Towards a Behavioral Analysis of Computer Algebra Programs*

(Extended Abstract)

Muhammad Taimoor Khan
DK Computational Mathematics
Johannes Kepler University
Linz, Austria
muhammad.khan@dk-compmath.jku.at

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Wolfgang.Schreiner@risc.jku.at

We present our initial results on the behavioral analysis of computer algebra programs. Computer algebra programs written in symbolic computation languages such as Maple and Mathematica sometimes do not behave as expected [5], e.g. by triggering runtime errors or delivering wrong results. There has been a lot of research on applying formal techniques to classical programming languages, e.g. Java [6], C# [1] and C [3] etc., but we aim to apply the same techniques to computer algebra languages. Therefore our goal is to design and develop a tool for the static analysis of computer algebra programs [12]. The tool will automatically find errors in programs annotated with extra information such as variable types and method contracts [11], in particular type inconsistencies and violations of method preconditions.

The task of applying formal techniques to widely used computer algebra languages (Maple and Mathematica) is more complex as these are fundamentally different from classical languages. In particular, we found the following challenges respectively differences to classical languages for formal type checking respectively specifying Maple programs (which are typical for most computer algebra languages):

- The language supports some non-standard types of objects, e.g. symbols, unevaluated expressions and polynomials.
- There is no clear difference between declaration and assignment. A global variable is introduced by an assignment; a subsequent assignment may modify the type information for the variable.
- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-tests which selects the respective code-block for further execution. This makes type inference more complex.
- The language allows runtime type checking by type annotations but these annotations are optional which give rise to type ambiguities. This also makes type inference more complex.
- Maple values are organized in a kind of polymorphic type system with a sub-typing relationship such that we can assign a value to different types. This also makes type inference more complex.

The challenge for a specification language for Maple is to overcome those particularities of the language that hinder static analysis because Maple was not designed for this purpose (type checking respectively verification).

There are various computer algebra languages, Mathematica and Maple being the most widely used by far [13], both of which are dynamically typed. We have in our work chosen Maple for the following reasons:

- Maple has an imperative style of programming while Mathematica has a rule-based programming style with more complex semantics.

*The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

- Maple has type annotations for runtime checking which can be directly applied for static analysis. (There are also parameter annotations in Mathematica but they are used for selecting the appropriate rule at runtime).

Still the results we derive with type checking Maple can be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple.

As a starting point, we have defined a subset of the computer algebra language Maple called *MiniMaple* [9, 10]. Since type safety is a prerequisite of program correctness, we have formalized a type system for *MiniMaple* and implemented a corresponding type checker. Furthermore, we have defined a specification language to formally specify the behavior of *MiniMaple* procedures and implemented a corresponding type checker. As the next step, we will develop a tool to automatically detect errors in *MiniMaple* programs with respect to their specifications.

In the following we will brief the main features of our work.

A Type System for *MiniMaple*: *MiniMaple* uses Maple type annotations for static analysis. Based on these annotations we defined a language of types and a corresponding type system. The type system supports the usual concrete data types, sets, lists and records. It also supports some non-standard types, e.g. the union type of various types, symbols, unevaluated expressions and polynomials etc. Type *anything* is the super-type of all types. The problem of statically type-checking *MiniMaple* programs is related to the problem of statically type-checking scripting languages such as Ruby [8], but there are also fundamental differences due to the different language paradigms.

In the following, we highlight the problems arising from type checking various *MiniMaple* programs.

- Global variables (declarations) can not be type annotated; therefore to global variables values of arbitrary types can be assigned in Maple. We introduce *global* and *local* contexts to handle the different semantics of the variables inside and outside of the body of a procedure respective loop.
  - In a *global* context new variables may be introduced by assignments and the types of variables may change arbitrarily by assignments.
  - In a *local* context variables can only be introduced by declarations. The types of variables can only be *specialized* i.e. the new value of a variable should be a sub-type of the declared variable type.
  - The sub-typing relation is observed while specializing the type of a variable.
- Maple supports type tests (i.e. type(I,T)) to direct the control flow of a program. Different branches of a conditional may have different pieces of type information for the same variable. We keep track of the type information introduced by the branches to allow only satisfiable tests.
- With the use of type-tests, the number of loop iterations might influence the type information and one cannot determine the concrete type by the static analysis. To handle this non-determination of types we put a reasonable upper bound (least fixed point) on the types of variables. As a special case this upper bound is the type of a variable prior to the body of a loop.

The type checker has been applied to the Maple package *DifferenceDifferential* [4]. No crucial typing errors have been found but some bad code parts have been identified that can cause problems.

A Specification Language for *MiniMaple*: Based on the formalism of our type system we have defined a formal specification language for *MiniMaple*. The specification language is a logical formula language that mainly uses Maple notations but also has its own notations. The language allows to formally specify the behavior of the procedures as a state relationship, e.g. by specifying pre/post-conditions of a procedure and other constraints. The specification language supports specification declarations, procedure and loop specifications and assertions. The language also supports abstract data types, while the existing specification languages are weaker in such specifications. Currently we are defining formal semantics of *MiniMaple*. Based on this semantics we will define formal semantics of the specification language so that it specifies the intended algebraic properties. The specification language aims to realize respectively bridge the gap between actual computer algebra algorithm and its corresponding implementation [4].

We may use this specification language to generate executable assertions that are embedded in *MiniMaple* programs and check at runtime the validity of pre/post conditions. Our main goal, however, is to use the specification language for static analysis, in particular to detect violations of method preconditions. Here we currently investigate two possibilities:

1. We may directly generate verification conditions and use Satisfiability Modulo Theories (SMT) solvers or interactive theorem provers to prove their correctness.

2. We may use some existing framework to generate verification conditions and prove the correctness, e.g. by the Boogie [2] framework developed by Microsoft and Why [7] by LRI-France. Here we need to translate our specification annotated *MiniMaple* program into an intermediate language of Boogie/Why and then use their various proving back-ends for verification.

The formal specification of the *DifferenceDifferential* package developed at our institute will be the main test for our specification language and checking framework.

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS '04), volume 3362 of LNCS*, pages 49–69. Springer, 2004.

[2] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of LNCS*, pages 364–387. Springer, 2006.

[3] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008. `http://frama-c.com/download/acsl_1.2.pdf`.

[4] Christian Dönch. Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2009.

[5] Richard J. Fateman. Why Computer Algebra Systems Sometimes Can't Solve Simple Equations. *SIGSAM Bulletin*, 30(2):8–11, 1996.

[6] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. A Tutorial, 2006. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[7] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification condition generator. Research report 1366, LRI - CNRS UMR 8623, Université Paris-Sud, France, March 2003.

[8] Jeffrey S. Foster Michael Furr, Jong-hoon (David) An and Michael Hicks. Static Type Inference for Ruby. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing, OOPS track*, Honolulu, HI, 2009.

[9] Muhammad Taimoor Khan. Software for *MiniMaple*. `http://www.risc.jku.at/people/mtkhan/dk10/`.

[10] Muhammad Taimoor Khan. A Type Checker for *MiniMaple*. RISC Technical Report 11-05, also DK Technical Report 2011-05, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2011.

[11] Bertrand Meyer. Applying Design by Contract. *Computer*, 25:40–51, October 1992.

[12] Wolfgang Schreiner. Project Proposal: Formally Specified Computer Algebra Software. Doktoratskolleg (DK), Johannes Kepler University, Linz, Austria, `http://www.risc.jku.at/projects/dk10`, 2007.

[13] Inna K. Shingareva and Carlos Lizárraga-Celaya. *Maple and Mathematica: A Problem Solving Approach for Mathematics*. Springer, 2nd ed. edition, September 2009.