Towards the Formal Specification and Verification 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. In this paper, we present our ongoing work and initial results on the formal specification and verification of *MiniMaple* (a substantial subset of Maple with slight extensions) programs. The main goal of our work is to find behavioral errors in such programs w.r.t. their specifications by static analysis. This task is more complex for widely used computer algebra languages like Maple as these are fundamentally different from classical languages: they support non-standard types of objects such as symbols, unevaluated expressions and polynomials and require abstract computer algebraic concepts and objects such as rings and orderings etc. As a starting point we have defined and formalized a syntax, semantics, type system and specification language for *MiniMaple*.

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 [10], C# [1] and C [2]; we aim to apply similar techniques to computer algebra languages, i.e. to design and develop a tool for the static analysis of computer algebra programs. This tool will find errors in programs annotated with extra information such as variable types and method contracts, in particular type inconsistencies and violations of methods preconditions.

As a starting point, we have defined the syntax and semantics of 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* [7] developed at

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

J. Jeuring et al. (Eds.): CICM 2012, LNAI 7362, pp. 231–247, 2012.

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* programs. As the next step, we will develop a verification calculus for *MiniMaple* respectively a corresponding tool to automatically detect errors in *MiniMaple* programs with respect to their specifications. An example-based short demonstration of this work is presented in the accompanying paper [16].

Figure 1 gives a sketch of the envisioned system (the verifier component is under development); 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.

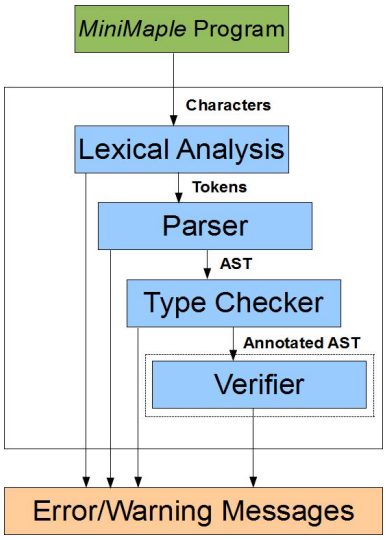

Fig. 1. Sketch of the System

There are various computer algebra languages, Mathematica and Maple being the most widely used by far, both of which are dynamically typed. We have chosen for our work Maple because of its simpler, more classical and imperative structure. Still we expect that the results we derive for type checking respective formal specification Maple can be applied to Mathematica, as Mathematica shares with Maple many concepts such as basic kinds of runtime objects.

During our study, we found the following special features for type checking respective formal specification of 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.
- 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.
- The language lacks in the use of abstract data types, which are necessary for the adequate specification of computer algebra functions.

The rest of the paper is organized as follows: in Section 2, we describe state of the art related to our work. In Section 3, we introduce the syntax of *MiniMaple* by an example. In Section 4, we briefly explain our type system for *MiniMaple*. In Section 5, we discuss our formal specification language for *MiniMaple*. In Section 6, we highlight the interesting features of a formal semantics of *MiniMaple*. Section 7 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 [17] introduced parameterized types in Maple. Gauss ran 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. In [6], partial evaluation 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. The problem of statically type-checking *MiniMaple* programs is related to the problem of statically type-checking scripting languages such as Ruby [13], but there are also fundamental differences due to the different language paradigms.

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 (JML) [10] for Java, Spec# [1] for C# and ACSL [2] for ANSI C: these specification languages are used by various tools for extended static checking and verification [8] of programs written in the corresponding languages. Also variously the application of formal methods to computer algebra has been investigated. For example [9] applied the formal specification language Larch [11] 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 [20] 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, which also supports some object-oriented features and allows the programmer to write formal specifications and proofs of programs. The work presented in [5] 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 [6].

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 to formalize abstract mathematical concepts, 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 [14]. 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.

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

Listing 1. An example *MiniMaple* program

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.

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.

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. 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 [4].

A type system in essence 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 4.1 sketches its design and Subsection 4.2 presents its implementation and application. A proof of the soundness of the type system remains to be performed.

```
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;
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 return [si,sf]; 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 return [si,sf]; 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]);
```

Listing 2. A *MiniMaple* procedure type-checked

4.1 Design

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

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

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 a sub-type of **rational** and the type **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.

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 types of variables.

Type Tests. 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 "then" branch is guarded by a test type(x,integer), in the "else" branch x has automatically type float. A warning is generated, if a test is redundant (always yields true or false).

For our example program our type system will generate the type information as depicted in Listing 2. The program is annotated with the type environment (a partial function from identifiers to their corresponding types) 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 **type**(E,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 lines *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 (fixed point) on the types of such variables, namely the type of a variable prior to the body of a loop.

4.2 Formalization

In this subsection we explain the typing judgments and typing rules 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

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

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

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

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, τ*set*, ϵ*set*, *rflag*) as type information". In the following we explain some typing rules to derive typing judgments for boolean expressions and conditional commands. These typing rules use different kinds of auxiliary functions and predicates as given below.

Auxiliary Functions

- *specialize*(π_1, π_2): specializes the identifiers of former type environment to the identifiers in the latter type environment w.r.t. their types.
- *combine*(π_1, π_2): combines the identifiers in the two environments with respect to their types.
- *superType*(τ_1, τ_2): returns the super-type between the two given types.

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.
- *superType*(τ_1, τ_2): returns true (in most cases) if the former type is general (super) type than the latter type. **Anything** is the super-type of all types.

Typing Rules. The typing rule for boolean expressions is as follows:

- **type**(I, T)

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

  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.

The typing rule for the conditional command is given below:

– **if** E **then** *Cseq Elif* **end if**

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

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 branches will be executed at runtime.

4.3 Application

Based on the type system sketched above we have implemented a type checker for *MiniMaple* [14] in Java (150+ classes and 15K+ lines of code). The type checker also handles the specification language of *MiniMaple*.

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...
...
**********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.
```

Fig. 2. Parsing and Type Checking the Program

The main test case for our type checker is the Maple package *Difference-Differential* [7] 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 [22].

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, e.g., variables that are declared but not used (and therefore cannot be type checked) and variables that have duplicate global and local declarations.

5 A Formal 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. The example for these quantifiers is explained later in the procedure specification of this section. The use of this specification language is described in the conclusions.

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 JML, abstract data types can be introduced to specify abstract concepts and notions from computer algebra.

At the top of *MiniMaple* program one 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*);) *decl*

is mainly borrowed from Maple. The phrase “**define**(*I,rules*);“ can be used for defining mathematical functions as shown in the following 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 "difference differential operator (DDO)".

‘type/DDO‘;

The task of formally specifying mathematical concepts using abstract data types is more simpler as compared to their underlying representation with concrete data types. Also other related facts and the access functions of abstract concept can be formalized for better reasoning.

Axioms can be introduced by the phrase "**assume**(*spec-expr*);" as the following example shows an axiom that an operator is a difference-differential operator, if its each term is a difference-differential term, where d is an operator:

```
assume(isDDO(d) equivalent forall(i::integer, 1<=i and i<=terms(d) implies
        isDDOTerm(getTerm(d,i,1),getTerm(d,i,2),
                  getTerm(d,i,3),getTerm(d,i,4));
```

Any predicate declaration can be introduced by the phrase "*I*(*spec-expr*);" as the following example shows a predicate that when a given field is supported:

```
inField(c);
```

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

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 3. 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; if the 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**).

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

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

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 (invariants) and total correctness (termination term).

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");

The implemented type checker also checks the correct typing of the formal specifications. We have used the specification language to formally specify parts of *DifferenceDifferential*. While the specifications are not yet formally checked, they demonstrate the adequacy of the language for the intended purpose.

6 Formal Semantics of *MiniMaple*

We have defined a formal denotational semantics of *MiniMaple* programs as a pre-requisite of a verification calculus which we are currently developing: the verification conditions generated by the verification calculus must be sound with respect to the semantics. There is no formally defined semantics for Maple and only the implementation of Maple can be considered as a basis for our work. However, our semantics definition attempts to depict the internal behavior of Maple. Based on this semantics, now we can ask the question about the correct behavior of any *MiniMaple* program. The complete definition of a formal semantics of *MiniMaple* is given in [15]. Its core features are as follows:

- *MiniMaple* has expressions with side-effects, which is not supported in functional programming languages like Haskel [12] and Miranda [21]. As a result the evaluation of an expression may change the state. The formal semantics of expression evaluation and command execution is therefore defined as a state relationship between pre- and post-states. A formal denotational semantics is defined as a state relationship is easier to integrate with non-uniquely specified procedures as compared to the function-based semantics definition [19].
- Semantic domains of values have some non-standard types of objects, for example symbol, uneval and union etc. *MiniMaple* also supports additional functions and predicates, for example type tests i.e. $\mathbf{type}(E, T)$, which are correspondingly modeled in semantics algebras.
- In *MiniMaple* a procedure is introduced by an assignment command, e.g. I := **proc**() ... **end proc**, such that assignments take the role of declarations in classical languages. Furthermore, static scoping is used in the definition of a *MiniMaple* procedure.

The denotational semantics is based on *semantic algebra* [18]. For example *Value* is a disjunctive union domain composed of all kinds of primitive semantic values (domains) supported in *MiniMaple*. It also defines some interesting domains i.e. *Module*, *Procedure*, *Uneval* and *Symbol*. The domain *Value* is a recursive domain, e.g. *List* is defined by *Value** as follows:

$List = Value*$
$Value = ...|List|...$

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meaning (semantic algebras) [18]. A valuation function D for a syntax domain D is usually formalized by a set of equations, one per alternative in the corresponding BNF rule for the *MiniMaple* syntactic domain. As the formal semantics of *MiniMaple* is defined as a state relationship so we define the result of valuation function as a predicate. For example the state relation (*StateRelation*) is defined as a power set of pair of pre- and post-states as follows:

$$StateRelation := \mathbb{P}(State \times StateU)$$

The valuation function for command sequence takes the abstract syntax of command sequence, a value of type Cseq and type environment *Environment* and results in a *StateRelation* as follows:

$$[\![\ \text{Cseq}\ ]\!] : Environment \rightarrow StateRelation$$

The denotational semantics of *MiniMaple* while-loop is defined as a relationship between a pre-state s and post-state s' as follows:

$$\begin{aligned}
&[\![\ \textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end do}\ ]\!](\text{e})(\text{s,s'}) \Leftrightarrow \\
&\quad \exists\ k \in Nat', t, u \in StateU* : t(1) = inStataU(s) \wedge u(1) = inStateU(s) \wedge \\
&\quad (\forall i \in Nat'_k : iterate(i, t, u, e, [\![E]\!], [\![Cseq]\!])) \wedge \\
&\qquad ((u(k) = inError() \wedge s' = u(k)) \vee \\
&\qquad (returns(data(inState(u(k)))) \wedge s' = t(k)) \vee \\
&\qquad (\exists\ v \in ValueU : [\![E]\!](e)(inState(t(k)), u(k), v) \\
&\qquad\quad \wedge v <> inValue(inBoolean(True)) \wedge \\
&\qquad\quad \text{IF } v = inValue(inBoolean(False)) \text{ THEN} \\
&\qquad\qquad \text{s'=t(k)} \\
&\qquad\quad \text{ELSE } s' = inError() \text{ END} \\
&\qquad) \\
&\quad)
\end{aligned}$$

The corresponding *iterate* predicate formalizes the aforementioned while-loop semantics. For the complete list of semantic algebras, domains and valuation functions, please see [15].

7 Conclusions and Future Work

In this paper we gave an overview of *MiniMaple* and its formal type system. We plan to automatically infer types as a future goal. Also we presented our initial work on a formal specification language for *MiniMaple* that can be used to specify the behavior of *MiniMaple* programs. As a main test case we have used our specification language to formally specify various abstract computer algebraic concepts used in the Maple package *DifferenceDifferential*, e.g. difference-differential operator and terms and various related access functions. 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 methods preconditions. For this purpose, based on the results of a prior investigation we intend to use the verification framework Why3 [3] to implement the verification calculus for *MiniMaple* as depicted in Fig. 3.

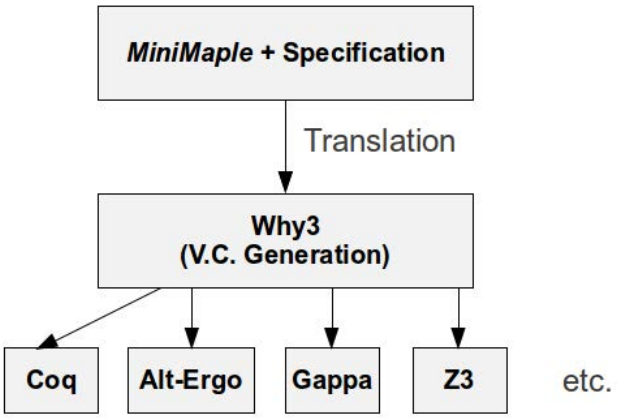

Fig. 3. Verification Calculus for *MiniMaple*

As one can see in the figure, here we need to translate our specification-annotated *MiniMaple* program into the intermediate language of Why3 and then use the various proving back-ends of Why3. Currently we are working on this translation.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C Specification Language (preliminary design V1.2), preliminary edn. (May 2008)
3. 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)
4. Cardelli, L.: Type Systems. In: Tucker, A.B. (ed.) The Computer Science and Engineering Handbook, pp. 2208–2236. CRC Press (1997)
5. Carette, J., Forrest, S.: Mining Maple Code for Contracts. In: Ranise, S., Bigatti, A. (eds.) Calculemus. Electronic Notes in Theoretical Computer Science. Elsevier (2006)

6. Carette, J., Kucera, M.: Partial Evaluation of Maple. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2007, pp. 41–50. ACM Press (2007)
7. Dönch, C.: Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2009)
8. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
9. Dunstan, M., Kelsey, T., Linton, S., Martin, U.: Lightweight Formal Methods For Computer Algebra Systems. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 1998, pp. 80–87. ACM Press (1998)
10. Leavens, G.T., Cheon, Y.: Design by Contract with JML. A Tutorial (2006), `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`
11. Guttag, J.V., Horning, J.J., Garl, W.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer (1993)
12. Hudak, P.: The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press (June 2000)
13. Foster, J.S., Furr, M., An, J.-H., Hicks, M.: Static Type Inference for Ruby. In: Proceedings of the 24th Annual ACM Symposium on Applied Computing, OOPS Track, Honolulu, HI (2009)
14. 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)
15. Khan, M.T.: Formal Semantics of MiniMaple. DK Technical Report 2012-01, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (January 2012)
16. Khan, M.T., Schreiner, W.: On Formal Specification of Maple Programs. In: Conferences on Intelligent Computer Mathematics, Systems and Projects Track (submitted, 2012)
17. Monagan, M.B.: Gauss: A Parameterized Domain of Computation System with Support for Signature Functions. In: Miola, A. (ed.) DISCO 1993. LNCS, vol. 722, pp. 81–94. Springer, Heidelberg (1993)
18. Schmidt, D.A.: Denotational Semantics: a methodology for language development. William C. Brown Publishers, Dubuque (1986)
19. Schreiner, W.: A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria (September 2008)
20. Boulmé, S., Hardin, T., Hirschkoff, D., Ménissier-Morain, V., Rioboo, R.: On the Way to Certify Computer Algebra Systems. In: Proceedings of the Calculemus Workshop of FLOC 1999 (Federated Logic Conference, Trento, Italie). ENTCS, vol. 23, pp. 370–385. Elsevier (1999)
21. Lambert, T., Lindsay, P., Robinson, K.: Using Miranda as a First Programming Language. Journal of Functional Programming 3(1), 5–34 (1993)
22. Zhou, M., Winkler, F.: Computing Difference-Differential Dimension Polynomials by Relative Gröbner Bases in Difference-Differential Modules. Journal of Symbolic Computation 43(10), 726–745 (2008)