2012 10th International Conference on Frontiers of Information Technology

On the Formal Semantics of *MiniMaple* and its Specification Language

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
Johannes Kepler University
A-4040 Linz, Austria
Email: muhammad.khan@dk-compmath.jku.at

***Abstract*—In this paper, we give a definition of the formal (denotational) semantics of *MiniMaple* (a substantial subset of a widely used computer algebra system Maple with slight modifications) and its specification language. Defining the formal semantics of *MiniMaple* and its specification language is rather a challenging task as these support non-standard types of objects, e.g. symbols, unions and polynomials, and additional predicates and functions, e.g. runtime type tests etc. Furthermore, the specification language also supports logical and sequence quantifiers and numeric quantifiers with a logical condition (property) to quantify a filtered range of values. This semantics is defined as a pre-requisite of a verification calculus which we are currently developing. The verification conditions generated by the verification calculus must be sound w.r.t. defined formal semantics.**

I. Introduction

We report on a complimentary result of a project whose goal is to find behavioral errors (type inconsistencies and violations of methods preconditions) in widely used computer algebra system Maple by static analysis. To an intermediate aim, we have defined a formal semantics of *MiniMaple* annotated programs [3] to formally describe the runtime behavior of such computer algebra programs. Currently we are developing a verification calculus where we need to translate annotated *MiniMaple* into a semantically equivalent Why3 [12] (an intermediate verification framework) program. Also the verification conditions generated by the calculus must be sound with respect to the semantics.

Computer algebra programs written in an annotated *MiniMaple* are semantically more complex than classical and functional programming and specification languages as they are fundamentally different from these languages. As a consequence, the denotational semantics of *MiniMaple* and its annotations which we have developed have the following features:

- The denotational semantics is defined as a state relationship between pre- and post-states.
- *MiniMaple* has expressions with side-effects, which is not supported in functional programming languages, e.g. Haskell [9] and Miranda [10]. As a result the evaluation of an expression may change the program execution state.
- In *MiniMaple*, a procedure is defined by an assignment command, e.g. I := **proc**() ... **end proc**. Furthermore, static scoping [11] is used to evaluate a *MiniMaple* procedure.
- The specification language supports abstract data types to formalize mathematical respectively computer algebra theories and concepts, while the existing specification languages are weaker in such specifications.
- It also supports numeral quantifiers to compute a certain binary operation (**add**, **mul**, **max** and **min**) for the filtered values (for a given property) of a specification expression.
- Also the specification language supports sequence quantifier (**seq**) to represent a sequence of values for a given expression.
- *MiniMaple* and its specification language share various semantic domains of values that have some non-standard types of objects, for example symbol, uneval and union etc. These languages also support additional functions and predicates, for example type tests i.e. **type**(*E,T*). For further details of the formal syntax of *MiniMaple* and its specification language, please see [1], [2].

The rest of the paper is organized as follows: Section 2 gives state of the art of formal semantics of programming and specification languages. In Section 3, we discuss the overview of the denotational semantics of *MiniMaple* and its specification language. Sections 4 defines the semantics of *MiniMaple* and Sections 5 and 6 give the definitions of the semantics of specification expression and annotation languages respectively. Section 7 presents conclusions and future work.

II. State of the art

In this section we first sketch a state of the art of various approaches of defining formal semantics of classical and functional programming languages and specification languages and then discuss their differences to the computer algebra programming and specification languages.

The main approaches of defining formal semantics are operational, denotational and axiomatic [24]. The denotational method is widely used as it defines the meaning of a program as a mathematical function that maps abstract syntax to its semantic value, a denotation. Usually this function is partial because of the fact that some syntactically correct programs may not have defined meanings. Since the denotational method

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

states the semantics of a syntactic phrase in terms of the denotations of its sub-phrases, the corresponding proof is more easier as an associated proof technique is structural induction.

To the best of our knowledge, the semantics of various popular programming languages have been incomplete or ill-defined in the literature. There are few widely used high level programming languages whose semantics are defined, e.g. Standard ML [30], Algol 60 [19], C++ [21], Cobol [23], Pascal [18], [22] and Prolog [29] to name a few. The Scott-Strachey notation [25] of denotational semantics was first formally used while designing Ada [20]. The static semantics (compile-time) were formulated only for the sequential Ada. Also a denotational variety of semantics was used to define the functional programming language Scheme [16]. Papaspyrou defined denotational semantics of C [32] by employing monads and monad transformers as denotations to achieve modularity and readability.

The denotational semantics for *MiniMaple* is different from the above approaches as it has some non-standard semantic domains of values (also common with its specification language), e.g. symbol, union and polynomial etc.; also it supports runtime type tests and has a polymorphic type system. Since *MiniMaple* has expressions with side effects in contrary to functional programming languages so here we define the denotational semantics as a mathematical function being a relationship between pre and post-states to best describe a program behavior. *MiniMaple* semantics are defined as dynamic semantics (runtime-semantics) though it includes static semantics. There is no formally defined semantics of Maple and we consider its current implementation as a basis of our work. Also we have developed satisfactorily complete and accurate semantics.

The semantic domains of specification languages are different from the conventional Scott-Strachey denotational domains. For specification languages, the semantic space can be considered as the world of "theories" and the meaning of a specification being the collection of all its models [27]. However, the relational variant of denotations can be used to define a specification semantics.

A variant of denotational notations is used to define the specification language Meta-IV used in VDM [17]. The goal here was to provide more general abstract syntax; for example declarations in the language do not respect order etc. As an application, an Ada compiler was semantically derived from the VDM denotations. Later the formal semantics of BSI/VDM [26] was defined by starting with all possible models and then restricting this set by examining the abstract syntax to the exact denotations. Formal semantics of Z is based on the notion of "variety" and makes use of typed set theory and relational algebra [28]. In [31] abstract interpretation is used to analyze whether a certain relationship holds between the two semantic interpretations of a Maple program for a particular property. One of the interpretations is used as a standard while the other as its abstract version with a certain property. The goal here was to explore the operational semantics of Maple for certain properties.

In comparison to the above approaches, the semantics space for a specification language of *MiniMaple* specification language has more complex structures, e.g. mathematical theories, procedure and loop specifications etc. In addition to basic expressions, the specification language also supports guarded numeral and sequence quantifiers. We give the semantics of our specification language as a relational variant of denotations to address the complexity of its semantic domains and other non-standard constructs.

Here our main goal is to formalize runtime behavior of *MiniMaple* annotated programs and also to define semantics as a pre-requisite of our verification calculus.

III. Semantics Overview

In this section, we discuss the different elements of denotational semantics, e.g. semantic algebras and valuation functions and then we demonstrate the semantics definition for various syntactic domains of *MiniMaple* and its specification language. For complete semantic domains and their operations, valuation functions and their definitions of *MiniMaple* and its specification language, please see [5], [6] respectively.

A. Semantic Algebras

The definition of a formal semantics is based on a collection of data structures. We use the framework of mathematical domain theory [13] to define these data structures. *Semantic domain* is a fundamental concept of domain theory; it represents a set of elements that share some common properties. Also a semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a *semantic algebra* [14]. The semantics of *MiniMaple* and its specification language makes use of several primitive and compound (disjoint sum and product) domains. In the following we discuss some interesting semantics domains of values for various syntactic domains of *MiniMaple* and its annotations.

1) Environment Values: The domain *Environment* holds the environment values of an annotated *MiniMaple* program and is defined as follows:

$Environment = Context \times Space$
$Context = Identifier \rightarrow EnvValue$
$EnvValue = Variable + Procedure + Function$
$Space = \mathbb{P}(Variable)$

Note here that the domain *Space* models the memory space.

2) State Values: The semantic domain *State* defines the state of the program by a tuple of *Store* and *Data* as follows:

$State = Store \times Data$
$Store = Variable \rightarrow Value$
$Data = Flag \times Exception \times Return$
$Flag = \{execute, exception, return, leave\}$

A *Store* is the most important part of the state and holds for every *Variable* a *Value*, while *Data* stores the control information of a particular state. *Exception* and *Return* give the corresponding values of the *Flag*.

3) Semantic Values: Value is a disjunctive union domain composed of all kinds of primitive semantic values (domains) supported in *MiniMaple*. Some of these domains, *Module*, *Procedure* and *Function* are explained in the next sub-sections. Also note that the domain *Value* is a recursive domain, e.g. *List* is defined by *Value**.

$$\begin{aligned} Value =& Module + Procedure + Function + List + Set \\ &+ ... + Symbol + Uneval + Value^* \end{aligned}$$

4) Module Values: The semantic domain *Module* formulates *MiniMaple* module values and maps identifiers to their corresponding values of the statements.

$$Module = Identifier\_Sequence \to Value^*$$

5) Procedure Values: The semantic domain *Procedure* is defined to represent a *MiniMaple* procedure. It is defined as a predicate of sequence of (parameter) values, pre- and post-states and a return value.

$$Procedure = \mathbb{P}(Value^* \times State \times State_\perp \times Value_\perp)$$

6) Function Values: The semantic domain *Function* defines and formalizes the mathematical functions supported in specification language. A predicate is a special case of mathematical function which returns a boolean value. A *Function* can be defined mathematically as:

$$Function = \bigcup_{n \in \mathbb{N}} Function^n$$

where $Function^n = Value^n \to Value$

i.e. maps a sequence of parameter values to its return value.

7) Unevaluated Values: Any term enclosed with single quotes represents an unevaluated value in *MiniMaple*. Each evaluation operation strips off one level of single quotes. The semantic domain *Uneval* is modeled as a string.

8) Lifted Values: The evaluation of some semantic domains might result in error (*State*) or undefinedness (*Value*). To address these unsafe evaluations we lifted the domains of *State* and *Value* to domains *State*$_\perp$ and *Value*$_\perp$, which are disjoint sums of the basic domains and the domains $\perp_s$ and $\perp_v$.

$State_\perp = State + \perp_s$ and $Value_\perp = Value + \perp_v$

Note that $\perp_s$ and $\perp_v$ are unit domains.

IV. Formal Semantics of *MiniMaple*

In this section, first we give high level syntactic domains of *MiniMaple* and then give the signatures of valuation functions for various syntactic domains. Finally, we demonstrate the semantics of some interesting phrases by defining the corresponding valuation functions.

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

Prog := *Cseq*
Cseq := EMPTY | *C;Cseq*
C := ... | **while** *E* **do** *Cseq* **end do** | ...
E := ... | **proc**($Pseq$) $S; R$ **end proc** | **type**(*E,T*) | ...

A. Signatures of Valuation Functions

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meanings (semantic algebras) [14]. A valuation function VF for a syntax domain VF 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. In this section we first give the definitions of various relations and functions that are used in the definition of valuation functions. For example the state relation (*StateRelation*) is defined as a power set of pair of pre- and post-states, where post state can be an $State_\perp$ state and is defined as follows:

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

In the following we give signatures of valuation functions for syntactic domains of program and expression.

1) Program: The valuation function for program takes the abstract syntax of a program, i.e. a value of type *Prog* and results in a *ComRelation*. The syntax of the valuation function for program is as follows:

⟦*Prog*⟧: *ComRelation*

where *ComRelation* = *Environment* $\to$ *StateRelation*

Informally a valuation function for a program takes an alternative of an abstract syntax of Program and an *Environment* and results in a power set of pairs of pre- and post-states of the execution of the command sequence (by program definition).

The valuation functions for the syntactic domains Command and Command_Sequence are the same as defined above for the Program.

2) Expression: The valuation function for abstract syntax domain expression values of *E* is defined as follows:

⟦*E*⟧: *ExpRelation*

where

ExpRelation = *Environment* $\to$ *StateValueRelation*
StateValueRelation = $\mathbb{P}(State \times State_\perp \times Value_\perp)$

Here the post-state or the evaluated value can be unsafe.

B. Definitions of Valuation Functions

In this section we give the definition of the formal semantics of the major syntactic domains of *MiniMaple*, e.g. Program, Command and Expression.

1) Program Semantics: The semantics of *MiniMaple* program are essentially the same as of command semantics.

2) Command Semantics: The semantics of command is a relationship between the pre- and post-state of the execution of any *MiniMaple* command. In the following we define the valuation function for a while-loop.

For While-loop: *MiniMaple* supports a typical while-loop. The semantics of the iterations of a *MiniMaple* while-loop is determined by two sequences of states *pre* and *post* [15]. Both of these states are constructed from the pre-state of the loop. Any ith iteration (execution of the body) of the loop transforms *pre(i)* state into *post(i+1)* state from which the *pre(i+1)* is constructed. No iteration is allowed from the $State_\bot$ as *pre* state. The loop terminates when the guard expression evaluates to *false* or body of the loop evaluates to an $State_\bot$ post-state. The corresponding valuation function is defined as follows:

```
$[\![$while $E$ do $Cseq$ end do$]\!](e)(s,s') \Leftrightarrow$
    $\exists\ k \in Nat', t, u \in State_\bot{}^*$ :
    $t(1) = inState_\bot(s) \wedge u(1) = inState_\bot(s) \wedge$
      $(\forall i \in Nat'_k : iterate(i,t,u,e,[\![E]\!], [\![Cseq]\!])) \wedge$
        $((u(k) = in \bot_s () \wedge s' = u(k)) \vee$
        $(returns(data(inState(u(k)))) \wedge s' = t(k)) \vee$
        $(\exists\ v \in Value_\bot : [\![E]\!](e)(inState(t(k)), u(k), v)$
          $\wedge v <> inValue(inBoolean(True)) \wedge$
          IF $v = inValue(inBoolean(False))$ THEN
            $s' = t(k)$
          ELSE $s' = in \bot_s ()$ END
        )
      )
```

The corresponding *iterate* predicate formalizes the aforementioned while-loop semantics and is defined as follows:

```
$iterate \subseteq Nat' \times State_\bot{}^* \times State_\bot{}^* \times Environment$
                      $\times StateValueRelation \times StateRelation$
$iterate(i,t,u,e,E,C) \Leftrightarrow$
cases $t(i)$ of
$is \bot_s () \rightarrow false$
$[]isState(m) \rightarrow executes(data(m)) \wedge$
  $\exists v \in Value_\bot, s' \in State_\bot : E(e)(m,s',v) \wedge$
  cases $s'$ of
    $is \bot_s () \rightarrow u(i+1) = in \bot_s () \wedge t(i+1) = u(i+1)$
    $[]isState(p) \rightarrow$
      cases $v$ of
        $is \bot_v () \rightarrow u(i+1) = in \bot_s () \wedge$
                    $t(i+1) = u(i+1)$
        $[]isValue(v') \rightarrow$
          cases $v'$ of
            $isBoolean(b) \rightarrow b \wedge C(e)(p, u(i+1)) \wedge$
                              $t(i+1) = u(i+1)$
            $[] ... \rightarrow u(i+1) = in \bot_s () \wedge t(i+1) = u(i+1)$
          END //cases-$v'$
      END //cases-$v$
  END //cases-$s'$
END //cases-$t(i)$
```

The $iterate$ defines the semantics of while-loop by employing a fixed-point theory [33]. The corresponding termination or leaving iteration i determines the corresponding least fixed-point. Informally, i is the least iteration when loop terminates or leaves.

3) Expression Semantics: The semantics of expression is a relationship among pre- post-states and a value of an expression. Any *MiniMaple* expression evaluation in a pre-state *State* yields some value *Value*$_\bot$ with the post-state *State*$_\bot$. In the following, we define the valuation function as the semantics of a *MiniMaple* procedure.

For Procedure: As mentioned in an earlier section, *MiniMaple* procedure is defined as a predicate. Since static scoping is used to evaluate a *MiniMaple Procedure*, it is evaluated at an invocation time. In the following we define the corresponding definition time valuation function where a procedure expression evaluates to a procedure predicate value p. In a procedure expression, *Pseq*, *S* and *R* represent parameter sequence (identifiers with corresponding types), procedure (identifiers) declarations and body (commands) of the procedure respectively.

```
$[\![$proc$(Pseq)\ S; R$ end proc$]\!](e)(s,s',v) \Leftrightarrow$
LET $p \in Procedure, p(valseq, s_0, s_1, v') \Leftrightarrow$
  LET $e' = push(e, identifiers(Pseq))$
    $\exists varseq \in Variable^*, s'', s_3 \in State_\bot,$
      $e'', e''' \in Environment : [\![Pseq]\!](e')(e'', valseq) \wedge$
        $[\![S]\!](e'')(s_0, s'', e''') \wedge$
        cases $s''$ of
          $is \bot_s () \rightarrow in \bot_s ()$
          $[]isState(s_4) \rightarrow \exists s_2 \in State, v'' \in Value_\bot$ :
                        $s_2 = update(s_4, varseq, valseq)$
                        $\wedge [\![R]\!](e''')(s_2, s_3, v'')$
        END
  IN cases $s_3$ of
      $is \bot_s () \rightarrow in \bot_s ()$
      $[]isState(s_5) \rightarrow$
        cases $v''$ of
          $is \bot_v () \rightarrow s_1 = in \bot_s () \wedge v' = in \bot_v ()$
          $[]isValue(v_1) \rightarrow s_1 = inState_\bot(s_5) \wedge$
                          $v' = inValue_\bot(v_1)$
        END
      END
  END
IN $s' = inState_\bot(s) \wedge v = inValue_\bot(p)$ END
```

The specification language for *MiniMaple* is designed with the following two level constructs:

1) specification expression language and
2) specification annotations language

In the following respective sections we define their semantics.

V. Formal Semantics of Specification Expression

In this section, first we give the syntactic domain of specification expression language and then give the signatures and definition of an example valuation function for a special expression, i.e. a numeral quantifier.

The syntactic definition of specification expression *SE* is as follows:

SE := ... | *IOp(SE, B, SE)* | ...
IOp := **add** | **mul** | **min** | **max**
B := I = *SE..SE* | I **in** *SE*

In the following sub-section we give the signatures of valuation function for specification expression.

A. Signature of Valuation Functions

As discussed in the previous section, the specification expression relation (*StateResValRelation*) is defined as a power set of a pre-state, post-state, (procedure) result value and an evaluated value of the expression, where the post-state can be $State_\bot$ and also the evaluated value can be $Value_\bot$. The corresponding relation is defined as follows:

StateResValRelation := $\mathbb{P}(State \times State_\bot \times Value \times Value_\bot)$

The valuation function for the abstract syntax domain specification expression values of *SE* is defined as follows:

$[\![SE]\!]$: *Environment* $\rightarrow$ *StateResValRelation*

StateResValRelation formulates the relationship of the evaluation of a specification expression. This relationship is a tuple of a pre-state, post-state, result value and the evaluated value of the specification. The result value refers to the return value of a procedure expression and the evaluated value is the truth evaluation of the corresponding specification expression. Here the post-state or the evaluated value can be unsafe.

B. Definitions of Valuation Functions

In this section we give the definition of the denotational semantics of a numeral quantifier.

For Numeral Quantifier: In additional to logical quantifiers, the specification language also supports a numeral quantifier to compute a binary arithmetic operation (*IOp*) of expression (*SE*$_1$) for its range of values (*B*) those satisfy a certain property (*SE*$_2$). The semantics is defined as follows:

$$\begin{array}{l}
[\![IOp(SE_1, B, SE_2)]\!](e)(s, s', r, v) \Leftrightarrow \\
\exists vseq \in Value^* : [\![B]\!](e)(s, s', r, inValue_\bot(vseq)) \wedge \\
\quad \exists k' \in Nat', e_1 \in Environment, vs \in Value^* : \\
\quad\quad e_1 = push(e, getIdentifiers(B)) \wedge \\
\quad\quad (\ \forall i \in Nat'_k : iterate(i, I, e_1, vseq, vs, [\![SE_1]\!], [\![SE_2]\!])\ ) \wedge \\
\quad\quad\quad (\ k' < length(vseq) \wedge \\
\quad\quad\quad\quad (\ access(k', vseq) = is \bot_v () \vee \\
\quad\quad\quad\quad \forall s \in State, r \in Value : \exists v_1 \in Value, n \in State_\bot : \\
\quad\quad\quad\quad\quad [\![SE_2]\!](e_1)(s, inState_\bot(s), r, inValue_\bot(v_1)) \wedge \\
\quad\quad\quad\quad inBoolean(v_1) = inFalse()\ ) \wedge v = in \bot_v () \\
\quad\quad\quad) \vee (\ k' = length(vs) \wedge v = doIterate(IOp, vs)\ )
\end{array}$$

First a B is evaluated to get the sequence of values, if none of them evaluates to $Value_\bot$ then, *Environment* is iteratively updated with an identifier (appeared in the B) to a next value in the (evaluated) value sequence. At each iteration the (filter) SE_2 is evaluated and if it holds (true) then SE_1 is evaluated and its value is collected. If all these evaluations are safe, then we get a range of those values of SE_1 for whom SE_2 holds true. And at the end we apply the operator IOp to these filtered values and compute the result value. The corresponding auxiliary predicate *iterate* formalizes the collection of filtered values.

VI. Formal Semantics of Specification Annotation

The elements of specification language are called specification annotations. These annotations include the syntactic domains of mathematical theory Declarations (*D*), Procedure Specifications (*PS*), Loop Specifications (*LS*) and Assertions (*A*) as defined below:

D ::= EMPTY | (**define**(I(*Itseq)::T,R*); | **'type/**I'; | **'type/**I':=*T*; | **assume**(*SE*);) *D*
PS := **requires** *SE*; **global** *Iseq*; **ensures** *SE; EC*;
LS := **invariant** *SE*; **decreases** *SE*;
A := **ASSERT**(*SE,"I"*);

Note that the semantics of a specification annotation is essentially the soundness statement for correctness of corresponding annotation.

In the following we give the signatures of a valuation function for a loop specification.

1) Loop Specification: The valuation function for abstract syntax domain of loop specification values *LS* is defined as:

$[\![LS]\!]$: *Environment* $\rightarrow \mathbb{P}(State \times State_\bot)$

Semantically and as a soundness for correctness, aforementioned loop specification must hold in a given environment and in pre- and post-states.

A. Definitions of Valuation Functions

In this section we define the denotational semantics of loop specification.

For Loop Specification: The semantics of a loop specification is a relationship between the pre-state (*s*) and post-state (*s'*) of the loop. *MiniMaple* supports four variations of a for-loop and a typical while-loop, we only define the semantics of a while-loop specification as below:

$$\begin{array}{l}
[\![\textbf{invariant } SE_1;\ \textbf{decreases } SE_2; \\
\quad\quad\quad \textbf{while } E \textbf{ do } Cseq \textbf{ end do;}]\!](e)(s, s') \Leftrightarrow \\
(\ \forall b \in Boolean, r \in Value : \\
\quad [\![SE_1]\!](e)(s, inState_\bot(s), r, inValue_\bot(b) \Rightarrow b = inTrue() \\
) \wedge \\
(\ \forall i \in Integer, r \in Value : \\
\quad [\![SE_2]\!](e)(s, inState_\bot(s), r, inValue_\bot(i)) \Rightarrow i > 0 \\
) \wedge \\
(\ \forall s_1, s_2 \in State, r \in Value : \\
\quad (\ \forall b_1 \in Boolean : \\
\quad [\![SE_1]\!](e)(s_1, inState_\bot(s_1), r, inValue_\bot(b_1)) \\
\quad\quad\quad \Rightarrow b_1 = inTrue()\ ) \wedge \\
\quad (\ \forall j \in Integer : \\
\quad [\![SE_2]\!](e)(s_1, inState_\bot(s_1), r, inValue_\bot(j)) \Rightarrow j > 0\ ) \wedge \\
\quad (\ \forall b_2 \in Boolean : \\
\quad [\![E]\!](e)(s_1, inState_\bot(s_1), r, inValue_\bot(b_2)) \\
\quad\quad \Rightarrow b_2 = inTrue()\ ) \wedge [\![Cseq]\!](e)(s_1, inState_\bot(s_2)) \\
\Rightarrow (\ \forall b_3 \in Boolean : \\
\quad [\![SE_1]\!](e)(s, inState_\bot(s_2), r, inValue_\bot(b_3)) \\
\quad\quad\quad \Rightarrow b_3 = inTrue()\ ) \wedge \\
\quad (\ \forall k \in integer : \\
\quad [\![SE_2]\!](e)(s_2, inState_\bot(s_2), r, inValue_\bot(k)) \\
\quad\quad\quad \Rightarrow k >= 0 \wedge k < j\ )\ )
\end{array}$$

i.e.

- in a pre-state (s) an invariant (a boolean value) SE_1 evaluates to true and
- the termination term (a numeral value) SE_2 evaluates to an integer value greater than or equal to zero and

- also for any arbitrary pre-state s_1 and post-state s_2, if we make an iteration step for the body of the loop (*Cseq*) where in the pre-state s_1
  - the loop's guard condition expression E holds and
  - the invariant SE_1 evaluates to true and
  - the termination term SE_2 evaluates to an integer value that is greater than or equal to zero

  then (after iteration step) in the post-state s_2
  - the invariant SE_1 evaluates to true and
  - the termination term SE_2 evaluates to an integer value greater than or equal to zero and
  - the value of the termination term in the post-state s_2 must be less than its value in the pre-state s_1.

VII. Conclusion

The overview of formal semantics of *MiniMaple* and its specification language is presented in this paper. Also we formulated the soundness statements for the correctness of specification annotations. The proof of the correctness of denotational semantics of *MiniMaple* (and annotations) can easily be practiced by employing structural induction, we plan it as a future goal, though.

We may use the denotational semantics sketched in this paper to standardize the official semantics of Maple to certify the truth of the compliance of a Maple implementation to a given standard. However, our main goal here is to define the formal semantics of *MiniMaple* and its specification language as a pre-requisite of our verification calculus for *MiniMaple*. For our verification calculus first we need to translate our *MiniMaple* annotated program into a semantically equivalent Why3 program. Since the verification conditions will automatically be generated by the Why3 conditions generator, these conditions must be sound w.r.t. denotational semantics of *MiniMaple* and its annotations. Later any back-end automatic or interactive theorem provers of Why3 can be used to prove the correctness of generated verification conditions. Here our main goal is to prove the correctness of methods preconditions.

Acknowledgment

The author would like to thank Wolfgang Schreiner for his invaluable discussions and guidance throughout this work.

References

[1] M.T. Khan, *A Type Checker for MiniMaple*, Doktratskolleg (DK) Technical Report 2011-05, Johannes Kepler University, Linz, Austria, 2011.

[2] M.T. Khan, *Software for MiniMaple*, http://www.risc.jku.at/people/mtkhan/dk10/, 2011.

[3] M.T. Khan, W. Schreiner, *Towards a Behavioral Analysis of Computer Algebra Programs*, In: Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT'11), Paul Pettersson and Cristina Seceleanu (eds.), pp. 42-44., Vasteras, Sweden, 2011.

[4] M.T. Khan, *Towards a Behavioral Analysis of Computer Algebra Programs*, DK Technical Report 2011-13, JKU, Linz, Austria, 2011.

[5] M.T. Khan, *Formal Semantics of MiniMaple*, DK Technical Report 2012-01, Johannes Kepler University, Linz, Austria, 2012.

[6] M.T. Khan, *Formal Semantics of a Specification Language for MiniMaple*, DK Technical Report 2012-06, JKU, Linz, Austria, 2012.

[7] M.T. Khan, W. Schreiner, *On Formal Specification of Maple Programs*, In: Intelligent Computer Mathematics, Johan Jeuring et al. (eds.), LNAI 7362, pp. 443–447. Springer, 2012.

[8] M.T. Khan, W. Schreiner, *Towards the Formal Specification and Verification of Maple Programs*, In: Intelligent Computer Mathematics, Johan Jeuring et al. (eds.), LNAI 7362, pp. 231–247. Springer, 2012.

[9] P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press 2000.

[10] T. Lambert et al., *Using Miranda as a First Programming Language*. Journal of Functional Programming 3(1), 534, 1993.

[11] L. Meertens, *On Static Scope Checking in ALGOL 68*. ALGOL Bulletin, vol 35, pp. 4558, 1973.

[12] F. Bobot et al., *Why3: Shepherd your herd of provers*. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocaw, Poland, 2011.

[13] S. Abramsky, A. Jung, *Domain Theory*. Handbook of Logic in Computer Science, vol 3, pp. 1–168, Oxford University Press, 1994.

[14] D.A.Schmidt, *Denotational Semantics: a methodology for language development*. William C. Brown Publishers, 1986.

[15] W. Schreiner, *A Program Calculus*, Technical Report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2008.

[16] J. Rees et al, *The revised^3 report on the algorithmic language Scheme*. ACM SIGPLAN Notices 21(12), pp. 37–79, 1986.

[17] D. Bjørner, C.B. Jones, editors, *Formal Specification & Software Development*. Prentice-Hall, 1982.

[18] D. Bjørner, C.B. Jones, *Pascal*, Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs, NJ, pp. 175–251, Chap. 7, 1982.

[19] D. Bjørner, C.B. Jones, *Algol 60*, Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs, NJ, pp. 141–173, Chap. 6, 1982.

[20] B. Jørgan, L. Schultz, *A denotational (static) semantics method for defining Ada context conditions*, In: Towards a Formal Description of Ada, D. Bjørner and O. Nest (eds.), Lecture Notes in Computer Science 98, pp. 21–212. Springer, 1980.

[21] C. Wallace, *The semantics of the C++ programming language*, In: E. Børger (Ed.), Specification and Validation Methods, Oxford University Press, Oxford, England, pp. 131164, 1995.

[22] R.D. Tennent, *A denotational definition of the programming language Pascal*. Technical report, Oxford University, Programming Research Group, 1978.

[23] M. Vale, *The evolving algebra semantics of COBOL, Part 1: Programs and control*. Technical Report CSE-TR-162-93, University of Michigan, EECS Department, Ann Arbor, MI, 1993.

[24] P.D. Mosses, *Formal Semantics of Programming Languages: An Overview*, Electronics Notes Theoretical Computer Science 148(1), pp. 41–73, 2006.

[25] J. Stoy, *Denotational semantics: the Scott-Strachey approach to programming language theory*, USA, MIT Press, 1977.

[26] P.G. Larsen et al., *Towards A Formal Semantics Of The BSI/VDM Specification Language*, In: Information Processing 89, pp. 95–100, North-Holland, 1989.

[27] R.M. Burstall, J.A. Goguen, *Algebras, Theories and Freeness: An Introduction to Computer Scientists*, In: Theoretical Foundations of Programming Methodology, M. Broy and G. Schmidt (Eds.), D. Reidel, 1982.

[28] M. Spivey, *Towards a Formal Semantics for the Z Notation*, Technical Monograph PRG-41, pages 28, Oxford University Computing Laboratory, UK, 1984.

[29] E. Börger, D. Rosenzweig, *A mathematical definition of full Prolog*, Science of Computer Programming 24 (3), pp. 249–286, 1995.

[30] R. Milner, M. Tofte, R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[31] J. Carette, S. Forrest, *Property inference for Maple: an application of Abstract Interpretation*, pages 5–19, Calculemus 2007.

[32] N.S. Papaspyrou, *Denotational Semantics of ANSI C*, Computer Standards and Interfaces, vol. 23, no. 3, pp. 169–185, 2001.

[33] S.C. Kleene, *Introduction to Metamathematics*, Van Nostrand Princeton, N.J. 1952.

[34] Z. Manna, J. Vuillemin, *Fixpoint approach to the theory of computation*, Communications of the ACM, pp. 528–536, 1972.