DK10 Wolfgang Schreiner: Formally Specified Computer Algebra Software

Abstract

We propose research on a semantic framework and supporting tools for the formal specification of computer algebra software written in statically untyped programming languages for the manipulation of expressions as they are used in the major computer algebra systems today. The focus of the work is to apply formal methods, rather than for verifying the correctness of the software, for finding and avoiding internal inconsistencies, in particular violations of method preconditions, which are typical indications of errors in the software (or at least of a lack of understanding of the developer).

Current State of Research

Computer algebra software has been repeatedly criticized concerning the correctness of the computed results [274, 228, 237, 241, 280]. While it is typically not the case that a result is "totally wrong", it is sometimes also not "really correct": it may only represent part of the answer or may be only valid under additional logical assumptions or in particular mathematical domains or under special interpretations of the arguments; however these caveats frequently remain unspecified. The main reasons seem limitations in the language between user and system incapable of expressing sufficient information such as domain information and logical dependencies [241], the unclear meaning of even fundamental notions such as "equality" [230] and most important a gross under-specification of the operations in the system documentation [280]. Consequently it is also for mathematical programmers difficult to build reliable software on top of computer algebra libraries or to employ computer algebra software as self-contained components [245] in contexts where no human is in the loop to continuously interpret the results and check their adequacy. This also the experience of the proposer and his coworkers who have in two recent FWF projects [262] worked on the development of an environment for formally describing, brokering, and executing mathematical services [204, 203, 202, 220], see also the supervised Ph.D. thesis [201].

The situation may be due to a separation between those users that are "just" interested in computing and those that are also inclined to deal with logical issues [241]. Nevertheless, there has also been considerable interest in combining the computational capabilities of computer algebra systems with the logical capabilities of theorem proving systems [218, 261, 211, 198, 221]. One branch of work provides *computer algebra systems with advanced reasoning capabilities* by connecting them to theorem proving systems: [196, 239] describes a Maple-PVS interface where Maple calls PVS to e.g. check the validity of a method call's precondition and postcondition; similarly [194, 195] presents a symbolic definite integral table where a given argument is matched against the entries in the table by invoking PVS to verify the necessary side conditions on the argument. Another branch of work provides *theorem provers with computing capabilities* by connecting them to computer algebra systems: in [243, 242] the proving assistant HOL uses Maple as an "oracle" to find answers to certain computational tasks which are then formally verified by HOL (checking an answer is often much easier than finding it); similarly [199] describes a connection of Isabelle and Maple and [200] a connection of Isabelle and the computer algebra

library Sum-It implemented in Aldor; here the answers are simply trusted to be correct. [273] presents a generic interface between the Omega proof planing system and computer algebra systems where the computer algebra system may return information that can be used to verify the computation. One may also attempt a much closer *integration of computing and reasoning*: e.g. the the reasoning system Theorema [276, 216, 215] built on top of Mathematica implements by its PCS method a proof heuristics that iterates phases of proving, computing, and solving on the basis of the algebraic methods provided by Mathematica; other examples of reasoning systems embedded into computer algebra systems are REDLOG [231] and Analytica [208, 226].

However, most of this work does in itself not contribute to increasing the trust in the correctness of the results returned by computer algebra software. At one extreme, one might simply accept this fact and ask for a separate check of every result and ultimately require that computer algebra software returns, together with its result, a *correctness certificate* that enables this check, e.g. an LF-style formal correctness proof like it is (optionally) returned by some arithmetic decision procedures [251, 263]; however, checking every result is costly and checking itself does not improve the correctness of the software. At the opposite side of the spectrum, one might also demand a radical change in the process of developing computer algebra software such that it becomes *correct by construction*: one may e.g. develop the software in a type-theoretic framework such that the specification of a method is denoted by its type and the fact that the method type-checks correctly guarantees its correctness with respect to the specification. A step in this direction is made in the Atypical project [279, 265, 266] where the type system of Aldor is modified such that its dependent types can be used to describe propositions and Aldor category specifications become axiomatic datatype specifications [278]; likewise [264] presents a type-theoretic elaboration of polynomial rings and the Gröbner bases algorithm. An alternative approach is to build a computer algebra system on top of a theorem proving system such that computations become rewriting proofs, see e.g. [223] for a computer algebra system built on top of HOL Light. One may also resort to program synthesis [207], where from a specification a program is constructed that is provably correct with respect to this specification, either manually by a sequence of stepwise refinement steps [249] or by some heuristically guided automatic process [214].

A more pragmatic approach is to equip a computer algebra system respectively language with a *formal specification language* and a corresponding logical framework: in [234, 235, 250] an integration of the behavioral interface specification language Larch with Aldor is presented; from Larch/Aldor programs, verification conditions are generated that are forwarded to the Larch prover for verification. The specification language of the Coq proving assistant has been variously used to define computer algebra algorithms and prove their correctness [277, 240, 259, 258, 260]; from the Coq definitions executable Ocaml programs can be automatically extracted. The FoCal (former Foc) project [267, 238, 213] has developed an axiomatic datatype specification language in which hierarchies of mathematical domains can be defined; a compiler extracts the computational parts as Ocaml programs and also generates verification conditions that can be interactively handled with the help a proving assistant; proofs are produced in the form of Coq scripts/terms that can be subsequently checked by Coq.

Much more than in computer algebra, in computer science interest in formal methods for modeling and reasoning has surged since the 1990s. Various specification languages have been developed, some of them programming-language independent e.g. VDM [248], Z [281], B [193],

Larch [256], Alloy [247], or also the Object Constraint Modeling Language OCL which is part of the UML standard [252]. Other formalisms are bound to particular programming languages such as Larch/C++ for C++ [256], Spark for Ada [205, 246], JML for Java [255], and Spec# for C# [205]. These languages are accompanied by corresponding tool sets that make use of automated reasoning techniques [217, 222] which has become possible due to the advances in SMT (satisfiability modulo theories) solving achieved since the late 1990s [272, 206]. The focus here is clearly not on verification of program correctness but on light-weight formal methods [269] e.g. to find by extended static checking [236, 224] possible runtime errors and internal inconsistencies in a program such as violations of specified method preconditions. Nevertheless, some environments provide integrated proving assistants that also support the interactive verification of complex correctness properties [210, 246].

Objectives and Methods

Admittedly the research described above has had up to now little impact on the actual practice of writing computer algebra software. Apart from those approaches that ask for fundamental changes in the computing principles or the development process, even the more pragmatic ones are bound to languages such as Aldor or Focal that are more advanced but also less used compared to the languages of systems like Maple, Mathematica, GAP, CoCoa, and others in which the vast majority of computer algebra software is written. Unlike languages with mathematically founded notions like categories (Aldor [197]) respectively species (Focal [267]) or magmas (Magma [212]) suitable for building abstract hierarchies of mathematical datatypes (see [257, 275, 219, 270, 225, 229] for related work) these languages are not even statically typed; they focus (in the tradition of Common Lisp) on the construction and manipulation of expressions compound of symbols and of values from specially supported datatypes such as unbounded integers. If formal methods shall influence the practice of writing computer algebra software, they have to be also applied to software written in these languages, and also provide some immediate advantage rather than only pointing towards a goal that is still far-away. The situation is analogous to that in computer science where remarkable success has been achieved since the focus has shifted from proving the full correctness of simple programs in rarely used languages towards the application of reasoning technologies to finding errors in complex programs in wide-spread languages.

The overall goal of the proposed research is therefore to work on a a specification language, a corresponding reasoning framework, and supporting tools for computer algebra languages that semantically operate on the level of expressions and for which any high-level interpretation as a mathematical datatype has to be especially constructed by a corresponding specification. This interpretation can be then used by the reasoning framework for checking the internal consistency of a program composed of multiple methods (and ultimately also form the basis for the verification of the program).

The core idea on which this work is based was already introduced by Hoare [244] and has been more precisely elaborated and further refined in [209, 253, 233, 232, 254]: in essence, for a concrete program type $\mathcal{C}$, a (partial) mapping $a : \mathcal{C} \to \mathcal{A}$ into an abstract mathematical type $\mathcal{A}$ is defined; a concrete program function $f : \mathcal{C} \to \mathcal{C}$ can be then specified by a precondition $P \subseteq \mathcal{A}$ and postcondition $Q \subseteq \mathcal{A} \times \mathcal{A}$ such that for every concrete argument $x \in \mathcal{C}$ with $P(a(x))$ the application $f(x)$ returns a concrete result $y \in \mathcal{C}$ such that $Q(a(x), a(y))$. The specification

(P, Q) of f thus operates, rather than on the concrete type $\mathcal{C}$, on the abstract type $\mathcal{A}$; the program function f has been thus specified as a mathematical function. The idea can be easily generalized to a heterogeneous scenario with multiple program types respectively abstract types, also the same program type may represent different abstract types in different contexts.

However, to turn the idea into a practical specification language, multiple questions have to be resolved. The first one, how to specify the semantics of the abstract type $\mathcal{A}$, is actually not crucial: any algebraic specification language in the tradition of OBJ3, Larch, CASL will do, provided that it supports a loose specifications semantics (the descriptions need denote only specifications, not implementations of the mathematical types). The languages mainly differ in the way in their organize specifications in the large, i.e. how they support modularization, genericity, and subtyping; these questions are orthogonal to our present discussion; it suffices that the specification yields a well-defined (possibly existentially quantified) type $\mathcal{A}$ with corresponding axioms.

A (for the reasoning framework) more critical question is how to specify the *abstraction function* a which is necessary if we want to derive from the information about the abstract $a(x)$ (provided by a specification) also information on the concrete x (needed for a verification). A very pragmatic solution is the one (implicitly) suggested by the model functions of JML [255]: the programmer defines a concrete program function in the implementation language that composes the abstract object in terms of the operations specified on $\mathcal{A}$. The properties of a required for reasoning are provided by a pre/post-condition pair on $\mathcal{A}$; if the implementation of the function can be verified against this specification, the consistency of this specification is guaranteed.

Since f and a operate on elements of C, understanding f and a can be considerably aided by introducing a *type discipline* also on the concrete program types. The type system shall be kept simple but suffice to keep an expression $x+1$ (with uninterpreted symbol x) apart from the integer $x+1$ (where x is an integer variable) and also indicate the syntax of expressions accepted and generated by a program (i.e. it shall allow to describe a suitable tree grammar [227]). More sophisticated constraints can be handled in the tradition of PVS and the proposer's RISC ProofNavigator [268, 271] by subtype predicates which give rise to type-checking conditions verified by a reasoner supporting the type checker.

All in all, the *specification of a program function* f then consists (backed by a collection previously defined abstract data types, abstraction mappings, and high-level properties on these) correctness lemmas (see below), a type signature for f (describing the concrete types of the arguments and results), a frame condition (mentioning any global variables assignable by f), a precondition, a postcondition (describing the relationship of the prestate to the poststate if the function has returned normally), and optionally an exceptional postcondition (describing the relationship of the prestate to the poststate, if the function has raised an exception); for (mutually) recursively defined functions, also termination terms may be provided that denote values from a well-founded ordering that must decrease with every invocation. Such a specification itself is already subject to reasoning: is it well-typed, is it trivial (equivalent to true), satisfiable (not equivalent to false), is it implied by another specification, etc?

To relate this specification to the actual implementation of f, a corresponding *type checking and reasoning calculus* for the underlying implementation language has to be devised. Since many computer algebra languages have a rather similar value and execution semantics, we will devise first a core language that captures the essential objects and constructs of these lan-

guages at a lower level and define the framework with respect to this core language. Compared to imperative/object-oriented languages, this is somewhat simplified because the semantics of basic data-types is simpler (e.g. unbounded integers rather than machine numbers) and the pointer-semantics of structures plays a less dominant role (most programs do not destructively update their arguments). In a second step, we will provide a translation from some high-level language subset(s) to this core language such that concrete programs can be treated. Analogously to ESC/Java2, there may be multiple translations of a language depending on the level of accuracy that the programs shall be modeled (e.g. by unfolding those loops that are not equipped with invariants by the user). The outcome of the framework are ultimately conditions that are necessary (and potentially also sufficient) be make the method meets its specification.

As for actually proving these conditions, in real-world scenarios we have to deal with the *partial under-specification* of abstract types respectively of notions defined on these types (e.g. undefined predicates) such that proofs of conditions depending on the semantics of these notions are actually not possible. Such situations will be typically detected in the course of these proofs; rather than just stating that a proof fails, it may be better to state the assumptions under which it would succeed (e.g. $A(x, y) \Rightarrow B(y)$ for undefined predicate B) and let the user, provided that he asserts the correctness of these assumptions or simpler ones, annotate the specification of a method with the assumptions and thus allow the proofs to make use of them. In this fashion, (previously also made but then implicit) correctness assumptions are now explicit and represent obligations for further formalization and proof.

We will work on a *supporting tool* that type-checks a program with respect to its type signature and, if this check succeeds, generates correctness conditions for the pre-conditions and (based on interfaces to automatic provers such as the Theorema system as well as to interactive proving assistants such as the proposer's RISC ProofNavigator) attempts to prove them. Again, the goal here is primarily to find internal inconsistencies in the program, post-conditions are just used as *assumptions* for the proofs of the pre-conditions of the subsequently called methods. To improve the level of automation, appropriate proving strategies are investigated and incorporated into the prover; if a proof fails, the tool provides the developer in a nice format with the knowledge available at a method call whose pre-condition could not be verified and requires further treatment (by manual verification or addition of a corresponding correctness assumption to the method); only when internal consistency with respect to the specification (and associated assumptions) is achieved, also the later verification of the program's postcondition may be attempted.

To guide our work, we will work with some program fragments from the CASA system (implemented in Maple) of the proposer of DK11. These are re-specified/re-written to our specification language and a corresponding subset of the implementation language to yield explicitly specified and internally consistent programs. Thus also errors in the original implementation will be detected and thus the reliability of the application will be improved.

Work plan

The organization of the Ph.D. work is sketched in the following table:

Months	Description
1–12	Course work with focus on CA, CA software, logic, formal methods Study of specification languages Sample specifications of CA methods
13–15	*Visit of an international institution*
16–24	Research on specification formalism, semantics and reasoning framework
25–27	*Visit of an international institution*
28–34	Work on specification checking prototype Application to sample specifications
35–36	Writing of Ph.D. thesis

After an initial training period with emphasis on courses related to computer algebra, software, logic, and formal methods, the Ph.D. student will start the investigation of prior work in formal specification and, based on an initial sketch of the proposed framework, work on sample specifications of concrete CA methods as available in the CASA system. Work on the actual details of the specification formalism, semantics, and reasoning framework, will proceed in close collaboration with the proposer, whose work during that time will focus on the development of a suitable software environment for education in program reasoning based on his prior work on the RISC ProofNavigator; the results of the PhD work will fit into this environment.

During the period of the actual research, the student will spend about six months total time for visits at institutions whose work is related to the Ph.D. topic. RISC has suitable contacts to various institutions that pursue the integration of computer algebra and logic, e.g. the Centre for Interdisciplinary Research in Computational Algebra in St. Andrews directed by Steve Linton (together with Ursula Martin who is now at the Queen Mary University of London director of a former project on embedded verification techniques for computer algebra) or the SPI research team at LIP6, Universite Pierre et Marie Curie Paris, directed by Therese Hardin (leader of the Focal project).