Towards a Behavioral Analysis of Computer Algebra Programs*

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
and
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Muhammad.Taimoor.Khan@risc.jku.at

November 15, 2011

Abstract

In this paper, we present our initial results on the behavioral analysis of computer algebra programs. The main goal of our work is to find typing and behavioral errors in such programs by static analysis of the source code. This task is more complex for widely used computer algebra languages (Maple and Mathematica) as these are fundamentally different from classical languages: for example they support non-standard types of objects, e.g. symbols, unevaluated expressions, polynomials etc.; moreover they use type information to direct the flow of control in the program and have no clear difference between declaration and assignment. For this purpose, we have defined the syntax and the formal type system for a substantial subset of the computer algebra language Maple, which we call *MiniMaple*. The type system is implemented by a type checker, which verifies the type correctness and respectively reports typing errors. We have applied the type checker to the Maple package *DifferenceDifferential* developed at our institute. Currently we are working on a formal specification language of *MiniMaple* and the specification of this package. The specification language will be used to find errors in computer algebra programs with respect to their specifications.

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

Contents

1 Introduction **3**

2 State of the art **4**

3 *MiniMaple* **6**

4 A Type System for *MiniMaple* **7**
4.1 Design 8
4.1.1 Global Variables 8
4.1.2 Type Tests 8
4.2 Typing Judgments 10
4.3 Typing Rules 11
4.3.1 Auxiliary Functions 11
4.3.2 Auxiliary Predicates 12
4.4 Application 13

5 A Formal Specification Language for *MiniMaple* **15**
5.1 Specification Declarations 15
5.2 Procedure Specification 16
5.3 Loop Specification 17
5.4 Assertions 18

6 Conclusions and Future Work **18**

1 Introduction

Computer algebra programs written in symbolic computation languages such as Maple and Mathematica sometimes do not behave as expected, 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 [13], C# [3] and C [5] etc., but we aim to apply the same techniques to computer algebra languages. Therefore our aim is to design and develop a tool for the static analysis of computer algebra programs [23]. The tool will find errors in programs annotated with extra information such as variable types and method contracts [20], in particular type inconsistencies and violations of method preconditions.

As a starting point, we have defined a subset of the computer algebra language Maple called *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* [9] developed at our institute for the computation of bivariate difference-differential dimension polynomials. Furthermore, we have defined a specification language to formally specify the behavior of *MiniMaple* procedures. As the next step, we will develop a tool to automatically detect errors in *MiniMaple* programs with respect to their specifications.

Figure 1 gives a sketch of the final system (the verifier component is to be developed); any *MiniMaple* program is parsed to generate an abstract syntax tree (AST). The AST is then annotated by type information and used by the verifier to check the correctness of a program. Error and information messages are generated by the respective components.

There are various computer algebra languages, Mathematica and Maple being the most widely used by far [24], both of which are dynamically typed. We have chosen in our work Maple for the following reasons:

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

During our study, we found the following special features for type checking 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.

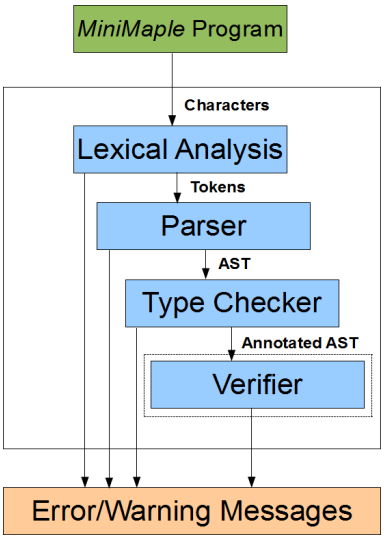

Figure 1: Sketch of the System

- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-checks 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.
- Maple values are organized in a kind of polymorphic type system [6] with a sub-typing relationship such that we can assign a value to different types. This also makes type inference more complex.

The rest of the paper is organized as follows: in Section II, we describe state of the art related to our work. In Section III, we introduce the syntax for *MiniMaple* by an example. In Section IV, we explain our type system for *MiniMaple*. In Section V, we discuss our formal specification language for *MiniMaple*. Section VI presents conclusions and future work.

2 State of the art

In this section we first sketch state of the art of type systems for Maple and then discuss the application of formal techniques to computer algebra languages.

Although there is no complete static type system for Maple; there have been several approaches to exploit the type information in Maple for various purposes. For instance, the Maple package Gauss [21] introduced parameterized types in Maple. Gauss run on top of Maple and allowed to implement generic algorithms

in Maple in an AXIOM-like manner. The system supported parameterized types and parameterized abstract types, however these were only checked at runtime. The package was introduced in Maple V Release 2 and later evolved into the *domains* package [1].

In [8], partial evaluation [25] is applied to Maple. The focus of the work is to exploit the available type information for generating specialized programs from generic Maple programs. The language of the partial evaluator has similar syntactic constructs (but fewer expressions) as *MiniMaple* and supports very limited types e.g. integers, rationals, floats and strings.

In comparison to the approaches discussed above, *MiniMaple* uses the type annotations provided by Maple for static analysis. It supports a substantial subset of Maple types in addition to named types.

Various specification languages have been defined to formally specify the behavior of programs written in standard classical programming languages, e.g. Java Modeling Language [13] for Java, Spec# [3] for C# and ACSL [5] for ANSI C: these specification languages are used by various tools for extended static checking [10] and verification [15, 11] of programs written in the corresponding languages.

Also variously the application of formal methods to computer algebra has been investigated. For example [12] applied the formal specification language Larch [14] to the computer algebra system AXIOM respective its programming language Aldor. A methodology for Aldor program analysis and verification was devised by defining abstract specifications for AXIOM primitives and then providing an interface between these specifications and Aldor code. The project FoCaLiZe [26] aims to provide a programming environment for computer algebra to develop certified programs to achieve high levels of software security. The environment is based on functional programming language FoCal [22], which also supports some object-oriented features and allows the programmer to write formal specifications and proofs of programs.

The work presented in [7] aims at finding a mathematical description of the interfaces between Maple routines. The paper mainly presents the study of the actual contracts in use by Maple routines. The contracts are statements with certain (static and dynamic) logical properties. The work focused to collect requirements for the pure type inference engine for existing Maple routines. The work was extended to develop the partial evaluator for Maple mentioned above [8]. The problem of statically type-checking *MiniMaple* programs is related to the problem of statically type-checking scripting languages such as Ruby [16], but there are also fundamental differences due to the different language paradigms.

The specification language for *MiniMaple* fundamentally differs from those for classical languages such that it supports some non-standard types of objects, e.g. symbols, unevaluated expressions and polynomials etc. The language also supports abstract data types, while the existing specification languages are weaker in such specifications. In contrast to the computer algebra specification languages above, our specification language is defined for the commercially supported language Maple, which is widely used but was not designed to support

static analysis (type checking respectively verification). The challenge here is to overcome those particularities of the language that hinder static analysis.

3 *MiniMaple*

MiniMaple is a simple but substantial subset of Maple that covers all the syntactic domains of Maple but has fewer alternatives in each domain than Maple; in particular, Maple has many expressions which are not supported in our language. The complete syntactic definition of *MiniMaple* is given in [18]. The grammar of *MiniMaple* has been formally specified in BNF from which a parser for the language has been automatically generated with the help of the parser generator ANTLR.

The top level syntax for *MiniMaple* is as follows:

Prog := *Cseq*;
Cseq := EMPTY | *C,Cseq*
C := ... | *I,Iseq* := *E,Eseq* | ...

A program is a sequence of commands, there is no separation between declaration and assignment.

Listing 1 gives an example of a *MiniMaple* program which we will use in the following sections for the discussion of type checking and behavioral specification. The program consists of an assignment initializing a global variable *status* and an assignment defining a procedure *prod* followed by the application of the procedure. The procedure takes a list of integers and floats and computes the product of these integers and floats separately; it returns as a result a tuple of the products. The procedure may also terminate prematurely for certain inputs, i.e. either for an integer value 0 or for a float value less than 0.5 in the list; in this case the procedure computes the respective products just before the index at which the aforementioned terminating input occurs.

In the procedure header, the variable *status* has no type information at the "global" declaration, because global variables cannot have type information in Maple. In the body of the procedure, the local variables *x*, *si* and *sf* are annotated with the types. In the body of the loop, the variable *x* can be of type **integer** or **float** depending on the type of the selected element of the list; respectively after the body of the loop *x* can have any of the two types (**integer** or **float**) depending upon the execution of the corresponding if-else branch.

As one can see from the example, we make use of the type annotations that Maple introduced for runtime type checking. In particular, we demand that function parameters, function results and local variables are correspondingly type annotated. Based on these annotations, we define

- a language of types and
- a corresponding type system

for the static type checking of *MiniMaple* programs.

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

Listing 1: An example *MiniMaple* program

4 A Type System for *MiniMaple*

A *type* is (an upper bound on) the range of values of a variable. A *type system* is a set of formal typing rules to determine the variables types from the text of a program. A type system prevents *forbidden errors* during the execution of a program [6]. It completely prevents the *untrapped errors* and also a large class of *trapped errors*. *Untrapped errors* may go unnoticed for a while and later cause an arbitrary behavior during execution of a program, while *trapped errors* immediately stop execution.

A type system is a decidable logic with various kinds of *judgments*; for example the typing judgment

$$\pi \vdash E{:}(\tau)\mathbf{exp}$$

can be read as "in the given type environment π, E is a well-typed expression of type τ". A type system is *sound*, if the deduced types indeed capture the program values exhibited at runtime.

In the following we describe the main properties of a type system for *MiniMaple*. Subsection A sketches its design, subsections B and C discuss its formal definition and subsection D presents its implementation and application. A proof of the soundness of the type system is still open.

4.1 Design

MiniMaple uses Maple type annotations for static type checking, which gives rise to the following language of types:

$$
\begin{aligned}
T ::= & \ \textbf{integer} \mid \textbf{boolean} \mid \textbf{string} \mid \textbf{float} \mid \textbf{rational} \mid \textbf{anything} \\
& \mid \textbf{\{}\ T\ \textbf{\}} \mid \textbf{list(}\ T\ \textbf{)} \mid [\ \mathit{Tseq}\ ] \mid \textbf{procedure}[\ T\ ](\ \mathit{Tseq}\ \textbf{)} \\
& \mid I(\ \mathit{Tseq}\ \textbf{)} \mid \textbf{Or(}\ \mathit{Tseq}\ \textbf{)} \mid \textbf{symbol} \mid \textbf{void} \mid \textbf{uneval} \mid I
\end{aligned}
$$

The language supports the usual concrete data types, sets of values of type T ($\{\ T\ \}$), lists of values of type T (**list**($\ T\ $)) and records whose members have the values of types denoted by a type sequence *Tseq* ([*Tseq*]). Type **anything** is the super-type of all types. Type **Or**(*Tseq*) denotes the union type of various types, type **uneval** denotes the values of unevaluated expressions, e.g. polynomials and type **symbol** is a name that stands for itself if no value has been assigned to it. User-defined data types are referred by I while I(*Tseq*) denotes tuples (of values of types *Tseq*) tagged by a name I.

A sub-typing relation ($<$) is defined among types, i.e. **integer** $<$ **rational** $<$ ... $<$ **anything**, such that **integer** is sub-type of **rational** and **anything** is the super-type of all types.

In the following, we demonstrate the problems arising from type checking *MiniMaple* programs using the example presented in the previous section.

4.1.1 Global Variables

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.

4.1.2 Type Tests

Maple supports type tests (**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 and as a result we combine the types of the variable from all the branches of a conditional. Our type system also allows negation of a type test in conditional. Negation is only possible if the identifier

has a union type, i.e. Or(Tseq) as declared type, as a result we subtract the (negated) type that appears in type test from the declared type.

For our example program our type system (described in the following) will generate the type information depicted in Listing 2.

```
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];
9.              status:=i;
10.             # π={..., i:integer, ..., status:integer}
11.             if type(x,integer) then
12.             # π={..., i:integer, x:integer, si:integer, ..., status:integer}
13.                     if (x = 0) then
14.                         return [si,sf];
15.                     end if;
16.                     si:=si*x;
17.             elif type(x,float) then
18.             # π={..., i:integer, x:float, ..., sf:float, status:integer}
19.                     if (x < 0.5) then
20.                         return [si,sf];
21.                     end if;
22.                     sf:=sf*x;
23.             end if;
24.             # π={..., i:integer, x:Or(integer,float),..., status:integer}
25.         end do;
26.         # π={..., i:symbol, x:Or(integer,float),..., status:anything}
27.         status:=-1;
28.         # π={..., i:symbol, x:Or(integer,float),..., status:integer}
29.         return [si,sf];
30.         end proc;
31. result := prod([1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);
32. print(result);
33. print(status);
```

Listing 2: A *MiniMaple* procedure type-checked

The program is annotated with the type environment (a partial function) of the form $\#\pi = \{variable{:}type,...\}$. For example, the type environment at line *6* shows the types of the respective variables as determined by the static analysis of parameter and identifier declarations (**global** and **local**).

The static analysis of the two branches of the conditional command in the body of the loop introduces the type environments at lines *12* and *18* respectively; the type of variable x is determined as **integer** and **float** by the conditional type-expressions respectively.

There is more type information to direct the program control flow for an identifier x introduced by an expression $\mathbf{type}(I, T)$ at lines *11* and *17*.

By analyzing the conditional command as a whole, the type of variable x is determined as **Or(integer, float)** (at line *24*), i.e. the union type of the two types determined by the respective branches.

The local type information introduced/modified by the analysis of body of loop does not effect the global type information. The type environment at line *6* and *26* reflects this fact for variables *status*, i and x. This is because of the fact that the number of loop iterations might have an effect on the type of the variable otherwise 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 such variables. As a special case this least fixed point is the type of a variable prior to the body of a loop. For example take the following program (which is not correct according to our type system):

```
local x::list(anything), y::list(anything), a::integer, b::integer;
...
while a < b do
i.      x:=y;
        if type(x,list(anything)) then
j.          y:=[x];
        end if;
end do;
```

In this program, the number of loop iterations influence the types of variables x and y at lines i and j respectively. The static analysis of the loop would give the following type information:

- iteration 1, π={x:**list(anything)**, y:**list(list(anything))** ,...}
- iteration 2, π={x:**list(list(anything))**, y:**list(list(list(anything)))**,...}

After each iteration the types of variables x and y grow such that one cannot determine the concrete types by the static analysis of the loop. To handle this non-determination of types we put a reasonable upper bound on the types of variables. This upper bound is the type of a variable prior to the body of a loop. In other words, while analyzing the loop we ignore the new type information introduced by the body of the loop.

4.2 Typing Judgments

In this subsection we explain the typing judgments for some expressions and commands of *MiniMaple*. These judgments use the following kinds of objects ("Identifier" and "Type" are the syntactic domains of identifiers/variables and types of *MiniMaple* respectively):

- π: Identifier $\rightarrow$ Type: a type environment, i.e. a (partial) function from identifiers to types.
- $c \in$ {global, local}: a tag representing the context to check if the corresponding syntactic phrase is type checked inside/outside of the procedure/loop.
- *asgnset* $\subseteq$ Identifier: a set of assignable identifiers introduced by type checking the declarations.

- $\epsilon set \subseteq$ Identifier: a set of thrown exceptions introduced by type checking the corresponding syntactic phrase.
- $\tau set \subseteq$ Type: a set of return types introduced by type checking the corresponding syntactic phrase.
- *rflag* $\in$ {aret, not_aret}: a return flag to check if the last statement of every execution of the corresponding syntactic phrase is a *return* command.

MiniMaple supports various types of expressions but boolean expressions are treated specially because of the test **type**(I,T) that gives additional type information about the expression. The typing judgment for boolean expressions

$$\pi \vdash E{:}(\pi_1)\textbf{boolexp}$$

can be read as "with the given π, E is a well-typed boolean expression with new type environment π_1". The new type environment is produced as a fact of type test that might introduce new type information for an identifier.

The typing judgment for commands

$$\pi,\ c,\ asgnset \vdash C{:}(\pi_1,\ \tau set,\ \epsilon set,\ rflag)\textbf{comm}$$

can be read as "in the given type environment π, context c and an assignable set of identifiers *asgnset*, C is a well-typed command and produces (π_1, τset, ϵset, *rflag*) as type information".

4.3 Typing Rules

In this subsection we explain some typing rules to derive typing judgments for boolean expressions and commands. These typing rules use different kinds of auxiliary functions and predicates as given below.

4.3.1 Auxiliary Functions

- $update(\pi, (Id, Idseq), (\tau, \tau seq))$: updates the type environment with the given identifiers and their types.
- $specialize(\pi_1, \pi_2)$: specializes the identifiers of former type environment to the identifiers in the latter type environment with respect to their corresponding types.
- $combine(\pi_1, \pi_2)$: combines the identifiers in the two environments with respect to their types.
- $override(\pi_1, \pi_2)$: overrides the identifiers of former type environment to those in the latter type environment with respect to their corresponding types.
- $restrict(\pi, idset)$: restricts the type environment to only those variables that are in the given set (of identifiers).
- $superType(\tau_1, \tau_2)$: returns the super-type between the two given types.

4.3.2 Auxiliary Predicates

- *canSpecialize*(π_1, π_2): returns true if all the common identifiers (in both type environments) have a super-type between their corresponding types.
- *matchType*(τ_1, τ_2): returns true (in most cases) if the former type is general (super) type than the latter type. Type **anything** matches every type being the super-type of all types.
- *isNotRepeated*$(I, Iseq)$: returns true if every identifier in the given sequence occurs only once in it.
- *isAssignable*$((I, Iseq), s)$: returns true if all the identifiers in the given sequence are also in the given set of identifiers s.

The typing rule for boolean expressions is as follows:

- **type**(I,T)

$$\frac{\pi \vdash I{:}(\tau_1)\mathbf{id} \quad \pi \vdash T{:}(\tau_2)\mathbf{type} \quad matchType(\tau_1,\tau_2)}{\pi \vdash \mathbf{type}(I,T){:}(\{\mathrm{I}{:}\tau_2\})\mathbf{boolexp}}$$

The phrase "**type**(I,T)" is a well-typed boolean expression if the declared type of identifier (τ_1) is the super-type of T (τ). The boolean expression may introduce new type information for the identifier.

Typing rules for assignment, conditional and loop commands are given below.

- *I,Iseq* := *E,Eseq*

For local context

$$\frac{\begin{array}{c}\pi \vdash I{:}(\tau_1)\mathbf{id} \quad \pi \vdash Iseq{:}(\tau seq_1)\mathbf{idseq} \quad isNotRepeated(I,Iseq) \\ \pi \vdash E{:}(\tau_2)\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq_2)\mathbf{expseq} \\ isAssignable((I,Iseq),\ asgnset) \quad matchTypeSeq((\tau_1,\tau seq_1),(\tau_2,\tau seq_2))\end{array}}{\begin{array}{c}\pi,\ local,\ asgnset \vdash I,Iseq := E,Eseq \\ :(update(\pi,(I,Iseq),(\tau_2,\tau seq_2)),\{\},\{\},not\_aret)\mathbf{comm}\end{array}}$$

The phrase "*I,Iseq* := *E,Eseq*" is a well typed assignment command in a local context only if the types of expressions (E and *Eseq*) are the subtypes of the declared identifiers types (I and *Iseq*). The assignment command updates the type environment with identifiers and their corresponding subtypes.

For global context

$$\frac{\begin{array}{c}\mathit{isNotRepeated(I,Iseq)}\\ \pi \vdash E{:}(\tau)\mathbf{exp}\\ \pi \vdash \mathit{Eseq}{:}(\tau seq)\mathbf{expseq}\end{array}}{\begin{array}{c}\pi,\ \mathit{global},\ \mathit{asgnset} \vdash \mathit{I,Iseq} :=\\ \mathit{E,Eseq}{:}(\mathit{update}(\pi,(\mathit{I,Iseq}),(\tau,\tau seq)),\{\},\{\},\mathit{not\_aret})\mathbf{comm}\end{array}}$$

The phrase "*I,Iseq* := *E,Eseq*" is a well typed assignment command in a global context; it allows to change the types of identifiers arbitrarily.

- **if** *E* **then** *Cseq Elif* **end if**

$$\frac{\begin{array}{c}\pi \vdash E{:}\ (\pi')\mathbf{boolexp} \quad \mathit{canSpecialize}(\pi,\pi')\\ \mathit{specialize}(\pi,\pi'),\ c,\ \mathit{asgnset} \vdash \mathit{Cseq}{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{cseq}\\ \pi,\ c,\ \mathit{asgnset} \vdash \mathit{Elif}{:}(\pi_2,\pi set,\ \tau set_2,\epsilon set_2,rflag_2)\mathbf{elif}\end{array}}{\begin{array}{c}\pi,\ c,\ \mathit{asgnset} \vdash \mathbf{if}\ E\ \mathbf{then}\ \mathit{Cseq}\ \text{Elif}\ \mathbf{end}\\ \mathbf{if}{:}(\mathit{combine}(\pi_1,\pi_2),\tau set_1 \cup \tau set_2,\epsilon set_1 \cup \epsilon set_2,\text{ret}(rflag_1,\ rflag_2))\mathbf{comm}\end{array}}$$

The phrase "**if** *E* **then** *Cseq Elif* **end if**" is a well typed conditional command if the type of expression *E* does not conflict global type information. The conditional command combines the type environment of its two conditional branches (*if* and *elif*), because we are not sure which of the branch will be executed at runtime.

- **for** *I* **from** E_1 **by** E_2 **to** E_3 **do** *Cseq* **end do**

$$\frac{\begin{array}{c}\pi \vdash E_1{:}\ (\mathbf{integer})\mathbf{exp} \quad \pi \vdash E_2{:}\ (\mathbf{integer})\mathbf{exp}\\ \pi \vdash E_3{:}\ (\mathbf{integer})\mathbf{exp}\\ \mathit{override}(\pi,\{I{:}\mathbf{integer}\}),\ \mathit{local},\ \mathit{asgnset}\\ \vdash \mathit{Cseq}{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{cseq}\end{array}}{\begin{array}{c}\pi,\ c,\ \mathit{asgnset} \vdash \mathbf{for}\ I\ \mathbf{from}\ E_1\ \mathbf{by}\ E_2\ \mathbf{to}\ E_3\ \mathbf{do}\ \mathit{Cseq}\ \mathbf{end}\\ \mathbf{do}{:}(\pi,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm}\end{array}}$$

The phrase "**for** *I* **from** E_1 **by** E_2 **to** E_3 **do** *Cseq* **end do**" is a well typed loop command if it does not influence the global type information. The loop command leaves the input type environment π unchanged because otherwise number of loop iterations might influence the type information.

4.4 Application

Based on the type system sketched above we have implemented a type checker for *MiniMaple* [17, 18] in Java (150+ classes and 15K+ lines of code).

Figure 2 shows that the output of the type checker applied to a file containing the source code of the example program from the previous section. It shows that the file has successfully parsed and also presents the type annotations for the first assignment command. In the second part, it shows the resulting type

environment with the associated program identifiers and their respective types introduced while type checking. The last message indicates that the program type checked correctly.

```
/home/taimoor/antlr3/Test6.m parsed with no errors.
Generating Annotated AST...
#commseq#
#comm#
#asgncomm#
#expression#
#idexp#
status
:=
#expression#
#numexp#
0

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END***********
;
GLOBAL CONTEXT FOR ASSIGNMENT-COMMAND

**********|ASSIGNMENT| COMMAND-ANNOTATION START**********
PI -> [
status:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
**********|ASSIGNMENT| COMMAND-ANNOTATION END***********

...

**********COMMAND-SEQUENCE-ANNOTATION START**********
PI -> [
prod:procedure[[integer,float]](list(Or(integer,float)))
status:integer
result:[integer,float]
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
**********COMMAND-SEQUENCE-ANNOTATION END***********

Annotated AST generated.
The program type-checked correctly.
```

Figure 2: Parsing and Type Checking the Program

The main test case for our type checker is the Maple package *Difference-Differential* [9] developed by Christian Dönch at our institute. The package provides algorithms for computing difference-differential dimension polynomials by relative Gröbner bases in difference-differential modules according to the method developed by M. Zhou and F. Winkler [27].

We manually translated this package into a *MiniMaple* package so that the type checker can be applied. This translation consists of

- adding required type annotations and

- translating those parts of the package that are not directly supported into logically equivalent *MiniMaple* constructs.

No crucial typing errors have been found but some bad code parts have been identified that can cause problems. These code parts refer to those variables

- that are declared but not used therefore cannot be type checked and
- that have duplicate declarations in global and local declarations.

5 A Formal 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 *MiniMaple* notations but also has its own notations. This specification language will be used as described in the conclusions.

Formula Language

In general the formula language consists of basic logical formulas/expressions but also supports logical (existential and universal), numerical (**add**, **mul**, **min** and **max**) and sequence (**seq**) quantifiers representing truth values, numeric values and sequence of values respectively. We have extended the Maple syntax for these quantifiers, e.g. logical quantifiers are quantified over typed variables. Furthermore the numerical quantifiers are extended so that they can be applied over filtered (logical condition) values of their corresponding specified range of quantified variables. The example for these quantifiers is explained later in the procedure specification of this section.

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 consists of the elements given below.

5.1 Specification Declarations

At the top of *MiniMaple* program we can declare respectively define mathematical functions, user-defined named and abstract data types and axioms. The syntax of specification declarations

decl ::= EMPTY
| (**define**(*I,rules*);
| **‘type/**I‘:=T; | **‘type/**I‘;
| **assume**(*spec-expr*); | I(*spec-expr*);) *decl*

is mainly borrowed from Maple. The phrase “**define**(*I,rules*);“ can be used for defining mathematical functions as shown in the following definition of the factorial function:

define(fac, fac(0) = 1, fac(n::integer) = n * fac(n -1));

User-defined data types can be declared with the phrase "**‘type/**I‘:=T;“ as shown in the following declaration of "ListInt" as the list of integers:

‘type/ListInt‘:=**list**(**integer**);

The phrase "**‘type/**I‘;" can be used to declare abstract data type with the name I, e.g. the following example shows the declaration of abstract data type “stack”.

‘type/stack‘;

Axioms can be introduced by the phrase “**assume**(*spec-expr*);“ as the following example shows an axiom for the computation of gcd of two integers, where n is a procedure parameter:

assume(**forall**(a::**integer**, b::**integer**,
1<=a **and** a<=n **and** 1<=b **and** b<=n
implies gcd(a,b) = gcd(b, a **mod** b)));

The entities introduced by the specification declarations can be used in the following specifications.

5.2 Procedure Specification

The purpose of a procedure specification is to constrain the behavior of a procedure. A procedure specification consists of a pre-condition, the set of global variables that can be modified and the post condition, describing the relationship between pre and post state. By an optional exception clause we can specify the exceptional behavior of a procedure. The procedure specification syntax is influenced by the Java Modeling Language:

proc-spec ::= **requires** *spec-expr*;
global *Iseq*;
ensures *spec-expr*; *excep-clause*

Listing 3 shows an example for the procedure specification. The specification is a big logical disjunction to formulate two possible behaviors of the procedure:

1. when the procedure terminates normally and
2. when the procedure terminates prematurely.

```
(*@
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 3: A *MiniMaple* procedure formally specified

The listing gives a formal specification of the example procedure introduced in Section III. 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).

From the example one can also notice the application of numerical quantifier **mul**. The quantifier multiplies only those elements of the input array l that satisfy the test **type**(e,**integer**).

5.3 Loop Specification

Loops can be specified by invariants and termination terms denoting non-negative integers as follows:

loop-spec := **invariant** *spec-expr*; **decreases** *spec-expr*;

An Invariant presents partial correctness of a loop and it must hold after each iteration of the loop, even if the loop does not execute at all. A termination term is added to specify the total correctness of the loops, it guarantees that after each iteration the value of the termination term decreases and respectively terminates.

The following example specifies the loop that iterates over integers from 1...100 respectively computes the sum.

```
i := 1; s := 0; n := 100;
while (i <= n) do{
(*@invariant s = OLD s + i - 1; decreases n-i;@*)
s := s + i; i := i + 1;
}
```

From the example one can see the relationship between the loop variables that holds after every iteration and that the value of the termination term decreases after every iteration.

Loop specifications help in reasoning about loops, i.e. about partial correctness [19] (invariants) and total correctness [19] (termination term).

5.4 Assertions

An assertion constrains the state of the execution at the point where it occurs. Furthermore, an assertion splits the verification proof into two parts,

1. a proof obligation and
2. an assumption for the rest of the proof.

Assertions have Maple borrowed syntax as given:

asrt := **ASSERT**(*spec-expr*, (EMPTY | "*I*"));

An assertion can be a logical formula or a named assertion. The following example shows a named assertion ("test failed").

x := 1; y := x; x := x + y;
ASSERT(**type**(y,**integer**), "test failed");

6 Conclusions and Future Work

In this paper we gave an overview of a substantial subset of Maple called *MiniMaple*. We have defined a formal type system which makes use of Maple annotations for the static (compile-time) type checking. We have implemented the corresponding type checker and applied it to a Maple package developed at our institute. As a result some problematic code parts were identified.

We also presented our initial work on a formal specification language for *MiniMaple* that can be used to specify the behavior of *MiniMaple* programs. 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 investigated two possibilities:

1. We may directly generate verification conditions and use Satisfiability Modulo Theories (SMT) [2] solvers or interactive theorem provers to prove their correctness. The former are fully automated semi-decision procedures that may give "don't know" results. The later require human assistance by adding additional information (loop invariants) and guiding the correctness proof.

2. We may use some existing framework to generate verification conditions and prove the correctness, e.g. by the Boogie [4] framework developed by Microsoft. Here we need to translate our specification annotated *MiniMaple* program into an intermediate language of Boogie and then use the various proving back-ends of Boogie.

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

Acknowledgment

The author cordially thanks Professor Wolfgang Schreiner for his valuable and constructive comments and suggestions.

References

[1] Maple Domain Package. `http://www.maplesoft.com/support/help/view.aspx?sid=47395`.

[2] Satisfiability Modulo Theories. `http://goedel.cs.uiowa.edu/smtlib/`.

[3] 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 Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[4] 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 Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[5] 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`.

[6] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[7] Jacques Carette and Stephen Forrest. Mining Maple Code for Contracts. In Silvio Ranise and Anna Bigatti, editors, *Calculemus*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006. `http://www.cas.mcmaster.ca/~carette/publications/MapleContracts.pdf`.

[8] Jacques Carette and Michael Kucera. Partial Evaluation of Maple. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and*

semantics-based program manipulation, PEPM '07, pages 41–50. ACM Press, 2007.

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

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. In *SRC Research Report 159, Compaq Systems Research Center*, 1998. `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-159.pdf`.

[11] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, June 2008.

[12] Martin Dunstan, Tom Kelsey, Steve Linton, and Ursula Martin. Lightweight Formal Methods For Computer Algebra Systems. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '98*, pages 80–87. ACM Press, 1998.

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

[14] J. V. Guttag, J. J. Horning, Withs. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and Tools for Formal Specification. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1993.

[15] Gerard J. Holzmann. Trends in Software Verification. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, Lecture Notes in Computer Science, pages 40–50. Springer, 2003.

[16] 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.

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

[18] 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.

[19] A. K. McIver and C. Morgan. Partial Correctness for Probabilistic Demonic Programs. *Theoretical Computer Science*, 266(1-2):513–541, 2001.

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

[21] Michael B. Monagan. Gauss: A Parameterized Domain of Computation System with Support for Signature Functions. In *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, DISCO '93, pages 81–94. Springer-Verlag, 1993.

[22] Virgile Prevosto. Certified Mathematical Hierarchies: the FoCal System. In Thierry Coquand and Henri Lombardi and Marie-Françoise Roy, editor, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

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

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

[25] Eijiro Sumii and Naoki Kobayashi. Online Type-Directed Partial Evaluation for Dynamically-Typed Languages. *Computer Software*, 17:38–62, 1999.

[26] Sylvain Boulmé and Thérèse Hardin and Daniel Hirschkoff and Valérie Ménissier-Morain and Renaud Rioboo. On the Way to Certify Computer Algebra Systems. In *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conference, Trento, Italie)*, volume 23 of *ENTCS*, pages 370–385. Elsevier, 1999.

[27] Meng Zhou and Franz Winkler. Computing Difference-Differential Dimension Polynomials by Relative Gröbner Bases in Difference-Differential Modules. *Journal of Symbolic Computation*, 43(10):726–745, October 2008.