Formal Semantics of a Specification Language for *MiniMaple*[*]

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

April 22, 2012

Abstract

In this paper, we give the complete definition of a formal semantics of a specification language for *MiniMaple*. This paper is an update of the previously reported formal (denotational) semantics of *MiniMaple*. As a next step we will develop a verification calculus for *MiniMaple* and its specification language. The verification conditions generated by the calculus must be sound with respect to both the formal semantics of *MiniMaple* and its corresponding specification language.

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

Contents

1 Introduction

Our goal here is to formally specify the behavior of computer algebra programs written in *MiniMaple* [4]. Therefore we have defined a formal semantics of a specification language for *MiniMaple* based on the formal semantics of *MiniMaple* and its type system. This paper is an extension of our previously defined formal semantics of *MiniMaple* [3]. For readability of this document, the term "semantics" is referred to the semantics of both *MiniMaple* programs and its specification language unless explicitly mentioned. This semantics is also a prerequisite of a verification calculus which we will develop as a next step. The verification conditions generated by the verification calculus must be sound with respect to these semantics.

The specification language for *MiniMaple* is semantically more complex than classical specification languages as they are fundamentally different from this language. As a consequence, the semantics of the specification language for *MiniMaple* which we have developed has the following features:

- It supports abstract data types to formalize mathematical concepts in general and computer algebra concepts in particular, while the existing specification languages are weaker in such specifications.
- The specification language 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 of a given expression.
- Semantic domains of values have some non-standard types of objects, for example symbol, uneval and union etc. Like *MiniMaple*, its specification language also supports additional functions and predicates, for example type tests i.e. **type**(E, T). For further details of the formal syntax of the specification language and *MiniMaple* and their corresponding type checkers, please see [2, 1].

The rest of the paper is organized as follows: in Section 2, we discuss the overview of our semantics of the specification language for *MiniMaple*. Section 3 presents conclusions and future work. Appendix A gives the formal syntax of the specification language and Appendix B contains the semantic algebras, definition of valuation functions and the definition of formal semantics of the specification language for *MiniMaple*.

2 Specification Expression Semantics

In this section, we describe the guidelines to read the different sections of Appendix B with the help of some examples. Each of the following subsections presents the corresponding section of the Appendix B. We start by giving the definition of different semantic algebras.

2.1 Semantic Algebras

The semantics of the specification expression makes use of several primitive and compound domains. In the following we enlist the semantic domains and their corresponding operations. For the readability of this document we also enlist the semantic domains of *MiniMaple*. Some operations are defined and some are just declared for the purpose of completeness of this document.

2.1.1 Truth Values

This subsection lists the primitive domain of boolean values and its operations.

2.1.2 Numeral Values

The primitive domains to represent numeral values (e.g. $\mathbb{Q}, \mathbb{N}$ etc.) and their operations are formalized in this section.

2.1.3 Environment Values

The domain *Environment* holds the environment values of a *MiniMaple* program and its specification language. *Environment* is formalized as a tuple of domains *Context* and *Space*. The domain *Context* is a mapping of identifiers to the environment values (*Variable*, *Procedure*, *Function* and *Type-Tag*), while the domain *Space* models the memory space.

2.1.4 State Values

This section defines the domain for the *State* of the program. A *Store* is the most important part of the state and holds for every *Variable* a *Value*. The value can be read and modified.

2.1.5 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*, mathematical *Function*, *Uneval* and *Symbol* are explained in the later sections. Also note that the domain *Value* is a recursive domain, e.g. *List* is defined by *Value** as discussed in the next section.

2.1.6 List Values

This section defines the structure of a typical semantic domain *List* as a finite sequence of semantic domain *Value*. The semantic domain *List* is used as a building block for some other domains, e.g. *Record* and *Set* as are discussed in the later sections. *List* and *Set* are defined as a sequence of values from a single domain.

2.1.7 Unordered Values

An unordered sequence of values is defined by the semantic domain *Set*. As a matter of fact, the domain *Set* is just defined as the domain *List*. In the semantics of *MiniMaple* set construction, the order of values in the construction of set is unknown. The elements of the domain *Set* are their permutation.

2.1.8 Tuple Values

The *Record* domain defines a tuple as a sequence of different semantic values, each representing one element of the tuple. *Record* is also defined by the semantic domain *List*.

2.1.9 Sequence Values

In this section we define a finite sequence of values (Value*) from the semantic domain *Value* and its operations.

2.1.10 Procedure Values

The semantic domain *Procedure* is defined to represent a *MiniMaple* procedures. It is defined as a predicate of sequence of (parameter) values, pre- and post-states and the return value. A *Procedure* is one of the values that can be stored in the *Environment*.

2.1.11 Module Values

The semantic domain *Module* defines the *MiniMaple* module values. *Module* maps identifiers to their corresponding values of the statements.

2.1.12 Identifier Values

The semantic domain *Identifier* defines the values of the corresponding syntactic domain of *MiniMaple* and its operations. It also defines the syntactic sequence of *Identifier* values.

2.1.13 Symbol Values

This section defines the structure of the semantic domain *Symbol*. The domain *Symbol* contains those names which are not assigned any value.

2.1.14 Character String Values

Character strings are defined as a semantic domain *String*.

2.1.15 Unevaluated Values

The semantic domain *Uneval* represents unevaluated values of the corresponding syntactic domain of *MiniMaple*. Any term enclosed with single quotes represents an unevaluated value in *MiniMaple*. Each evaluation operation strips off one level of single quotes.

2.1.16 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 *StateU* and *ValueU*, which are disjoint sums of the basic domains and domains *Error* respectively *Undefined*.

2.1.17 Parameter Values

The semantic domain *Parameter* defines the values of the corresponding syntactic domain of *MiniMaple* and its operations. It also defines the syntactic sequence of *Parameter* values.

2.1.18 Declaration Values

The semantic domain of *Declaration* defines the values of the global, local and exported identifiers as of the corresponding syntactic domain of *MiniMaple*.

2.1.19 Type-Tag Values

A *Type-Tag* is a disjoint union domain of type-tags, one per actual type supported by *MiniMaple*. Some values of *Type-Tag* are unit domains and some are recursively defined over the domain *Type-Tag* depending on the corresponding basic or extended *MiniMaple* types. This domain is used in the type tests used in various *MiniMaple* constructs.

2.1.20 Function Values

The semantics domain *Function* defines and formalizes the mathematical functions in the 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

$$Function^n = Value^n \rightarrow Value$$

2.2 Signatures of Valuation Functions

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meanings (semantic algebras) [7]. 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 each syntactic domain of specification expression.

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 specification expression relation (*StateResultValueRelation*) 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 an *Error* state and also the evaluated value can be *Undefined*. The corresponding relation is defined as follows:

$$StateResultValueRelation := \mathbb{P}(State \times StateU \times Value \times ValueU)$$

In this subsection, we define the various valuation functions for the syntactic domains of the specification expression (and its associated syntactic domains) of the specification language for *MiniMaple*.

2.2.1 Specification Expression

The valuation function for the abstract syntax domain specification expression values of spec-expr is defined as follows:

$$[\![\text{spec-expr}]\!] : \mathit{Environment} \rightarrow \mathit{StateResultValueRelation}$$

StateResultValueRelation 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.

2.2.2 Specification Expression Sequence

The valuation function for abstract syntax domain expression sequence values of eseq is defined as follows:

$$[\![\text{eseq}]\!] : \mathit{Environment} \rightarrow \mathit{StateResultValueSeqRelation}$$

The valuation function maps an *Environment* to *StateResultValueSeqRelation*. *StateResultValueSeqRelation* is the same as *StateResultValueRelation* except that it returns a sequence of values instead of a single value as an evaluated value.

2.2.3 Binding

The valuation function for the abstract syntax domains of binding values binding also results in *StateResultValueRelation* as shown above for procedure specification:

$$[\![\text{binding}]\!] : \mathit{Environment} \rightarrow \mathit{StateResultValueRelation}$$

The desired evaluated value of the binding is a sequence of values.

2.2.4 Identifier Typed

The evaluation of typed identifiers results in a new *Environment* and the valuation function for the abstract syntax domains of identifier typed It is defined as follows:

$$[\![\text{It}]\!] : \mathit{Environment} \rightarrow \mathit{Environment}$$

2.2.5 Identifier Typed Sequence

The valuation function for the sequence of typed identifiers (Itseq) is the same as shown above identifier typed:

$$[\![\text{Itseq}]\!] : \mathit{Environment} \rightarrow \mathit{Environment}$$

In the following section we define the auxiliary functions and predicates used in the formal semantics of the specification expression (and associated domains) of *MiniMaple*.

2.3 Auxiliary Functions and Predicates

In the following subsections auxiliary functions and predicates for the use in semantics definition of sequence, binding and special expressions are defined.

2.3.1 For Sequence

We defined the relation *seq* to be used later in this document for the semantic definitions of sequence expression.

2.3.2 For Binding

Also we have defined the relation *iterate* that formalizes the corresponding binding semantics. It will be used later in this document for the semantic definitions of binding (an associated domain of specification expression).

2.3.3 For Special Expression

This section defines the equality of two binary operators, i.e. *equalsOperator* and the modification function *subsop* for the semantic domains *List*, *Set* and *Record* etc.

2.4 Definitions of Valuation Functions

In this section we give the definition of the formal semantics of the syntactic domains (and associated domains) of the specification expression language for *MiniMaple*, e.g. Specification Expression, Binding, Specification Expression Sequence. The semantics of other domains of the specification language are very simple and can be easily rehearsed.

2.4.1 Specification Expression Semantics

The semantics of a specification expression is a relationship among the pre-state (s), post-state (s'), (procedure) result value (r) and an evaluated value (v). As a specification expression involves logical expressions and terms as well, so the result of the evaluation of a specification expression *spec-expr* results in a boolean or term value. The evaluation of a specification expression might be unsafe.

Logical A specification language for *MiniMaple* supports logical-and and logical-or, implication and equivalence binary expressions. The expressions *spec-expr*$_1$ and *spec-expr*$_2$ are evaluated, if they both yield to boolean values, then the corresponding logical-and or logical-or, implication or equivalence boolean operation of these values is evaluated. If any of them yields to *Undefined* value then the whole logical expression evaluates to *Undefined* value otherwise corresponding boolean value is the result of its evaluation.

Quantifier Also a specification language for *MiniMaple* supports universal and existential quantifiers. First the bounded variables *Itseq* are evaluated and the *Environment* is updated with these identifiers. The semantics of a universal quantifier evaluates to true if the *spec-expr* holds (true) in an *Environment* where the (bounded) identifiers are mapped to all their

possible values (with respect to types), otherwise it evaluates to false. And the semantics of an existential quantifier evaluates to true if the *spec-expr* holds (true) in an *Environment* where the (bounded) identifiers are mapped to any of their possible values (with respect to types), otherwise it evaluates to false.

Iterator In additional to logical quantifiers, the specification language also supports numeral quantifiers to compute a binary arithmetic operation (**add**, **mul**, **max** and **min**) over a range of values those satisfy a certain property. First a *binding* is evaluated to get the sequence of values, if none of them evaluates to *Undefined* then, *Environment* is iteratively updated with an identifier (appeared in the *binding*) to a next value in the (evaluated) value sequence. At each iteration the (filter) *spec-expr*$_2$ is evaluated and if it holds (true) then *spec-expr*$_1$ is evaluated and its value is collected. If all these evaluations are safe, then we get a range of those values of *spec-expr*$_1$ for whom *spec-expr*$_2$ holds true. And at the end we apply the operator $it - op$ (**add**, **mul**, **max** and **min**) to these filtered values and compute the result value. The corresponding auxiliary function *iterate* formalizes the collection of filtered values.

Local Definition The specification language supports an evaluation of a specification expression with a local definition (by corresponding LET-IN construct). First the local definitions (LET part) is evaluated, the specification expression sequence is evaluated, if none of them yields *Undefined* value then, *Environment* is updated with the identifiers ($Iseq$) mapped to the corresponding evaluated values (expression sequence). Then the specification expression *spec-expr* (IN part) is evaluated in the updated *Environment*, the result of the whole LET-IN construct is the evaluated value of *spec-expr*.

Conditional A conditional expression is supported in the specification language. The semantics of a conditional expression is to evaluate *spec-expr*$_1$ first, if it yields to true then specification expression *spec-expr*$_2$ is evaluated that gives the the result semantic value, otherwise specification expression *spec-expr*$_3$ is evaluated as a result value.

Call First, the argument specification expression sequence is evaluated, if any of them yields an *Undefined* value, the specification expression evaluates to *Undefined* value. Otherwise, the *Environment* is looked up for a mathematical function named *I*. This function is applied to the argument values to get the result. The result is the right side of the equation function definition for whom this equation holds true.

Old An **OLD** construct is an expression that refers to the value of an identifier in the previous state. The semantics of an old expression is the value looked up in the previous state.

Result An expression **RESULT** refers to the result (return) value of the evaluation of a *MiniMaple* procedure expression. It is provided as the third parameter of the predicate of the specification expression relationship.

The semantics of specification binary, unary and special expressions is the same as discussed for the corresponding binary, unary and special expression of

MiniMaple except that here the semantics is a relationship among the pre-state (s), post-state (s'), (procedure) result value (r) and an evaluated value (v). Also the semantics of the syntactic domains *Type* and *Type Sequence* are the same as discussed for the corresponding *MiniMaple* domains.

3 Specification Annotation Semantics

In this section, we define the semantics of the specification annotation for *MiniMaple*. The main specification annotations includes the syntactic domains of Specification Declaration, Procedure Specification, Loop Specification and Assertion.

3.1 Signatures of Valuation Functions

In the following subsections, we define various valuation functions for the syntactic domains of the above mentioned domains.

3.1.1 Specification Declaration

The valuation function for abstract syntax domain of procedure specification values of decl is defined as follows:

$$[\![\text{ decl }]\!] : \mathit{Environment} \to \mathit{Environment}$$

The specification declaration introduces a new environment that contains the mathematical function declarations/definitions.

3.1.2 Procedure Specification

The valuation function for abstract syntax domain of procedure specification values of proc-spec is defined as follows:

$$[\![\text{ proc-spec }]\!] : \mathbb{P}(\mathit{Environment})$$

The procedure specification holds in the given environment.

3.1.3 Loop Specification

The valuation function for abstract syntax domain of loop specification values of loop-spec is defined as follows:

$$[\![\text{ loop-spec }]\!] : \mathit{Environment} \to \mathbb{P}(\mathit{State} \times \mathit{StateU})$$

The loop specification must hold in the given environment and in the pre- and post-state.

3.1.4 Assertion

The valuation function for abstract syntax domain of assertion values of asrt is defined as follows:

$$[\![\text{ asrt }]\!] : \mathit{Environment} \to \mathbb{P}(\mathit{State})$$

The assertion holds in the given environment and state.

3.2 Definition of Valuation Functions

In this section we give the definition of the formal semantics of the main elements of the specification language for *MiniMaple*, e.g. Specification Declaration, Procedure Specification, Loop Specification and Assertion.

3.2.1 Specification Declaration Semantics

A specification declaration can be used to specify a mathematical theory and its semantics produces a new environment that has the corresponding theory declarations and definitions. We have defined the overall semantics of a specification declaration as a theory declaration and the individual semantics of each syntactic alternative of declaration returns a tuple of all its syntactic components. For an overall semantics of declaration, first from a given declaration (*decl*) all the function definitions (function identifiers and corresponding rules), axioms (specification expressions) and type declarations (type identifiers and corresponding types) are collected and then

- the type of each of the type identifier is evaluated that introduces a new environment where the type identifier is mapped with its corresponding type. The evaluation of all the type identifiers produces e_n environments and
- the environment e_n is updated (to the result environment e') with the function identifiers mapped to corresponding such *Function* values where each function is of some arity equal to the length of its corresponding parameters and
- in the updated environment e' all the rules must hold and
- also in an environment e' all the axioms evaluate to true.

In the following subsections, we give the semantics definition of each alternative as an the semantics of specification declaration.

Mathematical Function A mathematical function can be defined with the help of function signatures (parameter types and return type) and rules (**define** construct). A rule can be empty or an equation, in the former case it returns a boolean truth value (a truth declaration like a predicate) while in the latter case it returns the truth of the equation (equality). The left side of the equation is a function application (function name and its parameters) and right side is a specification expression that evaluates to a value of its declared return type.

User Defined Type *MiniMaple* programs may contain more complex concrete data types, any identifier can be used to represent that complex data type in the specification.

Abstract Data Type As the main goal of the specification language is to represent abstract computer algebraic concepts, so the abstract data type representation is used to model such mathematical concepts in the specification language.

Axiom The specification language for *MiniMaple* also supports axiom declaration. Here an axiom is a specification expression that returns a boolean truth value.

3.2.2 Procedure Specification Semantics

The semantics of a procedure specification is true in a given environment. For the semantics of a procedure specification, if for any arbitrary pre-state s_1 and post-state s_2

- we update the given environment e with all the identifiers (from the given parameter sequence *Pseq*) mapped to all their possible values (w.r.t. their types) and
- the precondition expression (*spec-expr*$_1$) holds in the pre-state s_1 and
- the evaluation of a procedure expression (**proc**(Pseq)::T; S;R; **end proc;**) in a pre-state s_1 evaluates to a procedure relation p and
- the procedure relation p holds for all the possible values of parameter identifiers

then

- the two states s_1 and s_2 are equal except for the values of identifiers *Iseq* and
- if the post-state s_2 is an exception-state then the exceptional behavior of the procedure *excep-clause* holds in the post-state s_2, otherwise normal behavior *spec-expr*$_2$ holds in the post-state s_2.

3.2.3 Loop Specification Semantics

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 while-loop, we only discuss the semantics of while-loop specification. The semantics of the for-loop specification can easily be practiced. The semantics of a while-loop specification is defined below:

- in a pre-state (s) an invariant (boolean specification expression) *spec-expr*$_1$ evaluates to true and
- the termination term (a numeral specification expression) *spec-expr*$_2$ 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 expression E holds and
  - the invariant *spec-expr*$_1$ evaluates to true and
  - the termination term *spec-expr*$_2$ 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 *spec-expr*$_1$ evaluates to true and
– the termination term *spec-expr*$_2$ 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

3.2.4 Assertion Semantics

The semantics of an assertion is the same as shown above for a boolean specification expression. The result of the evaluation of the boolean specification expression evaluates to true in the given *Environment* and state.

4 Conclusions and Future Work

In this paper we gave the definition of formal semantics of a specification language for *MiniMaple* including semantic domains, semantic algebras, declaration and definitions of valuation functions. For the readability of this document, sometimes we also included the related definitions/declarations about the formal semantics of *MiniMaple*. As a next step we will develop a verification calculus for *MiniMaple* programs and its specification language and the verification conditions generated by the verification calculus must be sound with respect to the formal semantics of both *MiniMaple* and its specification language. After our initial study and discussions we intend to use Why3 as an intermediate verification language for our verification calculus. Then the back-end provers of Why3 will be used to prove the correctness of verification conditions. Currently we are working on the translation of specification annotated *MiniMaple* program to corresponding Why3 program.

Acknowledgment

The author cordially thanks Wolfgang Schreiner for his valuable and constructive comments and suggestions throughout this work.

References

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

[2] 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.

[3] Muhammad Taimoor Khan. Formal Semantics of *MiniMaple*. DK Technical Report 2012-01, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, January 2012.

[4] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards a Behavioral Analysis of Computer Algebra Programs (Extended Abstract). In Paul Pettersson and Cristina Seceleanu, editors, *Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT'11)*, pages 42–44, Vasteras, Sweden, October 2011.

[5] Muhammad Taimoor Khan and Wolfgang Schreiner. On Formal Specification of Maple Programs. In *Conferences on Intelligent Computer Mathematics, Systems and Projects track*. 2012.

[6] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards the Formal Specification and Verification of Maple Programs. In *Conferences on Intelligent Computer Mathematics, Calculemus track*. 2012.

[7] Schmidt, David A. *Denotational Semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

Appendices

Appendix A gives the formal abstract syntax (language grammar) for the specification language of *MiniMaple* which is different than the one given in [5, 6] and Appendix B gives the complete definition of formal semantics of the specification language for *MiniMaple* declaration, procedure specification, loop specification, assertion and specification expressions with valuation functions and auxiliary functions and predicates.

A Formal Syntax of a Specification Language for *MiniMaple*

decl ∈ Declaration
proc-spec ∈ Procedure_Specification
loop-spec ∈ Loop_Specification
asrt ∈ Assertion
rules ∈ Rules
excep-clause ∈ Exception_Clause
eseq ∈ Specification_Expression_Sequence
spec-expr ∈ Specification_Expression
binding ∈ Binding
Itseq ∈ Identifier_Typed_Sequence
It ∈ Identifier_Typed
Iseq ∈ Identifier_Sequence
I ∈ Identifier
Bop ∈ Binary_Operator
Uop ∈ Unary_Operator
it-op ∈ Iteration_Operator
esop ∈ Especial_Operator
sel-op ∈ Selection_Operator
Tseq ∈ Type_Sequence
T ∈ Type
N ∈ Numeral

decl ::= EMPTY | (**define**(I(Itseq)::T,rules);
| **‘type/**I‘;
| **‘type/**I‘:=T;
| **assume**(spec-expr);) decl
proc-spec ::= **requires** spec-expr; **global** Iseq; **ensures** spec-expr; excep-clause
loop-spec := **invariant** spec-expr; **decreases** spec-expr;
asrt := **ASSERT**(spec-expr, (EMPTY | “I“));
rules ::= EMPTY | I(Itseq) = spec-expr, rules
excep-clause ::= EMPTY | **exceptions** ”I” spec-expr; excep-clause
eseq ::= EMPTY | spec-expr, eseq
spec-expr ::= I **(** eseq **)** | **type**(spec-expr,T) | spec-expr **and** spec-expr | spec-expr **or** spec-expr
| spec-expr **equivalent** spec-expr | spec-expr **implies** spec-expr
| **forall**(Itseq, spec-expr) | **exists**(Itseq, spec-expr)
| **(**spec-expr**)** | spec-expr Bop spec-expr | Uop spec-expr | esop

| it-op(spec-expr, binding, (EMPTY | spec-expr))
| **true** | **false** | **LET** Iseq=eseq **IN** spec-expr | **RESULT**
| **'if'**(spec-expr1, spec-expr2, spec-expr3) | I | I1:-I2 | **OLD** I | N
| spec-expr1 = spec-expr2 | spec-expr1 <> spec-expr2

binding ::= I = spec-expr1...spec-expr2 | I **in** spec-expr
Itseq ::= EMPTY | It, Itseq
It ::= I::T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Bop ::= +| − | / | ∗ | **mod** | < | > | ≤ | ≥ | = | <>
Uop ::= **not** | − |+
it-op ::= **add** | **mul** | **max** | **min** | **seq**
esop ::= **op(** spec-expr1, spec-expr2 **)** | **op(** spec-expr **)**
| **op(** spec-expr..spec-expr, spec-expr **)** | **nops(** spec-expr **)**
| **subsop(** spec-expr1=spec-expr2, spec-expr3 **)**
| **subs(** I=spec-expr1, spec-expr2 **)** | **“** spec-expr **“**
| I sel-op | [eseq] | { eseq } | I(eseq) | **eval(** I,1 **)**
sel-op ::= EMPTY | [eseq] sel-op
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | [Tseq] | **procedure**[T]**(** Tseq **)**
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **unevaluated** | I
N ::= a sequence of decimal digits

B Formal Semantics of a Specification Language for *MiniMaple*

This section gives the complete definition of the formal semantics of the specification language. For the completeness of this document, some of the auxiliary and domain functions are just declared. In this case the informal comments are added to give reader our intention.

Sections 2.1 and 2.3 are the updates to the corresponding previously reported sections on the formal semantics of *MiniMaple*. Please be noted, we have also not redefined some related semantic definitions for the specification language here, in this case the definitions are the same as defined for the *MiniMaple* semantics, so please see [3].

2.1 Semantic Algebras

2.1.1) Truth Values

Domain Tr = Boolean = {True, False}

Operations

true: Tr
false: Tr
and: Tr x Tr → Tr
or: Tr x Tr → Tr
implies: Tr x Tr → Tr
equivalence: Tr x Tr → Tr
not: Tr → Tr
length: Boolean → Nat'
length(b) = 1

2.1.2) Numeral Values

Domain Nat' = N\{0}, Nat = N, Integer = Z, Rational = Q, Float = R

Operations

length: Integer → Nat'
length(k) = 1

length: Rational → Nat'
length(r) = 1

length: Float → Nat'
length(f) = 2

iterations: Integer x Integer x Integer → Integer
iterations(x,y,z) = IF x+y <= z THEN 1 + iterations(x+y, y, z) ELSE 0

expRangeValues: Integer x Integer → Value*
expRangeValues(m,n) → IF m < n THEN
        cons(inValue(m), expRangeValues(m+1,n))
    ELSE
        cons(inValue(m), emptyValue)
    END //if-m+1

2.1.3) Environment Values

Domains

Environment = Context x Space
Context = Identifier → EnvValue
EnvValue = Value + Procedure + Function + Type-Tag
Space = P(Variable)
Variable := n, n \in N // represents location

Operations
space : Environment → Space
space(c,s) = s

context : Environment → Context
context(c,s) = c

environment : Context x Space → Environment
environment(c,s) = <c,s>

take : Space → Identifier x Space
take(s) = LET x= SUCH x: x \in s IN <x,s\{x}>

push : Environment x Identifier → Environment
push(e,I) = LET <x,s'> = take(space(e)) IN environment(context(e)[I |-> inVariable(x)], s') END

push : Environment x Identifier x Type-Tag→ Environment
push(e,I,t) = LET <x,s'> = take(space(e)) IN environment(context(e)[I |-> inType-Tag(x)], s') END

push : Environment x Identifier x Function→ Environment
push(e,I,f) = LET <x,s'> = take(space(e)) IN environment(context(e)[I |-> inFunction(x)], s') END

push : Environment x Identifier x Value→ Environment
push(e,I,v) = LET <x,s'> = take(space(e)) IN environment(context(e)[I |-> inValue(x)], s') END

push : Environment x Identifier Sequence → Environment
push(e,empty) = e
push(e,(I,Iseq)) = LET
<x,s'> = take(space(e))
e1 = environment(context(e)[I |-> inVariable(x)], s')
e2 = push(e1, Iseq)
IN e2 END

push : Environment x Identifier Sequence x Value*→ Environment

getExportValues : Environment x State x IdentifierSeq → Value*
getExportValues(e,EMPTY, s) = emptyValue
getExportValues(e,i,s) = cons(store(s)([[I]](e)), emptyValue)
getExportValues(e,<i,iseq>,s) = cons(store(s)([[I]](e)), getExportValues(e,iseq,s))

2.1.4) State Values
Domains
State = Store x Data
Store = Variable → Value
Data = Flag x Exception x Return
Flag = {execute, exception, return, leave}

Exception = Identifier x ValueU
Return = ValueU

Operations
state : Store x Data → State
state(s,d) = <s,d>

exception : Identifier x ValueU → Exception
exception(i,v) = <i,v>

ide : Exception → Identifier
ide(i,v) → i

valuee : Exception → ValueU
valuee(i,v) → v

data : State → Data
data(s,d) = d

store : State → Store
store(s,d) → s

flag : Data → Flag
flag(f,e,r) = f

exception : Data → Exception
exception(f,e,r) = e

return : Data → Return
return(f,e,r) = r

data : Flag x Exception x Return → Data
data(f,e,r) = <f,e,r>

execute : State → State
execute(s) = LET d = data(s) IN state(store(s), data(execute, exception(d), return(d))

exception : State x String x ValueU → State
exception(s,st,v) = LET d = data(s) IN state(store(s), data(exception, (st,v), return(d))

return : State x ValueU → State
return(s,v) = LET d = data(s) IN state(store(s), data(return, exception(d), v)

executes : P(Data)
executes(d) <=> flag(d) = execute

exceptions : P(Data)

```
exceptions(d) <=> flag(d) = exception

returns : P(Data)
returns(d) <=> flag(d) = return

update : State x Variable* x Value* → State
update(s,empty,empty) = s
update(s,r,v) = state(store(s)[r |-> v], data(s))
update(s,<r,rseq>,<v,vseq>) = update(state(store(s)[r |-> v], data(s)), <rseq>, <vseq>)

update : State x Variable* x List* x Value* → State
update(s,empty,empty, empty) = s
update(s,r,l,v) = IF empty<>l THEN
                        LET v' = store(s)(r)
                        IN
                                cases v' of
                                        isList(list) → state(store(s)[r |-> updateList(l,v,list)], data(s))
                                    [] isRecord(rec) → state(store(s)[r|->updateRecord(l,v,list)], data(s))
                                    [] isSet(set) → state(store(s)[r |-> updateSet(l,v,list)], data(s))
                                    [] isString(st) → state(store(s)[r |-> updateString(l,v,list)], data(s))
                                    [] isRational(rt) → state(store(s)[r |-> rt], data(s))
                                    [] isFloat(f) → state(store(s)[r |-> f], data(s))
                                    [] isInteger(j) → state(store(s)[r |-> j], data(s))
                                    [] isSymbol(sm) → state(store(s)[r |-> sm], data(s))
                                    [] isUneval(u) → state(store(s)[r |-> u], data(s))
                                    [] … → s
                        END//let-v'
                ELSE state(store(s)[r |-> v], data(s)) END

update(s,<r,rseq>,<l,lseq>, <v,vseq>) = update( update(s,r,l,v), rseq, lseq, vseq )

previous: Environment x State → State
```

2.1.5) Semantic Values

Domain

Value = Function + Procedure + Module + List + Set + Record + Boolean + Integer + String + Uneval + Value* + …

Operations

```
length: Value → Nat
length(v) =
        cases v1 of
                isList(l) → length(l)
                [] isSet(s) → length(s)
                [] isRecord(r) → length(r)
                [] isInteger(j) → length(j)
                [] isRational(t) → length(t)
```

```
                [] isFloat(f) → length(f)
                [] isString(st) → length(st)
                [] isBoolean(b) → length(b)
                [] isUneval(u) → length(u)
                [] isSymbol(sm) → length(sm)
                [] … → 0
        END //cases-v1

expValues: Value → Value*
expValues(v) →
                cases v of
                        isList(l) → accessAll(l)
                        [] isRecord(r) → accessAll(r)
                        [] isSet(s) → accessAll(s)
                        [] isInteger(i) → cons(i, emptyValue)
                        [] isRational(rat) → cons(rat, emptyValue)
                        [] isFloat(f) → cons(f, emptyValue)
                        [] isString(str) → cons(str, emptyValue)
                        [] isSymbol(sy) → cons(sy, emptyValue)
                        [] isUneval(u) → cons(u, emptyValue)
                        [] isValue*(vs) → cons(vs, emptyValue)
                        [] ... → vseq = emptyValue
                END //cases-v

add: Value x Value → ValueU
add(v1, v2) =
        cases v1 of
                isInteger(j) →
                        cases v2 of
                                isInteger(k) → inValueU(inInteger(j+k))
                                [] isRational(r) →
                                        LET
                                                res=j+r
                                        IN
                                                IF isDivisible(numerator(res), denominator(res)) THEN
                                                        inValueU(inInteger(res))
                                                ELSE
                                                        inValueU(inRational(res))
                                                END //if-isDivisible
                                        END //let-res
                                [] isFloat(f) → inValueU(inFloat(j+f))
                                [] … → inValueU(inUndefined())
                        END //cases-v2
                [] isRational(r) →
                        cases v2 of
                                isInteger(k) →
                                        LET
```

```
                                        res=r+k
                                    IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                            inValueU(inInteger(res))
                                        ELSE
                                            inValueU(inRational(res))
                                        END //if-isDivisible
                                    END //let-res
                                [] isRational(r1) →
                                    LET
                                        res=r+r1
                                    IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                            inValueU(inInteger(res))
                                        ELSE
                                            inValueU(inRational(res))
                                        END //if-isDivisible
                                    END //let-res
                                [] isFloat(f) → inValueU(inFloat(r+f))
                                [] … → inValueU(inUndefined())
                            END //cases-v2
                        [] isFloat(f) →
                            cases v2 of
                                isInteger(k) → inValueU(inFloat(f+k))
                                [] isRational(r) → inValueU(inFloat(f+r))
                                [] isFloat(f1) → inValueU(inFloat(f+f1))
                                [] … → inValueU(inUndefined())
                            END //cases-v2
                        [] … → inValueU(inUndefined())
                END //cases-v1

sub: Value x Value → ValueU
sub(v1, v2) =
        cases v1 of
            isInteger(j) →
                cases v2 of
                    isInteger(k) → inValueU(inInteger(j+k))
                    [] isRational(r) →
                        LET
                            res=j-r
                        IN
                            IF isDivisible(numerator(res), denominator(res)) THEN
                                inValueU(inInteger(res))
                            ELSE
                                inValueU(inRational(res))
                            END //if-isDivisible
                        END //let-res
```

```
                    [] isFloat(f) → inValueU(inFloat(j-f))
                    [] … → inValueU(inUndefined())
                END //cases-v2
            [] isRational(r) →
                cases v2 of
                    isInteger(k) →
                        LET
                            res=r-k
                        IN
                            IF isDivisible(numerator(res), denominator(res)) THEN
                                inValueU(inInteger(res))
                            ELSE
                                inValueU(inRational(res))
                            END //if-isDivisible
                        END //let-res
                    [] isRational(r1) →
                        LET
                            res=r-r1
                        IN
                            IF isDivisible(numerator(res), denominator(res)) THEN
                                inValueU(inInteger(res))
                            ELSE
                                inValueU(inRational(res))
                            END //if-isDivisible
                        END //let-res
                    [] isFloat(f) → inValueU(inFloat(r+f))
                    [] … → inValueU(inUndefined())
                END //cases-v2
            [] isFloat(f) →
                cases v2 of
                    isInteger(k) → inValueU(inFloat(f-k))
                    [] isRational(r) → inValueU(inFloat(f-r))
                    [] isFloat(f1) → inValueU(inFloat(f-f1))
                    [] … → inValueU(inUndefined())
                END //cases-v2
            [] … → inValueU(inUndefined())
        END //cases-v1

mul: Value x Value → ValueU
mul(v1, v2) =
        cases v1 of
            isInteger(j) →
                cases v2 of
                    isInteger(k) → inValueU(inInteger(j+k))
                    [] isRational(r) →
                        LET
                            res=j*r
```

```
                                IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                                inValueU(inInteger(res))
                                        ELSE
                                                inValueU(inRational(res))
                                        END //if-isDivisible
                                END //let-res
                        [] isFloat(f) → inValueU(inFloat(j*f))
                        [] … → inValueU(inUndefined())
                END //cases-v2
        [] isRational(r) →
                cases v2 of
                        isInteger(k) →
                                LET
                                        res=r*k
                                IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                                inValueU(inInteger(res))
                                        ELSE
                                                inValueU(inRational(res))
                                        END //if-isDivisible
                                END //let-res
                        [] isRational(r1) →
                                LET
                                        res=r*r1
                                IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                                inValueU(inInteger(res))
                                        ELSE
                                                inValueU(inRational(res))
                                        END //if-isDivisible
                                END //let-res
                        [] isFloat(f) → inValueU(inFloat(r+f))
                        [] … → inValueU(inUndefined())
                END //cases-v2
        [] isFloat(f) →
                cases v2 of
                        isInteger(k) → inValueU(inFloat(f*k))
                        [] isRational(r) → inValueU(inFloat(f*r))
                        [] isFloat(f1) → inValueU(inFloat(f*f1))
                        [] … → inValueU(inUndefined())
                END //cases-v2
        [] … → inValueU(inUndefined())
END //cases-v1

div: Value x Value → ValueU
div(v1, v2) =
```

```
cases v1 of
        isInteger(j) →
                cases v2 of
                        isInteger(k) →
                                IF k = 0 THEN
                                        inValueU(inUndefined())
                                ELSE
                                LET
                                        res=j/k
                                IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                                inValueU(inInteger(res))
                                        ELSE
                                                inValueU(inRational(res))
                                        END //if-isDivisible
                                END //let-res
                                END //if-k=0
                        [] isRational(r) →
                                IF denominator(r) = 0 THEN
                                        inValueU(inUndefined())
                                ELSE
                                LET
                                        res=j/r
                                IN
                                        IF isDivisible(numerator(res), denominator(res)) THEN
                                                inValueU(inInteger(res))
                                        ELSE
                                                inValueU(inRational(res))
                                        END //if-isDivisible
                                END //let-res
                                END //if-denominator(r)
                        [] isFloat(f) →
                                IF denominator(f) = 0.0 THEN
                                        inValueU(inUndefined())
                                ELSE
                                        inValueU(inFloat(j/f))
                                END //if-denominator
                        [] … → inValueU(inUndefined())
                END //cases-v2
        [] isRational(r) →
                IF denominator(r) = 0 THEN
                        inValueU(inUndefined())
                ELSE
                cases v2 of
                        isInteger(k) →
                                LET
                                        res=r/k
```

```
                IN
                    IF isDivisible(numerator(res), denominator(res)) THEN
                        inValueU(inInteger(res))
                    ELSE
                        inValueU(inRational(res))
                    END //if-isDivisible
                END //let-res
            [] isRational(r1) →
                IF denominator(r1) = 0 THEN
                    inValueU(inUndefined())
                ELSE
                LET
                    res=r/r1
                IN
                    IF isDivisible(numerator(res), denominator(res)) THEN
                        inValueU(inInteger(res))
                    ELSE
                        inValueU(inRational(res))
                    END //if-isDivisible
                END //let-res
                END //if-denominator(r1)
            [] isFloat(f) →
                IF f = 0 THEN
                    inValueU(inUndefined())
                ELSE
                    inValueU(inFloat(r/f))
                END //if-f
            [] … → inValueU(inUndefined())
        END //cases-v2
        END //if-denominator(r)
[] isFloat(f) →
        cases v2 of
            isInteger(k) →
                IF k = 0 THEN
                    inValueU(inUndefined())
                ELSE
                    inValueU(inFloat(f/k))
                END //if-k=0
            [] isRational(r) →
                IF denominator(r) = 0 THEN
                    inValueU(inUndefined())
                ELSE
                    inValueU(inFloat(f/r))
                END //if-denominator(r)
            [] isFloat(f1) →
                IF f1 = 0 THEN
                    inValueU(inUndefined())
```

```
                                ELSE
                                        inValueU(inFloat(f+f1))
                                END //if-f1
                        [] … → inValueU(inUndefined())
                END //cases-v2
        [] … → inValueU(inUndefined())
END //cases-v1

mod: Value x Value → ValueU
mod(v1, v2) =
        cases v1 of
                isInteger(j) →
                        cases v2 of
                                isInteger(k) →
                                        IF k = 0 THEN
                                                inValueU(inUndefined())
                                        ELSE
                                                inValueU(inInteger(mod(j,k)))
                                        END //if-k=0
                                [] isRational(r) →
                                        IF denominator(r) = 0 THEN
                                                inValueU(inUndefined())
                                        ELSE
                                        IF isDivisible(numerator(r), denominator(r)) AND
                                                isModularInverse(j,r) THEN
                                                inValueU(inInteger(mod(j,r)))
                                        ELSE
                                                inValueU(inUndefined())
                                        END //if-isDivisible
                                        END //if-denominator(r)
                                [] … → inValueU(inUndefined())
                        END //cases-v2
                [] isRational(r) →
                        cases v2 of
                                isInteger(k) →
                                        IF isDivisible(numerator(r), denominator(r)) AND
                                                isModularInverse(j,r) AND k <> 0 THEN
                                                inValueU(inInteger(mod(r,k)))
                                        ELSE
                                                inValueU(inUndefined())
                                        END //if-isDivisible
                                [] isRational(r1) →
                                        IF isDivisible(numerator(r), denominator(r)) AND
                                                isDivisible(numerator(r1), denominator(r1)) AND
                                                isModularInverse(j,r) AND denominator(r1) <> 0 THEN
                                                inValueU(inInteger(mod(r,r1)))
                                        ELSE
```

```
                                        inValueU(inUndefined())
                                    END //if-isDivisible
                                [] … → inValueU(inUndefined())
                            END //cases-v2
                    [] … → inValueU(inUndefined())
            END //cases-v1

equals: Value x Value → Tr
equals(v1, v2) =
cases v1 of
        isInteger(j) →
                cases v2 of
                        isInteger(k) → inTr(j=k)
                        [] isRational(r) → inTr(j=r)
                        [] isFloat(f) → inTr(j=f)
                        [] … → inTr(False)
                END //cases-v2
        [] isRational(r) →
                cases v2 of
                        isInteger(k) → inTr(r=k)
                        [] isRational(r1) → inTr(r=r1)
                        [] isFloat(f) → inTr(r=f)
                        [] … → inTr(False)
                END //cases-v2
        [] isFloat(f) →
                cases v2 of
                        isInteger(k) → inTr(f=k)
                        [] isRational(r) → inTr(f=r)
                        [] isFloat(f1) → inTr(f=f1)
                        [] … → inTr(False)
                END //cases-v2
        [] isBoolean(b1) →
                cases v2 of
                        isBoolean(b2) → inTr(b1=b2)
                        [] … → inTr(False)
                END //cases-v2
        [] isString(s1) →
                cases v2 of
                        isString(s2) → inTr(s1=s2)
                        [] … → inTr(False)
                END //cases-v2
        [] isList(l1) →
                cases v2 of
                        isList(l2) → inTr(l1=l2)
                        [] … → inTr(False)
                END //cases-v2
        [] isRecord(r1) →
```

```
                cases v2 of
                        isRecord(r2) → inTr(r1=r2)
                        [] … → inTr(False)
                END //cases-v2
        [] isSet(st1) →
                cases v2 of
                        isSet(st2) → inTr(st1=st2)
                        [] … → inTr(False)
                END //cases-v2
        [] isSymbol(sy1) →
                cases v2 of
                        isSymbol(sy2) → inTr(sy1=sy2)
                        [] … → inTr(False)
                END //cases-v2
        [] ... → inTr(False)
END //cases-v1

notequals: Value x Value → Tr
notequals(v1, v2) = not(equals(v1,v2))

lessthan: Value x Value → Tr
lessthan(v1, v2) =
cases v1 of
        isInteger(j) →
                cases v2 of
                        isInteger(k) → inTr(j<k)
                        [] isRational(r) → inTr(j<r)
                        [] isFloat(f) → inTr(j<f)
                        [] … → inTr(False)
                END //cases-v2
        [] isRational(r) →
                cases v2 of
                        isInteger(k) → inTr(r<k)
                        [] isRational(r1) → inTr(r<r1)
                        [] isFloat(f) → inTr(r<f)
                        [] … → inTr(False)
                END //cases-v2
        [] isFloat(f) →
                cases v2 of
                        isInteger(k) → inTr(f<k)
                        [] isRational(r) → inTr(f<r)
                        [] isFloat(f1) → inTr(f<f1)
                        [] … → inTr(False)
                END //cases-v2
        [] isBoolean(b1) →
                cases v2 of
                        isBoolean(b2) → inTr(b1<b2)
```

```
                [] … → inTr(False)
            END //cases-v2
        [] isString(s1) →
            cases v2 of
                isString(s2) → inTr(s1<s2)
                [] … → inTr(False)
            END //cases-v2
        [] isList(l1) →
            cases v2 of
                isList(l2) → inTr(l1<l2)
                [] … → inTr(False)
            END //cases-v2
        [] isRecord(r1) →
            cases v2 of
                isRecord(r2) → inTr(r1<r2)
                [] … → inTr(False)
            END //cases-v2
        [] isSet(st1) →
            cases v2 of
                isSet(st2) → inTr(st1<st2)
                [] … → inTr(False)
            END //cases-v2
        [] isSymbol(sy1) →
            cases v2 of
                isSymbol(sy2) → inTr(sy1<sy2)
                [] … → inTr(False)
            END //cases-v2
        [] ... → inTr(False)
END //cases-v1

greaterthan: Value x Value → Tr
greaterthan(v1, v2) =
cases v1 of
        isInteger(j) →
            cases v2 of
                isInteger(k) → inTr(j>k)
                [] isRational(r) → inTr(j>r)
                [] isFloat(f) → inTr(j>f)
                [] … → inTr(False)
            END //cases-v2
        [] isRational(r) →
            cases v2 of
                isInteger(k) → inTr(r>k)
                [] isRational(r1) → inTr(r>r1)
                [] isFloat(f) → inTr(r>f)
                [] … → inTr(False)
            END //cases-v2
```

```
        [] isFloat(f) →
                cases v2 of
                        isInteger(k) → inTr(f>k)
                        [] isRational(r) → inTr(f>r)
                        [] isFloat(f1) → inTr(f>f1)
                        [] … → inTr(False)
                END //cases-v2
        [] isBoolean(b1) →
                cases v2 of
                        isBoolean(b2) → inTr(b1>b2)
                        [] … → inTr(False)
                END //cases-v2
        [] isString(s1) →
                cases v2 of
                        isString(s2) → inTr(s1>s2)
                        [] … → inTr(False)
                END //cases-v2
        [] isList(l1) →
                cases v2 of
                        isList(l2) → inTr(l1>l2)
                        [] … → inTr(False)
                END //cases-v2
        [] isRecord(r1) →
                cases v2 of
                        isRecord(r2) → inTr(r1>r2)
                        [] … → inTr(False)
                END //cases-v2
        [] isSet(st1) →
                cases v2 of
                        isSet(st2) → inTr(st1>st2)
                        [] … → inTr(False)
                END //cases-v2
        [] isSymbol(sy1) →
                cases v2 of
                        isSymbol(sy2) → inTr(sy1>sy2)
                        [] … → inTr(False)
                END //cases-v2
        [] ... → inTr(False)
END //cases-v1

lessequal: Value x Value → Tr
lessequal(v1, v2) = or(equals(v1,v2), less(v1,v2))

greaterequal: Value x Value → Tr
greaterequal(v1, v2) = or(equals(v1,v2), greater(v1,v2))

numerator: Value → Value
```

denominator: Value → Value
isDivisible: Value x Value → Tr
isModularInverse: Value x Value → Tr

// returns the corresponding type of the given value
getType: Value → Type-TagU

2.1.6) List Values
Domain List = Value*

Operations
emptyList: List
vseq2List: Value* → List
cons: Value x List → List
head: List → Value
tail: List → List
length: List → Nat'

head(cons(v, l)) = v
tail(cons(v, l)) = l

length(emptyList) = 0
length(cons(v, emptyList)) = 1
length(cons(v, l)) = length(l) + 1

permutation \subin List x List
permutation(emptyList, emptyList) <=> inBoolean(True)
permutation(l1, cons(v2, l2)) <=> \exists e \in List: extract(l1, v2, e) AND permutation(e, l2)

extract \subin List x Value x List
extract(cons(v1, l1), v1, l1) <=> inBoolean(True)
extract(cons(v1, l1), v2, cons(v1, l2)) <=> extract(l1, v2, l2)

addElement: Nat' x Value x List → List
addElement(j, v, [a1,a2, ...,aj, …, an]) = [a1,a2,..., aj/v, …, an], if 1<=j<=n

updateList: List x Value* x List → List
update: Integer x Value x List → List

access: Nat' x List → Value
access(j, [a1,a2, ...,aj, …, an]) = aj , if 1<=j<=n
accessAll: List → Value*

listToValSeq: List → Value*
listToValSeq(emptyList) = empty
listToValSeq(cons(v, emptyList)) = v
listToValSeq(cons(v, l)) = <v, listToValSeq(l)>

2.1.7) Unordered Values
Domain Set = List

Operations
emptySet: Set
list2Set: List → Set
cons: Value x Set → Set
length: Set → Nat'

memberOf: Value x Set → Tr
union: Set x Set → Set
intersection: Set x Set → Set
minus-set: Set x Set → Set
updateSet: List x Value* x Set → Set
update: Integer x Value x Set → Set

access: Nat' x Set → Value
access(j, [a1,a2, ..., aj, …, an]) = aj , if 1<=j<=n // the value of aj will not be the same at every
// access because elements of set are the
// permuted.
accessAll: Set → Value*

length(emptySet) = 0
length(cons(v, emptySet) = 1
length(cons(v, l)) = length(l) + 1

2.1.8) Tuple Values
Domain Record= List

Operations
emptyRecord: Record
list2Record: List → Record
cons: Value x Record → Record
length: Record → Nat'
updateRecord: List x Value* x Record → Record
update: Integer x Value x Record → Record

access: Nat' x Record → Value
access(j, [a1,a2, ...,aj, …, an] = aj, if 1<=j<=n

accessAll: Record → Value*

addRecord: Value* x Record → Record

upateElement: Nat' x Value x Record → Record
updateElement(j, v, [a1,a2,...,aj, …, an]) = [a1,a2,..., aj/v, …, an], if 1<=j<=n

length(emptyRecord) = 0
length(cons(v, emptyRecord)) = 1
length(cons(v, r)) = length(r) + 1

2.1.9) Sequence Values
Domain Value*

Operations
emptyValue: Value*
cons: Value x Value* → Value*

// function generates a sequence of integer values from the former integer value to the latter value
buildRangeValSeq: Integer x Integer → Value*

length: Value* → Nat
length(empty) = 0
length(<v,vs>) = length(<vs>) + 1

access: Integer x Value* → Value
access: Integer x Integer x Value* → Value
updateValue: Value* x Value x Value* → Value*
update: Integer x Value x Value* → Value*
append: Integer x Value x Value* → Value*

2.1.10) Procedure Values
Domain Procedure = P(Value* x State x StateU x ValueU x Type-Tag x Type-Tag*)

2.1.11) Module Values
Domain Module = P(IdentifierSeq x ValueU)

Operations
moduleValue: Nat x Module → ValueU
moduleValue(j, <iseq, vseq>) =
    cases access(j, vseq) of
        isModule(m) → inUndefined()
        [] … (v') → inValueU(v')
    END //cases-v

evalMProc: Value x Value* x State → State x ValueU
evalMProc(v, vseq, s) = cases v of
        isProcedure(p) →
            LET
                p \in Procedure
                \exists s' \in StateU, v' \in ValueU: p(vseq, s, s', v')
            IN
                (s', v')

```
                                    END //let-
                                [] … → inUndefined()
                            END //cases-v

evalMValue: Value → ValueU
evalMValue(v) =     cases v of
                            isModule(p) →inUndefined()
                            [] isProcedure(p) → inUndefined()
                            [] … → inValueU(v)
                        END //cases-v
```

2.1.12) Identifiers
Domains Identifier, IdentifierSeq

Operations

```
length: IdentifierSeq → Nat
length(empty) = 0
length(I,Iseq) = 1+length(Iseq)

indexOf: Identifier x IdentifierSeq → Nat
indexOf(I, empty) = 0
indexOf(Ij, <I1,I2, …, Ij, …, In>) = j          , if 0 < j <= n
indexOf(Im, <I1,I2, …, Ij, …, In>) = 0          , if m > n
```

2.1.13) Symbol Value
Domain Symbol

Operations

```
length: Symbol → Nat'
length(s) = 1
```

2.1.14) Character Strings
Domain String

Operations

```
A, B, C … Z:String
emptyString: String
concat: String x String → String
length: String → Nat'
length(st) = 1

substring : String x String → Tr
substring(x,y) =    inTr(True),     if x is an initial substring of y
                    inTr(False),    else

updateString: List x Value* x String → String
update: Integer x Value x String → String
```

lengthOfPlaceHldrs: String → Nat

replacePlaceHolders: String x State → String
replacePlaceHolders("...%1 … %j … %n ...", s) =
LET
vseq = (data(s)↓2)↓2
IN
"...%1/access(1,vseq) … %j/access(j,vseq) … %n/access(n,vseq) ..."
END //let

hasPlaceHolders \subin String → Tr

2.1.15) Unevaluated Values
Domain Uneval

Operations
length: Uneval → Nat'
length(u) = 1

eval: Uneval → ValueU
eval("E") → inValueU(inUneval('E'))
eval('E') → inUndefined()

2.1.16) Lifted Value domain
Domains ValueU = Value + Undefined, Undefined = Unit, StateU = State + Error, Error = Unit

Operations
hasUndefinedValue : ValueU* → Tr
hasUndefined(empty) <=> inTr(True)
hasUndefinedValue(<v,vseq>) <=> inValue(v) AND hasUndefinedValue(vseq)

2.1.17) Parameter Values
Domains Parameter, ParameterSeq

Operations
identifiers: Pseq → IdentifierSeq
identifiers(empty) = empty
identifiers((P,Pseq)) = identifier(P), identifiers(Pseq)

identifier: P → Identifier
identifier(I) → I
identifier(I::M) → I

2.1.18) Declaration Values
Domain S

Operations
getExported: S → Identifier Sequence
getExported(local …) = empty
getExported(global …) = empty
getExported(uses …) = empty
getExported(export I,Iseq) = I,Iseq

2.1.19) Type-Tag Values
Domain Type-Tag = ADT-Tag + Integer-Tag + Rational-Tag + Float-Tag + Boolean-Tag + String-Tag + Type-Tag* …
where ADT-Tag = Integer-Tag = Rational-Tag = Float-Tag = …. = Unit and
List-Tag = Set-Tag = … = Type-Tag
Record-Tag = Type-Tag*
Procedure-Tag = Type-Tag x Type-Tag

Type-TagU = Type-Tag + Error-Tag

Operations
cons: Type-Tag x Type-Tag → Type-Tag
cons: Type-Tag x Type-Tag x Type-Tag→ Type-Tag

emptyList-Tag: Type-Tag
emptySet-Tag: Type-Tag
emptyRecord-Tag: Type-Tag
emptyOr-Tag: Type-Tag
emptyProcedure-Tag: Type-Tag
…

hasErrorTag : Type-TagU* → Tr
hasErrorTag(empty) <=> inTr(True)
hasErrorTag(<t,tseq>) <=> inType-Tag(t) AND hasErrorTag(tseq)

access: Integer x Type-Tag* → Type-Tag
isTypeSeq: Type-Tag* x Value* → Boolean

isType: Type-Tag x Value → Boolean
isType(tag, val) =
cases tag of
    isInteger-Tag() →
        cases val of
            isInteger(j) → true
            [] isRational(r) → true
            [] … → false
        END //cases-val
    [] isRational-Tag() →
        cases val of
            isInteger(j) → true

```
                [] isRational(r) → true
                [] … → false
        END //cases-val
[] isFloat-Tag() →
        cases val of
                isFloat(f) → true
                [] isRational(r) → true
                [] … → false
        END //cases-val
[] isBoolean-Tag() →
        cases val of
                isBoolean(b) → true
                [] … → false
        END //cases-val
[] isString-Tag() →
        cases val of
                isString(st) → true
                [] … → false
        END //cases-val
[] isList-Tag(lt) →
        cases val of
                isList(list)→\forall x:1 >= x AND x <= length(list) AND isType(lt, access(x,list))
                [] … → false
        END //cases-val
[] isRecord-Tag(rt) →
        cases val of
        isRecord(r)→\forall x:1 >= x AND x <= length(list) AND isType(rt↓x, access(x,r))
        [] … → false
        END //cases-val
[] isSet-Tag(st) →
        cases val of
                isSet(st)→\forall x:1 >= x AND x <= length(list) AND isType(st, access(x,st))
                [] … → false
        END //cases-val
[] isSymbol-Tag() →
        cases val of
                isSymbol(sy) → true
                [] … → false
        END //cases-val
[] isUneval-Tag() →
        cases val of
                isUneval(u) → true
                [] … → false
        END //cases-val
[] isOr-Tag(tags) →
        cases val of
```

```
            [] … (v) → \exists x: 1 >= x AND x <=length(tags) AND isType(access(x, tags), v)
            END //cases-val
        [] isProcedure-Tag(ptag) →
            cases val of
                isProcedure(p)→
                    cases p↓3  of
                        isUndefined() → false
                        [] isValue(v) →
                            IF hasUndefinedValue(p↓1) THEN
                                false
                            ELSE
                                isType(ptag↓1, v) AND isTypeSeq(ptag↓2, p↓1)
                            END //if-hasUndefinedValue
                    END //cases-p3
                [] … → false
            END //cases-val
        ...
        [] isAnything-Tag() → true
END //cases-tag
```

2.1.20) Mathematical Function Values
Domain Function = U_{n \in N} Function^n
Function^n = (Value^n) → Value

2.2 Signatures of Valuation Functions for *MiniMaple Specification Language (other than MiniMaple)*

StateEnvRelation = P(Environment x State x StateU)
EnvRelation = Environment → StateEnvRelation
BindRelation = Environment → P(State x StateU x Value*)
StateResultValueRelation = P(State x StateU x Value x ValueU)
StateResultValueSeqRelation =P(State x StateU x Value x ValueU*)
SpecExpRelation = Environment → StateResultValueRelation
SpecExpSeqRelation = Environment → StateResultValueSeqRelation

2.2.1) For Specification Expression:
[[spec-expr]] : SpecExpRelation

2.2.2) For Specification Expression Sequence:
[[eseq]] : SpecExpSeqRelation

2.2.3) For Binding:
[[binding]] : BindRelation

2.2.4) For Identifier Typed:

[[Idt]] : Environment → Environment

2.2.5) For Identifeir Typed Sequence:
[[Idtseq]] : Environment → Environment

2.3 Auxiliary Functions and Predicates

2.3.1) // seq ...
seq \subin Nat' x Identifier x Environment x Value* x Value* x StateResultValueRelation

seq(i, I, e, vseq, vs, spec-expr) <=>
\exists e1 \in Environment: e1 = push(e, I, access(i,vseq)) AND
\forall s \in State, r \in Value:
\exists v' \in Value: spec-expr(e1)(s,inStateU(s),r,inValueU(v')) AND access(i,vs) = v'

2.3.2) // iterator operations ...
iterate \subin
Nat' x Identifier x Environment x Value* x Value* x
StateResultValueRelation1 x StateResultValueRelation2 x Integer
iterate(i, I, e, vseq, vs, spec-expr1, spec-expr2, k) <=>
\exists e1 \in Environment: e1 = push(e, I, access(i,vseq)) AND
\forall s \in State, r \in Value: \exists v' \in Value:spec-expr1(e1)(s,inStateU(s),r,inValueU(v'))
AND spec-expr2(e1)(s,inStateU(s),r,inValueU(inBoolean(inTrue()))) AND access(i,vs) = v'

2.3.3) For special functions

equalsOperator \subin Operator x Operator → Tr
equalsOperator(o1,o2) <=> IF o1=+ AND o2=+ THEN inTr(True)
ELSE IF o1=- AND o2=- THEN inTr(True)
ELSE IF o1=/ AND o2=/ THEN inTr(True)
ELSE IF o1=* AND o2=* THEN inTr(True)
ELSE IF o1=mod AND o2=mod THEN inTr(True)
ELSE IF o1=< AND o2=< THEN inTr(True)
ELSE IF o1=> AND o2=> THEN
inTr(True)
ELSE IF o1=<= AND o2=<= THEN
inTr(True)
ELSE IF o1=>= AND o2=>= THEN
inTr(True)
ELSE inTr(False)
END //if->=
END //if-<=
END //if->
END //if-<

```
                                        END //if-mod
                                    END //if-*
                                END //if-/
                            END //if--
                        END //if-+

subsop: Integer x Value x Value → ValueU
subsop(j, v1, v2) =
    IF j > 0 AND j <= length(v2) THEN
        cases v2 of
            isList(list) → inValueU(update(j, v1, list))
            [] isRecord(r) → inValueU(update(j, v1, r))
            [] isSet(s) → inValueU(update(j, v1, s))
            [] isValue*(vs) → inValueU(update(j, v1, vs)
            [] isInteger(k) → inValueU(k)
            [] isRational(rat) → inValueU(rat)
            [] isFloat(f) → inValueU(f)
            [] isBoolean(b) → inValueU(b)
            [] isString(st) → inValueU(st)
            [] isSymbol(sm) → inValueU(sm)
            [] isUneval(u) → inValueU(u)
            [] … → inUndefined()
        END //cases-v2
    ELSE
        inUndefined()
    END //if
```

2.3.4) Miscellaneous

```
// function returns the identifiers representing their corresponding types from the specification
// declaration
getTypeIdentifiersAndTypes: Declaration → Identifier Sequence x Type Sequence

// function returns the function identifiers and their corresponding return types from the
// specification declaration
getFunctionIdentifiersAndTypes: Declaration → Identifier Sequence x Type Sequence

// function returns the axioms in the declaration as a specification expression sequence
getAxioms → Declaration → Specification Expression Sequence

// function extracts all the rules from the specification declaration
getRules → Declaration → Rules

// function returns true, if the former state is equal to later state except for the values of the
// identifiers given in the Identifier Sequence, otherwise returns false
equalsExcept: State x State x Identifier Sequence → Boolean
```

getIdentifier: Binding → Identifier
getIdentifier(I = spec-expr1 … spec-expr2) = I
getIdentifier(I **in** spec-expr) = I

// function extracts the identifiers from the given identifier typed sequence
getIdentifiers: Identifier Typed Sequence → Identifier Sequence

// function extracts the identifiers and their corresponding types from the given identifier
// typed sequence
getIdentifiersAndTypes: Identifier Typed Sequence → Identifier Sequence x Type Sequence

// function extracts the identifiers and corresponding types from the given parameter sequence
getIdentifiersAndTypes: Parameter Sequence → Identifier Sequence x Type Sequence

// returns the length of a given Identifier Sequence
length: Identifier Sequence → N

// returns the length of a given Identifier Typed Sequence
length: Identifier Typed Sequence → N

// extracts all the identifiers (that appear in a given specification expression) and their corresponding
// types
getExpressionIdentifiersAndTypes: Specification Expression → Identifier Sequence x Type Sequence

// extracts all the identifiers (that appear in a given specification expression sequence) and their
// corresponding types
getExpressionSequenceIdentifiersAndTypes: Specification Expression Sequence→
Identifier Sequence x Type Sequence

getOp: it-op → String
getOp(add) = add
getOp(mul) = mul
getOp(max) = max
getOp(min) = min

doIterate: String x Value* → ValueU
doIterate(add, vseq) = addSeq(vseq)
doIterate(mul, vseq) = mulSeq(vseq)
doIterate(max, vseq) = maxSeq(vseq)
doIterate(min, vseq) = minSeq(vseq)

// performs an arithmetic add operation on the values of a given sequence (of values) and returns the
// result
addSeq: Value* → ValueU

// performs corresponding multiplication operation on the values of a given sequence (of values) and
// returns the result

mulSeq: Value* → ValueU

// computes a maximum value among the values of a given sequence
maxSeq: Value* → ValueU

// computes a minimum value among the values of a given sequence
minSeq: Value* → ValueU

2.4 Definition of Valuation Functions

2.4.1) CASE: Specification Expression

```
[[ type(spec-expr, T) ]](e)(s,s',r,v) <=>
\exists v1, v2 \in Value, tag \in Type-Tag: [[ spec-expr ]](e)(s,s',r,inValueU(v1))
        AND isIdentifier(inValue(v1)) AND [[T]](e)(tag) AND v2 = [[expr2Identifier(v1)]](e)
        AND v = inValueU(inBoolean(isType(tag,v2)))

[[ spec-expr1 and spec-expr2 ]](e)(s,s',r,v') <=>
\exists b1, b2 \in Boolean: [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b1)))
        AND [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b2)))
        AND v'=inValueU(inValue(and(b1,b2)))

[[ spec-expr1 or spec-expr2 ]](e)(s,s',r,v) <=>
\exists b1, b2 \in Boolean: [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b1)))
        AND [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b2)))
        AND v'=inValueU(inValue(or(b1,b2)))

[[ spec-expr1 implies spec-expr2 ]](e)(s,s',r,v) <=>
\exists b1, b2 \in Boolean: [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b1)))
        AND [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b2)))
        AND v'=inValueU(inValue(implies(b1,b2)))

[[ spec-expr1 equivalent spec-expr2 ]](e)(s,s',r,v) <=>
\exists b1, b2 \in Boolean: [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b1)))
        AND [[spec-expr1]](e)(s,s',r,inValueU(inValue(inBoolean(b2)))
        AND v'=inValueU(inValue(and(b1,b2)))

[[ forall (Itseq, spec-expr) ]](e)(s,s',r,v) <=>
\exists s1 \in StateU, e1 \in Environment: [[Itseq]](e)(e1) AND
                LET
                        (iseq, Tseq) = getIdentifiersAndTypes(Itseq)
                        \exists b \in Boolean: AND
                        b = ( \forall valseq \in [[Tseq]], v' \in ValueU, e' \in Environment:
                                e' = push(e, iseq, valseq) AND [[spec-expr]](e')(s,s',r,v') AND
                                v' = inValueU(inBoolean(inTrue()))
                            )
                IN
                        v=inValueU(inValue(b))
                END //let-(iseq,Tseq)

[[ exists (Itseq, spec-expr) ]](e)(s,s',v) <=>
\exists s1 \in StateU, e1 \in Environment: [[Itseq]](e)(e1) AND
                LET
                        (iseq, Tseq) = getIdentifiersAndTypes(Itseq)
```

```
                        \exists b \in Boolean: AND
                        b = ( \exists valseq \in [[Tseq]], v' \in ValueU, e' \in Environment:
                                e' = push(e, iseq, valseq) AND [[spec-expr]](e')(s,s',r,v') AND
                                v' = inValueU(inBoolean(inTrue()))
                            )
                IN
                        v=inValueU(inValue(b))
                END //let-(iseq,Tseq)

[[ it-op (spec-expr1, binding, spec-expr2) ]](e)(s,s',r,v) <=>
\exists vseq \in Value*: [[binding]](e)(s,s',r,inValueU(vseq)) AND
        \exists k' \in Integer, e1 \in Environment, vs \in Value*: e1=push(e,getIdentifier(binding)) AND
                ( \forall i \in Nat'_k:iterate(i,I,e1,vseq,vs,[[spec-expr1]],[[spec-expr2]]) ) AND
                        ( k' < length(vseq)
                                AND
                                ( access(k', vseq)=isUndefined()
                                OR \forall s \in State, r \in Value:
                                        \exists v1\in Value, n \in StateU:
                                                [[spec-expr2]](e1)(s,inStateU(s),r,inValueU(v1))
                                                        AND inBoolean(v1) = inFalse()
                                )
                                AND v=inUndefined()
                        )
                        OR
                        ( k' = length(vs) AND v=doIterate(getOp(it-op),vs) )

[[LET Iseq=eseq IN spec-expr ]](e)(s,s',r,v') <=>
\exists vs \in ValueU*: [[eseq]](e)(s,s',r,vs) AND
                IF hasUndefinedValue(vs) THEN
                        v'=inUndefined()
                ELSE
                        \exists e1 \in Environment, s2 \in State: e1 = push(e, Iseq, vs) AND
                                AND [[spec-expr2]](e1)(s,s',v')
                END //if-hasUndefinedValue

[[ 'if'(spec-expr1, spec-expr2, spec-expr3 ) ]](e)(s,s',r,v') <=>
\exists v1 \in ValueU: [[spec-expr1]](e)(s,s',r,v1) AND
        IF v1=inValueU(inValue(inBoolean(inTrue()))) THEN
                [[spec-expr2]](s,s',r,v')
        ELSE
                [[spec-expr3]](s,s',r,v')
        END //if-b1=inTrue()

[[ RESULT ]](e)(s,s',r,v') <=> v'=inValueU(r) AND s'=inStateU(s)

[[ OLD I ]](e)(s,s',r,v') <=>
v' = inValueU(store(s)([[I]](e))) AND s'=inStateU(s)
```

```
[[ I(eseq) ]](e)(s,s',r,v) <=>
LET vseq \in ValueU*: [[eseq]](e)(s,s',r,vseq)
IN
        IF hasUndefinedValue(vseq) THEN
                v=inUndefined()
        ELSE
                cases [[I]](e) of
                        isFunction(f) → \exists ax \in Axiom^n, v' \in Value: n=length(vseq) AND
                                [[ax]](e)(inValueU(v')) AND f(vseq)=v' AND v=inValueU(v')
                                AND s'=inStateU(s)
                        [] … → v=inUndefined()
                END //cases-[[I]]
        END //if-hasUndefinedValue
END //let-vseq

[[ I1:-I2 ]](e)(s,s',r,v) <=>
LET
        v' = store(s)[[I1]](e)
IN
        cases v' of
                isModule(m) →
                        LET
                                j = indexOf(I2, iseq)
                        IN
                                IF j > 0 THEN
                                        cases moduleValue(j,  m) of
                                                isUndefined() → v=inUndefined()
                                                [] isValue(mv) →
                                                        cases evalMValue(mv) of
                                                                isUndefined() → v=inUndefined()
                                                                [] isValue(v1) → v=inValueU(v1)
                                                        END //cases-evalMValue
                                        END //cases-mv
                                ELSE
                                        v=inUndefined()
                                END //if-hasIdentifier
                        END //let-index
                [] … → v=inUndefined()
        END //cases-v'
END //let-v'

[[ I1:-I2(eseq) ]](e)(s,s',r,v) <=>
\exists vs \in Value*: [[eseq]](e)(s,s',r, vs) AND
        IF hasUndefinedValue(vseq) THEN
                v=inUndefined()
        ELSE
```

```
        LET
                v' = store(s)[[I1]](e)
        IN
        cases v' of
                isModule(m) →
                        LET
                                j = indexOf(I2, iseq)
                        IN
                                IF j > 0 THEN
                                        cases moduleValue(j,  m) of
                                                isUndefined() → v=inUndefined()
                                                [] isValue(mv) →
                                                        LET    sm \in StateU, vm \in ValueU
                                                                (sm, vm) = evalMProc(mv)
                                                        IN
                                                                v=vm
                                                        END //let-
                                                END //cases-mv
                                ELSE
                                        v=inUndefined()
                                END //if-hasIdentifier
                        END //let-index
                [] … → v=inUndefined()
        END //cases-v'
        END //let-v'
        END //if-hasUndefinedValue

[[ spec-expr1 = spec-expr2 ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr1]](e)(s,s',r,v1) AND
        cases v1 of
                isUndefined() → v=inUndefined()
                [] isValue(v11) → \exists v2 \in ValueU: [[spec-expr2]](e)(s, s',r, v2) AND
                        cases v2 of
                                isUndefined() → v=inUndefined()
                                [] isValue(v22) → v=inValueU(equals(v11,v22))
                        END //cases-v2
        END //cases-v1

[[ spec-expr1 <> spec-expr2 ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr1]](e)(s,s',r,v1) AND
        cases v1 of
                isUndefined() → v=inUndefined()
                [] isValue(v11) → \exists v2 \in ValueU: [[spec-expr2]](e)(s, s', r, v2) AND
                        cases v2 of
                                isUndefined() → v=inUndefined()
                                [] isValue(v22) → v=inValueU(notequals(v11,v22))
                        END //cases-v2
```

END //cases-v1

Case: Bop

[[Bop]](v',v'')(v) <=>
IF equalsOperator(Bop,+) THEN v=inValueU(add(v',v''))
ELSE IF equalsOperator(Bop,-) THEN v=inValueU(sub(v',v''))
ELSE IF equalsOperator(Bop,/) THEN v=inValueU(div(v',v''))
ELSE IF equalsOperator(Bop,*) THEN v=inValueU(mul(v',v''))
ELSE IF equalsOperator(Bop,mod) THEN v=inValueU(mod(v',v''))
ELSE IF equalsOperator(Bop,<) THEN v=inValueU(less(v',v''))
ELSE IF equalsOperator(Bop,>) THEN v=inValueU(greater(v',v''))
ELSE IF equalsOperator(Bop,<=) THEN v=inValueU(lessequal(v',v''))
ELSE IF equalsOperator(Bop,>=) THEN v=inValueU(greaterequal(v',v'')) END //if->=
END //if-<=
END //if->
END //if-<
END //if-mod
END //if-*
END //if-/
END //if--
END //if-+

[[spec-expr1 Bop spec-expr2]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr1]](e)(s,s',r,v1) AND
cases v1 of
isUndefined() → v=inUndefined()
[] isValue(v11) → \exists v2 \in ValueU: [[spec-expr2]](e)(s, s', r, v2) AND
cases v2 of
isUndefined() → v=inUndefined()
[] isValue(v22) →\exists v' \in Value: [[Bop]](v11, v22)(v') AND
v=inValueU(v')
END //cases-v2
END //cases-v1

Case: Uop

[[Uop]](v')(v) <=>
IF equalsOperator(Uop,+) THEN v=inValueU(plus(v'))
ELSE IF equalsOperator(Uop,-) THEN v=inValueU(minus(v'))
ELSE IF equalsOperator(Uop,not) THEN v=inValueU(not(v'))
ELSE v=inUndefined()
END //if-not
END //if--
END //if-+

[[Uop spec-expr]](e)(s,s',r,v) <=>

```
\exists v1 \in ValueU: [[spec-expr]](e)(s,s',r,v1) AND
        cases v1 of
                isUndefined() → v=inUndefined()
                [] isValue(v'') → \exists v' \in Value: [[Uop]](v'')(v') AND v=inValueU(v')
        END //cases-v1
```

Case: Special Expressions

```
// list construction
[[ [eseq] ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[eseq]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v'') →
                        cases v'' of
                                isValue*(vs) → IF hasUndefinedValue(vs) THEN
                                                        v=inUndefined()
                                                ELSE
                                                        v=inValueU(vseq2List(vs))
                                                END //if-hasUndefinedValue
                                [] … → v=inUndefined()
                        END //cases-v''
        END //cases-v'

// record construction
[[ [eseq] ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[eseq]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v'') →
                        cases v'' of
                                isValue*(vs) → IF hasUndefinedValue(vs) THEN
                                                        v=inUndefined()
                                                ELSE
                                                        v=inValueU(list2Record(vseq2List(vs)))
                                                END //if-hasUndefinedValue
                                [] … → v=inUndefined()
                        END //cases-v''
        END //cases-v'

// set construction
[[ {eseq} ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[eseq]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v'') →
                        cases v'' of
```

```
                        isValue*(vs) → IF hasUndefinedValue(vs) THEN
                                            v=inUndefined()
                                       ELSE
                                      \exists pl \in List: permutation(cons(vs,emptyList), pl)
                                            AND v=inValue(list2Set(pl))
                                       END //if-hasUndefinedValue
                        [] … → v=inUndefined()
                  END //cases-v"
        END //cases-v'

[[ “spec-expr” ]](e)(s,s',r,v) <=> IF hasPlaceHolders(“spec-expr”) THEN
                                        v=inValueU(replacePlaceHolders(“spec-expr”,s))
                                   ELSE
                                        v=inValueU(concat(“spec-expr”,emptyString))
                                   END //if-hasPlaceHolders

[[ op(spec-expr1,spec-expr2) ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[spec-expr2]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v1) →
                        LET
                                vseq = expValues(v1)
                                k = length(vseq)
                        IN
                        \exists v3 \in ValueU: [[spec-expr1]](e)(s, s', r, v3) AND
                                cases v3 of
                                        isUndefined() → v=inUndefined()
                                        [] isValue(v33) →
                                        cases v33 of
                                                isInteger(n) →
                                                IF n > 0 AND n <= k THEN
                                                                v=inValueU(access(n,vseq))
                                                ELSE v=inUndefined() END
                                                [] … → v=inUndefined()
                                        END //cases-v33
                        END //let-vseq
        END //cases-v'

[[ op(spec-expr) ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[spec-expr]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v1) → v' = expValues(v1)
        END //cases-v'

[[ op(spec-expr1...spec-expr2,spec-expr3) ]](e)(s,s',r,v) <=>
```

```
\exists v' \in ValueU: [[spec-expr3]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v1) →
                                LET
                                        vseq = expValues(v1)
                                        k = length(vseq)
                                IN
                                \exists v3 \in ValueU: [[spec-expr1]](e)(s, s', r, v3) AND
                                        cases v3 of
                                                isUndefined() → v=inUndefined()
                                                [] isValue(v33) →
                                                cases v33 of
                                                        isInteger(n) →
                                \exists v4 \in ValueU: [[spec-expr2]](e)(s, s', r, v4) AND
                                        cases v4 of
                                                isUndefined() → v=inUndefined()
                                                [] isValue(v44) →
                                                        isInteger(m) →
                                                        IF n <= m AND n > 0 AND m <= k THEN
                                                                v=inValueU(access(n, m ,vseq))
                                                        ELSE v=inUndefined() END
                                                        [] … → v=inUndefined()
                                        END //cases-v4
                                                        [] … → v=inUndefined()
                                                END //cases-v33
                                        END //cases-v3
                                END //let-vseq
        END //cases-v'

[[ nops(spec-expr) ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[spec-expr]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v=inUndefined()
                [] isValue(v1) → v=inValueU(length(v1))
        END //cases-v'

[[ subsop(spec-expr1=spec-expr2,spec-expr3) ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr3]](e)(s, s', r, v1) AND
        cases v1 of
                isUndefined() → v=inUndefined()
                [] isValue(v11) →
                                \exists v2 \in ValueU: [[spec-expr1]](e)(s, s', r, v2) AND
                                        cases v2 of
                                                isUndefined() → v=inUndefined()
                                                [] isValue(v22) →
                                                cases v22 of
```

```
                                        isInteger(j) →
                                    \exists v3 \in ValueU: [[spec-expr2]](e)(s, s', r, v3)
                                    AND
                                    cases v3 of
                                        isUndefined()→ v=inUndefined()
                                        [] isValue(v33) →
                                            LET    vs \in ValueU
                                                   vs = subsop(j, v33, v11)
                                            IN
                                            cases vs of
                                            isUndefined() →v=inUndefined()
                                            [] isValue(vs1) →v=inValueU(vs1)
                                            END //cases-vs
                                            END //let-vs
                                    END //cases-v3
                                        [] … → v=inUndefined()
                                    END //cases-v22
                                END //cases-v2
        END //cases-v1

[[ subs(I=spec-expr1,spec-expr2) ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr1]](s, s', r, v1) AND
        cases v1 of
            isUndefined() → v=inUndefined()
            [] isValue(v1) → \exists e1 \in Environment: e1 = push(I, e1,v1) AND
                                    [[spec-expr2]](e1)(s, s', r, v)
        END //cases-v1

[[ "spec-expr" ]](e)(s,s',r,v) <=> v=inValueU(inUneval("spec-expr'))

[[ eval(I,1) ]](e)(s,s',r,v) <=>
            LET
                v' = store(s)([[I]](e))
            IN
                cases v' of
                    isUneval(u) →
                        cases eval(u) of
                            isUneval(u')→v=inValueU(u')
                            [] … → v=inUndefined()
                        END //cases-eval
                    [] … → v=inUndefined()
                END //cases-v'
            END //let-in

[[ seq(spec-expr1,I=spec-expr2...spec-expr3) ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[spec-expr2]](e)(s,s',r,v') AND
        cases v' of
```

```
                isUndefined() → v = inUndefined()
                [] isValue(v'') →
                        cases v'' of
                                isInteger(m)→\exists v'' \in ValueU:[[spec-expr3]](e)(s,s',r,v'') AND
                        cases v''' of
                                isUndefined() → v=inUndefined()
                                isValue(v4) →
                                cases v4 of
                                isInteger(n) →
                        LET
                                vseq = expRangeValues(m,n)
                        IN
                        \exists k' \in Integer, e1 \in Environment, vs \in Value*: e1 = push(I,e) AND
                                ( \forall i \in Nat'_k: seq(i, I, e1, vseq, vs, [[spec-expr1]]) ) AND
                                        ( k' < length(vseq) AND v=inUndefined() )
                                        OR
                                        ( k' = length(vs) AND v=inValueU(vs) )
                        END //let-vseq
                                [] .. → v=inUndefined()
                                END //cases-v4
                        END //cases-v'''
                        [] … → v=inUndefined()
                        END //cases-v''
        END //cases-v'

[[ seq(spec-expr1, I in spec-expr2) ]](e)(s,s',r,v) <=>
\exists v' \in ValueU: [[spec-expr2]](e)(s,s',r,v') AND
        cases v' of
                isUndefined() → v = inUndefined()
                [] isValue(v'') →
                        LET
                                vseq = expValues(v'')
                        IN
                                IF hasUndefinedValue(vseq) THEN
                                        v=inUndefined()
                                ELSE
                        \exists k' \in Integer, e1 \in Environment, vs \in Value*:e1 = push(I,e)AND
                                ( \forall i \in Nat'_k: seq(i,I,e1,vseq,vs,[[spec-expr1]]) ) AND
                                        ( k' < length(vseq) AND access(k', vseq)=isUndefined()
                                                        AND v=inUndefined()
                                        ) OR
                                        ( k' = length(vs) AND v=inValueU(vs) )
                                END //if-hasUndefinedValues
                        END //let-vseq
        END //cases-v'
```

2.4.2) CASE: Binding

```
[[ I = spec-expr1 … spec-expr2 ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[spec-expr1]](e)(s,s',r,v1) AND
                cases v1 of
                        isUndefined() → v=emptyValue()
                        isValue(v11) →
                                cases v11 of
                                        isInteger(i)→\exists 2 \in ValueU:[[spec-expr2]](e)(s,s',r,v2) AND
                                                cases v2 of
                                                        isUndefined() → v=emptyValue()
                                                        isValue(v22) →
                                                                cases v22 of
                                                                        isInteger(j) → greaterthan(j,i)
                                                                        AND s'=inStateU(s)
                                                                        AND v=buildRangeValSeq(i,j)
                                                                        [] … → v=emptyValue()
                                                                END //cases-v22
                                                END //cases-v2
                                END //cases-v11
                END //cases-v1

[[ I in spec-expr ]](e)(s,s',r,v) <=>
\exists v1 \in ValueU: [[ spec-expr ]](e)(s,s',r,v1) AND
        cases v1 of
                isUndefined() → v=emptyValue()
                isValue(v11) →
                        LET
                                vseq=expValues(v11)
                        IN
                                IF hasUndefinedValue(vseq) THEN
                                        v=emptyValue()
                                ELSE
                                        v=inValueU(vseq)
                                END //if-vseq
                        END //let-vseq
        END //cases-v1
```

CASE: Identifier Typed Sequence

Can be easily practiced.

CASE: Identifier Typed

Can be easily rehearsed.

CASE: Identifier Sequence

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

CASE: Identifier

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

CASE: Selection Operator

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

CASE: Type Sequence

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

CASE: Type

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

CASE: Numeral

Same as for MiniMaple, please see the corresponding formal semantics of MiniMaple

3.1 Signatures of Valuation Functions for Specification Semantics

3.1.1) Specification Declaration

[[proc-spec]] : Environment → Environment

For Rules:

[[rules]] : Environment → Value

3.1.2) Method Specification

[[proc-spec]] : P(Environment)

For Exception Clause:

[[excep-clause]] : SpecExpRelation

3.1.3) Loop Specification

[[loop-spec]] : Environment → P(State x StateU)

3.1.4) Assertion

[[asrt]] : Environment → P(State)

3.2 Definition of Valuation Functions for Specification Semantics

3.2.1) CASE: Declaration

```
[[ decl ]](e)(e') <=>
LET
        (id1,...,idn, T1,...,Tn) = getFunctionIdentifersAndTypes(decl)
        (iseq1,...,iseqn, Tseq1,...,Tseqn) = getFunctionParametersAndTypes(decl)
        (i1,...,in, Td1,...,Tdn) = getTypeIdentifiersAndTypes(decl)
        (ax1,...,axn) = getAxioms(decl)
        (r1,...,rn) = getRules(decl)

IN
        \exists f1,...,fn = Function^n1,...,Function^nn, n1,...,nn \in Integer, tag1,...,tagn \in Type-Tag,
                e1,...,en \in Environment:
                n1=length(iseq1) AND … AND nn=length(iseqn) AND
                [[Td1]](e)(inType-TagU(tag1)) AND e1=push(e,i1,tag1) AND … AND
                [[Tdn]](en-1)(inType-TagU(tagn)) AND en=push(e,in,tagn) AND
                e' = push(en, id1,...,idn, f1,...,fn) AND [[r1]](e') AND … AND [[rn]](e')
                AND
                ( \forall b1,...,bn \in Boolean, s \in State, r \in Value:
                        [[ax1]](e')(s,inStateU(s),r,inValueU(inValue(b1))) AND ...
                        [[axn]](e')(s,inStateU(s),r,inValueU(inValue(bn)))
                 => b1 = inTrue() AND … AND bn=inTrue()
                )
END //let-in

[[ EMPTY ]] = <>

[[ define(I(Itseq)::T, rules) ]] = <I, Itseq, T, rules>

[[ `type/I` ]] = I

[[ `type/I`:=T ]] = <I,T>

[[ assume(spec-expr) ]] = spec-expr
```

CASE: Rules

```
[[ EMPTY ]](e) <=> True

[[ I(Itseq) = spec-expr ]](e) <=>
LET     f = [[I]](e)
        n = length(Itseq)
        (iseq, Tseq) = getIdentifersAndTypes(Itseq)
IN
```

```
        \forall vals \in [[Tseq]], s \in State, r \in Value, e' \in Environment: e' = push(e, iseq, vals) AND
                [[spec-expr]](e')(s,inStateU(s),r,inValueU(v')) => f(vals)=v'
END //let-in
```

3.2.2) CASE: Method Specification

```
[[requires spec-expr1; global Iseq; ensures spec-expr2; excep-clause; proc(Pseq)::T S;R end]](e) <=>
LET    (iseq,Tseq) = getIdentifiersAndTypes(Pseq)
IN
        \forall valseq \in [[Tseq]], e1 \in Environment, s1,s2 \in State,
                v,r \in Value, b, b1 \in Boolean: e1=push(e, iseq, valseq) AND
                [[spec-expr1]](e1)(s1,inStateU(s1), r, inValueU(inValue(b))) AND b = inTrue() AND
                \exists p \in Procedure, tag \in Type-Tag, tagseq \in Type-Tag*:
                        [[proc(Pseq)::T; S;R;]](e1)(s1,inStateU(s1),inValueU(inValue(p)))
                AND p(valseq, s1, inStateU(s2), inValueU(v), tag, s-tag) AND isType(v,tag)
        => equalsExcept(s1,s2,Iseq) AND
            IF exceptions(data(s2)) THEN
                [[excep-clause]](e1)(s2,inStateU(s2),v,inValueU(inValue(b1))) AND b1=inTrue()
            ELSE
                [[spec-expr2]](e1)(s2,inStateU(s2),v,inValueU(inValue(b1))) AND b1=inTrue()
            END //if-exceptions(data(inState(s2)))
END //let-(iseq,Tseq)
```

CASE: Exceptions

```
[[EMPTY ]](e)(s,s',r,v') <=> v'=inValueU(inValue(inBoolean(inTrue())))

[[exceptions "I" spec-expr, excep-clause ]](e)(s,s',r,v') <=>
IF exceptions(data(inState(s'))) AND substring( "I", ide(exception(data(inState(s')))) ) THEN
        \exists v1 \in ValueU: [[spec-expr]](e)(s,s',r,v')
                AND v'=inValueU(inValue(inBoolean(inTrue())))
ELSE
        [[excep-clause]](e)(s,s',r,v')
END
```

3.2.3) CASE: Loop Specification

```
[[invariant spec-expr1; decreases spec-expr2; while E do Cseq end do;]](e)(s,s') <=>
(\forall b \in Boolean: [[spec-expr1]](e)(s,inStateU(s),r,inValueU(inValue(inBoolean(b)))
        => b = inTrue() )
AND
(\forall i \in Integer: [[spec-expr2]](e)(s,inStateU(s),r,inValueU(inValue(i))) => i > 0 )
AND
( \forall s1, s2 \in State:
                ( \forall b1 \in Boolean:
                        [[spec-expr1]](e)(s1,inStateU(s1),r,inValueU(inValue(inBoolean(b1)))
```

```
                    => b1=inTrue()
            )
            AND
            ( \forall j \in Integer: [[spec-expr2]](e)(s1,inStateU(s1),r,inValueU(inValue(j)))
                    => j >= 0
            )
            AND
            ( \forall b2 \in Boolean: [[E]](e)(s1,inStateU(s1),inValueU(b3)) => b3=inTrue() )
            AND [[Cseq]](e)(s1,inStateU(s2))
    =>      ( \forall b3 \in Boolean:
                    [[spec-expr1]](e)(s,s2,r,inValueU(inValue(inBoolean(b3))) => b3=inTrue()
            )
            AND
            ( \forall k \in Integer:
                    [[spec-expr2]](e)(s2,inStateU(s2),r,inValueU(inValue(k)))
                            => k >= 0 AND k < j
            )
    )
```

3.2.4) CASE: Assertion

[[**assert**(spec-expr, “I”)]](e)(s) <=>
\forall r \in Value, b \in Boolean:
    [[spec-expr]](e)(s,inStateU(s),r,inValueU(inBoolean(b))) => b = inTrue()