On Formal Specification of Maple Programs*

Muhammad Taimoor Khan[1] and Wolfgang Schreiner[2]

[1] Doktoratskolleg Computational Mathematics
[2] Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
muhammad.khan@dk-compmath.jku.at,
Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at/people/mtkhan/dk10/

Abstract. This paper is an example-based demonstration of our initial results on the formal specification of programs written in the computer algebra language *MiniMaple* (a substantial subset of Maple with slight extensions). The main goal of this work is to define a verification framework for *MiniMaple*. Formal specification of *MiniMaple* programs is rather complex task as it supports non-standard types of objects, e.g. symbols and unevaluated expressions, and additional functions and predicates, e.g. runtime type tests etc. We have used the specification language to specify various computer algebra concepts respective objects of the Maple package *DifferenceDifferential* developed at our institute.

1 Introduction

We report on a project whose goal is to design and develop a tool to find behavioral errors such as type inconsistencies and violations of method preconditions in programs written in the language of the computer algebra system Maple; for this purpose, these programs need to be annotated with the types of variables and methods contracts [8].

As a starting point, we have defined a substantial subset of the computer algebra language Maple, which we call *MiniMaple*. Since type safety is a prerequisite of program correctness, we have formalized a type system for *MiniMaple* and implemented a corresponding type checker. The type checker has been applied to the Maple package *DifferenceDifferential* [2] developed at our institute for the computation of bivariate difference-differential dimension polynomials. Furthermore, we have defined a language to formally specify the behavior of *MiniMaple* programs. As the next step, we will develop a verification calculus for *MiniMaple*. The other related technical details about the work presented in this paper are discussed in the accompanying paper [7]. For project details and related software, please visit http://www.risc.jku.at/people/mtkhan/dk10/.

The rest of the paper is organized as follows: in Section 2, we briefly demonstrate formal type system for *MiniMaple* by an example. In Section 3, we

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

J. Jeuring et al. (Eds.): CICM 2012, LNAI 7362, pp. 442–446, 2012.

introduce and demonstrate the specification language for *MiniMaple* by an example. Section 4 presents conclusions and future work.

2 A Type System for *MiniMaple*

MiniMaple procedure parameters, return types and corresponding local (variable) declarations needs to be (manually) type annotated. Type inference would be partially possible and is planed as a later goal. The results we derive with type checking Maple can also be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple.

Listing 1 gives an example of a *MiniMaple* program which we will use in the following section for the discussion of type checking respective formal specification. Also the type information produced by the type system is shown by the mapping π of program variables to types. For other related technical details of the type system, please see [4].

```
1. status:=0;
2. prod := proc(l::list(Or(integer,float)))::[integer,float];
3.         # π={l:list(Or(integer,float))}
4.         global status;
5.         local i, x::Or(integer,float), si::integer:=1, sf::float:=1.0;
6.         # π={..., i:symbol, x:Or(integer,float),..., status:anything}
7.         for i from 1 by 1 to nops(l) do
8.             x:=l[i]; status:=i;
9.             # π={..., i:integer, ..., status:integer}
10.            if type(x,integer) then
11.            # π={..., i:integer, x:integer, si:integer, ..., status:integer}
12.                    if (x = 0) then
13.                        return [si,sf];
14.                    end if;
15.                    si:=si*x;
16.            elif type(x,float) then
17.            # π={..., i:integer, x:float, ..., sf:float, status:integer}
18.                     if (x < 0.5) then
19.                         return [si,sf];
20.                     end if;
21.                     sf:=sf*x;
22.            end if;
23.            # π={..., i:integer, x:Or(integer,float),..., status:integer}
24.        end do;
25.        # π={..., i:symbol, x:Or(integer,float),..., status:anything}
26.        status:=-1;
27.        # π={..., i:symbol, x:Or(integer,float),..., status:integer}
28.        return [si,sf];
29.        end proc;
30. result := prod([1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);
```

Listing 1. The example *MiniMaple* procedure type-checked

The following problems arise from type checking *MiniMaple* programs:

- Global variables (declarations) can not be type annotated; therefore values of arbitrary types can be assigned to global variables in Maple. Therefore we introduce *global* and *local* contexts to handle the different semantics of the variables inside and outside of the body of a procedure respectively 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 types of variables.
- A predicate **type**(*E,T*) (which is true if the value of expression E has type T) may direct the control flow of a program. If this predicate is used in a conditional, then different branches of the conditional may have different type information for the same variable. We keep track of the type information introduced by the different type tests from different branches to adequately reason about the possible types of a variable. For instance, if a variable x has type Or(integer,float), in a conditional statement where the "if" branch is guarded by a test type(x,integer), in the "else" branch x has automatically type float. This automatic type inferencing only applies if an identifier has a union type. A warning is generated, if a test is redundant (always yields true or false).

The type checker has been applied to the Maple package *DifferenceDifferential* [2]. No crucial typing errors have been found but some bad code parts have been identified that can cause problems, e.g., variables that are declared but not used (and therefore cannot be type checked) and variables that have duplicate global and local declarations.

3 A Specification Language for *MiniMaple*

Based on the type system presented in the previous section, we have developed a formal specification language for *MiniMaple*. This language is a logical formula language which is based on Maple notations but extended by new concepts. The formula language supports various forms of quantifiers, logical quantifiers (**exists** and **forall**), numerical quantifiers (**add**, **mul**, **min** and **max**) and sequential quantifier (**seq**) representing truth values, numeric values and sequence of values respectively. We have extended the corresponding Maple syntax, e.g., logical quantifiers use typed variables and numerical quantifiers are equipped with logical conditions that filter values from the specified variable range.

Also the language allows to formally specify the behavior of procedures by pre- and post-conditions and other constraints; it also supports loop specifications and assertions. In contrast to specification languages such as Java Modeling Language [3], abstract data types can be introduced to specify abstract concepts and notions from computer algebra.

```
(*@
requires true;
global status;
ensures
   (status = -1 and RESULT[1] = mul(e, e in l, type(e,integer))
   and RESULT[2] = mul(e, e in l, type(e,float))
   and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)
   and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5))
   or
   (1<=status and status<=nops(l) and RESULT[1] = mul(l[i], i=1..status-1, type(l[i],integer))
   and RESULT[2] = mul(l[i], i=1..status-1, type(l[i],float))
   and ((type(l[status],integer) and l[status]=0) or (type(l[status],float) and l[status]<0.5))
   and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
   and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5));
@*)
proc(l::list(Or(integer,float)))::[integer,float]; ... end proc;
```

Listing 2. The example *MiniMaple* procedure formally specified

Listing 2 gives a formal specification of the example procedure introduced in Section 2. The procedure has no pre-condition as shown in the **requires** clause; the **global** clause says that a global variable *status* can be modified by the body of the procedure. The normal behavior of the procedure is specified in the **ensures** clause. The post condition specifies that, if the complete list is processed then we get the result as the product of all integers and floats in the list but if procedure terminates pre-maturely then we only get the product of integers and floats till the value of variable *status* (index of the input list). For the complete syntax and other details of the formal specification language see [6]. To test the specification language, we have formally specified some parts of the Maple package *DifferenceDifferential* [2] developed at our institute as the main test for the specification language.

4 Conclusions

We may use the specification language sketched in this short paper 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 to verify the correctness of *MiniMaple* annotated programs by static analysis, in particular to detect violations of methods preconditions. For this purpose, based on the results of a prior investigation, we intend to use the verification framework Why3 [1] to implement the verification calculus for *MiniMaple*, i.e., to translate *MiniMaple* into the intermediate language of Why3 and to apply its verification condition generator to generate verification conditions and prove their correctness with various back-end provers. Since the verification calculus must be sound, we have defined a formal semantics of *MiniMaple* [5] such that the correctness of the transformation can be shown.

References

1. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
2. Dönch, C.: Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2009)
3. Leavens, G.T., Cheon, Y.: Design by Contract with JML. A Tutorial (2006), `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`
4. Khan, M.T.: 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)
5. Khan, M.T.: Formal Semantics of MiniMaple. DK Technical Report 2012-01, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (January 2012)
6. Khan, M.T., Schreiner, W.: Towards a Behavioral Analysis of Computer Algebra Programs (Extended Abstract). In: Pettersson, P., Seceleanu, C. (eds.) Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT 2011), Vasteras, Sweden, pp. 42–44 (October 2011)
7. Khan, M.T., Schreiner, W.: Towards the Formal Specification and Verification of Maple Programs. In: Conferences on Intelligent Computer Mathematics, Calculemus Track (submitted, 2012)
8. Meyer, B.: Applying Design by Contract. Computer 25, 40–51 (1992)