΩMEGA's command line in interactive proof development.

```
(com~defcommand split-monomials-plus
 (argnames l2 l1 pos x y)
 (argtypes ndline ndline position term term)
 (arghelps "a line containg x*a+y*a"
           "a line containing z*a"
           "the position of the term"
           "the first coefficient"
           "the second coefficient")
 (function taco=split-monomials-plus)
 (frag-cats tactics)
 (defaults)
 (log-p t)
 (help "Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y."))


(defun taco=split-monomials-plus
 (l2 l1 pos x y)
 (infer~compute-outline 'split-monomials-plus
  (list l2 l1)
  (list pos x y)))
```

Then the code for each outline pattern is generated. Declarations of functions are suffixed by a number that refers to the order in which outlines have been declared. The suffix 1 refers to the first outline pattern `(nonexistent existent)`. First the tactic is defined for this pattern.

```
(tac~deftactic split-monomials-plus-1 split-monomials-plus
 (in real)
 (parameters
  (pos pos+position "the position of the term")
  (x term+term "the first coefficient")
  (y term+term "the second coefficient"))
 (premises l1)
 (conclusions l2)
 (computations
  (l2
   (taco=split-monomials-plus-1-l2
    (formula l1)
```

```
    pos x y)))
 (sideconditions
  (taco=split-monomials-plus-1-p
   (formula l1)
   pos x y))
 (description "Apply tactic split-monomials-plus
               to pattern (nonexistent existent)."))
```

In pattern `(nonexistent existent)` the premise `l1` is already existent in the proof plan, while conclusion `l2` is not. Thus a function to instantiate `l2` from line `l1` and parameters `pos`, `x` and `y` is required to apply the tactic. In slot `computations` this function is declared to be `taco=split-monomials-plus-1-l2`. Furthermore a predicate function is needed to test whether the tactic is applicable in a given context. In slot `sideconditions` this is declared to be `taco=split-monomials-plus-1-p`. In the following the code of these two functions is generated according to the algorithm described in sections 4.3.2 and 4.3.3. Variables have the prefix `taco-` to avoid name clashes. The instantiation function is thus

```
(defun taco=split-monomials-plus-1-l2
 (taco-l1 taco-pos taco-x taco-y)
 (let*
  ((taco-x0
    (data~struct-at-position taco-l1 taco-pos))
   (taco-x1
    (data~appl-arguments taco-x0))
   (taco-a
    (nth 1 taco-x1))
   (taco-x7
    (list taco-y taco-a))
   (taco-times
    (data~appl-function taco-x0))
   (taco-x6
    (data~appl-create taco-times taco-x7))
   (taco-x5
    (list taco-x taco-a))
   (taco-x4
    (data~appl-create taco-times taco-x5))
   (taco-x3
    (list taco-x4 taco-x6))
   (taco-plus
    (env~lookup-object :plus
     (pds~environment omega*current-proof-plan)))
   (taco-x2
    (data~appl-create taco-plus taco-x3)))
  (data~replace-at-position taco-l1 taco-pos taco-x2)))
```

and the predicate function is

```
(defun taco=split-monomials-plus-1-p
```

```
(taco-l1 taco-pos taco-x taco-y)
(and
 (and (data~primitive-p taco-x )
    (numberp (keim~name taco-x )))
 (and (data~primitive-p taco-y )
    (numberp (keim~name taco-y )))
 (let*
  ((taco-x0
    (data~struct-at-position taco-l1 taco-pos))
   (taco-num
    (env~lookup-object :num
     (pds~environment omega*current-proof-plan)))
   (taco-times
    (env~lookup-object :times
     (pds~environment omega*current-proof-plan))))
  (and
   (data~appl-p taco-x0)
   (term~taco-equal taco-times
    (data~appl-function taco-x0))
   (let*
    ((taco-z
      (term~constant-create
   (+ (keim~name taco-x ) (keim~name taco-y )) taco-num))
      (taco-x1
       (data~appl-arguments taco-x0)))
     (and
      (and (data~primitive-p taco-z )
     (numberp (keim~name taco-z )))
      (listp taco-x1)
      (=
       (list-length taco-x1)
       2)
      (term~taco-equal taco-z
       (nth 0 taco-x1)))))))))
```

The same procedure is repeated for patterns `(existent nonexistent)` and `(existent existent)`. For these patterns, apart from the tactic's declaration, the following functions are generated for pattern `(existent nonexistent)`:

```
taco=split-monomials-plus-2-l1 (taco-l2 taco-pos taco-x taco-y)

taco=split-monomials-plus-2-p (taco-l2 taco-pos taco-x taco-y)
```

and for pattern `(existent existent)`:

```
taco=split-monomials-plus-3-p (taco-l2 taco-l1 taco-pos taco-x taco-y)
```

Note that the argument lists of these functions are adapted to the corresponding PAI situation. For pattern `(existent nonexistent)`, proof line `l2` is already instantiated, for pattern `(existent existent)` both proof lines `l1` and `l2` are instantiated. All further arguments are parameters of the tactic and therefore available for all of the patterns. As all proof lines are already instantiated for pattern `(existent existent)`, no instantiation function has to be generated and only a predicate function is required. The complete code generated by TACO can be found in appendix B.

While this code example can be reasonably considered a deterring example, its adaption to a modified functionality is easy in TACO. As already mentioned, this is a simplified form of *Split-Monomials-Plus*. Unlike the "real" tactic, the coefficient 1 is not suppressed if it occurs. In TACO, this feature can easily be added by changing the definition of the premise from

```
(l2 (formula phi
      (plus (times x a) (times y a))
      pos))
```

to

```
(l2 (formula phi
      (plus
       {if (= (keim~name ?x) 1)
         ?a ?(times x a)}
       {if (= (keim~name ?y) 1)
         ?a ?(times y a)})
      pos))
```

where both coefficients `x` and `y` are examined and occur in the new instantiated line only if they do not equal 1. The according code is generated in an instant by TACO.

The generation of the expansion function is omitted here, because it is the subject of the following section 4.4.

4.4 Expansion of Tactics

In the previous sections the generation of the code for a tactic with respect to its application was described. However to apply a tactic developed this way without threatening the logical correctness of the resulting proof, a logical justification of the result has to be provided, too. To do so, the concept of tactics as it is used in the ΩMEGA system consists of two parts: first the application of a tactic, i.e. the functionality of analysis of a proof situation, the generation of new proof lines and their correct insertion into the proof plan; second the mechanism to generate a justification for the application of the tactic. In general a tactic application can be justified by the application of a sequence of less complex inference steps, or, vice versa, a tactic is used to package a more or less complex algorithm that is based on a set of inference steps into the single entity of an inference step.

The mechanism to establish a logical justification of the results brought about by a tactic's application is called expansion. Expansion means that the ΩMEGA system refines the justification of proof lines by tactics: starting with the original outline of the tactic, the expansion mechanism, which is a tool to make a proof plan hierarchical, starts an algorithm that results

into a subproof that justifies the conclusions of the outline and whose premises are those of the outline. Within the ΩMEGA system such expansion algorithms are implemented in native COMMON LISP.

The approach used by the TACO system to simplify the implementation of these expansion algorithms and thus to provide an easier access to the development of provably correct tactics is to base their implementation on a simple procedural syntax to describe a sequence of inference steps, along with the possibility to implement conditional branching. Apart from the adaptions to be made to fit the context of proof development, this syntax has the characteristics of a simple imperative language. Such an abstract *expansion* is described by the following grammar:

expansion	:=	*step* | *step expansion*
step	:=	*inference* | *conditional*
inference	:=	`(inference` *infer outline [parameters]*`)`
conditional	:=	`(case` *term case**`)` | `(if` *cond expansion expansion*`)`
case	:=	`(`*term expansion*`)`

where *term* denotes a POST expression, *cond* can either be an embedded code fragment or an equation of the form *term* = *term*, which is analogous to conditions within a tactic's specification described in 4.2. A single inference step is specified by the name *infer* of the tactic to be applied, *outline* denotes the outline of the tactic and the optional argument *parameters* is used to specify additional parameters.

The inference steps of such an expansion algorithm are to be executed sequentially. In case of an *inference* statement the respective inference step is applied, in case of a conditional statement the condition is evaluated and an expansion sequence is chosen accordingly. The purpose of TACO in this context is to translate the abstract specification of an expansion algorithm to COMMON LISP respectively KEIM code. In detail this requires TACO to adapt eventually applied inference steps and their outlines and parameters to an application within an actual proof plan and also to extract information needed to instantiate meta variables that are required to evaluate conditions in conditional statements. When doing so, the name space of meta variables is the same as for the specification of the tactic's application, i.e. variables used to define an expansion algorithm can refer to variables used in the specification described in section 4.2.

Concerning an application of an inference step, an adaption to an actual proof plan requires to control the instantiation of abstract proof lines in the specification with proof plan nodes. In general we can assume an initial set of abstract proof lines $\mathcal{I}_0$ to be instantiated, i.e. a node in the actual proof plan is assigned to each abstract line $l \in \mathcal{I}_0$. When applying a sequence $s = [infer_1, ..., infer_n]$ of inference steps, each step $infer_m$ may produce additional instantiations. Since each step $infer_m$ is applied in a PAI situation, some of the lines of its outline $\mathcal{O}_m$ may already have instantiations, some may not. According to the set $\mathcal{I}_{m-1}$ of lines that are instantiated prior to application of $infer_m$, an application scheme is chosen, and its application completes the instantiation of abstract lines in $\mathcal{O}_m$. Thus the set of abstract lines that have an instantiation after step $infer_m$ can be determined by $\mathcal{I}_m = \mathcal{I}_{m-1} \cup \mathcal{O}_m$, and furthermore for each sequence of expansion steps s, given an initial set of instantiated proof lines $\mathcal{I}_0$, the set of lines $\mathcal{I}_s$ that have an instantiation after execution of this sequence can be determined.

While this scheme applies for sequences of inference steps, some adaptions have to be made for conditional statements. When a conditional statement is processed, the condition of

the statement is evaluated and one of the conditional expansion sequences $s_1, ..., s_n$ is chosen accordingly. Note that each of these sequences may lead to different sets $\mathcal{I}_{s_i}$ of proof lines that have instantiations after processing the whole conditional statement. As this makes it difficult to analyse the expansion algorithm, the following restriction is imposed to the use of conditional statements: the set of proof line variables that are not instantiated in all branches $s_1...s_n$ of a conditional statement are required to be distinct from the set of proof line variables used in following statements (this is furthermore of importance when reusing the expansion algorithm to create code for argument instantiation and applicability predicates, see section 4.5). The consequence is that all variables that are used in later statements can be consistently tested for being instantiated yet or not. To do so the intersection of the set of instantiated proof lines of all branches $\mathcal{I}_{cond} = \bigcap_{i=1}^{n} \mathcal{I}_{s_i}$ is determined. Now for all proof line variables l in later statements applies either $l \in \mathcal{I}_{cond}$, in which case the l has an instantiation after processing the conditional statement, or $l \notin \mathcal{I}_{cond}$, in which case it has not (or the above restriction has been violated).

Therefore for each occurrence of a proof line variable l it can be determined whether or not an instantiation is computed previously, the arguments for each inference of the tactic's expansion can be supplied in a straightforward way: If a line has not yet been instantiated, `nil` is supplied, the instantiated proof node otherwise.

Thus the expansion definition of *Split-Monomials-plus*, given by:

```
(inference expand-num (l3 l1)({pos~add-end ?pos 1} x y))
(inference distribute-right (l2 l3) (pos))
```

can be processed in a straightforward way. In the tactic's declaration a function is declared that implements the expansion steps. Its arguments are the tactic's outline and its parameters. Within this function, the expansion is initialised by the function `tacl~init` and ended by `tacl~end`. Inbetween, tactics are applied using the function `tacl~apply`, its arguments are the outline the tactic is applied to and its parameters. For *Split-Monomials-Plus*, the following function code is generated:

```
(defun taco=expand-split-monomials-plus (outline parameters)
 (let* ((taco-l2 (nth 0 outline))
        (taco-l1 (nth 1 outline))
        (taco-pos (nth 0 parameters))
        (taco-x (nth 1 parameters))
        (taco-y (nth 2 parameters)))
  (tacl~init outline)
   (let* ((outline1 (tacl~apply 'expand-num
                      (list nil taco-l1)
                      (list (pos~add-end taco-pos 1) taco-x taco-y)))
          (taco-l3 (nth 0 outline1)))
    (tacl~apply 'distribute-right
      (list taco-l2 taco-l3)
      (list taco-pos)))
  (tacl~end)))
```

In this function, each proof line in the outline and each parameter is bound to a variable first. Then the expansion is initialised by `tacl~init`. Then the tactics in the expansion

declaration are sequentially applied. As the result may be used to instantiate new proof lines for further processing, all of these tactics are applied in a `let*` statement. Only the last tactic will certainly not produce an outline that has to be processed any further. Here the tactic *expand-num* computes proof line `taco-l3` which is then selected from the new outline of *expand-num* and used to apply *distribute-right*. Finally the expansion is ended by `tacl~end`.

In general, the code of the expansion function that is generated by TACO will be of the following form:

```
(let* variable instantiations
      (tacl~init)
      (let* ((outline1 apply tactic_1)
             process outline1
             (outline2 apply tactic_2)
             process outline2
             ...  )
       apply tactic_n)
      (tacl~end)
```

Here alternately a tactic is applied, then the computed new outline is used to instantiate the proof lines for the application of further tactics. The outline computed by the last application of a tactic, $tactic_n$, is not processed any further and is therefore placed in the body of the `let*` statement. *Variable instantiations* means here not only the instantiation of proof lines and parameter variables from the tactic's outline and the list of its parameters, but can also instantiate further variables that have occurrences in the expansion declaration. Variables can occur here either in parameter specification for expansion tactics or in the condition of a conditional statement.

An example is the expansion of the modified tactic *Split-Monomials-Plus* with the additional feature of 1-elimination. The expansion declaration is now:

```
(inference expand-num (l3 l1)({pos~add-end ?pos 1} x y))
(inference distribute-right (l4 l3) (pos))
(if {= (keim~name ?x) 1}
  (inference 1*e (l5 l4)({pos~add-end ?pos 1}))
  (inference same (l5 l4)))
(if {= (keim~name ?y) 1}
  (inference 1*e (l2 l5)({pos~add-end ?pos 2}))
  (inference same (l2 l5)))
```

In this case two conditional statements are added to eliminate a multiplication by 1 using tactic *1*e*. If the coefficient does not equal 1, a no-operation tactic *Same* is employed to avoid a violation of the requirement that all branches of a conditional statement have to instantiate the same set of proof lines. This requirement allows to treat a conditional statement like a tactic, where the set of possibly instantiated proof lines corresponds to the outline. In code generation, the code schema of the expansion function as described above is modified to encode conditional branches. The computed new outlines are used to assemble the outline of the whole branch. The modified code schema is the following:

```
(let* ((outline1 apply tactic_1)
       process outline1
       (outline2 apply tactic_2)
       process outline2
       ...  )
       (outline_n apply tactic_n)
       process outline_n
 assemble branch outline)
```

To implement the whole conditional statement, a code fragment according to this schema is generated for each conditional branch, the correct branch is selected by the LISP conditional `if` respectively `cond` (to implement `case` statements), which is then inserted like a tactic application using `tacl~apply`. For the modified example of *Split-Monomials-Plus*, TACO translates the conditional statement

```
(if {= (keim~name ?x) 1}
  (inference 1*e (l5 l4)({pos~add-end ?pos 1}))
  (inference same (l5 l4)))
```

to the code fragment

```
(if (= (keim~name taco-x) 1)
  (let* ((outline1
          (tacl~apply '1*e
            (list nil taco-l4)
            (list (pos~add-end taco-pos 1))))
         (taco-l5 (nth 0 outline1)))
   (list taco-l5 taco-l4))
  (let* ((outline1
          (tacl~apply 'same
           (list nil taco-l4)))
         (taco-l5 (nth 0 outline1)))
   (list taco-l5 taco-l4))))
```

which can be used like a tactic application in the example of the simplified version of *Split-Monomials-Plus*. Outline and parameters of tactic applications are instantiated analogously to the previous example, and the whole `if` statement returns a new outline, here `(taco-l5 taco-l4)`. In the generated code, these conditional statements can occur in any place a statement of the form `(tacl~apply` *tactic outline parameters*`)` may occur, i.e. its return value can be used the same way to instantiate proof lines for further processing in the expansion function.

4.5 Development of Algorithms

The automated development of algorithms is based on an idea that was first presented at the Calculemus conference by the author and Sorge [76]. The level of abstractness in the

implementation of expansion algorithms described in the previous section is higher than that provided by an implementation in native COMMON LISP. This allows, to a certain degree, to use the information gained from the analysis of such implementations not only to generate code to expand a tactic, but also for the generation of code for the application of a tactic.

To do so, it is necessary to understand the differences of the typical circumstances and the purposes of a tactic's expansion on the one hand side and of its application on the other. In general a tactic's outline is fully instantiated at the time the tactic is expanded. Therefore the expansion algorithm can be executed in a straightforward way, i.e. the inference steps of the algorithm are applied in the order they are given, with respect to the correct branching when processing conditional statements. The expansion of a tactic therefore resembles in many aspects the sequential execution of a programming language. Additionally, of course, aspects of proof construction and representation have to be respected, for instance new proof lines that are generated during the process of expansion have to be correctly inserted into the proof plan and it is necessary to keep track of dependencies between proof lines. Nevertheless the generation of executable code from an abstract specification of a tactic's expansion mechanism is still straightforward.

When generating executable code for a tactic's application, however, some uncertainties in the context have to be dealt with. In first line it is in general not assured that all proof lines in the tactic's outline exist in the current proof plan at the time a tactic is intended to be applied. As furthermore the applicability of particles from an abstract expansion algorithms, in general the (conditional) application of an inference step, requires some of its arguments to be instantiated before, the order of application of these particles is critical and depends on the tactic's application scheme that has to be constructed.

In this approach the construction of code for a tactic's application from an abstract specification of its expansion is based on two premises: First the inference steps in a tactic's expansion specification that are used to construct the code for its application are restricted to tactics generated by TACO. This restriction allows to use information about implementational matters, that are naturally available to the TACO system at the time these inference steps are generated. Not only is the set of application schemes these inference steps can be applied to known, but TACO's standardisation of function naming makes internal LISP functions in the generated code also available when generating code for complex combinations of such tactics.

Second the adaption of particles of code to different application schemes can be undertaken by TACO's algorithm for code generation that has been described above. To bring both foundations together it is necessary to fit the particles of code gained from an expansion specification into the scheme of constraint rules and construction rules described in section 4.3.1, which is explained in the following.

A simple example is the implementation of an algorithm to add two natural numbers in Peano arithmetic. It is based on two rewrite rules:

$$\begin{aligned} &\textit{Peano-Plus-Step}: && s(x) + y \rightarrow x + s(y) \\ &\textit{Peano-Plus-End}: && 0 + y \rightarrow y \end{aligned}$$

In TACO, these two rules are implemented as tactics. The according abstract declarations are for *Peano-Plus-Step*:

- *Premises*:

```
(l1 (formula phi (plus (s a) b) pos))
```

- *Conclusions*:

```
(l2 (formula phi (plus a (s b)) pos))
```

- *Parameters*:

```
(pos position)
```

- *Patterns*:

```
(existent nonexistent)
(nonexistent existent)
```

and for *Peano-Plus-End*:

- *Premises*:

```
(l1 (formula phi (plus 0 a) pos))
```

- *Conclusions*:

```
(l2 (formula phi a pos))
```

- *Parameters*:

```
(pos position)
```

- *Patterns*:

```
(existent nonexistent)
(nonexistent existent)
```

When TACO has generated the tactic's code accordingly, the LISP functions to implement argument instantiation and applicability predicates are known by name. Function naming under TACO is schematic, function names consist of the name of the tactic and a suffix to specify their purpose. This suffix is composed from the number of the outline in the order outlines are declared, followed by "`-p`", if the function implements an applicability predicate, or the name of the proof line to be instantiated by this function. Arguments of these functions are the formulae of all proof lines that already have an instantiation in the respective outline and the tactic's parameters.

In tactic *Peano-Plus-Step*, the function to find an instantiation for proof line `l2` in outline pattern `(nonexistent existent)`, the second pattern in the declaration, will thus be named `taco=peano-plus-step-2-l2 (l1 pos)`, the applicability predicate function for *Peano-Plus-End* in pattern `(existent nonexistent)` will be named `taco=peano-plus-end-1-p (l2 pos)`. To make use of these functions in TACO's algorithm for code generation, these functions are treated like code snippets. For the two above functions, the according *construction* respectively *constraint rules* are inserted into the construction graph:

```
construction_rule(target: l2
                  base:   [l1,pos]
                  code:   'taco=peano-plus-step-2-l2 (l1 pos)')
```

and

```
constraint_rule(base: [l2,pos]
                code: 'taco=peano-plus-end-1-p (l2 pos)')
```

Except for conditional statements, this is all that has to be done to extend TACO's code generation algorithm to make use of a tactic's expansion declaration. As proof lines, like `l2` in the above example, are variables that represent the formula of the proof line, the integration of functions for argument instantiation and applicability predicates fits naturally in TACO's code construction. Furthermore, this mechanism may be used recursively, i.e. instantiation and predicate functions of the tactic currently processed may be used as well. This allows to define a tactic *Peano-Plus* by its expansion declaration to repeatedly apply tactic *Peano-Plus-Step*. If its the expansion declaration is

```
(inference peano-plus-step (l3 l1) (pos))
(inference peano-plus (l2 l3) (pos))
```

where the outline of *Peano-Plus* is `(l2 l1)`, its only parameter is `pos`, and the outline pattern currently processed is the second pattern `(nonexistent existent)`, the following code will be generated to instantiate `l2`:

```
(let* ((l3 (taco=peano-plus-step-2-l2 (l1 pos))))
  (taco=peano-plus-2-l2 (l3 pos)))
```

The predicate function's generation is analogous:

```
(and (taco=peano-plus-step-2-p (l1 pos))
     (let* ((l3 (taco=peano-plus-step-2-l2 (l1 pos))))
       (taco=peano-plus-2-p (l3 pos))))
```

As this tactic implements an infinitely repeated application of tactic *Peano-Plus-Step*, it will not be applicable in any proof situation. To implement the tactic to end with an application of *Peano-Plus-End* requires a conditional branching. The specification of *Peano-Plus* modified this way is now:

- *Premises*:

  ```
  (l1 (formula phi a pos))
  ```

- *Conclusions*:

  ```
  (l2 (formula phi b pos))
  ```

- *Parameters*:

```
(pos position)
```

- *Patterns*:

```
(existent nonexistent)
(nonexistent existent)
```

- *Expansion*:

```
(case a
  ((plus (s x) y)
    (inference peano-plus-step (l3 l1) (pos))
    (inference peano-plus (l2 l3) (pos)))
  ((plus 0 y)
    (inference peano-plus-end (l2 l1) (pos))))
```

In this case there is a branching of the tactic's application depending on the expression found at position `pos` in proof line `l1` is of the form `(plus (s x) y)` or `(plus 0 y)`. As for the expansion function described in section 4.4, each branch of the conditional statement is treated like a single tactic, i.e. it is assigned an outline and a list of parameters. The outline is again the intersection of the outlines of all branches, here `(l2 l1)`. The parameters are all variables used for evaluation of the condition or required to supply parameters for a tactic's application within the statement. There is furthermore a scope mechanism which allows to use local variables, here `x` and `y`. All variables that are not declared in the tactic's variable declarations are considered local, thus `x` and `y` should not be declared here. The remaining variables or constants and therewith parameters of the conditional statement are in this case `a`, `s`, `0`, `plus` and `pos`. The proceeding is now similar to code generation for tactics: The condition of each branch is evaluated using the algorithm to generate applicability predicates, i.e. in the first branch of the above example this predicate function has to assure that the equation `(a = (plus (s x) y))` holds, which is done by checking the structure of `a` and checking whether `plus` and `s` occur at the right positions. Furthermore the applicability predicates of *Peano-Plus-Step* and *Peano-Plus* have to be evaluated (in the way it is described above). As there is no explicitly defined set of outline patterns for the conditional statements, every possible outline is tested, i.e. here the outline is `(l2 l1)`, thus code generation is attempted for patterns `(nonexistent existent)`, `(existent nonexistent)` and `(existent existent)`. If this code generation for a given outline is successful for each branch of the conditional statement, i.e. instantiation function for every non-instantiated parameter can be constructed, the outline is considered a valid outline pattern of the statement. Now the algorithm produces the according instantiation and predicate functions for each valid outline pattern. These functions are then inserted in form of *construction* respectively constraint rules in the construction graph. In the example of *Peano-Plus*, the following rules will be generated, where `x0` is a variable introduced by TACO for the numeric constant 0:

```
construction_rule(target: l2
                  base:   [l1,pos,a,s,x0,plus]
                  code:   'instantiation function for (nonexistent existent)')
```

```
constraint_rule(base: [l1,pos,a,s,x0,plus]
                code: 'predicate function for (nonexistent existent)')

construction_rule(target: l1
                  base:   [l2,pos,a,s,x0,plus]
                  code:   'instantiation function for (existent nonexistent)')

constraint_rule(base: [l2,pos,a,s,x0,plus]
                code: 'predicate function for (existent nonexistent)')

constraint_rule(base: [l2,pos,a,s,x0,plus]
                code: 'predicate function for (existent existent)')
```

For pattern `(existent existent)`, no *construction rule* has to be generated, as there is no argument to be instantiated. The schema of the code generated from a `case` statement is for *construction rules*:

```
(cond ((predicate code for branch 1
        instantiation code for branch 1)
       (predicate code for branch 2
        instantiation code for branch 2)
        ...
       (predicate code for branch n
        instantiation code for branch n))
```

where the code of every branch depends on the same set of variables. The code schema for *constraint rules* is:

```
(or predicate code for branch 1
    predicate code for branch 2
    ...
    predicate code for branch n)
```

which assures that at least one branch of the `case` statement is applicable. In code generation these *constraint rules* come to application whenever it is possible to apply the *construction rule* of the respective conditional's outline in a given context, because both rely on the same set of variables.

The proceeding for `if` statements is analogous. Nested conditional statements can be processed, too, by executing the above procedure recursively. Integrating *construction* and *constraint rules* from conditional statements as described above, TACO's code generation algorithm can be employed to construct the tactic's code in the usual way.

Code generation based upon expansion declarations is a switchable feature in TACO, i.e. to avoid the generation of redundant code, the integration of expansion declarations in the process of code generation has to be explicitly activated. While this way of tactic design is not suitable to implement tactics whose application is not synchronous to its expansion (e.g. to use more efficient programming at application time, where the expansion only has

to verify the result) nor is it possible to integrate tactics other than those generated by TACO, it is a very comfortable way to rapidly development algorithmic approaches in tactics by combining lower level tactics and the control structure provided by TACO's syntax for expansion declaration. This is of special interest for the implementation of common knowledge bases for the integration of a CAS, as it helps to decouple the CAS algorithms from their verification. This is desirable whenever a CAS algorithm makes use of efficient programming techniques where the computation is hard to remodel in a formal and readable way. In these cases parts of the computation's remodelling can be bridged by a reasoning-orientated reimplementation of these algorithms in TACO.

4.6 Graphical User Interface

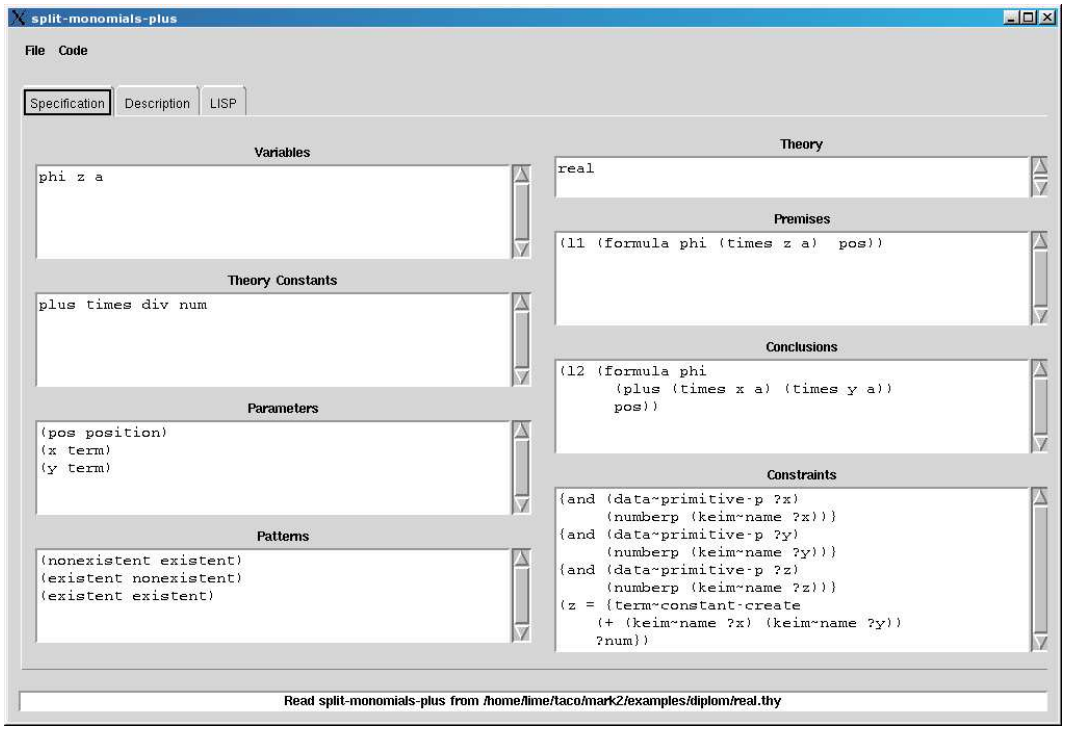


Figure 4.4: The TACO User Interface.

The focus of the TACO system is to provide an easy to use support for the development of inference rules. The way to reach this aim is to provide a scheme to specify these inference rules that is mostly independent of the underlying implementation, this scheme is described in the previous sections. The core of this philosophy is to hide away implementational matters and to have the user confronted with nothing but a specification in a syntax that is close to that of the calculus used in the targeted deduction system; the consequence of this philosophy of an easy to use tool is to provide a graphical user interface (see figure 4.4).

The graphical user interface (GUI) provides a structured environment to develop tactics. Each slot of a tactic's abstract specification as described in section 4.2 corresponds to a text-box in the GUI, equipped with editor features like a check for matching brackets. Apart form

this the GUI provides a page to view and edit the original LISP code produced by TACO. This page consists of two sections, one for generated code, which is overwritten each time TACO's code generation engine is started, and a second so called *protected* area which is part of the tactic's specification and is not overwritten. This part is used to include LISP code that is used in embedded code snippets, e.g. user implemented function definitions.

Beyond a structured representation of tactic specifications, the GUI provides file management facilities. In general several tactics are implemented in a single file, which makes it sometimes hard to keep an overview over tactics stored in a file. TACO's GUI provides a facility to select the tactic to be revised from a list presentation of tactics stored in the respective file.

To bring a tactic to application from the specification developed in TACO's GUI, the tactic is stored in a file. This file is loaded in the ΩMEGA system. Abstract specifications and applicable LISP respectively KEIM code are stored in the same file. The specification is written to file as LISP comments, structured by special tags. The specification is therefore readable for the GUI, but is ignored by ΩMEGA.

4.7 Conclusion

TACO turned out to be a useful tool during the development of the prototypical CAS MASS and its integration into the ΩMEGA system. Its main strength is rapid development of simple inference steps, here ΩMEGA tactics. TACO allows the development of tactics at an abstract level and thus frees the unexperienced user from the necessity to be conversant with the underlying system architecture. Instead of programming at system level, which means in the case of ΩMEGA at the level of the underlying KEIM library, tactics are developed at an abstract level, in which however LISP or KEIM code snippets can be integrated to implement special functionalities. All actual implementation, i.e. among others the coding of headers to declare new tactics and the definition of commands for use from ΩMEGA's command line in interactive proof development, is left to TACO and generated automatically. This considerable reduces the development time for a tactic to be ready for use in a running system.

A further aspect of decoupling the definition of inferences and system architecture is of course portability. In the current implementation of TACO, only embedded code snippets are dependent on the underlying system architecture. As otherwise the major part of adaption of abstract specifications to an actual deduction system is automated, an adaption to other reasoning systems like ISABELLE should be possible with reasonable effort, allowing tactics to be ported from one system to another. But even within a single system the development of inference mechanisms at an abstract level could help to avoid multiple implementations for different purposes. Such multiple implementations are both costly and a possible source of errors.

A thinkable further development would be to make TACO compatible to existing standards for the description of mathematical objects, like OPENMATH [1, 17] and OMDOC [45]. Here it would be desirable to make the abstract definition within TACO independent of a specific system, although the code snippets that can be embedded in TACO proved to be are very helpful feature. Thus to make TACO declarations free of system dependent elements, a standardised and possibly formalised simple programming language, as already advocated in section 3.7, could be a solution for system independent implementation of non-standard features. An example is again the formalisation of the Java Virtual Machine in ISABELLE [66, 6, 62]. The

non-formal attempt to integrate programming language elements in a formal environment by using the control structures of expansion declarations to develop complex tactics (see section 4.5) is a small step in a similar direction.

The development of formal-methods in a semi-formal environment which is type-free, employs snippets of native LISP code and outputs native LISP code may call forth the question for formal correctness of the approach. However TACO proved to be a workable solution. An example of similar character is PHP, a programming language for web applications [49]. PHP is type free, and PHP code is usually a mixture of native HTML code and PHP code fragments that are processed by the webserver. These code fragments are used to produce HTML code, with the only requirement of being interpretable by a Browser. This means that it is possible to use PHP code to produce HTML code with embedded JavaScript [13] functions, whose execution again modifies the resulting web page. While the certainty given by strict type systems in programming languages is lost, PHP is very easy to use, sets very few boundaries on its applications and is one of the most widespread languages for web applications. In the case of TACO, the open characteristics of the development environment proved to be an advantage concerning usability, too. Especially the lack of a type system is reasonable: First, in a real application all issues of type system and formal correctness are backed by the underlying deduction system and its type checker or proof checker. Second a type free environment and simplicity of description is an advantage for portability.

Of theoretical interest is the representation of abstract specifications in a graph as it is used in TACO. Concerning related approaches to represent terms in graph structures, Lafont's Interaction Nets [46] and term indexing techniques as described by Stickel [74], which are widely used in automated theorem provers, e.g. E [68], and have been adapted to higher order logic by Pientka [65], are to be mentioned. A possible application of similar techniques for interfacing purposes is described in chapter 5. In TACO, the implementation of a first order matching algorithm is based upon a graph structure. By association of the resulting graph representation of terms and their relations with programming language elements for term analysis and synthesis, it is possible to implement a two phase matching algorithm that automates the adaption of a matching to different situations. In a first phase, the abstract specification of a tactic is analysed and represented in a graph structure. This graph representation is used to automatically generate an efficient implementation of a matching of these specification against actual expressions in native LISP respectively KEIM code. This allows to considerably reduce the amount of code which has to be executed during proof development, as the possibly costly analysis of the matching specification has to be performed only once and *before*, not during an actual proof search.

Chapter 5

Reactive Behaviour on Shared Terms

5.1 Motivation

While the subject of the previous chapters was mainly the integration of parts of proof generated by a Computer Algebra System *after* its computations have been executed, i.e. the translation of computations into partial proof plans, this chapter will present some thoughts of functionalities that an interface has to provide *before* an integrated Computer Algebra System is invoked. As already described the SAPPER interface provides the functionality to invoke an integrated CAS on various levels of ΩMEGA's proof architecture, and CAS generated proofs can be integrated in various ways into the proof data structure $\mathcal{PDS}$. However, SAPPER has its limitations and there are still functionalities lacking that are desirable for an integration of Computer Algebra algorithms into a state of the art mixed initiative theorem prover, especially during interactive use of the theorem prover.

In detail the SAPPER interface provides the following features for an integration of a CAS into the ΩMEGA environment:

- An abstract representation of a CAS along with the algorithms it provides.
- A mechanism to translate and pass suitable arguments and further relevant information (e.g. the term position where a CAS rewrite step is applied) to the CAS
- A suggestion mechanism to determine whether and how a CAS may be applied to handle a focused goal

The functionality of the SAPPER interface as it is provided by its current implementation allows the rewrite of one term or subterm at a time. An abstract representation of a CAS has to provide a mapping of function symbols as defined in ΩMEGA's theories to native CAS commands respectively algorithms along with suitable translation functions to translate expressions from POST syntax to the CAS's native syntax and vice versa. Using this abstract representation, a suggestion mechanism which employs the SAPPER interface can lookup occurrences of function symbols that can be processed by an external CAS. When the CAS is called to evaluate the respective expression, a rewrite step is applied replaces the original expression by the CAS result or, in verbose mode, a linear sequence of inference step is applied to justify this result.

However, this strategy imposes considerable restrictions to the type of algorithms that can be encoded: the preconditions of a CAS invocation have to be encoded within a single expression, and the restriction to linear rewriting hampers to some extent efficient computational methods like the reuse of partial results as it is used e.g. in a variety of divide-and-conquer-style algorithms and furthermore improves the readability of the resulting proofs. Especially the implementation of a suggestion mechanism could profit from a refined description for external systems and their algorithms. A desirable solution is a description of preconditions for an algorithm in the style of a tactic's outline as described in chapter 4. Of course this approach requires a technically much more complex architecture, the payoff however is a variety of additional features and possibilities.

In the following I will describe an interface mechanism that meets the requirements of such an approach. It is based on a variant of the coordinate indexing and path indexing method described by M. E. Stickel [74] which is used to implement a blackboard architecture that allows a number of external systems to interact with a deduction system and among each other. The blackboard is implemented as a database to store terms and requests and provides the capability to match both.

The mechanism allows the specification of an algorithm's applicability preconditions as a set of first order formulae with meta variables. This extends the functionality of the interface from rewriting a single expression to more complex algorithms which require several preconditions which have not necessarily to be made available in a single proof line. The interface is therefore not only capable to establish the communication between the deduction system and external subsystems, but can also store constraints until the set of preconditions to invoke a given algorithm is completed. The combination of a device to collect constraints and a CAS has already been shown to work well in experiments where the constraint solver CoSIE was used and supported by external CASs (Maple and Mass) [57].

Apart from collecting constraints until sufficient information to invoke an external CAS is available, such a blackboard architecture offers a further possibility for the interaction between deduction system and CAS: allowing to specify a set preconditions of a CAS call instead of a single precondition can not only be used to wait for this set of preconditions to be fulfilled, but also to check such a set for lacking parts of information. This can be used to introduce these parts of a specification as new subgoals to be solved either by passing them to another CAS, or by using another algorithm that is specified in the blackboard, or by returning these subgoals to the deduction system, such that their communication offers mutual support in both directions. Furthermore the refined specification of eventually applicable CAS algorithms allows a better suggestion mechanism that, based on an efficient implementation using indexing techniques and running as a background thread, meets the requirements of complex multi-threaded and modular deduction systems like ΩMEGA.

The reason to choose a variation of path indexing respectively coordinate indexing as the technical basis for such an architecture was in first line its overwhelming success in first order theorem provers. Almost all state of the art first order theorem provers are based on these techniques, like the award winning theorem prover E [68], Waldmeister [37] or SPASS [78]. Term sharing and indexing techniques considerably increased the efficiency of these systems and are responsible for a huge speedup. For a deduction system that operates on a high level of abstraction, like the ΩMEGA system however, the requirements that a matching algorithm has to meet are different. While the application of this technique is mainly focused on fast retrieval of terms matching a given request, the challenge for a highly abstract and complex system that offers a variety of applicable strategies, methods and subsystems is rather to find

a suitable method to tackle a given focused part of a proof. Thus the requirement that a matching algorithm has to meet here is rather to find requests that match a given term. In the following I will introduce an adaption of coordinate and path indexing, that opens the fast and efficient technique of indexing and term sharing to a complex high level deduction system.

A benefit of shared representation of terms is the uniqueness of representation within the data structure, i.e. syntactically equal terms have a single representation. Thus, associating a term with a single fixed node in the representation graph, allows to quickly retrieve all data related to that special term and to identify superterms in which this term occurs. A further advantage is the reduction of cost to evaluate properties of terms, e.g. for type checking, as such evaluations have to be computed at most once for each syntactical structure. Finally a shared graph representation of terms allows to define a mapping of all syntactical structures that are currently in use to a set of natural numbers, which dramatically cuts the cost for term passing and especially for (syntactical) equality testing. Many operations on terms can be reduced to operations on sets of natural numbers, for which very efficient algorithms exist. An actual examination of terms can be avoided in most cases.

Apart from this, a reduction of cost is obtained by the employment of a positional tree for coordinate indexing. It is especially helpful to identify superterms in which specified terms occur and to identify terms according to the occurrence of specified syntactical structures in particular positions, as it provides direct access, i.e. access at constant cost, to information related to a specific term position, e.g. all syntactical structures that have occurrences in this position within some term. An association of syntactic structures and related information has not to be actively established, but is a genuine part of the representation, which is similar to the concept of associations in human thought.

A third aspect of this work is the development of a reactive database, i.e. a database which reacts when terms meeting a given term pattern are inserted. The purpose fulfilled by such a structure is of interest to the resort of mathematical knowledge management. Similar functionality has been implemented using agentified approaches where agents are autonomously searching a database according to term patterns. Unlike active agents however, the reactive behaviour of a database is based mainly on passive data, actual computation is triggered by insertion of suitable structures, and operation on unsuitable terms is therefore reduced by a considerable degree.

5.2 Data Structure

The concept of term sharing presented in this work is based upon the idea of considering terms as relations of symbols rather than strings or syntax trees. Assuming symbols being identifiers of unique and constant objects, an occurrence of a symbol in a term can be considered a reference to this object. As these objects are constant, not only symbols have a fix semantics, but so do any terms build around these symbols. Thus we can assign an identifier to such a term and consider its syntactical structure the object that is denoted by this identifier. Furthermore an occurrence of this term as a subterm in a superterm can as well as for symbols be treated as a reference to this object. The consequence is that any possible subterm occurring in a set of terms to be represented has a unique identifier. Using these identifiers are the nodes of a graph, the structure of the terms is represented in the edges of this graph. An edge denotes here the relation of a subterm and its superterm. The consequence is that

multiple occurrences of the same syntactical structure have only a single representation, i.e. any property of this structure has to be analysed at most once, and, provided that the edges of this graph can be searched in both directions, all occurrences of this structure in a whole set of terms can be efficiently looked up.

In the following I will introduce a data structure that implements this concept along with basic operations a term database should support, i.e. database operations like adding, deleting or finding terms as well as term manipulations like copying or modifying terms. The term structure employed here is the most basic notion of expressions: every expression is either a symbol or it is constructed from other expressions by λ-abstraction or application. A similar representation is employed in OpenMath as the most general structure to represent mathematical objects. All further specification of terms, e.g. typing of terms or discriminating constants and variables, is omitted. This lax handling of type system and semantic issues is reasonable, as it preserves the data structure's openness for various purposes. It's applicability ranges from a device to filter terms for interfacing purposes over blackboard architectures to databases for proof objects, and if needed, all operations can be backed by a type checker or an examination of semantic issues.

However an important requirement for a specific term system is the uniqueness of notation, i.e. identical objects are denoted by identical terms.

5.2.1 Notation

To begin, I will give a quick overview of symbols that are used in the following.

The main structure of the concept is a graph structure, or accurately two graphs, whose nodes are labelled by elements from two sets of identifiers $\mathsf{ID}_{\mathcal{G}}$ to identify term nodes and $\mathsf{ID}_{\mathcal{D}}$ to identify nodes of the indexing tree. $\mathsf{ID}_{\mathcal{G}}$ is furthermore composed from the following subsets: Σ denoting the alphabet of symbols, $\mathcal{T}$ denoting the set of non-primitive terms and $\mathcal{X}$ denoting the set of bound variables.

5.2.2 Terms

The set of terms is defined over an alphabet of symbols $\Sigma \subset \{s_n | n \in I\!N\}$, this is the set of all symbols that may have occurrences in a term. The set of terms $\mathcal{F}$ is inductively defined:

- every symbol is a term.

  $\forall s{\in}\Sigma.s{\in}\mathcal{F}$

- an application of terms is a term. Note that an application is n-ary, i.e. it is constructed from a function term and n arguments.

  $\forall f, a_1, ..., a_n \in \mathcal{F}.f(a_1, ..., a_n) \in \mathcal{F}$

  A further notation of an application is

  $\mathsf{apply}_n(f, a_1, ..., a_n) = f(a_1, ..., a_n)$

  where both notations are equivalent. Note that the arity of an application (which does not necessarily equal the arity of the function denoted by the function term), is explicitly specified here.

- an abstraction of a term is a term. Unlike the common notation of abstractions, a lambda binder does not specify a bound variable, but the mapping of bound variables

to the appropriate binder is determined by the name of the bound variable, see below for details.

$\forall a \in \mathcal{F}.\lambda.a \in \mathcal{F}$

and

- elements of the set of bound variables $x \in \mathcal{X} = \{x_n | n \in I\!N\}$ are terms

  $\forall x \in \mathcal{X}.x \in \mathcal{F}$

To obtain a uniform representation of abstraction terms and their subterms, bound variables are denoted by numbered special identifiers x_n. One of the special strengths of the data structure is bottom-up search, i.e. the efficient identification of superterms of a given expression in the database. This requires a uniform notation of expressions with occurrences of bound variables. The solution is a deBruijn-like numbering of bound variables [25]. The numbering of the bound variable denotes here the distance of variable and binder in terms of scopes. Considering the expression

$$\lambda a.\Phi_0(\lambda b.\Phi_1(\lambda c.\Phi_2(a))),$$

each Φ_n is the scope of one of the λ-binders. Thus all occurrences of a in Φ_0 are in the direct scope of the binder λa and therefore the scope distance is 0. For occurrences of x in Φ_1, there is one binder λb between the variable and its binder, thus the scope distance is 1. Analogously the scope distance increments to 2 for Φ_2, thus the correct transformation of the above expression into anonymous notation is

$$\lambda.\Phi_0(\lambda.\Phi_1(\lambda.\Phi_2(x_2))).$$

Considering the expressions in anonymous notation

$$\lambda.\Phi_0(\lambda.\Phi_1(\lambda.\Phi_2(x_2)))$$

and

$$\lambda.\Psi_0(\lambda.\Psi_1(\lambda.\Phi_2(x_2))),$$

both have the common subterm $\Phi_2(x_2)$. This structure has the same representation independent of the superterm in which it occurs. Therefore this notation is suitable for term sharing.

5.2.3 Shared Terms Graph

The nodes of the graph are denoted by either symbols $s \in \Sigma \cup \mathcal{X}$ or by identifiers $\texttt{id} \in \mathcal{T} = \{t_n | n \in I\!N\}$, i.e. the set of possible identifiers is

$$\mathsf{ID}_{\mathcal{G}} = \Sigma \cup \mathcal{X} \cup \mathcal{T}.$$

These nodes have the following meaning:

- symbols in $\mathcal{F}$, i.e. symbols from the alphabet or identifiers of bound variables $s \in \Sigma \cup \mathcal{X}$ denote themselves and have no special attributes.

- node identifiers $t \in \mathcal{T}$ denote non-primitive terms. Their attributes consist of a flag whose value is either `abstraction`, in which case a further attribute is the identifier $\mathtt{id} \in \mathsf{ID}_{\mathcal{G}}$ of the respective subterm in the range of the abstraction, or `application`, which requires as further attributes a number m denoting its arity and a list $[\mathtt{id}_0, ..., \mathtt{id}_m] \in \mathsf{ID}_{\mathcal{G}}^m$ denoting its function and argument subterms.

A further attribute that is common to all node identifiers is a list of direct superterms in which the respective term occurs and in case of an application a number $n \geq 0$ denoting its position in the superterm. The lists of subterms and superterms of a node `id` will be denoted $\mathtt{id}.T_{sub}$ respectively $\mathtt{id}.T_{super}$ in the following.

The result is a structure that is similar to the syntax tree of the represented term with respect to unique representation of subterms, i.e. if a subterm has multiple occurrences within the term to be represented, the resulting structure is not a tree, but a graph.

An interesting side effect is that infinite term structures can be easily represented by cyclic graphs. As this however may not necessarily be useful, but may also be extremely harmful to the properties of the graph, the consequences of an application of such structures should be evaluated carefully before using them.

Adding new Terms

When adding new terms to the graph, the invariant of having only a single representation for each distinct syntactical structure has to be carefully kept, as a violation of the uniqueness of term identifiers threatens the correctness of the concept. The procedure $\mathtt{add} : \mathcal{F} \rightarrow \mathsf{ID}_{\mathcal{G}}$ takes one argument $f \in \mathcal{F}$ and returns a possibly new node identifier $\mathtt{id} \in \mathsf{ID}_{\mathcal{G}}$, furthermore it performs all necessary modifications to the graph.

When beginning to encode a set of terms, the nodes of the graph is the set of symbols $\Sigma \cup \mathcal{X}$, and the graph features no edges. A graph representing a set $\{f_1, ..., f_n\}$ of terms can be constructed by sequentially adding these terms to an empty graph. The procedure `add` has the following effect if applied to graph Γ and argument $f \in \mathcal{F}$:

- if $f \in \Sigma \cup \mathcal{X}$, i.e. if the term is a primitive, then $\mathtt{add}(f) = f$. Γ is not modified.
- if $f \notin \Sigma \cup \mathcal{X}$, i.e. if the term is a non-primitive, then $\mathtt{add}(f) = t$, where $t \in \mathcal{T}$ is either a node of the graph or a new node that is then inserted in the graph, depending on whether f is already represented in the graph. To test whether a non-primitive term is already represented in Γ it is recursively added, e.g. for an n-ary application $\mathsf{application}_n(f_0, ..., f_n)$, where f_0 denotes its function term and $f_1, ..., f_n$ its arguments, each subterm is added by evaluating $\mathtt{add}(f_i)$. An abstraction $\mathsf{abstraction}(f_0)$ is treated analogously like an application of arity 0. If the term f is already represented, then it is, due to the uniqueness of term identifiers, the only element in the intersection of the subterms' sets of superterms:

$$t = \begin{cases} t_f & \text{if } \bigcap_{i=0}^{n} \mathtt{add}(f_i).T_{super} = \{t_f\} \\ & \text{with respect to relative positions} \\ & \text{of subterm and superterm} \\ \\ t_{new} & \text{if } \bigcap_{i=0}^{n} \mathtt{add}(f_i).T_{super} = \emptyset \\ & \text{again with respect to} \\ & \text{the relative positions} \end{cases}$$

Checking the relative position of superterm and subterms is necessary to avoid a confusion of permutated terms, e.g. the term identifier t denoting the term $\mathsf{apply}_2(+, a, b)$ can be found by intersection:

$$t \in +.T_{super} \cap a.T_{super} \cap b.T_{super}$$

In this case ignoring the relative positions of subterms and superterms, i.e. that a really is the first argument of the application denoted by t, may cause a confusion of $\mathsf{apply}_2(+, a, b)$ and $\mathsf{apply}_2(+, b, a)$. Further confusions may be provoked by ignoring the arity of the application in question.

If the returned identifier t denotes a new node, then t has finally to be added to the subterms' list of pointers to their superterms, and the type flag of the new node has to be set to denote either an abstraction or an n-ary application.

This procedure is not only an algorithm to insert new terms, but also one to find terms. The concept of tracking the syntax tree bottom up allows some refinement for efficient lookup of nodes. The number of necessary intersections of sets of node identifiers e.g. can be reduced by choosing subterms that occur in few superterms and therefore require intersection of small sets. If the set of candidate identifiers of some point of the syntax tree can be narrowed to an empty set, then a new node has to be inserted and so have all identifier nodes of its superterms. Strategies for an efficient search of the graph and the properties of this representation of a term database will be subject to a later section.

In the following some further database and term manipulation operations will be outlined, where the main strategy of the algorithms will also be some kind of navigation through the graph.

Rebuilding Terms

Rebuilding terms is a retransformation of a node in the graph to a term $f \in \mathcal{F}$. To do so a simple procedure $\mathtt{get} : \mathsf{ID}_{\mathcal{G}} \to \mathcal{F}$ is used. This procedure recursively rebuilds a term starting from the root of its syntax tree denoted by $t \in \mathsf{ID}_{\mathcal{G}}$.

The procedure $\mathtt{get}(t)$ is defined as follows:

- if $t \in \Sigma \cup \mathcal{X}$ then the node denotes a primitive and the identifier itself is returned.

  $\mathtt{get}(t) = t$

- if $t \in \mathcal{T}$ and its type flag's value is $(\mathtt{application}, n)$ then the node represents an n-ary application. Let $t.T_{sub} = [t_{f0}, ..., t_{fn}]$ the list of its subterms, then an application $\mathsf{application}_n$ is returned.

  $\mathtt{get}(t) = \mathsf{apply}_n(\mathtt{get}(t_{f0}), ..., \mathtt{get}(t_{fn}))$

- if $t \in \mathcal{T}$ and its type flag's value is $\mathtt{abstraction}$ then the node represents an abstraction. Let $t.T_{sub} = t_{f0}$ its subterm then an application $\mathsf{abstraction}$ is returned.

  $\mathtt{get}(t) = \lambda.\mathtt{get}(t_{f0})$

In this case the graph is only tracked down, which is always deterministic and requires no search. In further operations both upward search and downward tracking is applied, which increases the complexity of the algorithms.

Deleting Terms

To delete a term, the corresponding node is removed from the graph, and so are recursively its subterms. Note that this is only possible and necessary, if the term in question is not a subterm of other terms, which can be checked by examining its list of superterms.

The procedure to do so is $\texttt{delete} : \mathsf{ID}_{\mathcal{G}} \rightarrow \emptyset$, where $\texttt{delete}(t)$ has the following effect:

- if $t \in \Sigma \cup \mathcal{X}$, i.e. if t is a primitive, then the node is removed if it has no occurrences in any superterm, in this case is $t.T_{super} = \emptyset$. If this is not the case, the graph stays unmodified.
- if $t \in \mathcal{T}$ and its type flag's value is $(\texttt{application}, n)$, then t is removed from its subterms' superterm lists, i.e.

  $\forall t_n \in t.T_{sub} : t_n.T'_{super} = t_n.T_{super} \ominus t.$

  Furthermore all subterms whose list of superterms has been emptied this way are deleted:

  $\forall t_n \in t.T_{sub} : (t_n.T'_{super} = \emptyset) \Rightarrow \texttt{delete}(t_n)$
- if $t \in \mathcal{T}$ and its type flag's value is `abstraction` then it is processed in the same way as a 0-ary application.

Note that apart from nodes being subterms of other nodes, some nodes may be of "external interest", i.e. they may be referenced to from outside the system. In this case these nodes should not be removed either, and an implementation should keep track of external references.

When using the graph to implement deductional purposes, then the `delete` procedure should be used to avoid underperformance caused by information overload. It probably will pay off to focus on a selection of proof lines that is cleaned up and updated regularly.

Copying Terms

Within the graph, it is not necessary to copy terms, it actually is even explicitly forbidden, as it would violate the uniqueness of representation.

When using a shared terms graph within the context of a reasoning system however, a selection of term identifiers will be referenced from outside the graph. Multiple references to a single identifier may occur in this case, but the effort necessary to copy and paste terms is reduced to constant cost, as this only requires to operate on the term identifier.

Substituting Subterms

Substitution is a potentially costly operation, as it requires to identify a path of unknown length between two nodes, actually the path from the term's root to the subterm that is to be substituted. In case of multiple occurrences of this subterm one path has to be processed for each occurrence, and attention has to be payed to possible intersections of these paths. The costs to identify these paths can be considerably cut if using indexing techniques.

Given a path $[t_1, ..., t_n]$, where t_1 is the direct superterm of the subterm t_0 to be substituted and t_n is the root of the term, then the subterm of the affected position of each node t_i is substituted by the new identifier, and if there is no node in the graph that equals this substituted node, then a new node is created.

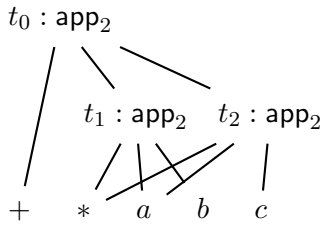


Figure 5.1: An example Term.

Let t'_0 be the term to be substituted for the former subterm t_0, $t_i.T_{sub} = [t_{i,0}, ..., t_{i,m}]$ the list of direct subterms of $t_i \in [t_1, ..., t_n]$ and j the position at which the substitution takes place, then at each node t_i of the path the next step of the substitution is iteratively computed:

$$t'_i = \mathsf{add}(\mathsf{application}_m(t_{i,0}, ..., t_{i,j-1}, t'_{i-1}, t_{i,j+1}, ..., t_{i,m}))$$

Finally the modified root node t'_n is returned where this node denotes the substituted term $t'_n = [t_0 \rightarrow t'_0]t_n$.

In case of an λ-abstraction, the procedure is executed analogously to a 0-ary application, symbols cannot occur on the path.

A special case of substitution is global rewriting where only the pointers between all direct superterms and the subterm in question have to be modified.

Searching for Occurrences of Expressions

One of the main advantages of a shared representation of terms is the speed up that is obtained when searching specified terms in a database. Searching a term in this context means to check whether a term $f \in \mathcal{F}$ has been added to the data structure. Apart from fully specified terms, e.g. $P(a) \vee Q(a)$, terms that have occurrences of meta variables may be of interest, like $P(\alpha) \vee Q(\alpha)$ with meta variable α. In both cases we can make use of the uniqueness of the representation, i.e. we can start a search for expressions as those given above at the graph nodes that represent occurring symbols P, Q or $\vee$. As these are uniquely represented, all nodes that represent expressions in which these symbols occur can be reached by a path starting in the corresponding node.

Thus the basic elements of any search are either tracking down the graph from a node to its subterms or searching the graph upwards by following pointers to superterms. In case of an upward search the result of the operation is in general a set of nodes rather than a single node. If the task is to find a substructure in the graph, there is usually a selection of nodes that are known to be contained in the graph and easy to locate, e.g. symbols $s \in \Sigma$. These are suitable points to start a search.

Apart from the symbols occurring in an expression that is searched for, the queries may contain application nodes and abstraction nodes. For each of these nodes a corresponding node of the graph has to be identified. This can be incrementally done by following pointers from neighboured nodes. In case of an upward search, not a node is identified, but rather a set of candidate nodes. Such sets of candidate nodes can be narrowed for nodes that denote

n-ary applications with $n \geq 1$ by intersection of the candidate sets. These candidate sets can furthermore be used for downward tracking analogously to a single node, the difference is that this operation is not of type $\mathsf{ID}_{\mathcal{G}} \rightarrow \mathsf{ID}_{\mathcal{G}}$, but rather $\mathcal{P}(\mathsf{ID}_{\mathcal{G}}) \rightarrow \mathcal{P}(\mathsf{ID}_{\mathcal{G}})$, i.e. arguments and results are elements of the power set of $\mathsf{ID}_{\mathcal{G}}$.

The complexity of such a search depends on the structure of the graph and of the strategy that is applied, e.g. the order in which candidate sets for expression nodes are chosen for intersection, furthermore it has to be considered under which circumstances downward tracking of a candidate set has to be preferred to upward search from a node or upward search from a candidate set, which is potentially the most expensive strategy.

Usually it pays off to chose small candidate sets to continue the search rather than big ones. If the set of candidates from a specified node narrows to the empty set then the corresponding term is not represented in the graph, and unless it is intended to add the term in question to the database, the search for the whole expression will fail. Therefore the attempt to narrow candidate sets to the empty set can be chosen as a search strategy to quickly bring about negative results.

As an example, I will describe the search for a term $t = a * b + a * c$. The representation of t within the term sharing graph is sketched in figure 5.1.

Nodes denoting symbols, here $+$, $*$, a, b and c, can be immediately identified, then candidate sets for superterms can be narrowed step by step. Candidate sets C for t_1 and t_2 can be determined by intersection, as candidates have to be a direct superterm of all of it subterms, where in this case all subterms are symbols:

$$\begin{aligned} & C_{t_1} \subseteq \\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 0) \in * . T_{super}\} \\ \cap\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 1) \in a . T_{super}\} \\ \cap\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 2) \in b . T_{super}\} \end{aligned}$$

and

$$\begin{aligned} & C_{t_2} \subseteq \\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 0) \in * . T_{super}\} \\ \cap\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 1) \in a . T_{super}\} \\ \cap\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 2) \in c . T_{super}\} \end{aligned}$$

In the next iteration, the candidate sets C_{t_1} and C_{t_2} are propagated to narrow C_{t_0}:

$$\begin{aligned} & C_{t_0} \subseteq \\ & \{t_{super} | (t_{super}, \texttt{application}, 2, 0) \in + . T_{super}\} \\ \cap\ & \bigcup_{t \in C_{t_1}} \{t_{super} | (t_{super}, \texttt{application}, 2, 1) \in t . T_{super}\} \\ \cap\ & \bigcup_{t \in C_{t_2}} \{t_{super} | (t_{super}, \texttt{application}, 2, 2) \in t . T_{super}\} \end{aligned}$$

Note that in this example the query contains no meta variables. The candidate set C_{t_0} therefor narrows either to a singleton set or to the empty set in case there are no occurrences of the term. If a term is not fully specified, i.e. it contains meta variables, the same algorithm is applied. As in this case however some of the superterm sets would not be available, the narrowing of the set of candidates may yield a greater number of terms.

Having identified the node that represents a term, all superterms can be looked up by simply following all superterm pointers upwards, which is useful to find terms specified by a given subterm. Note that in general only a selection of terms found this way is of relevance to the user, e.g. terms that represent a proof line's formula. These nodes should be attributed by additional marks, and generally it is useful to filter the result with respect to these marks.

5.2.4 Positional Tree

The data structure presented so far should be useful for an efficient term sharing and will speed up some of the operations it implements. However some of the procedure are costly and sometimes possibly more expensive than in a non-termsharing implementation. Mainly a search from a node upwards to identify a special superterm may cause considerable effort.

An attempt to cure this is the application of positional trees that allow to identify sets of terms by structural properties. In the following I will describe a positional tree to efficiently look up terms by using their prefixes as keys (unlike an upward search in the term sharing graph in which the suffix is used as a key).

The aim is to maintain a tree that provides enough extra information for a considerable speed up of operations with an acceptable consumption of space and time for its maintenance. Thus a new discrimination node is created no sooner than the first term is added that actually features the corresponding position. The tree is maintained whenever terms are added to or deleted from the representation graph. The tree's structure and how to maintain it is described in the following.

Each of the tree's nodes represents a position in a terms syntax tree and can be used to access all identifiers that denote terms in which this special position exists. The information about these terms is kept as pairs of term identifiers, one denoting the root node of the term, the other denoting the subterm in that special position, i.e. assuming the term $(a + c) + b$ has been added to the graph and is denoted by note t_1, and the term $(a + c)$ is denoted by node t_2, then the node of the positional tree that corresponds to the first argument of an application that is the topmost structure of a term is associated to the pair (t_1, t_2) denoting that term t_1 has an occurrence of term t_2 in this special position.

The pointer to the subterm is bidirectional again, i.e. not only term nodes can be accessed from a specified position node, but also nodes of the positional tree can be accessed from term nodes that are related to it. The set of positions the terms occurs in is recorded in the additional term node slot $t.D_{this}$.

Furthermore each of these nodes is either the root of a positional tree that can be accessed using the subterms in this special position as keys, or in case of symbols $s \in \Sigma \cup \mathcal{X}$, s is kept in this position.

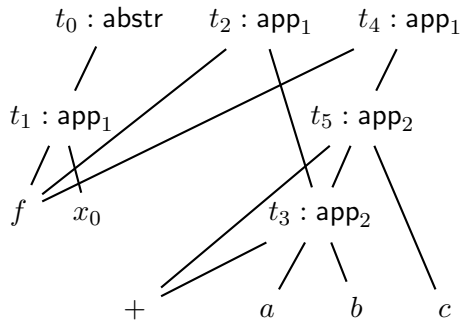
Nodes of positional trees will be denoted by $d \in \mathcal{D} = \{d_n | n \in I\!N\}$ in the following, where d_0 is the root of the tree denoting root position ϵ. Every node d has the attributes $d.T_{this} = \{(t_1, k_1), ..., (t_m, k_m)\}$ to denote the terms related to the node, a pointer to its parent node $d.D_{super} = d_{super}$ and a mapping

$$d.D_{sub} \in \\ \Sigma \cup \mathcal{X} \cup \{\mathsf{abstraction}\} \cup \{\mathsf{application(m, n)} | m, n \in I\!N\} \\ \times \mathcal{D}$$

that is used to map the topmost structure of the key term to an according child node, when accessing the positional tree. In case of an application the mapping discriminates according to its arity m and the position n of the function or argument subterm that is used as a key term in the next iteration.

Figure 5.2 shows an example of a discrimination tree. The tree is result of an empty graph in which the terms $\lambda.f(x_0)$, $f(a + b)$, and $f((a + b) + c)$ have been added. The graph representation of these terms is also shown in figure 5.2. A node

a graph representation of $\lambda.f(x_0)$, $f(a+b)$, and $f((a+b)+c)$:



the according positional tree:

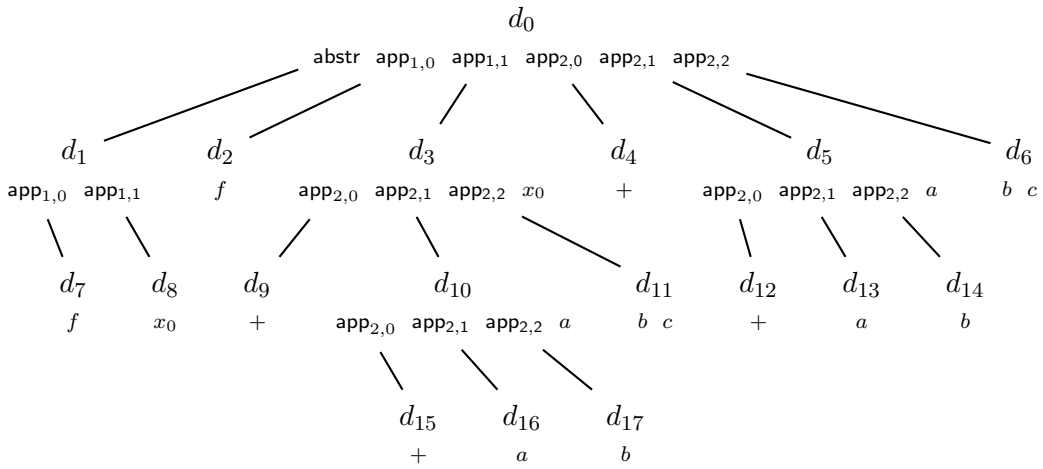


Figure 5.2: An example Positional Tree

$$\underset{k_0 \ldots k_n}{d_n}$$

consists of its identifier d_n and the set of all valid keys $k_0 ... k_n$ at this position. Every node in the tree denotes a unique position, and the set of valid keys is the set of all possible head symbols that occur in any term at this position, e.g. we can see from in figure 5.2 that d_7 denotes the function term of a 1-ary function that is the range of an abstraction. The only subterm that has occurred so far in this position is the symbol f.

The complete list of the sets of pairs (t, k), where t is the root term and k the key used in this special position, is the following:

$$\begin{aligned}
d_0.T_{this} &= \{(t_0, t_0), (t_1, t_1), (t_2, t_2), (t_3, t_3), (t_4, t_4), (t_5, t_5)\} \\
d_1.T_{this} &= \{(t_0, t_1)\} \\
d_2.T_{this} &= \{(t_2, f), (t_4, f)\} \\
d_3.T_{this} &= \{(t_1, x_0), (t_2, t_3), (t_4, t_5)\} \\
d_4.T_{this} &= \{(t_3, +), (t_5, +)\} \\
d_5.T_{this} &= \{(t_3, a), (t_5, t_3))\} \\
d_6.T_{this} &= \{(t_3, b), (t_5, c)\} \\
d_7.T_{this} &= \{(t_0, f)\} \\
d_8.T_{this} &= \{(t_0, x_0)\} \\
d_9.T_{this} &= \{(t_2, +), (t_4, +)\} \\
d_{10}.T_{this} &= \{(t_2, a), (t_4, t_3)\} \\
d_{11}.T_{this} &= \{(t_2, b), (t_4, c)\} \\
d_{12}.T_{this} &= \{(t_5, +)\} \\
d_{13}.T_{this} &= \{(t_5, a)\} \\
d_{14}.T_{this} &= \{(t_5, b)\} \\
d_{15}.T_{this} &= \{(t_4, +)\} \\
d_{16}.T_{this} &= \{(t_4, a)\} \\
d_{17}.T_{this} &= \{(t_4, b)\}
\end{aligned}$$

The position of occurrences are furthermore recorded in term nodes:

$$\begin{aligned}
t_0.D_{this} &= \{d_0\} \\
t_1.D_{this} &= \{d_0, d_1\} \\
t_2.D_{this} &= \{d_0\} \\
t_3.D_{this} &= \{d_0, d_3, d_5, d_{10}\} \\
t_4.D_{this} &= \{d_0\} \\
t_5.D_{this} &= \{d_0, d_3\} \\
x_0.D_{this} &= \{d_3, d_8\} \\
f.D_{this} &= \{d_2, d_7\} \\
a.D_{this} &= \{d_5, d_{10}, d_{13}, d_{16}\} \\
b.D_{this} &= \{d_6, d_{11}, d_{14}, d_{16}\} \\
c.D_{this} &= \{d_6, d_{11}\}
\end{aligned}$$

Note that the positional tree's root node d_0 features the set of all terms that have been inserted so far. Note further that adding a term to the positional tree is fully recursive, i.e. not only the term itself is added, but also each of its subterms.

Assuming an empty set of terms is represented by a tree consisting of an empty tree, i.e. a single root node for position ϵ attributed with $d_0.T_{this} = \emptyset$ to signal that there are no terms related to this position, and $d_0.D_{sub} = \emptyset$, as no subtrees have been inserted yet, then a positional tree can be constructed by sequentially adding terms.

Although the maintenance of this positional tree causes considerable cost, it pays off when searching terms in a database. The insertion of all possible subterms is justified by the advantage gained from this extra information when checking the occurrence of subterms at a non-specified position, as it is required e.g. for efficient application of substitutions.

Adding Terms

To add a new term to the positional tree, I define a procedure `d_add`. The arguments of a call to `d_add` are the term to add, the key to find the appropriate position node to do so and the current node. When calling `d_add`, term and key are identical, while d_0 is the starting node, then the key is sequentially processed by recursively calling `d_add`.

A call to the procedure `d_add`(t, k, d) with term t, key k and discrimination node d will have the following effect:

- in any case (t, k) is added to the set of terms related to the discrimination node d, $d.T'_{this} = d_n.T_{this} \cup \{(t, k)\}$.
- if the key node is a symbol, i.e. $k \in \Sigma \cup \mathcal{X}$, then the procedure terminates this branch, as there are no further subterms according to which a discrimination is possible.
- if the key node is an application, i.e. $k \in \mathcal{T}$ and t's type flag is $(\texttt{application}, n)$, then each of t's subterms is inserted into $d.D_{sub}$:

  For each node t_i in $t.T_{sub} = [t_{sub,0}, ..., t_{sub,n}]$ it is checked whether $d.D_{sub}$ contains a pair $(\mathsf{application}(n, i), d_{sub})$.

  If the key $\mathsf{application}(n, i)$ is pointing to a discrimination node d_{sub}, then the procedure is recursively called:

  `d_add`$(t, t_{sub,i}, d_{sub})$

  If there is no such pair, a new discrimination node d_{new} is created, the mapping is updated

  $d.D'_{sub} = d.D_{sub} \cup \{\mathsf{application}(n, i), d_{new}\}$

  and the procedure recurses:

  `d_add`$(t, t_{sub,i}, d_{new})$

  According to this scheme the positional tree will insert a new discrimination node only if a term is added that actually features the corresponding position.
- if the key node is an abstraction, then the proceeding is the same as for a 0-ary application.

The procedure is called each time a new term is inserted into the term graph. As a term t can only be inserted *after* all of its subterms have been inserted, the procedure `d_add` has already been executed for these subterms at the time t is inserted.

Having added a number of terms to the positional tree, the tree will have a node d for each position that occurs in some term that has been inserted. A unique key to access a discrimination node is a single path in the syntax tree of an arbitrary term. The target position corresponds to the position in which the path ends.

In general a term t is added to the positional tree by inserting it at root node d_0 with itself as the key: $\texttt{d\_add}(t, t, d_0)$.

Therefore $d_0.T_{this}$ is always the list of all terms that have been inserted so far, actually a pair $(t, t) \in d_0.T_{this}$ exists for each term t that is represented in the graph.

Looking up of Sets of Terms

The positional tree allows an efficient lookup of terms according to three criteria. The first criterion is the occurrence of a given expressions at a specified position. To lookup a term of the form $f(\alpha + \beta)$, where α and β are meta variables, in the positional tree depicted in figure 5.2, the proceeding is the following:

- The term position and the according position node of all symbols that are no meta variables are determined, here f and $+$. The according position nodes are d_2 for f(i.e. the function of a unary application) and d_9 for $+$ (i.e. the function of a binary application which itself is the argument of a unary application). Although in this examples only symbols $f, + \in \Sigma$ occur, this can be extended to occurrences of terms t_i which have already representations in the term graph.

- The according root term is looked up. In $d_2.T_{this}$, there are two entries where f is the key symbol, namely (t_2, f) and (t_4, f). The root symbols here are t_2 and t_4. For $+$, the corresponding entries are $(t_2, +), (t_4, +) \in d_9.T_{this}$.

- The sets of root symbols are intersected to find the root terms that have all symbols in the right position. Here the set of root terms is for both symbols $\{t_2, t_4\}$, thus this is the set of terms that fit the pattern $f(\alpha + \beta)$. This is correct, as $t_2 = f(a + b)$ and $t_4 = f((a + b) + c)$ match the query term. The according instantiations for meta variables are $[\alpha \rightarrow a, \beta \rightarrow b]$ respectively $[\alpha \rightarrow t_3, \beta \rightarrow c]$, where $t_3 = (a + b)$.

The lookup procedure can be implemented very efficiently, as all operations on sets of terms can be reduced to operations on sets of natural numbers.

The second criterion is the occurrence of the same meta variable at several term positions. In this the set of candidate terms is further reduced by examining the relevant term positions. The equality test can again be reduced to equality testing on natural numbers. If the pattern in the above example is changed to $f(\alpha + \alpha)$, the set of matching terms is reduced to $\emptyset$, as none of the terms found above have the same instantiation for α and β.

A special feature is the third criterion, the efficient lookup of terms that have the form $\Phi(x)$ for some expression x that occurs at a non-specified position. This is useful for CAS operations that perform rewrite steps at some position within terms. To lookup e.g. all terms that match pattern $\Phi(a + b)$, the procedure consists of two steps:

- First the pattern $a + b$ is looked up according to the procedure for the first criterion. The only term to match this pattern is t_3. Although there are no meta variables in $a + b$, this is no necessary restriction. Occurrences of meta variables are treated in the same way as described above.

- Then in all position nodes recorded in $t_3.D_{this} = \{d_0, d_3, d_5, d_{10}\}$, the according pairs (t_i, t_3) are looked up. The record $d.T_{this}$ in the according position nodes yields:

$$\begin{array}{rcl}(t_3, t_3) & \in & d_0.T_{this} \\ (t_2, t_3) & \in & d_3.T_{this} \\ (t_5, t_3) & \in & d_5.T_{this} \\ (t_4, t_3) & \in & d_{10}.T_{this}\end{array}$$

  Thus the the set of terms matching $\Phi(a+b)$ is $\{t_2, t_3, t_4, t_5\}$ respectively $\{f(a+b), (a+b), f((a+b)+c), ((a+b)+c)\}$.

This third criterion can furthermore be combined with the first, where the occurrence of a given subterm has to be *below* a given position. A pattern which combines both criteria is $f(\Phi(a+b))$. Now the procedure for the first criterion is applied to pattern $f(\alpha)$, the according candidate terms are $\{t_2, t_4\}$. If there is a suitable term for $f(\Phi(a+b))$, then $\Phi(a+b)$ has to occur at the position corresponding to d_3. Thus there has to be a term $t_j \in \{t_2, t_3, t_4, t_5\}$ (i.e. the result for the lookup $\Phi(a+b)$) where $(t_i, t_j) \in d_3.T_{this}$ for some term $t_i \in \{t_2, t_4\}$. In this case there are two suitable records $(t_2, t_3), (t_4, t_5) \in d_3.T_{this}$. Thus the result for the query $f(\Phi(a+b))$ is $\{t_2, t_4\}$ respectively $f(a+b)$ and $f((a+b)+c)$. The combination of criteria one and three requires two further intersection of sets, namely the intersection of the result of the third criterion with the set of key terms t_j that occur in some pair $(t_i, t_j) \in d_k.T_{this}$, where d_k is the position of Φ, and second the intersection of the set of according superterms t_i with the result of criterion one.

Of course all of the criteria can be combined, too. In this case, the procedure to combine criteria one and three are followed by a check whether all occurrences of the same meta variable have the same value.

In general the order in which sets of terms are intersected has no effect on the result, but on the performance. Thus an efficient implementation of the data structure should make sure that small sets are intersected first.

5.2.5 Reactive Behaviour

The basic idea of a reactive database of terms is to store not only terms in the database but also requests. Instead of finding suitable terms for a given request, the task is now to find a suitable request for a given term. This functionality is useful for the implementation of interfaces in heterogeneous logical environments like the ΩMEGA system, where external systems have to be coordinated. In such a database of term requests the actual computation of term matching is triggered by syntactical events like occurrence of a common subterm within a term and a possibly matching request.

As the operations on the term sharing graph as well as on the positional tree are based on some sort of navigation through the graph, there are always nodes that are explicitly travelled when doing so. This allows to attach reactive nodes to the graph to enable the data structure to show some kind of reactive behaviour, as each node can be marked to cast a signal when visited. Thus a term that is added to the database can be matched against a number of term patterns, where the matching is triggered by graph nodes that are visited when doing so, and the consideration of a particular term pattern in the matching process is triggered by insertion of matching subterms.

This offers a possibility to trap terms matching a specified pattern at the time they are added to the data structure. To do so the nodes d_i of the positional tree can be extended by a new attribute

$$d_i.R \subset \{(t_j, m_k)|j, k \in I\!N\}$$

to specify a meta nodes m_k that is notified whenever a new pair of terms (t_l, t_j) is inserted at this node. The message that is sent to m_j is of the form $\texttt{notify}(d_i, t_l)$, i.e. for every meta node m_k the root term t_l that features the respective subterm t_j at the specified position d_i is passed.

This trigger mechanism can be used to implement a reactive version of the first criterion of the lookup mechanism in section 5.2.4. In the following I will describe some examples of reactive term matching in the graphs in figure 5.2. To enable a reaction whenever a term matches pattern $f(\alpha + \beta)$, a new meta node m_1 is introduced, and the respective search requests are added to $d_i.R$ at the according positions. In this example, the consequence is that (f, m_1) is added to $d_2.R$ and $(+, m_1)$ to $d_9.R$. In case a suitable term, say $t_6 = f(c + c)$, is inserted to the term graph, m_1 receives two notifications, namely $\texttt{notify}(d_2, t_6)$ and $\texttt{notify}(d_9, t_6)$.

Each meta node m_k is attributed by the list of positions

$$m_k.D = \{d_1, ..., d_n\}$$

that have to send a notification for a complete identification of a suitable term, here $m_1.D = \{d_2, d_9\}$. As now all position nodes in $m_1.D$ have sent a notification with the same term t_6, t_6 can be identified as a suitable term for the pattern represented by m_1.

The second criterion, the consistence of meta variable instantiations, is applied analogously to section 5.2.4. The description of meta nodes is extended by an attribute

$$m.V = \{(v_1, \{d_1, ..., d_{k_1}\}), ..., (v_n, \{d_1, ..., d_{k_n}\})\}$$

to record meta variables and their respective positions. In case the first criterion is fulfilled by a term t, meta variables are evaluated. If the term t, which has been found according to the first criterion, has for each $(v_i, \{d_1, ..., d_{k_i}\}) \in m.V$ the same subterm t_{sub} in all positions $d_1, ..., d_{k_i}$, t matches the pattern represented by m and for meta variable v_i the instantiation t_{sub} has been found.

To implement the third criterion of section 5.2.4, a further meta node attribute is required to record terms and meta variable instantiations. This attribute is given by

$$m.I = \{(t_i, \{(v_1, t_1), ..., (v_n, t_n)\})|i, n \in I\!N\}$$

where t_i is the matching term and $t_1, ..., t_n$ are instantiations found for meta variables $v_1, ..., v_n$. Each time an instantiation has been found a new instance of $(t_i, (v_1, t_1), ..., (v_n, t_n))$ is added to $m.I$. The meta node attributes are completed by the list of superterms that have to be notified if a new instantiation has been found:

$$m.M_{super} = \{m_1, ..., m_n\},$$

and the list of subterms and their positions:

$$m.M_{sub} = \{(m_1, d_1), ..., (m_n, d_n)\},$$

If now a meta node m_{sub} where $m_{sub}.M_{super} \neq \emptyset$ finds a matching term for the pattern it represents, it will notify all nodes $m_{super} \in m_{sub}.M_{super}$. If a meta node m_{super} has received a notification from all of its subnodes m_i (i.e. all meta nodes that have a record $(m_i, d_i) \in m_{super}.M_{sub}$), the evaluation of the third criterion is executed analogously to section 5.2.4: For each of the subnodes m_i it is evaluated whether it occurs in a suitable position. The evaluation is triggered by m_{super} and not by m_{sub}, because the purpose of each m_{sub} is to match a subterm of the term matched by m_{super}, and thus m_{super} will be the last node to be triggered by a new term.

At the time m_{super} is triggered by a term t_{super}, some of the subterm nodes m_{sub} may have found several instantiations $(t, \{(v_1, t_1), ..., (v_n, t_n)\}) \in m_{sub}.I$. Among these, suitable instantiations, if there are any, are identified:

- The meta variable instantiations are consistent. Thus for each instantiated variable v_i it is evaluated if an instantiation has been found by m_{super}, too, and if both instantiations are consistent. Inconsistent instantiations are dropped.
- For each of the remaining instantiations $(t, \{(v_1, t_1), ..., (v_n, t_n)\})$, it is evaluated if there is a term t_d such that $(t_d, t) \in d.T_{This}$ for some $d \in t.D_{This}$, where $(t_{super}, t_d) \in t_d.D_{This}$, i.e. if t is a subterm of t_{super} below the specified position d. This is done by two intersections of term sets analogous to section 5.2.4, respectively one intersection and one membership test, because one of the sets is the singleton $\{t_d\}$.

For an example, the pattern in the above example is modified to find in the graph of figure 5.2 a term matching $f(\varphi(\Phi(\varphi(a, \beta)), \gamma))$, i.e. a term $f(\varphi(\Phi), \gamma))$ where somewhere in Φ there is an occurrence of the subterm $\varphi(a, \beta)$, where the function symbol φ is the same in both occurrences. The matching is implemented by the following setting:

A first meta node m_1 is employed to wait for the whole term. f is the only symbol that is no meta variable. It has to be found in position d_2:

$$m_1.D = \{d_2\}$$
$$d_2.R = \{(f, m_1)\}$$

The meta variables φ, denoting a binary function, and γ can be instantiated by the subterms in positions d_9 and d_{11}:

$$m_1.V = \{(\varphi, \{d_9\}), (\gamma, \{d_{11}\})\}$$

Finally a subterm to be identified by a second meta node m_2 has to occur below position d_{10}:

$$m_1.M_{sub} = \{(m_2, d_{10})\}$$

This second node m_2 has the purpose to identify a term matching $\varphi(\alpha, \beta)$. Its supernode is m_1:

$$m_2.M_{super} = \{m_1\}$$

Symbols and variables are defined analogously to m_1, here a has to occur in position d_5, φ in d_4 and β in d_6:

$$m_2.D = \{d_5\}$$
$$d_5.R = \{(a, m_2)\}$$
$$m_2.V = \{(\varphi, \{d_4\}), (\gamma, \{d_6\})\}$$

If the suitable term t_4 is inserted, it is trapped by m_1 and m_2. The node m_2 will be be triggered by t_3, because a is inserted in node d_5 with root term t_3. Thus d_5 sends a notification $\texttt{notify}(d_5, t_3)$ to m_2. As this is the only position m_2 is waiting for, the variables are instantiated according to $m_2.V$:

$$m_2.I = \{(t_3, \{(\varphi, +), (\beta, b)\})\}$$

Furthermore m_1 is notified that m_2 has found a matching term. Later m_1 will be triggered by t_4. Here d_2 will send the notification $\texttt{notify}(d_2, t_5)$. As d_2 is the only position m_1 is waiting for, variables are instantiated here, too:

$$m_1.I = \{(t_4, \{(\varphi, +), (\gamma, c)\})\}$$

As furthermore the only subnode m_2 has already sent a notification, it is evaluated whether the instantiation found by m_2 is suitable. First the variable instantiations are checked, φ is the only variable that is instantiated in both nodes, and it is in both cases bound to the symbol $+$.

Thus it is evaluate whether t_3 is at a suitable term position in t_5. The term t_3 occurs in positions d_0, d_3, d_5 and d_{10}, the root terms which are recorded with key term t_3 in these positions are t_3, t_2, t_5 and t_4. As m_2 is associated with d_{10} in m_1, the subterm of t_4 in d_{10} has to be looked up. It is t_3, and as $t_3 \in \{t_3, t_2, t_5, t_4\}$, the term identified by m_2 is obviously a suitable subterm in a suitable position. Thus m_1 has identified a term matching pattern $f(\varphi(\Phi(\varphi(a, \beta)), \gamma))$ along with the suitable meta variable instantiations:

$$(t_4, \{(\varphi, +), (\beta, b), (\gamma, c)\})$$

In case a term does not match, as for $t_2 = f(a + b)$ in figure 5.2, the matching will fail at some point. For t_2, both nodes m_1 and m_2 will be triggered: m_1 by t_2 and m_2 by $t_3 = (a+b)$. The variable instantiations are consistent, as again φ will be bound to symbol $+$. The last step however, in which it is determined if t_3 is at a suitable term position, will fail, because there is no entry (a, t_3) at any position $d \in t_3.D_{This}$ with root term a and key term t_3, i.e. t_3 is no subterm of a. This is the worst case of a failing matching, in general the unsuccessful branches of the algorithm should be cut at earlier stages.

The reactive mechanism for term matching here has two characteristics that makes it suitable for the identification of possibly applicable inference steps: it is lazy and it is exhaustive. Every time a new term is inserted into the graph structure, it is matched against *every* pattern that is encoded in meta nodes, but unsuccessful branches of the matching are cut early or are even not executed at all.

A reasonable extension to use the graph structure, e.g. for interfacing purposes in an environment like ΩMEGA are nodes that encode logical gates. Inference steps that are defined by an outline of proof lines could be encoded by combining several meta nodes. AND gates

can be assigned several input nodes and can pass a tuple of instantiations if the variable instantiations are consistent, and OR nodes can pass every instantiation that is supplied by one of its input nodes. A combination of both can implement a ΩMEGA tactic that is applicable for a set of different PAI situations.

5.2.6 Efficiency

The strength of this data structure is the speed of matching a number of terms against specified patterns. The most expensive operations are those who require an update to the positional tree, but this is reasonable as usually complex operations like matching are replaced by a number of simpler operations to maintain the order of the graph.

As pointed out by Stickel [74], a detailed analysis of the complexity of indexing techniques is difficult, as their performance depends heavily on the structure of terms in the database. Thus an overall evaluation of the complexity is omitted here.

However the complexity of parts of the computation can be analysed. For a term of length n, the depth of the syntax tree depends on its branching rate. Assuming a fixed branching rate greater than 1, the depth of the syntax tree will be $O(\log n)$. Based upon this assumption, the complexity of basic operations can be estimated:

- to add terms to the term graph, $O(n)$ intersections of terms have to be evaluated to ensure a perfect term sharing. This has to be repeated for subterms at each level of the syntax tree. As the syntax tree is of depth $O(\log n)$, $O(n \log n)$ intersections of term sets have to be evaluated.

- to add terms to the positional tree, for each of its elements two node entries have to be updated. This is again recursively repeated for each of the subterms. The complexity of the update of the positional tree is therefore $O(n \log n)$ entry updates for a term of length n. Updating of both the term graph and the positional tree, which is both necessary to insert a term into the graph structure, still has complexity $O(n \log n)$ node entries.

- to lookup a pattern requires the intersection of m term sets, where $m \leq n$ is the number of fixed symbols or subterms in a fixed position. For each occurrence of a subterm at a unspecified position, two further intersections of term sets are required. Furthermore $k \leq n$ equality tests have to be evaluated to ensure the consistence of variable instantiations. As equality tests can be performed in constant time, the complexity of a term lookup is that of $O(n)$ intersections of term sets.

- the reactive behaviour requires little extra computation. Each time a position node is updated, which happens $O(n \log n)$ times when inserting a term of length n, a membership test has to be performed and all meta nodes have to be notified. The number of meta nodes that are to be notified in each node depends on the term structure. For each notified node it is tested whether all necessary notifications have been received and the record of notifications is updated. This requires constant time, assuming that the maximum number of subnodes in each meta node is fixed. In case a meta node and all of its subnodes are triggered, for each sub node one intersection and one membership test have to be performed.

Hard to predict elements in this estimation are the set sizes for intersections and meta node notification. The size of sets which are accumulated in some node entries are heavily depending on the structure of terms in the database. In the worst case, sets will be of size $O(n)$, where n is the number of meta nodes.

However, all intersections of term sets and equality and membership tests can be reduced to operations on natural numbers. Thus an equality test is computed in constant time, and there are efficient algorithms for set operations. Implementations of integer sets as bit vectors in LEDA [59] are of complexity $O(1)$ for membership tests and insertion of elements, intersections of sets and similar operations are of complexity $O(b-a+1)$, where set elements are in a range $[a...b]$.

5.3 Reference

The following section gives a short overview of the objects of the data structure, i.e. in first line types of graph nodes and their attributes. After a summary about graphs and their semantics, an overview of the operations on these graphs will follow

5.3.1 Graphs

The set of possible identifiers ID is composed from a number of subsets, each of which has a number of attributes. Apart from the standard attributes that are listed here, arbitrary additional attributes can be used for purposes like typing of terms. Actually ID itself can be expanded, too, to increase the expressiveness of the system, e.g. by introducing reactive nodes as mentioned above. If doing so, care has to be taken of the paradigms required to maintain the system's correctness.

Standard node identifiers, attributes and membership in one of the two main structures, i.e. term sharing graph or positional tree, is listed here. The sets of nodes of the term sharing graph and the positional tree are denoted as $\mathsf{ID}_{\mathcal{G}}$ respectively $\mathcal{D}$.

- **alphabet symbols**

  $\Sigma = \{s_n | n \in I\!N\}$ is an alphabet of symbols. There is always a mapping $\Sigma \rightarrow I\!N$, therefore Σ can be reasonably identified with a set of natural numbers, furthermore is Σ subject to dynamic modification, as symbols may be introduced to or deleted from the environment at arbitrary time.

  The alphabet is a subset of the term sharing graph $\Sigma \subset \mathsf{ID}_{\mathcal{G}}$.

  These are the standard attributes of a node $s \in \Sigma$:

  - $s.T_{super} \subset \mathcal{T}$ is the set of direct superterms. Note that $s.T_{super}$ can be classified according to a set of key symbols $\{\texttt{abstraction}\} \cup \{(\texttt{application}, m, n) | m, n \in I\!N\}$, denoting the type of the superterm t, and in case of an `application` its arity and the position of the occurrence of s in t.
  - $s.D_{this} \subset \mathcal{D}$ is the set of discrimination nodes whose set of related terms features a pair (t, s), i.e. a node in which s has been used as a key when inserting a term to the discrimination tree, see below for details.

- **bound variables**

  $\mathcal{X} = \{x_n | n \in I\!N\}$ is the set of bound variables. The semantics of the index n defines the relation between occurrence of x_n and its binder; the binder can be found by ascending the term tree n scopes upwards.

  Node functionality of a node $x \in \mathcal{X}$ is the same as for alphabet symbols and so is its graph membership $\mathcal{X} \in \mathsf{ID}_\mathcal{G}$ and its attributes:

  - $x.T_{super} \subset \mathcal{T}$ and
  - $x.D_{this} \subset \mathcal{D}$.

- **non-primitive terms**

  $\mathcal{T} = \{t_n | n \in I\!N\}$ is the set of non-primitive terms.

  $\mathcal{T} \subset \mathsf{ID}_\mathcal{G}$ is also part of the term sharing graph, a node $t \in \mathcal{T}$ has the same attributes as primitive terms,

  - $t.T_{super} \subset \mathcal{T}$ and
  - $t.D_{this} \subset \mathcal{D}$,

  and apart from these two further attributes:

  - $t.T_{sub} \in \mathcal{T}^n$ is the list of subterms ordered according to their position.
  - $t.T_{type} \in \{\texttt{abstraction}\} \cup \{(\texttt{application}, n) | n \in I\!N\}$ is the node's type flag. Note that the list length of $t.T_{sub}$ must equal 1 in case of value `abstraction` and $n+1$ in case of $(\texttt{application}, n)$.

Therewith the list of standard types of nodes in $\mathsf{ID}_\mathcal{G}$ is completed. Every node in $\mathsf{ID}_\mathcal{G}$ represents a unique term, i.e. there is a bijective mapping $\mathsf{ID}_\mathcal{G} \rightarrow \mathcal{F}$ with respect to terms that have been added to the data structure.

The second main data structure is the positional tree $\mathcal{D}$, whose elements are discrimination nodes.

- **discrimination nodes**

  $\mathcal{D} = \{d_n | n \in I\!N\}$ is the set of discrimination nodes.

  A node $d \in \mathcal{D}$ has the following attributes:

  - $d.T_{this} \subset \mathcal{T} \times \mathcal{T}$ is the set of pairs $(t, k) \in \mathcal{T} \times \mathcal{T}$ where term t has been inserted at discrimination node d with key remainder k.
  - $d.D_{sub} \subset \mathcal{K} \times \mathsf{ID}$, where
    $\mathcal{K} = \Sigma \cup \mathcal{X} \cup \{\texttt{abstraction}\} \cup \{(\texttt{application}, m, n) | m, n \in I\!N\}$
    is the set of possible key symbols.
  - $d.D_{super} \in \mathcal{D}$ is the pointer to d's parent node.

A special discrimination node is $d_0 \in \mathcal{D}$, the root node of the positional tree. It is assigned to the root position of a term.

5.3.2 Procedures

The following is an overview over procedures for general graph maintenance. This comprises procedures to add and to delete terms. Furthermore a procedure to implement substitution for terms in the graph structure is introduced. Procedures to process queries to the database are omitted in this place, because their implementation is non-trivial. Processing of queries requires in general the intersection of term sets where the order in which these intersections are computed has an effect on the performance. Thus it is reasonable to employ elaborated mechanisms to guide the computation, which would be outside the scope of this work.

- **adding a term**

  The procedure `add` : $\mathcal{F} \rightarrow \mathsf{ID}$ adds a term to the graph and returns its identifier `id`. A term $f \in \mathcal{F}$ is recursively inserted, while pointers to subterms and superterms are updated, and all new nodes are added to the discrimination tree. This is the full function definition:

```
function add(f)
   case f ∈ Σ ∪ X
      then return f
   case f ∈ {application_n(f_0, ..., f_n) | n ∈ IN, f_i ∈ ID}
      then declare
            subterms:ID^n
            temp:ID
         for each [f_0, ..., f_n] do
            temp := add(f_i)
            subterms(i) := temp
            end for
         if ⋂_{i=0}^{n} {t_super | (t_super, application, n, i) ∈ subterms(i).T_super} = {t}
         then temp := t
         else temp := new node t_new
            temp.T_sub := subterms
            d_add(temp,temp,d_0)
         end if
         for i=0 to n do
            subterms(i).T_super := subterms(i).T_super ∪ {(temp, application, n, i)}
            end for
         return temp
   case f ∈ {abstraction(f_0) | f_0 ∈ ID}
      then declare temp:ID
            subterm := add(f_0)
         if (t, abstraction, 0, 0) ∈ subterm.T_super
         then temp := t
         else temp := new node t_new
            temp.T_sub := [subterm]
            d_add(temp,temp,d_0)
         end if
         subterm.T_super := subterm.T_super ∪ {(temp, abstraction)}
```

```
        return temp
```

The combination of `add` and `d_add` still offers possibilities for optimisation.

- **looking up a term**

  This is the code of the procedure $\texttt{get} : \mathsf{ID} \rightarrow \mathcal{F}$ that is used to lookup the term f that is denoted by an identifier `id`. This function recursively follows subterm pointers until the leaves of the term's syntax tree, actually symbols, have been found.

```
function get(t)
   case t ∈ Σ ∪ X
      then return t
   case t ∈ T
      then if (t.T_type == (application, n))
         then return application_n([get(t.T_sub,0), ..., get(t.T_sub,n)])
         else return abstraction(get(t.T_sub,0))
```

- **deleting a term**

  The procedure $\texttt{delete}(t)$ removes a term t from a graph. Terms can only be removed when unused.

```
function delete(t)
   if t.T_super = ∅
   then for each id ∈ t.T_sub,n do
      id.T_super = id.T_super ⊖ {t}
      delete(id)
      end for
   d_delete(t, t, d_0)
   remove node t
   end if
```

- **adding a term to the positional tree**

  The procedure $(\texttt{d\_add})(t, k, d)$ adds a term $t \in \mathsf{ID}_{\mathcal{G}}$ to a discrimination node $d \in \mathcal{D}$ using key $k \in \mathsf{ID}_{\mathcal{G}}$. This includes also the mechanism to expand the positional tree whenever needed.

```
function d_add(t, k, d)
   d.T_this = d.T_this ∪ {(t, k)}
   case k ∈ Σ ∪ X
      then d.D_sub(k).T_this = d.D_sub(k).T_this ∪ {(t, ε)}
   case t ∈ T
      then if (t.T_type == (application, n))
```

```
then for each id_i ∈ t.T_sub
   if ∃d_sub ∈ 𝒟.((application, n, i), d_sub) ∈ d.D_sub
   then d_add(t, id_i, d_sub)
   else declare temp := new node d_new
      temp.D_super := d
      d.D_sub := d.D_sub ∪ {(application, n, i), temp)}
      d_add(t, id_i, temp)
else declare id := k.T_sub
   if ∃d_sub ∈ 𝒟.(abstraction, d_sub) ∈ d.D_sub
   then d_add(t, id, d_sub)
   else declare temp := new node d_new
      temp.D_super := d
      d.D_sub := d.D_sub ∪ {(abstraction, temp)}
      d_add(t, id, temp)
end if
```

- **deleting terms from the positional trees**

  The procedure (`d_delete`)(t, k, d) deletes a term $t \in \mathsf{ID}_{\mathcal{G}}$ from a discrimination node $d \in \mathcal{D}$ using key $k \in \mathsf{ID}_{\mathcal{G}}$. This is the inverse to `d_add`. Whenever possible, the positional tree is contracted.

```
function d_delete(t, k, d)
   for each i ≤ n do, where n is the arity of k
      delete(t, k.T_sub,i, d_next), where ((k.T_type, n, i), d_next) ∈ d.D_sub
   end for
   d.T_this := d.T_this ⊖ {(t, k)}
   if (d.T_this == ∅)
   then d.D_super.D_sub := d.D_super.D_sub ⊖ {(∗, d)}
      remove node d
   end if
```

- **finding paths between subterms and superterms**

  As sometimes an uninformed search in the set of superterms of an identifier may cause considerable cost, there are tasks where it is reasonable to consider more efficient techniques to bring up a solution. An example is finding a path from the root of a term t_0 to a specified subterm t_{sub}, as it is required to substitute subterms.

  For this purpose a procedure `sniff`$(t_{sub}, t_0) : \mathsf{ID} \times \mathsf{ID} \rightarrow \mathsf{ID}^n$ is used that "sniffs" the path from the subterm to the terms root by following a non-branching trace on the positional tree:

```
function sniff(t_sub, t_0)
   if (t_0, t) ∈ t_sub.D_this.D_super.T_this
      and (t_0, t_sub) ∈ t_sub.D_this.T_this
```

```
then return append(t,sniff(t, t_0))
end if
```

The procedure terminates in $\texttt{sniff}(t_0, t_0)$ where (t_0, t_0) is categorised in the root of the positional tree, $(t_0, t_0) \in d_0.T_{this}$ where d_0 has no parent discrimination node.

- **substitution**

  The substitution of subterms is a fundamental operation in a term system. In the following I will describe two procedures for first global rewriting and second the application of a substitution to a single term.

  Global rewriting is the simpler one, as only pointers related to a single node have to be modified. This is done by a procedure $\texttt{global\_rewrite}(t_1, t_2)$ that replaces all occurrences of t_1 in the whole context with t_2:

```
function global_rewrite(t_1, t_2)
      for each {t|(t, type, n, i) ∈ t_1.T_super} do
         t.T_sub,i := t_2
      end for
      for each {(d, t)|d ∈ t_1.D_this, (t, t_1) ∈ d_1.T_this} do
         d_delete(t, t_1, d)
         d_add(t, t_2, d)
      end for
      t_2.T_super := t_2.T_super ∪ t_1.T_super
      t_1.T_super := ∅
      delete(t_1)
```

  The deletion of t_1 does not necessarily cause the node to be removed from the graph, it is kept if e.g. it is still referenced by external pointers. Within the data structure however t_1 is unused after a global rewrite.

  Note that updating the positional tree is potentially costly, as it depends on the overall number of occurrences of t_1 in the whole context. However the terms relevant to a rewriting are looked up in virtually no time.

  If there is however no need to maintain a positional tree, this feature can be abandoned. The result is a very efficient environment optimised towards purposes mainly relying on global rewriting.

  Apart from global rewriting, local substitution is a common operation on terms. The procedure $\texttt{substitute}(t_1, t_2, t_{root})$ replaces all occurrences of t_1 in t_{root} by t_2.

```
function substitute(t_1, t_2, t_root)
      declare modified := {(t_1, t_2)}
      for each t ∈ sniff(t_1, t_root) do
         declare subterms := t.T_sub
         for each i ≤ n do, where n is the arity of t
```

```
      if (subterms(i), t_new) ∈ modified
      then subterms(i) := t_new
      end if
   end for
   modified := modified∪{(t, add(t.T_type(subterms)))}
end for
declare subterms := t_root.T_sub
for each i ≤ n do, where n is the arity of t_root
   if (subterm(i), t_new) ∈ modified
   then subterm(i) := t_new
   end if
end for
return add(t_root.T_type(subterms))
```

In case of t_1 having no occurrences in t_{root}, `sniff` returns an empty list and t_{root} is returned unmodified. Note that otherwise `sniff` returns the substituted nodes in right order, i.e. subterms before their superterms.

5.4 Conclusion

Unfortunately an actual evaluation of the speed up that can be obtained by the data structure proposed in this work is still do be undertaken. However the results of implementations of similar approaches are quite encouraging. In general the speed up gained by term sharing and indexing techniques was dramatic. For first order logic there are several implementations of indexing techniques, one example is the award winning E-Prover [68], where term sharing and term indexing techniques lead to a considerable speed up. While the technique is rather well explored for the first order case, the evaluation of its use for higher order logic is still in progress. However, at least the parts of matching that are similar to first order matching will certainly experience a dramatic speed up, and subclasses of higher order terms can be well handled. An example is an implementation of substitutional tree indexing by Brigitte Pientka [65] based in higher order patterns, where the speed up was betweem 100% and 800%. These results are so convincing, that the techniques used here will probably become standard in automated reasoning soon.

The adaption that has been made to the indexing technique as it is used in other approaches is the addition of a reactive element. Automated first order theorem provers usually employ strong machine oriented proof strategies, in which indexing is used to quickly lookup suitable terms. At the time of such a lookup, there is in general one pattern for which a corresponding term has to be looked up. For knowledge based systems, e.g. ΩMEGA, the situation is different, as there are in general a considerable number of strategies that may be applicable. Thus the task becomes now to match a possibly large number of patterns against a possibly large number of proof lines. Therefore the indexing technique was adapted to react on insertion of new terms and have matching triggered by occurrences of matching subterms, and thus to avoid superficial lookups. The same applies for an interface to a CAS: a number of algorithms are specified that may be applicable in several places.

While this work is mainly focused on the implementation of an interface between a deduction system and its external subsystems, the technique described here is of relevance to

a variety of purposes where term matching is involved. The data structure is in general applicable to the purpose of automated search for applicable inference steps within a proof situation. This problem is central to any high level deduction system and is addressed e.g. by Sorge's agent based ΩANTS mechanism [9] in the ΩMEGA system. While this approach is much more general and is less rigid in the the way inference steps are described (the core of my data structure implements first order style term matching), it however associates inference rules and agents, each of which actively searches the database and therefore causes costly computations. A lazy approach where the actual process of matching is triggered by certain criteria being met avoids unsuccessful matching to a considerable degree. A further reduction of cost is obtained by pre-evaluation of relations between terms, which is a general aspect of term sharing techniques. Here a combination of the efficient term matching of my data structure and the freedom of programmable agents for special purposes is thinkable. Indexing technique can be used here as a filter for suitable terms, which then are supplied to the agents' special purpose evaluation.

The term structure of the database is not bound to any specific formal system, but rather implements a minimum of structural elements. This makes it open for implementation in various contexts, i.e. in different system, but also for various purposes.

While this reduction in cost in the processing of terms already pushes the limit towards solving more complex problems by being able to master greater knowledge bases in acceptable time, the representation of terms that is used in this work and in similar approaches may help to reveal completely new aspects of term processing. As the term sharing technique proposed in this work relates expressions and their elements rather than simply assembling expressions, it allows e.g. to search syntactical structures bottom up, i.e. given say a variable it is possible to lookup all expressions that have occurrences of this variable without examination of data that is not related to that variable. A further aspect is the representation of equivalence classes, which are much easier to implement as there is only a single representation of each syntactical structure, thus it is easy to associate all occurrences of an expressions to an equivalence class.

A final aspect of this work is that it is possible to encode a knowledge base along with the basic functionality of term matching into a single graph. This allows a compact implementation e.g. for various interfacing purposes and infrastructural issues, term filters or blackboard architectures with matching functionalities. The data structure can furthermore be used for all sorts of evaluation of sets of terms that are used for heuristics or even control at system level. Thinkable is e.g. an association of mathematical theories and the occurrence of certain term structures, where the database could be used to trigger the system to load additional mathematical theories into working memory according to the type of new expressions that are inserted into a proof plan. By extending this principle by e.g. feeding back results of this matching into the graph or by integration of several database into a single system, it is possible to provide a framework to very easily implement basic techniques of automated reasoning. As this would allow to implement reasoning systems by only providing term patterns to specify inference rules and possibly graphically designing the dataflow infrastructure, it may even help to provide a very flexible and easy to use development kit for reasoning techniques and possibly make these techniques accessible to users that are not too familiar with the peculiarities of a concrete theorem prover and do not want to become expert in the use of a full scale reasoning system.

Chapter 6

Conclusion

Subject of this work is the white box integration into a deduction system. Several aspects of such an integration have been examined. All experiments described here have been implemented in and around the ΩMEGA system [70] and make use of the already existing SAPPER interface by Sorge [71].

In chapter 3 I describe the implementation of the prototypical computer algebra system MASS and its integration into the ΩMEGA environment. Unlike many other approaches to integrate a CAS into a deduction system, MASS does not act as a black box, but provides sufficient information to remodel its computation within ΩMEGA's formalism. Thus these computations can be further processed by ΩMEGA's proof handling facilities. This is in first line ΩMEGA's proof checker to verify the correctness of MASS' computations, but also other facilities e.g. for proof representation and explanation can be used. This interesting e.g. to use proofs that are developed under participation of a CAS in computer supported mathematical education. Technically MASS is similar to its predecessor μCAS by Sorge [71], it is however more robust and offers a wider applicability than μCAS, which was necessary for a further evaluation of this kind of white box architecture.

The increased robustness and applicability of MASS made it a suitable tool for further experiments. A novelty hereby was the combination of a CAS that is fully integrated into a deduction system's formalism and a commercial CAS that behaves like a black box system. The combination of MAPLE's computational strength in non-trivial computations and MASS' strength in verification helped to make non-trivial algebraic computations verifiable, while it was neither necessary to formalise a full grown CAS like MAPLE nor to advance the development of the prototypical CAS like MASS to a level where sophisticated data structures and algorithms are required. MASS was employed in experiments in the domain of limit proofs [54] and in the exploration of properties of residue classes [53].

As a white box integration requires to maintain a common mathematical database that is accessible to both the CAS and the deduction system and has furthermore to be perfectly synchronised with the CAS' algorithms, this issue is addressed in chapter 4. The solution proposed her is the mathematical authoring tool TACO for the development of tactics in ΩMEGA. While the maintenance of CAS algorithms and analogue inference steps in the deduction system is still left to the human developer, TACO allows to develop tactics at a high level of abstraction within a graphical user interface without losing the power of a programming language. The approach in TACO to develop tactics at an abstract level offers furthermore a comfortable way to make the information available to different systems or

different modules within a single system, thus avoiding multiple implementation for different purposes. A special feature of TACO is the facility to combine already implemented tactics and a set of simple formalised control structures to develop simple algorithms.

The last aspect of the integration of a CAS into a deduction system that has been examined in this work is how the SAPPER interface could be extended to allow a finer interaction between CAS and deduction system. Although starting from the issue of CAS integration, the data structure described in chapter 5 and its application for interfacing purposes is of relevance for the general problem of identification of suitable proof strategies in a given proof situation. The approach is based on term indexing techniques [74], which have been adapted to implement a reactive database, as it could serve for the implementation of various interface architectures, blackboard mechanisms, constraint collectors or similar devices that require a mechanism for term matching. Although the efficiency of the approach has not been evaluated yet, the revolutionary success of indexing techniques in first order logic theorem proving [68] suggest a reasonable performance of the data structure proposed here in practical application.

The challenge for future work is to advance the development of the systems MASS and TACO especially concerning the independence of a specific system. While MASS is already system independent except for the requirement of a common mathematical database, TACO could serve to bridge this gap by offering possibilities to adapt abstract inference definitions to different systems. An adaption to standardised mathematical description languages like OPENMATH or OMDOC is thinkable, too. Of interest is furthermore the integration of formalised programming language elements in a logical environment as it is done e.g. to formalise the Java Virtual Machine in ISABELLE [66, 6, 62]. This could serve to implement a virtual machine that allows both to execute the code of algorithms and to reason about it. An integration of this approach within the systems described in this work could serve to advance system independency. With respect to CAS integration into formal environments, this could also help to avoid costly and possibly erroneous double implementation of computer algebra algorithms and the formalisation of their computations. A further subject of my interest is the application of graph structures to implement operations on terms, because it sometimes opens completely new ways to deal with terms.

Appendix A

The Proof of LIM+

This the complete proof of the LIM+ problem introduced in section 3.6.2. Note that the schema of *Complex-Estimate* was slightly modified for better understanding in section 3.6.2. The actual method is, as indicated by the name, rather complex. The corresponding proof line here is L28.

L34.	$\mathcal{H}_1$	$\vdash M\_X_{2\,[\nu]} = X_{[\nu]}$	(Tellcs-M)
L33.	$\mathcal{H}_1$	$\vdash Less_{[(\nu,\nu)\to o]}(0, M\_M_{[\nu]})$	(Tellcs-M)
L58.	$\mathcal{H}_2$	$\vdash Leq_{[(\nu,\nu)\to o]}(M\_E_{1\,[\nu]}, (Div_{[(\nu,\nu)\to\nu]}(1,2)\cdot_{[(\nu,\nu)\to\nu]} E_{[\nu]}))$	(Tellcs-M)
L57.	$\mathcal{H}_2$	$\vdash M\_X_{1\,[\nu]} = X$	(Tellcs-M)
L46.	L46	$\vdash \forall X_{1\,[\nu]}[(0 < M\_E_1) \Rightarrow [(0 < D_{1\,[\nu]}) \wedge [[(Absval_{[\nu\to\nu]}((X_1 -_{[(\nu,\nu)\to\nu]} A_{[\nu]})) < D_1) \wedge Greater_{[(\nu,\nu)\to o]}(\lvert(X_1{-}A)\rvert, 0)] \Rightarrow (\lvert(F_{[\nu\to\nu]}(X_1){-}Limit_{1\,[\nu]})\rvert < M\_E_1)]]]$	(Hyp)
L48.	L46	$\vdash [(0 < M\_E_1) \Rightarrow [(0 < D_1) \wedge [[(\lvert(M\_X_1{-}A)\rvert < D_1) \wedge (\lvert(M\_X_1{-}A)\rvert > 0)] \Rightarrow (\lvert(F(M\_X_1){-}Limit_1)\rvert < M\_E_1)]]]$	(Foralle-Meta-M L46)
L49.	$\mathcal{H}_2$	$\vdash (0 < M\_E_1)$	(Tellcs-M)
L51.	$\mathcal{H}_2$	$\vdash [(0 < D_1) \wedge [[(\lvert(M\_X_1{-}A)\rvert < D_1) \wedge (\lvert(M\_X_1{-}A)\rvert > 0)] \Rightarrow (\lvert(F(M\_X_1){-}Limit_1)\rvert < M\_E_1)]]$	($\Rightarrow E$ L49,L48)
L53.	$\mathcal{H}_2$	$\vdash [[(\lvert(M\_X_1{-}A)\rvert < D_1) \wedge (\lvert(M\_X_1{-}A)\rvert > 0)] \Rightarrow (\lvert(F(M\_X_1){-}Limit_1)\rvert < M\_E_1)]$	(Ande-M L51)
L8.	L8	$\vdash [(\lvert(X{-}A)\rvert < M\_D_{[\nu]}) \wedge (\lvert(X{-}A)\rvert > 0)]$	(Hyp)
L11.	L8	$\vdash (\lvert(X{-}A)\rvert > 0)$	(Ande-M L8)
L60.	$\mathcal{H}_2$	$\vdash (\lvert(X{-}A)\rvert > 0)$	(Weaken-M L11)
L62.	$\mathcal{H}_2$	$\vdash (M\_D \leq D_1)$	(Tellcs-M)
L61.	$\mathcal{H}_2$	$\vdash True_{[o]}$	(Truei-M)
L10.	L8	$\vdash (\lvert(X{-}A)\rvert < M\_D)$	(Ande-M L8)
L59.	$\mathcal{H}_2$	$\vdash (\lvert(X{-}A)\rvert < D_1)$	(Solve*<-M L10,L61,L62)
L54.	$\mathcal{H}_2$	$\vdash [(\lvert(M\_X_1{-}A)\rvert < D_1) \wedge (\lvert(M\_X_1{-}A)\rvert > 0)]$	(Andi-M L59,L60)
L56.	$\mathcal{H}_2$	$\vdash (\lvert(F(M\_X_1){-}Limit_1)\rvert < M\_E_1)$	($\Rightarrow E$ L54,L53)
L55.	$\mathcal{H}_2$	$\vdash (\lvert(F(X){-}Limit_1)\rvert < ((1/2)\cdot E))$	(Solve*<-M L56,L57,L58)
L50.	$\mathcal{H}_2$	$\vdash (\lvert(F(X){-}Limit_1)\rvert < ((1/2)\cdot E))$	(Impe-Open-M L54,L53,L55)
L47.	$\mathcal{H}_2$	$\vdash (\lvert(F(X){-}Limit_1)\rvert < ((1/2)\cdot E))$	(Impe-Open-M L49,L48,L50)
Limit-F.	Limit-F	$\vdash \forall E_{1\,[\nu]}\exists D_1\forall X_{1\,[\nu]}[(0 < E_1) \Rightarrow [(0 < D_1) \wedge [[(\lvert(X_1{-}A)\rvert < D_1) \wedge (\lvert(X_1{-}A)\rvert > 0)] \Rightarrow (\lvert(F(X_1){-}Limit_1)\rvert < E_1)]]]$	(Hyp)

L45.	Limit-F	$\vdash \exists D_{1[\nu]\blacksquare}\forall X_{1[\nu]\blacksquare}[(0 < M\_E_1) \Rightarrow [(0 < D_1) \wedge [[(\lvert(X_1 - A)\rvert < D_1) \wedge (\lvert(X_1 - A)\rvert > 0)] \Rightarrow (\lvert(F(X_1) - Limit_1)\rvert < M\_E_1)]]]$	(Foralle-Meta-M Limit-F)
L44.	$\mathcal{H}_1$	$\vdash (\lvert(F(X) - Limit_1)\rvert < ((1/2) \cdot E))$	(Existse-M L45,L47)
L39.	$\mathcal{H}_1$	$\vdash (\lvert(F(X) - Limit_1)\rvert < ((1/2) \cdot E))$	(Mset-Focus-M Limit-F,L44)
L32.	$\mathcal{H}_1$	$\vdash (\lvert(F(X) - Limit_1)\rvert < (E/2))$	(Simplify-M L39)
L31.	$\mathcal{H}_1$	$\vdash (M\_E_{2[\nu]} \leq (E/(2 \cdot M\_M)))$	(Tellcs-M)
L30.	$\mathcal{H}_1$	$\vdash (\lvert 1\rvert \leq M\_M)$	(Tellcs-M)
L19.	L19	$\vdash \forall X_{2[\nu]\blacksquare}[(0 < M\_E_2) \Rightarrow [(0 < D_{2[\nu]}) \wedge [[(\lvert(X_2 - A)\rvert < D_2) \wedge (\lvert(X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G_{[\nu \to \nu]}(X_2) - Limit_{2[\nu]})\rvert < M\_E_2)]]]$	(Hyp)
L21.	L19	$\vdash [(0 < M\_E_2) \Rightarrow [(0 < D_2) \wedge [[(\lvert(M\_X_2 - A)\rvert < D_2) \wedge (\lvert(M\_X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(M\_X_2) - Limit_2)\rvert < M\_E_2)]]]$	(Foralle-Meta-M L19)
L22.	$\mathcal{H}_1$	$\vdash (0 < M\_E_2)$	(Tellcs-M)
L24.	$\mathcal{H}_1$	$\vdash [(0 < D_2) \wedge [[(\lvert(M\_X_2 - A)\rvert < D_2) \wedge (\lvert(M\_X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(M\_X_2) - Limit_2)\rvert < M\_E_2)]]$	($\Rightarrow E$ L22,L21)
L26.	$\mathcal{H}_1$	$\vdash [[(\lvert(M\_X_2 - A)\rvert < D_2) \wedge (\lvert(M\_X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(M\_X_2) - Limit_2)\rvert < M\_E_2)]$	(Ande-M L24)
L64.	$\mathcal{H}_1$	$\vdash (\lvert(X - A)\rvert > 0)$	(Weaken-M L11)
L66.	$\mathcal{H}_1$	$\vdash (M\_D \leq D_2)$	(Tellcs-M)
L65.	$\mathcal{H}_1$	$\vdash True$	(Truei-M)
L63.	$\mathcal{H}_1$	$\vdash (\lvert(X - A)\rvert < D_2)$	(Solve*<-M L10,L65,L66)
L27.	$\mathcal{H}_1$	$\vdash [(\lvert(M\_X_2 - A)\rvert < D_2) \wedge (\lvert(M\_X_2 - A)\rvert > 0)]$	(Andi-M L63,L64)
L29.	$\mathcal{H}_1$	$\vdash (\lvert(G(M\_X_2) - Limit_2)\rvert < M\_E_2)$	($\Rightarrow E$ L27,L26)
L28.	$\mathcal{H}_1$	$\vdash (\lvert((F(X) +_{[(\nu,\nu) \to \nu]} G(X)) - Limit_1) - Limit_2)\rvert < E)$	(Mcomplexestimate<-M L29,L30,L31,L32,L33,L34)
L23.	$\mathcal{H}_1$	$\vdash (\lvert((F(X) + G(X)) - Limit_1) - Limit_2)\rvert < E)$	(Impe-Open-M L27,L26,L28)
L20.	$\mathcal{H}_1$	$\vdash (\lvert((F(X) + G(X)) - Limit_1) - Limit_2)\rvert < E)$	(Impe-Open-M L22,L21,L23)
Limit-G.	Limit-G	$\vdash \forall E_{2[\nu]\blacksquare}\exists D_{2\blacksquare}\forall X_{2[\nu]\blacksquare}[(0 < E_2) \Rightarrow [(0 < D_2) \wedge [[(\lvert(X_2 - A)\rvert < D_2) \wedge (\lvert(X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(X_2) - Limit_2)\rvert < E_2)]]]$	(Hyp)
L18.	Limit-G	$\vdash \exists D_{2[\nu]\blacksquare}\forall X_{2[\nu]\blacksquare}[(0 < M\_E_2) \Rightarrow [(0 < D_2) \wedge [[(\lvert(X_2 - A)\rvert < D_2) \wedge (\lvert(X_2 - A)\rvert > 0)] \Rightarrow (\lvert(G(X_2) - Limit_2)\rvert < M\_E_2)]]]$	(Foralle-Meta-M Limit-G)
L17.	$\mathcal{H}_3$	$\vdash (\lvert((F(X) + G(X)) - Limit_1) - Limit_2)\rvert < E)$	(Existse-M L18,L20)
L14.	$\mathcal{H}_3$	$\vdash (\lvert((F(X) + G(X)) - Limit_1) - Limit_2)\rvert < E)$	(Mset-Focus-M Limit-G,L17)
L9.	$\mathcal{H}_3$	$\vdash (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)$	(Simplify-M L14)
L7.	$\mathcal{H}_4$	$\vdash [[(\lvert(X - A)\rvert < M\_D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)]$	(Impi-M L9)
L6.	$\mathcal{H}_4$	$\vdash (0 < M\_D)$	(Tellcs-M)
L5.	$\mathcal{H}_4$	$\vdash [(0 < M\_D) \wedge [[(\lvert(X - A)\rvert < M\_D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)]]$	(Andi-M L6,L7)
L3.	Limit-F, Limit-G	$\vdash [(0 < E) \Rightarrow [(0 < M\_D) \wedge [[(\lvert(X - A)\rvert < M\_D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)]]]$	(Impi-M L5)
L2.	Limit-F, Limit-G	$\vdash \forall X_{\blacksquare}[(0 < E) \Rightarrow [(0 < M\_D) \wedge [[(\lvert(X - A)\rvert < M\_D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)]]]$	(Foralli-M L3)
L1.	Limit-F, Limit-G	$\vdash \exists D_{[\nu]\blacksquare}\forall X_{[\nu]\blacksquare}[(0 < E) \Rightarrow [(0 < D) \wedge [[(\lvert(X - A)\rvert < D) \wedge (\lvert(X - A)\rvert > 0)] \Rightarrow (\lvert((F(X) + G(X)) - (Limit_1 + Limit_2))\rvert < E)]]]$	(Existsi-M L2)

Thm. Limit-F, Limit-G $\vdash \forall Dc{-}_{375\,[\nu]\blacksquare} \exists Dc{-}_{376\,[\nu]\blacksquare} \forall Dc{-}_{377\,[\nu]\blacksquare} [(0 < Dc{-}_{375}) \Rightarrow [(0 < Dc{-}_{376}) \wedge [[(|(Dc{-}_{377}{-}A)| < Dc{-}_{376}) \wedge (|(Dc{-}_{377}{-}A)| > 0)] \Rightarrow (|((F(Dc{-}_{377}){+}G(Dc{-}_{377})){-}(Limit_1{+}Limit_2))| < Dc{-}_{375})]]]$ (Foralli-M L1)

$\mathcal{H}_1$ = Limit-F, Limit-G, L4, L8, L19
$\mathcal{H}_2$ = Limit-F, Limit-G, L4, L8, L19, L46
$\mathcal{H}_3$ = Limit-F, Limit-G, L4, L8
$\mathcal{H}_4$ = Limit-F, Limit-G, L4

Appendix B

Generated Code for *Split-Monomials-Plus*

This is the complete code of the tactic *Split-Monomials-Plus* as it is written to a file by TACO. The code presented here serves two purposes: First it can be read by TACO. The specifications relevant for TACO are encoded as LISP comments in the first part of the code, separated by special tags. Second, the code can be loaded into ΩMEGA. Here the native LISP code in the second code is interpreted, while the comments containing the abstract specification are ignored.

```
; TACO TACTIC split-monomials-plus

; TACO Variables

;phi z a

; TACO Theory Constants

;plus times div num

; TACO Parameters

;(pos position)
;(x term)
;(y term)

; TACO Patterns

;(nonexistent existent)
;(existent nonexistent)
;(existent existent)

; TACO Theory

;real
```

```
; TACO Premises

;(l1 (formula phi (times z a)  pos))

; TACO Conclusions

;(l2 (formula phi
;      (plus (times x a) (times y a))
;      pos))

; TACO Constraints

;{and (data~primitive-p ?x)
;     (numberp (keim~name ?x))}
;{and (data~primitive-p ?y)
;     (numberp (keim~name ?y))}
;{and (data~primitive-p ?z)
;     (numberp (keim~name ?z))}
;(z = {term~constant-create
;    (+ (keim~name ?x) (keim~name ?y))
;    ?num})

; TACO General Help

;Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y.

; TACO Argument Help

;(l2 "A Line containg x*a+y*a")
;(l1 "A Line containing z*a")
;(pos "The position of the term")
;(x "The first coefficient")
;(y "The second coefficient")

; TACO Expansion

;(inference expand-num (l3 l1)({pos~add-end ?pos 1} x y))
;(inference distribute-right (l2 l3) (pos))

; TACO Code

(infer~deftactic split-monomials-plus
 (outline-mappings
  (((nonexistent existent)
    split-monomials-plus-1)
   ((existent nonexistent)
```

```
    split-monomials-plus-2)
   ((existent existent)
    split-monomials-plus-3)))
 (expansion-function taco=expand-split-monomials-plus)
 (parameter-types position term term)
 (help "Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y."))


(com~defcommand split-monomials-plus
 (argnames l2 l1 pos x y)
 (argtypes ndline ndline position term term)
 (arghelps "a line containg x*a+y*a"
           "a line containing z*a"
           "the position of the term"
           "the first coefficient"
           "the second coefficient")
 (function taco=split-monomials-plus)
 (frag-cats tactics)
 (defaults)
 (log-p t)
 (help "Rewrite z*a=x*a+y*a where x,y,z are numbers and z=x+y."))


(defun taco=split-monomials-plus
 (l2 l1 pos x y)
 (infer~compute-outline 'split-monomials-plus
  (list l2 l1)
  (list pos x y)))


(tac~deftactic split-monomials-plus-1 split-monomials-plus
 (in real)
 (parameters
  (pos pos+position "the position of the term")
  (x term+term "the first coefficient")
  (y term+term "the second coefficient"))
 (premises l1)
 (conclusions l2)
 (computations
  (l2
   (taco=split-monomials-plus-1-l2
    (formula l1)
    pos x y)))
 (sideconditions
  (taco=split-monomials-plus-1-p
   (formula l1)
   pos x y))
```

```
 (description "Apply tactic split-monomials-plus
               to pattern (nonexistent existent)."))


(defun taco=split-monomials-plus-1-l2
 (taco-l1 taco-pos taco-x taco-y)
 (let*
  ((taco-x0
    (data~struct-at-position taco-l1 taco-pos))
   (taco-x1
    (data~appl-arguments taco-x0))
   (taco-a
    (nth 1 taco-x1))
   (taco-x7
    (list taco-y taco-a))
   (taco-times
    (data~appl-function taco-x0))
   (taco-x6
    (data~appl-create taco-times taco-x7))
   (taco-x5
    (list taco-x taco-a))
   (taco-x4
    (data~appl-create taco-times taco-x5))
   (taco-x3
    (list taco-x4 taco-x6))
   (taco-plus
    (env~lookup-object :plus
     (pds~environment omega*current-proof-plan)))
   (taco-x2
    (data~appl-create taco-plus taco-x3)))
  (data~replace-at-position taco-l1 taco-pos taco-x2)))


(defun taco=split-monomials-plus-1-p
 (taco-l1 taco-pos taco-x taco-y)
 (and
  (and (data~primitive-p taco-x )
     (numberp (keim~name taco-x )))
  (and (data~primitive-p taco-y )
     (numberp (keim~name taco-y )))
  (let*
   ((taco-x0
     (data~struct-at-position taco-l1 taco-pos))
    (taco-num
     (env~lookup-object :num
      (pds~environment omega*current-proof-plan)))
    (taco-times
```

```
     (env~lookup-object :times
      (pds~environment omega*current-proof-plan))))
   (and
    (data~appl-p taco-x0)
    (term~taco-equal taco-times
     (data~appl-function taco-x0))
    (let*
     ((taco-z
       (term~constant-create
    (+ (keim~name taco-x ) (keim~name taco-y )) taco-num))
       (taco-x1
        (data~appl-arguments taco-x0)))
     (and
      (and (data~primitive-p taco-z )
     (numberp (keim~name taco-z )))
      (listp taco-x1)
      (=
       (list-length taco-x1)
       2)
      (term~taco-equal taco-z
       (nth 0 taco-x1)))))))


(tac~deftactic split-monomials-plus-2 split-monomials-plus
 (in real)
 (parameters
  (pos pos+position "the position of the term")
  (x term+term "the first coefficient")
  (y term+term "the second coefficient"))
 (premises l1)
 (conclusions l2)
 (computations
  (l1
   (taco=split-monomials-plus-2-l1
    (formula l2)
    pos x y)))
 (sideconditions
  (taco=split-monomials-plus-2-p
   (formula l2)
   pos x y))
 (description "Apply tactic split-monomials-plus
               to pattern (existent nonexistent)."))


(defun taco=split-monomials-plus-2-l1
 (taco-l2 taco-pos taco-x taco-y)
 (let*
```

```
  ((taco-x2
    (data~struct-at-position taco-l2 taco-pos))
   (taco-x3
    (data~appl-arguments taco-x2))
   (taco-x6
    (nth 1 taco-x3))
   (taco-x7
    (data~appl-arguments taco-x6))
   (taco-a
    (nth 1 taco-x7))
   (taco-num
    (env~lookup-object :num
     (pds~environment omega*current-proof-plan)))
   (taco-z
    (term~constant-create
    (+ (keim~name taco-x ) (keim~name taco-y )) taco-num))
   (taco-x1
    (list taco-z taco-a))
   (taco-times
    (data~appl-function taco-x6))
   (taco-x0
    (data~appl-create taco-times taco-x1)))
  (data~replace-at-position taco-l2 taco-pos taco-x0)))


(defun taco=split-monomials-plus-2-p
 (taco-l2 taco-pos taco-x taco-y)
 (and
  (and (data~primitive-p taco-x )
     (numberp (keim~name taco-x )))
  (and (data~primitive-p taco-y )
     (numberp (keim~name taco-y )))
  (let*
   ((taco-x2
     (data~struct-at-position taco-l2 taco-pos))
    (taco-num
     (env~lookup-object :num
      (pds~environment omega*current-proof-plan)))
    (taco-times
     (env~lookup-object :times
      (pds~environment omega*current-proof-plan)))
    (taco-plus
     (env~lookup-object :plus
      (pds~environment omega*current-proof-plan))))
   (and
    (data~appl-p taco-x2)
    (term~taco-equal taco-plus
```

```
 (data~appl-function taco-x2))
(let*
 ((taco-z
   (term~constant-create
(+ (keim~name taco-x ) (keim~name taco-y )) taco-num))
  (taco-x3
   (data~appl-arguments taco-x2)))
 (and
  (and (data~primitive-p taco-z )
 (numberp (keim~name taco-z )))
  (listp taco-x3)
  (=
   (list-length taco-x3)
   2)
  (let*
   ((taco-x6
     (nth 1 taco-x3))
    (taco-x4
     (nth 0 taco-x3)))
   (and
    (data~appl-p taco-x6)
    (data~appl-p taco-x4)
    (term~taco-equal taco-times
     (data~appl-function taco-x6))
    (term~taco-equal taco-times
     (data~appl-function taco-x4))
    (let*
     ((taco-x7
       (data~appl-arguments taco-x6))
      (taco-x5
       (data~appl-arguments taco-x4)))
     (and
      (listp taco-x7)
      (=
       (list-length taco-x7)
       2)
      (listp taco-x5)
      (=
       (list-length taco-x5)
       2)
      (term~taco-equal taco-y
       (nth 0 taco-x7))
      (term~taco-equal taco-x
       (nth 0 taco-x5))
      (let*
       ((taco-a
         (nth 1 taco-x7)))
```

```
           (term~taco-equal taco-x5
            (list taco-x taco-a))))))))))))


(tac~deftactic split-monomials-plus-3 split-monomials-plus
 (in real)
 (parameters
  (pos pos+position "the position of the term")
  (x term+term "the first coefficient")
  (y term+term "the second coefficient"))
 (premises l1)
 (conclusions l2)
 (computations)
 (sideconditions
  (taco=split-monomials-plus-3-p
   (formula l2)
   (formula l1)
   pos x y))
 (description "Apply tactic split-monomials-plus
               to pattern (existent existent)."))


(defun taco=split-monomials-plus-3-p
 (taco-l2 taco-l1 taco-pos taco-x taco-y)
 (and
  (and (data~primitive-p taco-x )
     (numberp (keim~name taco-x )))
  (and (data~primitive-p taco-y )
     (numberp (keim~name taco-y )))
  (let*
   ((taco-x2
     (data~struct-at-position taco-l2 taco-pos))
    (taco-x0
     (data~struct-at-position taco-l1 taco-pos))
    (taco-num
     (env~lookup-object :num
      (pds~environment omega*current-proof-plan)))
    (taco-times
     (env~lookup-object :times
      (pds~environment omega*current-proof-plan)))
    (taco-plus
     (env~lookup-object :plus
      (pds~environment omega*current-proof-plan))))
   (and
    (data~appl-p taco-x2)
    (data~appl-p taco-x0)
    (term~taco-equal taco-plus
```

```
 (data~appl-function taco-x2))
(term~taco-equal taco-times
 (data~appl-function taco-x0))
(let*
 ((taco-z
   (term~constant-create
(+ (keim~name taco-x ) (keim~name taco-y )) taco-num))
  (taco-x3
   (data~appl-arguments taco-x2))
  (taco-x1
   (data~appl-arguments taco-x0)))
 (and
  (and (data~primitive-p taco-z )
 (numberp (keim~name taco-z )))
  (listp taco-x3)
  (=
   (list-length taco-x3)
   2)
  (listp taco-x1)
  (=
   (list-length taco-x1)
   2)
  (term~taco-equal taco-z
   (nth 0 taco-x1))
  (let*
   ((taco-x6
     (nth 1 taco-x3))
    (taco-x4
     (nth 0 taco-x3))
    (taco-a
     (nth 1 taco-x1)))
   (and
    (data~appl-p taco-x6)
    (data~appl-p taco-x4)
    (term~taco-equal taco-times
     (data~appl-function taco-x6))
    (term~taco-equal taco-times
     (data~appl-function taco-x4))
    (let*
     ((taco-x5
       (list taco-x taco-a))
      (taco-x7
       (list taco-y taco-a)))
     (and
      (term~taco-equal taco-x6
       (data~appl-create taco-times taco-x7))
      (term~taco-equal taco-x4
```

```
          (data~appl-create taco-times taco-x5)))))))))))

(defun taco=expand-split-monomials-plus (outline parameters)
 (let* ((taco-l2 (nth 0 outline))
        (taco-l1 (nth 1 outline))
        (taco-pos (nth 0 parameters))
        (taco-x (nth 1 parameters))
        (taco-y (nth 2 parameters)))
  (tacl~init outline)
   (let* ((outline1 (tacl~apply 'expand-num
                      (list nil taco-l1)
                      (list (pos~add-end taco-pos 1) taco-x taco-y)))
          (taco-l3 (nth 0 outline1)))
    (tacl~apply 'distribute-right
      (list taco-l2 taco-l3)
      (list taco-pos)))
  (tacl~end)))

; TACO END
```

Bibliography

[1] J. Abbott, A. van Leeuwen, and A. Strotmann. Objectives of OpenMath. Technical Report 12, RIACA, Technische Universiteit Eindhoven, Jun 1996.

[2] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, UK, September 2001. Springer-Verlag.

[3] P. B. Andrews, M. Bishop, and C. E. Brown. System description: Tps: A theorem proving system for type theory. In McAllester [50], pages 164–169.

[4] C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between isabelle and maple. In A. H. M. Levelt, editor, *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'95)*, pages 150–157. ACM Press, 1995.

[5] H. Barendregt and A. M. Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *J. Symb. Comput.*, 32(1/2):3–22, 2001.

[6] D. Basin, S. Friedrich, M. Gawkowski, and J. Posegga. Bytecode model checking: An experimental analysis, 2002.

[7] M. Beeson and F. Wiedijk. The meaning of infinity in calculus and computer algebra systems. In *AISC*, pages 246–258, 2002.

[8] C. Benzmüller and M. Kohlhase. System description: Leo - a higher-order theorem prover. In C. Kirchner and H. Kirchner, editors, *CADE*, volume 1421 of *Lecture Notes in Computer Science*, pages 139–144. Springer Verlag, 1998.

[9] C. Benzmüller and V. Sorge. ΩANTS – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus-2000)*, St. Andrews, UK, 6–7 August 2001. AK Peters, New York, NY, USA.

[10] A. V. Bocharov. DEliA: A System of Exact Analysis of Differential Equations using S. Lee Approach. Technical Report OWIMEX, Program Systems Institute, Pereslavl-Zalessky, UdSSR, 1989.

[11] G. Boole. *An Investigation of The Laws of Thought.* Macmillan, Barclay, & Macmillan, Cambridge, Großbritannien, 1854.

[12] W. Bosma and J. Cannon. *Handbook of Magma Functions.* Sydney, 1994.

[13] J. Brent W. Benson. Javascript. *SIGPLAN Not.*, 34(4):25–27, 1999.

[14] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9^{th} International Conference on Automated Deduction (CADE–9)*, volume 310 of *LNCS*, Argonne, Illinois, USA, 1988. Springer Verlag, Berlin.

[15] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The OYSTER-CLAM system. In M. E. Stickel, editor, *Proceedings of the 10^{th} International Conference on Automated Deduction (CADE–10)*, volume 449 of *LNCS*, pages 647–648, Kaiserslautern, Deutschland, 1990. Springer Verlag, Berlin.

[16] H. J. Bürckert. Unifikationstheorie. In K. H. Bläsius and H. J. Bürckert, editors, *Deduktionssysteme: Automatisierung des logischen Denkens*, chapter I, pages 112–125. Oldenbourg Verlag, München, zweite edition, 1992.

[17] O. Caprotti and A. M. Cohen. Connecting proof checkers and computer algebra using OpenMath. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, volume 1690 of *Lecture Notes in Computer Science*, pages 109–112, Nice, France, September 1999. Springer.

[18] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *First leaves: a tutorial introduction to Maple V.* Springer Verlag, Berlin, 1992.

[19] L. Cheikhrouhou and V. Sorge. $\mathcal{PDS}$ — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, 22–24 March 2000.

[20] A. Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1, 1936.

[21] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[22] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1986.

[23] J. H. Davenport. The AXIOM system. AXIOM Technical Report TR5/92 (ATR/3) (NP2492), Numerical Algorithms Group, Inc., Downer's Grove, IL, USA and Oxford, UK, Dezember 1992.

[24] M. Davis. The prehistory and early history of automated deduction. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 1–28. Springer, Berlin, Heidelberg, 1983.

[25] N. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[26] A. Fiedler. *P.rex*: An interactive proof explainer. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer Verlag.

[27] A. Franke and M. Kohlhase. System description: Mbase, an open mathematical knowledge base. In McAllester [50], pages 455–459.

[28] G. Frege. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle, 1879.

[29] B. Fuchssteiner et al. (The MuPAD Group). *MuPAD User's Manual*. John Wiley and sons, Chichester, New York, erste edition, März 1996.

[30] H. Gericke. *Geschichte des Zahlbegriffs*. Bibliographisches Institut, Mannheim, Germany, 1970.

[31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931.

[32] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, Berlin, 1979.

[33] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, UK, 1993.

[34] C. Grannell, D. Powers, and G. McLachlan. *Foundation Macromedia Dreamweaver MX 2004*. friends of ED, 2004.

[35] T. S. Group. *SIMATH Manual*. Fachbereich Mathematik, Universität des Saarlandes, Saarbrücken, 1994.

[36] A. C. Hearn. Reduce user's manual: Version 3.3. Technical report, Rand Corporation, Santa Monica, CA, USA, 1987.

[37] T. Hillenbrand, A. Jaeger, and B. Löchner. System Description: Waldmeister : Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 232–236, Trento, Italy, July 1999. Springer Verlag.

[38] K. Homann and J. Calmet. Combining theorem proving and symbolic mathematical computing. In J. Calmet and J. A. Campbell, editors, *Integrating Symbolic Mathematical Computation and Artificial Intelligence*, volume 958 of *LNCS*, pages 18–29, Second International Conference, AISMC-2 Cambridge, United Kingdom, 3.–5. August 1994. Springer Verlag, Berlin, 1995.

[39] K. Homann and J. Calmet. An Open Environment for Doing Mathematics. In M. Wester, S. Steinberg, and M. Jahn, editors, *Proceedings of 1st International IMACS Conference on Applications of Computer Algebra*, Albuquerque, USA, 1995.

[40] X. Huang, M. Kerber, M. Kohlhase, E. Melis, D. Nesmith, J. Richts, and J. Siekmann. KEIM: A toolkit for automated deduction. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE–12)*, volume 814 of *LNAI*, pages 807–810, Nancy, Frankreich, 1994. Springer Verlag, Berlin.

[41] R. Kaiser. *C++ With Borland C++Builder: An Introduction to the Ansi/Iso Standard and Object-Oriented Windows Programming*. SpringerVerlag, 2004.

[42] S. E. Keene. *Object–Oriented Programming in* COMMON LISP*: A Programmer's Guide to* CLOS. Addison–Wesley Publishing Company, 1989.

[43] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra with Proof Planning. In J. Calmet and C. Limongelli, editors, *Design and Implementation of Symbolic Computation Systems; International Symposium, DISCO '96, Karlsruhe, Germany, September 18-20, 1996; Proceedings*, volume LNCS1128 of *Lecture Notes in Computer Science*, Berlin;Heidelberg;New York, 1996. Springer.

[44] H. Kirchner and C. Ringeissen, editors. *Frontiers of Combining Systems, Third International Workshop, FroCoS 2000, Nancy, France, March 22-24, 2000, Proceedings*, volume 1794 of *Lecture Notes in Computer Science*. Springer, 2000.

[45] M. Kohlhase. Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In *AISC '00: Revised Papers from the International Conference on Artificial Intelligence and Symbolic Computation*, pages 32–52, London, UK, 2001. Springer-Verlag.

[46] Y. Lafont. Interaction nets. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 95–108, New York, NY, USA, 1990. ACM Press.

[47] G. W. Leibniz. Projet et essais pour arriver à quelque certitude pour finir une bonne partie des disputes et pour avancer l'art d'inventer. In K. Berka and L. Kreiser, editors, *Logiktexte*, chapter I.2, pages 16–18. Akademie-Verlag, deutsche "Ubersetzung, 1983, Berlin, 1686.

[48] G. W. Leibniz. *Neue Abhandlung über den menschlichen Verstand*. Reclam, Ditzingen, 1993.

[49] R. J. Lerdorf, K. Tatroe, B. Kaehms, and R. McGredy. *Programming Php*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[50] D. McAllester, editor. *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *LNAI*, Pittsburgh, PA, USA, 2000. Springer Verlag.

[51] J. L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[52] W. McCune. Otter 2.0. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction (CADE–10)*, volume 449 of *LNAI*, pages 663–664, Kaiserslautern, Deutschland, 1990. Springer Verlag, Berlin.

[53] A. Meier, M. Pollet, and V. Sorge. Comparing approaches to the exploration of the domain of residue classes. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, October 2002. S. Linton and R. Sebastiani, eds.

[54] E. Melis. The "limit" domain. In *AIPS*, pages 199–207, 1998.

[55] E. Melis and A. Meier. Proof Planning with Multiple Strategies. In J. Loyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, and Y. S. P. Stuckey, editors, *First International Conference on Computational Logic (CL-2000)*, volume 1861 of *LNAI*, pages 644–659, London, UK, 2000. Springer-Verlag.

[56] E. Melis, J. Zimmer, and T. Müller. Integrating constraint solving into proof planning. In Kirchner and Ringeissen [44], pages 32–46.

[57] E. Melis, J. Zimmer, and T. Müller. Integrating constraint solving into proof planning. In H. Kirchner and C. Ringeissen, editors, *Frontiers of Combining Systems – Third International Workshop, FroCoS 2000*, volume 1794 of *LNAI*, pages 32–46, Nancy, France, March 2000. Springer.

[58] M. Minsky. Steps toward artificial intelligence. In *Computers & thought*, pages 406–450. MIT Press, Cambridge, MA, USA, 1995.

[59] S. Näher and K. Mehlhorn. Leda-manual version 3.0. Research Report MPI-I-93-109, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, February 1993.

[60] D. Nesmith. KEIM–*Manual, Version 1.2*. Universität des Saarlandes, Im Stadtwald, Saarbrücken, 1994.

[61] A. Newell, C. Shaw, and H. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proceedings of the 1957 Western Joint Computer Conference*, New York, USA, 1957. McGraw-Hill. Reprinted in Computers and Thought, Edward A. Feigenbaum, Julian Feldman, Hrsg., New York, USA, 1963.

[62] T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[63] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *CAV-96*, volume 1102 of *LNCS*, pages 411–414. Springer Verlag, Berlin, 1996.

[64] L. C. Paulson. Isabelle: The next 700 theorem provers. *Logic and Computer Science*, pages 361–386, 1990.

[65] B. Pientka. Higher-order substitution tree indexing. In C. Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2003.

[66] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.

[67] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[68] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[69] J. Siekmann, S. M. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. $\mathcal{LOUI}$: $\mathcal{L}$ovely ΩMEGA $\mathcal{U}$ser $\mathcal{I}$nterface. *Formal Aspects of Computing*, 11(3):326–342, 1999.

[70] J. H. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.-P. Wirth, and J. Zimmer. Proof development with omega. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 144–149, London, UK, 2002. Springer-Verlag.

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

[72] V. Sorge. Non-trivial symbolic computations in proof planning. In Kirchner and Ringeissen [44], pages 121–135.

[73] G. L. Steele. COMMON LISP*: The Language – second edition.* Digital Press, zweite edition, 1990.

[74] M. E. Stickel. The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.

[75] The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, Mar. 2002.

[76] F. Theiß and V. Sorge. Automatic generation of algorithms and tactics. In O. Caprotti and V. Sorge, editors, *Calculemus 2002 — 10th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning — Work in Progress Papers*, number SR-02-04 in Seki Report, pages 74–75, Marseille, France, June 3-5 2002. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany.

[77] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4:2–22, 1960.

[78] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in LNAI, pages 275–279. Springer Verlag, 2002.

[79] A. N. Whitehead and B. Russell. *Principia Mathematica*, volume I. Cambridge University Press, Cambridge, Great Britain, second edition, 1910.

[80] S. Wolfram. *Mathematica: a System for Doing Mathematics by Computer.* Addison-Wesley, 1991.