Formal Semantics of *MiniMaple*[*]

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

January 11, 2012

Abstract

In this paper, we give the complete definition of a formal (denotational) semantics of a subset of the language of the computer algebra systems Maple which we call *MiniMaple*. As a next step we will develop a verification calculus for this language. The verification conditions generated by the calculus must be sound with respect to the formal semantics.

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

Contents

1 Introduction

Our goal is to formally describe the runtime behavior of computer algebra programs written in *MiniMaple* [8]. Therefore we have defined a formal semantics of *MiniMaple* programs. This semantics is also a pre-requisite 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 the semantics.

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

- The formal semantics is defined as a state relationship between pre- and post-states.
- *MiniMaple* has expressions with side-effects, which is not supported in functional programming languages [5, 12]. As a result the evaluation of an expression may change the state.
- Semantic domains of values have some non-standard types of objects, for example symbol, uneval and union etc. *MiniMaple* also supports additional functions and predicates, for example type tests i.e. **type**(E,T). For further details of the formal syntax of *MiniMaple* and its corresponding type checker, please see [7, 6].
- In *MiniMaple* a procedure is defined by an assignment command, e.g. I := **proc**() ... **end proc**. Furthermore, static scoping (context) [9] is used to evaluate a *MiniMaple* procedure.

The rest of the paper is organized as follows: in Section 2, we discuss the overview of our semantics of *MiniMaple*. Section 3 presents conclusions and future work.

2 Semantics Overview

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

2.1 Semantic Algebras

The definition of a formal semantics is based on a collection of data structures. We use the framework of domain theory [2] to define these data structures. *Semantic domains* are a fundamental concept of domain theory; they represent set of elements that share some common properties. A semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a *semantic algebra* [10]. The *MiniMaple* semantics makes use of several primitive and compound domains.

In the following we enlist the semantic domains and their corresponding operations. 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. *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* 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*, *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.

2.2 Signatures of Valuation Functions

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meanings (semantic algebras) [10]. A valuation function VF for a syntax domain VF is usually formalized by a set of equations, one per alternative in the corresponding BNF rule for the *MiniMaple* syntactic domain.

As the formal semantics of *MiniMaple* is defined as a state relationship, so we define the result of valuation function as a predicate. In this section we first give the definitions of various relations and functions that are used in the definition of valuation functions. For example the state relation (*StateRelation*) is defined as a power set of pair of pre- and post-states, where post state can be an *Error* state and is defined as follows:

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

In the following sections, we give define the various valuation functions.

2.2.1 Program

As a *MiniMaple* program is defined as a command sequence, so the valuation function for program takes the abstract syntax of a program and a value of type Prog and results in a *ComRelation*. The command relation (*ComRelation*) maps an *Environment* to *StateRelation*. The syntax of the valuation function for program is as follows:

$$[[\text{ Prog }]] : \mathit{ComRelation}$$

If we unfold the definition of the above valuation function and rewrite its signature, the above function can be rewritten as follows:

$$[[\ ]] : \text{Prog} \rightarrow \mathit{Environment} \rightarrow \mathit{StateRelation}$$

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

2.2.2 Command

The valuation function for commands has the same result as they are shown above for programs. Here the valuation function is defined over the abstract syntax domain command of values C:

$$[[\ \mathrm{C}\ ]] : \mathit{ComRelation}$$

2.2.3 Command Sequence

The valuation function for abstract syntax domain command sequence (Cseq) is defined as follows:

$$[[\ \mathrm{Cseq}\ ]] : \mathit{ComRelation}$$

2.2.4 Expression

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

$$[[\ \mathrm{E}\ ]] : \mathit{ExpRelation}$$

The valuation function results in an *ExpRelation*, which maps an *Environment* to *StateValueRelation*. *StateValueRelation* formulates the relationship of the evaluation of an expression with the given expression (E) and an *Environment*. This relationship is a tuple of pre-state, post-state and the evaluated value of an expression E. Here the post-state or the evaluated value can be unsafe.

2.2.5 Expression Sequence

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

$$[[\ \mathrm{Eseq}\ ]] : \mathit{ExpSeqRelation}$$

The valuation function results in an *ExpSeqRelation*, which maps an *Environment* to *StateValueSeqRelation*. *StateValueSeqRelation* is the same as *StateValueRelation* except that it returns a sequence of values instead of a single value.

2.2.6 Elif

The valuation function for abstract syntax domain elif conditional of values Elif is defined as follows:

$$[[\ \mathrm{Elif}\ ]] : \mathit{ComRelation} \times \mathrm{Tr}$$

The evaluation of an Elif construct results as a tuple of command relation *ComRelation* and a truth value Tr. The truth value is *true* if the evaluation of any of the conditional of the Elif is *true* and *false* otherwise. It is used while the evaluation of the else part of the conditional that includes Elif construct.

2.2.7 Identifier

The valuation function for an identifier I results in its corresponding variable (a location address). It is defined as follows:

$$[[\text{ I }]] : \textit{Environment} \rightarrow \textit{Variable}$$

2.2.8 Identifier Sequence

Like identifier, the valuation function for identifier sequence *Iseq* results in a sequence of variables as follows:

$$[[\text{ Iseq }]] : \textit{Environment} \rightarrow \textit{Variable*}$$

2.2.9 Identifier with Subscript

The valuation function for an identifier *IS* results as a tuple of its corresponding variable and a list of subscripts. The elements of the list represent the corresponding nesting level of the domain. It is defined as follows:

$$[[\text{ IS }]] : \textit{Environment} \rightarrow (\textit{Variable} \times \textit{List})$$

2.2.10 Identifier Sequence with Subscripts

The valuation function for an identifier *ISseq* results as a tuple of a sequence of variables and a sequence of lists (of subscripts corresponding to each identifier). It is defined as follows:

$$[[\text{ ISseq }]] : \textit{Environment} \rightarrow (\textit{Variable*} \times \textit{List*})$$

2.2.11 Catch

As Catch is a command so it has the same valuation function as shown above for the command:

$$[[\text{ Catch }]] : \textit{ComRelation}$$

2.2.12 Declaration Sequence

The valuation function for *MiniMaple* declarations S (global, local and export) is defined as follows:

$$[[\text{ S }]] : \textit{Environment} \rightarrow \textit{StateEnvRelation}$$

StateEnvRelation is a power set of post-environment, pre- and post-states. The post-environment is produced by the declarations of new identifiers, while the post-state represents for the assigned values to these identifiers, if any.

2.2.13 Recurrence

The valuation function for the value of the body of the *MiniMaple* procedure/-module R is defined as follows:

$$[[\text{ R }]] : \textit{ExpRelation}$$

The evaluation of the body a *MiniMaple* procedure/module is same as of evaluation expression, because syntactically it is a command sequence followed by an expression.

2.2.14 Parameter

The valuation function for a parameter P is defined as follows:

$$[[\text{ P }]] : \textit{Environment} \rightarrow (\textit{Environment} \times \textit{Variable})$$

The evaluation of a parameter result as a tuple of a post-environment and a variable.

2.2.15 Parameter Sequence

The evaluation of parameter sequence (Pseq) results as a tuple of a post-environment and a sequence of variables (for corresponding identifiers). The valuation function is defined as follows:

$$[[\text{ Pseq }]] : \textit{Environment} \rightarrow (\textit{Environment} \times \textit{Variable*})$$

2.2.16 Binary Operator

The evaluation of a binary operator (Bop) might results an undefined value (*Undefined*) if the binary operator is not defined for any of the two given values. The valuation function for the value of binary operator Bop is defined as follows:

$$[[\text{ Bop }]] : \textit{Value} \times \textit{Value} \rightarrow \textit{ValueU}$$

2.2.17 Unary Operator

Like binary operator, the evaluation of a unary operator (Uop) might also results an undefined value (*Undefined*) if it is not defined for the given value. The valuation function for the value of a unary operator Uop is defined as follows:

$$[[\text{ Uop }]] : \textit{Value} \rightarrow \textit{ValueU}$$

2.2.18 Type (Predicate)

The evaluation of a type structure in a given environment *Environment* gives the corresponding *Type-Tag* of the type. The valuation function for the value of type T is defined as follows:

$$[[\text{ T }]] : \textit{Environment} \rightarrow \textit{Type-Tag}$$

The *Type-Tag* is used for the *MiniMaple* type tests.

2.2.19 Type (Assignment)

For the type assignment to any semantic *Value* maps a type-structure to storage allocation actions in the current state. The evaluation for a given type (T), *Environment* and *State* results as a tuple of post-value, post-state and an associated *Type-Tag*. The valuation function for the value of type T is defined as follows:

$$[[\text{ T }]] : \textit{Environment} \times \textit{State} \rightarrow (\textit{ValueU} \times \textit{StateU} \times \textit{Type-Tag})$$

In the following section we define the auxiliary functions and predicates used in the formal semantics of *MiniMaple*.

2.3 Auxiliary Functions and Predicates

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

2.3.1 For Loop/Sequence

MiniMaple has four variations of for-loop and also supports sequence expression. In order to understand their semantics more simpler, we defined relations *iterate* and *seq* to be used later in this document for the semantic definitions of loops and sequence expression. The heart of the loop semantics is modeled by *iterate* and likewise the semantics of sequence expression is modeled by *seq*.

2.3.2 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 major syntactic domains of *MiniMaple*, e.g. Program, Command and Expression. The semantics of other domains are very simple and can be easily rehearsed.

2.4.1 Program Semantics

The semantics of *MiniMaple* program are essentially the same as of command semantics.

2.4.2 Command Semantics

The semantics of command is a relationship between the pre- and post-state of the execution of any *MiniMaple* command.

Assignment The *MiniMaple* assignment command is a simultaneous assignment. First the expressions on the right side of an assignment are evaluated; if none of them evaluates to *Undefined*, then the values obtained by the evaluations of the expressions are stored for the variables on the left side of the assignment.

While Loop *MiniMaple* supports the typical while-loop. The semantics of the iterations of a *MiniMaple* while-loop is determined by two sequences of states *pre* and *post* [11]. Both these states are constructed from the pre-state of the loop. Any *ith* iteration (execution of the body) of the loop transforms *pre(i)* state into *post(i+1)* state from which the *pre(i+1)* is constructed. No iteration is allowed from the *Error* as *pre* state. The loop terminates when the guard expression evaluates to false or body of the loop evaluates to an *Error* post-state. The corresponding *iterate* predicate formalizes the aforementioned while-loop semantics.

For Loops Also *MiniMaple* supports following four variations of for-loops.

1. The first variation is a for-loop **for** I **in** E **do** $Cseq$ **end do**. The semantics of the for-loop is the same as shown above for while-loop but here expression E is evaluated first. In each iteration *pre* state overrides the value of I as a next value/element of the expression E. The loop terminates when all the element of the value of expression E are iterated or the body of the loop executes to an *Error* post-state. The corresponding *iterate* predicate formalizes the aforementioned for-loop semantics.
2. The second variation is a for-while-loop **for** I **in** E_1 **while** E_2 **do** $Cseq$ **end do**. The iteration semantics of the for-while-loop is the same as shown above for while-loop but here expression E_1 is evaluated before the execution of the loop. The expression E_2 is tested at the beginning of each iteration; if it does not evaluate to *true*, the loop terminates. In each iteration the *pre* state overrides the value of I as a next value/element of the expression E_1. The loop terminates when the expression E_2 does not evaluate to *true* or all the element of the value of expression E_1 are iterated or the body of the loop evaluates to the *Error* post-state. The corresponding *iterate* predicate formalizes the aforementioned for-while-loop semantics.
3. The third variation is another typical for-loop **for** I **from** E_1 **by** E_2 **to** E_3 **do** $Cseq$ **end do**. The iteration semantics of this variation of for-while is the same as shown above for first variation of for-loop but here to-expression E_3 is evaluated only once at the beginning of the loop and the value of expression E_3 is used as a termination test. In each iteration the *pre* state overrides the value of I as a next value of from-expression E_2 incremented by the value of by-expression E_3. The loop terminates when the test against the value of to-expression E_3 fails or the body of the loop executes to an *Error* post-state. The corresponding *iterate* predicate formalizes the aforementioned for-loop semantics.
4. The fourth variation is another for-while-loop **for** I **from** E_1 **by** E_2 **to** E_3 **while** E_4 **do** $Cseq$ **end do**. The iteration semantics of this variation is the same as shown above for third variation of for-loop but here while-expression E_4 is evaluated before every loop iteration and the values of while-expression E_4 and to-expression E_3 are used as termination test orderly. The loop terminates when the test against any of the value of to-expression E_3 or while-expression E_4 fails or the body of the loop executes to an *Error* post-state. The corresponding *iterate* predicate formalizes the aforementioned for-while-loop semantics.

One-sided Conditional First the conditional expression is evaluated. If this yields to a boolean value *true* then the body of the conditional (command sequence) is executed. If the conditional expression evaluates to *false* then Elif-construct (if present) is evaluated. If any of the above evaluations is unsafe, the corresponding command immediately terminates with an *Error* as post-state.

Two-sided Conditional First the conditional expression is evaluated. If this yields to a boolean value *true* then the body of the conditional (command sequence) is executed. If the conditional expression evaluates to *false* then Elif-construct is executed, if present. At the end if Elif-construct (if present) also evaluates to *false*, then the body (command sequence) of the else branch is executed. If any of the above evaluations is unsafe, the corresponding command immediately terminates with *Error* as post-state.

Return A return command occurs in a procedure, which immediately returns to the point of invocation of the current procedure. If the evaluation of the expression sequence of the return statement does not yields to *Undefined* value, then these evaluated values are returned as the procedure value, otherwise it results in an *Error* post-state. In the case of anonymous return, value of the last command before the return statement is returned as the value of the called procedure.

Error The error command raises an exception. Execution of the current command sequence is interrupted, and the exception handler is encountered, if present or the execution returns with an *Error* as post-state. The string part of the error command is a string value that may contain numbered parameters of the form $\%n$ where n is an integer. These parameters (*MiniMaple* expressions) are one or more arbitrary *MiniMaple* objects that are substituted for the numbered parameters in the string value of an error command. For example, the error "f has a 2nd argument, x, which is missing" is specified by the string "%1 has a %-2 argument, %3, which is missing", and the expression sequence (parameters) f, 2, and x [1]. If any expression among the expression sequence yields an *Undefined* value, the command immediately terminates in the *Error* post-state. An anonymous error command raises an anonymous exception and the execution results an *Error* as a post-state. Exceptions can be caught by using the exception handler facility of *MiniMaple*.

Exception Handler The try-command provides a mechanism for executing commands in a controlled environment. When an exception is thrown, then the try-command sequence is executed trying to catch thrown exception. If no exception occurs during the execution of the try-command sequence, execution continues with the finally clause (if it was specified). After that, execution continues with the following command of the try-catch clause. If an exception does occur during the execution of the try-command sequence, execution of try-catch clause terminates immediately. The exception string that corresponds to the exception is compared against each catch-string in turn until a match is found. If a matching catch clause is found, the catch-command sequence of that catch clause is executed, and the exception is considered to have been caught. If no matching catch clause is found, the exception is considered not-caught, and executes with an *Error* as post-state. Under normal execution, the command sequence of the finally clause (if specified) is always executed before the control leaves the try-catch clause.

When looking for a matching catch clause, the following definition of "matching" is used [1]: The catch-strings are considered to be prefixes of the exception-string. If a catch-string has n characters, only the first n characters of the exception-string have to match the catch-string. This also allows one to define *classes of exceptions*.

Procedure Call First, the argument expression sequence is evaluated; if any of them yields an *Undefined* value, the command terminates with an *Error* as post-state. Otherwise, the *Environment* is looked up for the procedure named *I*. This procedure is applied to the argument values which yields a command behavior and the post-state of the command sequence execution is set to the post-state of the procedure call statement.

Type Declaration *MiniMaple* supports user-defined type declaration by the type-command '**type**/I':=T. First, the *Type-Tag* for type T is obtained and then the *type-identifier I* is mapped to the obtained *Type-Tag*. The corresponding mapping of *type-identifier* is stored in the *Environment*.

2.4.3 Expression Semantics

The semantics of expression is a relationship among the pre- post-states and the value of the expression. Any *MiniMaple* expression evaluation in a pre-state *State* yields some value *ValueU* with the post-state *StateU*.

Procedure *MiniMaple* procedure is defined as a predicate over the followings:

- *Value** as a sequence of parameter values
- *State* as a pre-state
- *StateU* as a post-state
- *ValueU* as a return value of the procedure
- *Type-Tag* as a type tag for the procedure's return value
- *Type-Tag** as a sequence of type tags for the corresponding procedure parameter values

Procedure is evaluated at the invocation time. Further, static scoping is used to evaluate *MiniMaple* procedure.

Module Also *MiniMaple* module is defined as a predicate over exported identifiers and their corresponding (command sequence) values. *Module* is evaluated at its definition time. First, the declaration S of a module is evaluated and if yields an *Error* post-state then the module expression evaluates to *Undefined* value and changes post-state to an *Error*. Otherwise from the declaration list of exported identifiers is collected and the body of the module R is evaluated. If the evaluation of the body does not yields to an *Undefined* value and all of the exported identifiers have been assigned some value then the module expression evaluates to a *Module* value.

Binary *MiniMaple* supports typical arithmetic (addition, subtraction, multiplication, division and mod) and logical (less, greater, less equal and greater equal) binary operations. As a special case binary operations of equality and non-equality are defined separately. First the expressions E_1 and E_2 are evaluated in a pre-state. If any of them does not yields to *Undefined* value, then corresponding binary operation is performed on the yielded values that yields a result expression value. Remember that the subject binary operation is performed only if it is defined for the yielded values.

Unary Also *MiniMaple* supports typical arithmetic (plus and minus) and logical (negation) unary operations. First the expression E is evaluated in a pre-state. If it does not yields to *Undefined* value, then corresponding unary operation is performed for the yielded value. The value of the unary expression is the value of the operated expression. Also remember here that the unary operation is performed only if it is defined for the yielded value.

Procedure Call First, the argument expression sequence is evaluated, if any of them yields an *Undefined* value, the expression evaluates to *Undefined* value and with an *Error* as post-state. Otherwise, the *Environment* is looked up for the procedure named I. This procedure is applied to the argument values which yields a command behavior and the post-state of the command sequence execution is set to the post-state of the procedure call expression and the procedure call expression evaluates to the value of the procedure.

Module Call First, the I_1 is looked up in the *store* of the pre-state. If it yields to a *Module*, then if I_2 is in the exported identifiers its corresponding value in the *Module* is executed. The module call expression yields to the value of the corresponding value of I_2 in *Module*. If I_1 does not yields to a *Module* value, or I_2 is not in the exported identifiers of the *Module* then the expression evaluates to *Undefined* value and with an *Error* as post-state.

Logical *MiniMaple* supports logical-and and logical-or binary expressions. The expressions E_1 and E_2 are evaluated, if they both yield to boolean values, then the corresponding logical-and or logical-or boolean expression of these values is evaluated. If any of them yields to *Undefined* value or to a non-boolean value then the logical expression evaluates to *Undefined* value with *Error* as post-state.

Equality Also *MiniMaple* supports binary expressions of equality and non-equality. The expressions E_1 and E_2 are evaluated, if they both do not yield to *Undefined* values, then the equality of these values is evaluated. If any of them yields to *Undefined* value then the equality expression evaluates to *Undefined* value with an *Error* as post-state. The non-equality of expression values is defined as the negation of the equality of the corresponding expression values.

Type-Test A type-test is a special feature of *MiniMaple*, which tests if the expression E is of type T. First the expression E is evaluated, if it does not yield to *Undefined* value, then it checks if the value of the expression conforms to the corresponding type-tag.

Special Expressions *MiniMaple* also supports some special expressions. This includes the followings:

Constructors The constructors are the special expressions for the construction of the domain *List*, *Set* and *Record* values. As mentioned in one of the previous sections, the constructor for the domain *List* is defined as a sequence of values i.e. *Value**, while the domain constructors for *Set* and *Record* are defined by *List*. The *List* and *Set* is constructed by a sequence of single-domain values, e.g. list of integers etc., while a *Record* is constructed by a sequence of many-domain values, as each element of the tuple can be of different domain value. The elements of a *Set* is their permutation.

Operands In general the special expression **op** selects the values (elements) of the expression. *MiniMaple* also supports to select a range of values of the expression. The value of the **op** special expression is a sequence of values (Value*) if none of these values yields to *Undefined* value. Otherwise the expression evaluates to *Undefined* and results in an *Error* post-state. Please note that the unit domains (e.g. Integer, Float etc.) of *MiniMaple* also supports selects operation, but in this case only legal index and range are allowed. The select operation over the domain *Set* gives an unknown value because of the fact that *Set* element are their permutation.

Length The special expression **nops** is a typical length operation of the expression. The value of the **nops** special expression is a natural number if the expression does not yields to *Undefined* value. Otherwise the expression evaluates to *Undefined* and results in an *Error* post-state. Please note that the length of the unit domains (e.g. Integer, Rational etc.) of *MiniMaple* is one, but the length of *Float* expression is two.

Substitution The special expressions **subs** and **subsop** are update operations that substitute an expression for the given expression in the target expression. The first variation (**subs**) substitutes an identifier I for expression E_1 in the expression E_2. It is like a simple application of replacing a symbol by a value in a formula. The second variation (**subsop**) is used to replace specified operands/elements of an expression with given new values. In this case, result is obtained by replacing the expression E_1 by the expression E_2 in the expression E_3. The operation does the simultaneous substitutions. Moreover the action of substitution is usually not followed by evaluation.

Unevaluated Expression A *MiniMaple* expression enclosed with (right single) *unevaluation* quotes delays the evaluation of the expression. This expression is called unevaluated expression. The semantic domain *Uneval* represents unevaluated values. The corresponding evaluation **eval**(I,1) strips off one level of unevaluation quotes. But remember that this evaluation is not simplification.

Sequence *MiniMaple* supports sequence operator **seq** that generates a finite sequence of values. There are two variations of sequence operator in *MiniMaple*. The first variation **seq**(E_1, I=E_2..E_3) generates the sequence of values of pattern E_1. In each term of the generated sequence, the value of identifier I is replaced by the value of E_2 to value of E_3. The length of the sequence is the range from value of E_2 to value of E_3. The second variation **seq**(E_1, I **in** E_2) generates a sequence by applying the values of E_1 to each operand or entry of value of E_2. Here, E_2 would most commonly be a *Set* or *List*, but it could be any other data structure to which **op** can be applied. The semantics of **seq** expression is related to the for-loop semantics.

2.5 Type Predicate Semantics

The semantics of type predicate takes *Environment* and gives a corresponding *Type-Tag*.

Basic For the basic/primitive types of *MiniMaple*, the corresponding *Type-Tag* is defined as a unit tag.

Extended *MiniMaple* supports various extended types, complex types e.g. **procedure**[T](Tseq) etc and user-defined types e.g. I and $I(Tseq)$ etc. The *Type-Tag* for complex type *Procedure* is defined as a tuple of *Type-Tag* and *Type-Tag** for its return type and parameters types respectively. The *Type-Tag* for user-defined type is looked up in the *Environment* for the given identifier I.

3 Conclusions and Future Work

In this paper we gave the definition of formal semantics of *MiniMaple* programs including semantic domains, semantic algebras, declaration and definitions of valuation functions. As a next step we will develop a verification calculus for *MiniMaple* and the verification conditions generated by the verification calculus must be sound with respect to the formal semantics of *MiniMaple*. Here we have investigated various existing frameworks, e.g. Boogie2 [3] by Microsoft and Why3 [4] by LRI. After studying some formal literature about Boogie2 and Why3 and also from the discussions with the people at LRI, we currently 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.

Acknowledgment

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

References

[1] Exception Handling in Maple. `http://www.maplesoft.com/support/help/Maple/view.aspx?path=try`.

[2] Abramsky, Samson and Jung, Achim. *Domain Theory*, volume 3. Oxford University Press, Oxford, UK, 1994.

[3] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

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

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

[7] 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.

[8] 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.

[9] Meertens, L. On Static Scope Checking in ALGOL 68. *ALGOL Bulletin*, pages 45–58, March 1973.

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

[11] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008.

[12] Tim Lambert and Peter Lindsay and Ken Robinson. Using Miranda as a First Programming Language. *Journal of Functional Programming*, 3(1):5–34, 1993.

Appendices

In the following subsection we give the complete definition of formal semantics of *MiniMaple* programs, commands and expressions with valuation functions and auxiliary functions and predicates.

A Formal Semantics of *MiniMaple*

Semantic Algebras

1) Truth Values
Domain Tr = Boolean = {True, False}

Operations
true: Tr
false: Tr
and: Tr x Tr → Tr
or: Tr x Tr → Tr
not: Tr → Tr
length: Boolean → Nat'
length(b) = 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
```

3) Environment Values
Domains
Environment = Context x Space
Context = Identifier → EnvValue
EnvValue = Variable + Procedure + 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) = LET <x,s'> = take(space(e)) IN environment(context(e)[I |-> inType-Tag(x)], s') END

push : Environment x IdentifierSeq → 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

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))

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 )
```

5) Semantic Values

Domain

Value = 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

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)>

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

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

9) Sequence Values
Domain Value*

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

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

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
```

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

13) Symbol Value
Domain Symbol

Operations
length: Symbol → Nat'
length(s) = 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

15) Unevaluated Values
Domain Uneval

Operations
length: Uneval → Nat'

length(u) = 1

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

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)

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

18) Declaration Values
Domain S

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

19) Type-Tag Values
Domain Type-Tag = Integer-Tag + Rational-Tag + Float-Tag + Boolean-Tag + String-Tag + Type-Tag* …
where 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

Valuation Functions

StateRelation = P(State x StateU)
ExpRelation = Environment → StateValueRelation
ExpSeqRelation = Environment → StateValueSeqRelation
ComRelation = Environment → StateRelation
StateValueRelation = P(State x StateU x ValueU)
StateValueSeqRelation =P(State x StateU x ValueU*)
StateEnvRelation = P(Environment x State x StateU)
catchEnv = Identifier → (State x StateU)

For Program:
[[Prog]] : ComRelation

For Command:
[[C]] : ComRelation

For Command Sequence:
[[Cseq]] : ComRelation

For Expression:
[[E]] : ExpRelation

For Expression Sequence:
[[Eseq]] : ExpSeqRelation

For Elif:
[[Elif]] : ComRelation x Tr
// where Tr has value TRUE if the body of "Elif" was executed, otherwise enters the "else" branch.

For Identifier:
[[I]] : Environment → Variable

For Identifier Sequence:
[[Iseq]] : Environment → Variable*

For Subscripted Identifier:
[[IS]] : Environment → (Variable x List) // where List is the list of indexes, e.g. I[1][3] gives as [1,3]

For Subscripted Identifier Sequence:
[[ISseq]] : Environment → (Variable* x List*)
//where List* contains the sequence of list of indexes of each of the Identifiers in ISseq

For Catch:
[[Catch]] : ComRelation

For Declaration (Sequence):
[[S]] : Environment → StateEnvRelation

For Recurrence:
[[R]] : ExpRelation

For Parameter:
[[P]] : Environment → Environment x Variable

For Parameter Sequence:
[[Pseq]] : Environment → Environment x Variable*

For Binary Operator:
[[Bop]] : Value x Value →ValueU

For Unary Operator:
[[Uop]] : Value →ValueU

For Type:
[[T]] : Environment→ Type-TagU

Auxiliary Functions and Predicates

1) For loops
// while loop iterator ...

```
iterate \subin Nat' x StateU* x StateU* x Environment x StateValueRelation x StateRelation
iterate(i, t, u, e, E, C) <=>
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND \exists v \in ValueU, s' \in StateU : E(e)(m,s',v) AND
          cases s' of
               isError() → u(i+1)=inError() AND t(i+1)=u(i+1)
               [] isState(p) →
               cases v of
                    isUndefined() → u(i+1)=inError() AND t(i+1)=u(i+1)
                    [] isValue(v') → cases v' of
                         isBoolean(b) → b AND C(e)(p,u(i+1)) AND t(i+1)=u(i+1)
                         [] … → u(i+1)=inError() AND t(i+1)=u(i+1)
                         END //cases-v'
               END //cases-v
          END //cases-s'
END //cases-t(i)
```

// for-1 loop iterator, i.e. for I \in E do ...

```
iterate \subin Nat' x StateU* x StateU* x Identifier x Environment x Value* x StateRelation
iterate(i, t, u, I, e, vseq, C) <=>
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND \exists s \in State: s = update(m, [[I]](e), access(i,vseq))
                        AND C(e)(s1,u(i+1)) AND t(i+1)=u(i+1)
END //cases-t(i)
```

\\ for-2 loop itertator, for I in E1 while E2 do ...

```
iterate \subin
Nat' x StateU* x StateU* x Identifier x Environment x Value* x StateValueRelation x StateRelation
iterate(i, t, u, I, e, vseq, E, C) <=>
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND
                \exists v \in ValueU, s' \in StateU, s1 \in State: s1 = update(m, [[I]](e), access(i,vseq))
                        AND E(e)(s1,s',v) AND
                cases s' of
                        isError() → u(i+1)=inError() AND t(i+1)=u(i+1)
                        [] isState(p) →
                        cases v of
                                isUndefined() → u(i+1)=inError() AND t(i+1)=u(i+1)
                                [] isValue(v') → cases v' of
                                        isBoolean(b) → b AND C(e)(p,u(i+1)) AND t(i+1)=u(i+1)
                                        [] … → u(i+1)=inError() AND t(i+1)=u(i+1)
                                        END //cases-v'
                        END //cases-v
                END //cases-s'
END //cases-t(i)
```

// for-3 loop iterator, i.e. for I from E1 by E2 to E3 do ...

```
iterate \subin Nat' x StateU* x StateU* x Identifier x Environment x Nat x Nat x ComRelation
iterate(i, t, u, I, e, y, z, C) <=>
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND
                LET
                        x = add( store(m)([[I]](e)), y)
                        s1 = write(m, [[I]](e), x)
                IN
                IF x <= z THEN
                        [[C]](e)(s1,u(i+1)) AND t(i+1)=u(i+1)
                END //if-less
                END //let-x
END //cases-t(i)
```

// for-4 loop iterator, i.e. for I from E1 by E2 to E3 while E4 do ...

```
iterate \subin
Nat' x State* x StateU* x Identifier x Environment x Nat' x Nat' x Nat' x ExpRelation x ComRelation
iterate(i, t, u, I, e, x, y, z, E, C) <=>
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND
          LET
               x = add( store(m)([[I]](e)), y)
               s1 = write(m, [[I]](e), x)
          IN
          IF x <= z THEN
          \exists s' \in StateU, v \in ValueU: E(e)(s1,s',v) AND
          cases s' of
               isError() → u(i+1)=inError() AND t(i+1)=u(i+1)
               [] isState(p) →
               cases v of
                    isUndefined() → u(i+1)=inError() AND t(i+1)=u(i+1)
                    [] isValue(v') → cases v' of
                         isBoolean(b) → b AND C(e)(p,u(i+1)) AND t(i+1)=u(i+1)
                         [] … → u(i+1)=inError() AND t(i+1)=u(i+1)
                         END //cases-v'
               END //cases-v
          END //cases-s'
          END //if-less
          END //let-x
END //cases-t(i)
```

// seq ...

```
seq \subin Nat' x StateU* x StateU* x Identifier x Environment x Value* x Value* x StateValueRelation
iterate(i, t, u, I, e, vseq, vs, E) <=>
cases t(i) of
isError() → false
[] isState(m) → \exists s1 \in State: s1 = update(m, [[I]](e), access(i,vseq)) AND
          \exists v' \in ValueU: E(e)(s1,u(i+1),v') AND
               cases v' of
                    isUndefined() → u(i+1)=inError() AND t(i+1)=u(i+1)
                    [] isValue(v1) → access(i,vs) = v1 AND u(i+1)=inStateU(u')
                                        AND t(i+1)=u(i+1)
               END //cases-v'
END //cases-t(i)
```

2) 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
```

Semantics

CASE: Program

[[P]](e)(s,s') <=> [[Cseq]](e)(s,s')

CASE: Command Sequence

```
[[ C, Cseq ]](e)(s,s') <=>
\exists s" \in StateU: [[C]](e)(s,s") AND
        cases s" of
                isError() → s' = inError()
                isState(p) →   IF executes(data(p)) THEN
                                        [[Cseq]](e)(p,s')
                               ELSE s' = inStateU(p)
                               END //if-executes
        END //cases-s"
```

CASE: Commands

```
[[ IS,ISseq := E,Eseq ]](e)(s,s') <=>
\exists v \in ValueU, s" \in StateU: [[E]](e)(s,s",v)
        AND cases v of
                isUndefined() → s' = inError()
                [] isValue(v') → cases s" of
                               isError() →s' =  inError()
                               [] isState(p) → \exists v" \in ValueU*, s"' \in State: [[Eseq]](e)(p,s"',v")
                                        AND cases s"' of
                                                isError() → s' = inError()
                                                [] isState(p1)
                                                        → IF undefinedSeq(v") THEN
                        \exists var \in Variable, l1 \in List, vars \in Variable*, ls \in List*:
                               [[IS]](e)(var, l) AND [[Isseq]](e)(vars, ls) AND
                               s' = inValueU( update(<var, vars>, <l,ls>, <v',valSeq(v")>), p1) )
                                                          ELSE s' = inError()
                                                          END //if
                                                END //cases-s"'
                               END //cases-s"'
                END //cases-v
```

[[**if** E **then** Cseq Elif **end if**]](e)(s,s') <=>

```
\exists v \in ValueU, s" \in StateU: [[E]](e)(s,s",v) AND
cases v of
        isUndefined() → s' = inError()
        [] isValue(v1)
                → cases s" of
```

```
                    isError() → s' = inError()
                    [] isState(p)
                              →cases v1 of
                                        isBoolean(v2) → IF v2 THEN [[Cseq]](e)(p,s')
                                                  ELSE \exists v' \in Tr, p' \in StateU: [[Elif]](e)(s,p',v') AND
                                                  cases p' of
                                                            isError() → s' = inError()
                                                            [] isState(p'')→ IF v'=inTr(True) THEN
                                                                                      s' = inStateU(p'')
                                                                               ELSE s' = s
                                                                               END
                                                  END
                                                  END
                              END
          END
END

[[ if E then Cseq1 Elif else Cseq2 end if]](e)(s,s') <=>
\exists v \in ValueU, s'' \in StateU: [[E]](e)(s,s'',v) AND
          cases v of
                    isUndefined() → s' = inError()
                    [] isValue(v1)
                              → cases s'' of
                                        isError() → s' = inError()
                                        [] isState(p) →
                                                  cases v1 of
                                                            isUndefined() → s' = inError()
                                                            [] isBoolean(b) →
                                                            IF b THEN [[Cseq1]](e)(p,s')
                                                            ELSE \exists v' \in Tr, p' \in StateU: [[Elif]](e)(s,p',v') AND
                                                            cases p' of
                                                                      isError() → s' = inError()
                                                                      [] isState(p'')→ IF v'=inTr(True) THEN
                                                                                                s' = inStateU(p'')
                                                                                         ELSE [[Cseq2]](e)(s,s')
                                                                                         END //if-v'
                                                            END //cases-p'
                                                            END //if-b
                                                   END //cases-v1
                              END //cases-s''
          END //cases-v

[[ while E do Cseq end do ]](e)(s,s') <=>
\exists k \in Nat', t, u \in StateU*:
          t(1)=inStataU(s) AND u(1)=inStateU(s) AND
          ( \forall i \in Nat'_k: iterate(i, t, u, e, [[E]], [[Cseq]]) ) AND
          ( ( u(k)=inError() AND s'=u(k) ) OR ( returns(data(inState(u(k)))) AND s'=t(k) ) OR
```

(\exists v \in ValueU: [[E]](e)(inState(t(k)), u(k), v)
AND v <> inValue(inBoolean(True)) AND
IF v = inValue(inBoolean(False)) THEN
s'=t(k)
ELSE s' = inError() END //if-v
)
)

[[**for** I **in** E **do** Cseq **end do**]](e)(s,s') <=>
\exists v \in ValueU, s'' \in StateU: [[E]](e)(s,s'',v) AND
cases s'' of
isError() → s'=inError()
[] isState(p) → cases v of
isUndefined() → s'=Error()
[] isValue(v') →
LET
vseq = expValues(v')
IN
IF hasUndefinedValue(vseq) THEN
s'=inError()
ELSE
LET
k = length(vseq)
IN
\exists k' <= k, t, u \in StateU*, e1 \in Environment:
t(1)=inStateU(s) AND u(1)=inStateU(s) AND e1 = push(I,e) AND
\forall i \in Nat'_k': iterate(i,t,u,I,e1,vseq,[[Cseq]]) AND
(u(k')=inError() OR returns(data(inState(u(k')))) OR k' = k
) AND IF returns(data(u(k))) OR k'=k THEN
s'=t(k')
ELSE s'=inError() END //if-returns
END //let-k
END //if-hasUndefinedValues
END //let-vseq
END //cases-v
END //cases-s''

[[**for** I **in** E1 **while** E2 **do** Cseq **end do**]](e)(s,s') <=>
\exists v \in ValueU, s'' \in StateU: [[E1]](e)(s,s'',v) AND
cases s'' of
isError() → s'=inError()
[] isState(p) → cases v of
isUndefined() → s'=Error()
[] isValue(v') →
LET
vseq = expValues(v')
IN

```
                                        IF hasUndefinedValue(vseq) THEN
                                                s'=inError()
                                        ELSE
                                        LET
                                                k = length(vseq)
                                        IN
                                        \exists k' <= k, t, u \in StateU*, e1 \in Environment:
                                        t(1)=inStateU(s) AND u(1)=inStateU(s) AND e1 = push(I,e) AND
                                        ( \forall i \in Nat'_k': iterate(i, t, u, I, e1, vseq, [[E2]], [[Cseq]]) ) AND
                                                ( ( u(k')=inError() AND s'=u(k') ) OR
                                                                ( returns(data(u(k'))) AND s'=t(k') ) OR
                                                        ( \exists v \in ValueU: [[E]](e1)(inState(t(k')), u(k'), v)
                                                                AND v <> inValue(inBoolean(True)) AND
                                                                IF v = inValue(inBoolean(False)) THEN
                                                                        s'=t(k')
                                                                ELSE  s' = inError() END //if-v
                                                        )
                                                )
                                        END //let-k
                                        END //if-hasUndefinedValues
                                END //let-vseq
                        END //cases-v
END //cases-s''

[[ for I from E1 by E2 to E3 do Cseq end do ]](e)(s,s') <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
isError() → s'=inError()
[] isState(p1) →
        cases v1 of
                isUndefined() → s'=inError()
                [] isValue(v11) →
                        cases v11 of
                                isInteger(m) →
                                        LET     e1 = push(e,I)
                                                p11 = update(p1, [[I]](e1), m)  IN
                                        \exists s2 \in StateU,v2 \in ValueU: [[E2]](e1)(p11,s2,v2) AND
                                        cases s2 of
                                                isError() → s'=inError()
                                                [] isState(p2) →
                                                        cases v2 of
                                                                isUndefined() → s'=inError()
                                                                [] isValue(v22) →
                                                                cases v22 of
                                                                isInteger(n) →
                                        \exists s3 \in StateU, v3 \in ValueU: [[E3]](e1)(p2,s3,v3) AND
                                        cases s3 of
```

```
                                        isError() → s'=inError()
                                        [] isState(p3) →
                                                cases v3 of
                                                        isUndefined() → s'=inError()
                                                        [] isValue(v33) →
                                                        cases v33 of
                                                        isInteger(x) →
                LET
                        k = iterations(m,n,x)
                IN
                \exists k' <= k \in Nat', t \in State*, u \in StateU*:
                t(1)=inStateU(s) AND u(1)=inStateU(s) AND
                        ( \forall i \in N'_k': \exists s1 \in State, s2 \in StateU:
                                iterate(i, t, u, I, e1, n, x, [[Cseq]]) ) AND
                                ( u(k')=inError() OR returns(data(inState(u(k')))) OR k' = k )
                                AND  IF returns(data(u(k'))) OR k'=k THEN
                                                s'=t(k')
                                         ELSE s'=inError() END //if-returns
                        )
                END //let-k'
                                                        [] … → s'=inError()
                                                        END //cases-v33
                                                END //cases-v3
                                END //cases-s3
                                                        [] … → s'=inError()
                                                        END //cases-v22
                                                END //cases-v2
                                END //cases-s2
                [] … → s'=inError()
                END //cases-v11
        END //cases-v1
END //cases-s1
```

[[**for** I **from** E1 **by** E2 **to** E3 **while** E4 **do** Cseq **end do**]](e)(s,s') <=>

```
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
isError() → s'=inError()
[] isState(p1) →
        cases v1 of
                isUndefined() → s'=inError()
                [] isValue(v11) →
                        cases v11 of
                                isInteger(m) →
                                        LET     e1 = push(e,I)
                                                p11 = update(p1, [[I]](e1), m)  IN
                                        \exists s2 \in StateU,v2 \in ValueU: [[E2]](e1)(p11,s2,v2) AND
                                        cases s2 of
```

```
                        isError() → s'=inError()
                        [] isState(p2) →
                            cases v2 of
                                isUndefined() → s'=inError()
                                [] isValue(v22) →
                                cases v22 of
                                isInteger(n) →
                    \exists s3 \in StateU, v3 \in ValueU: [[E3]](e1)(p2,s3,v3) AND
                    cases s3 of
                        isError() → s'=inError()
                        [] isState(p3) →
                            cases v3 of
                                isUndefined() → s'=inError()
                                [] isValue(v33) →
                                cases v33 of
                                isInteger(x) →
            LET
                k = iterations(m,n,x)
            IN
            \exists k' <= k \in Nat', t \in State*, u \in StateU*:
            t(1)=inStateU(s) AND u(1)=inStateU(s) AND
                ( \forall i \in N'_k': \exists s1 \in State, s2 \in StateU:
                    iterate(i, t, u, I, e1, n, x, [[E4]], [[Cseq]]) ) AND
                    ( ( u(k')=inError() AND s'=u(k') )
                        OR ( returns(data(u(k'))) AND s'=t(k') ) OR
                        ( k' = k AND s'=t(k') ) OR
                        ( \exists v \in ValueU: [[E4]](e1)(inState(t(k')), u(k'), v)
                            AND v <> inValue(inBoolean(True)) AND
                            IF v = inValue(inBoolean(False)) THEN
                                s'=t(k')
                            ELSE  s' = inError() END //if-v
                        )
                    )
                )
            END //let-k
                                [] … → s'=inError()
                                END //cases-v33
                            END //cases-v3
                    END //cases-s3
                                [] … → s'=inError()
                                END //cases-v22
                            END //cases-v2
                    END //cases-s2
            [] … → s'=inError()
            END //cases-v11
    END //cases-v1
END //cases-s1
```

```
[[return E]](e)(s,s') <=>
\exists s" \in StateU, v \in ValueU: [[E]](e)(s,s",v) AND
        cases v of
                isUndefined() → s' = inError()
                [] isValue(v') →
                                cases s" of
                                        isError() → s' = inError()
                                        [] isState(p) → s' = inStateU(return(p,v'))
                                END //cases-s"
        END //cases-v

[[return]](e)(s,s') <=> s'=inStateU(return(s, inUndefined()))

[[error "I", Eseq]](e)(s,s') <=> \exists v \in ValueU*, s" \in StateU: [[Eseq]](e)(s,s",v) AND
        cases s" of
                isError() → s' = inError()
                []isState(p)→IF hasUndefinedValue(v) THEN
                                        s'=inError()
                                 ELSE
                                        IF lengthOfPlaceHldrs("I")=length(v) THEN
                                                s' = inStateU(exception(p,inIdentifier(I),valueSeq(v)))
                                        ELSE s'=inError()
                                        END //if-length
                                 END //if-hasUndefinedValue(v)
        END //cases-s"

[[error]](e)(s,s') <=> s' = inStateU(exception(s, "anonymous", inUndefined()))

[[try Cseq Catch end]](e)(s,s') <=> \exists s" \in StateU: [[Cseq]](e)(s,s") AND
        cases s" of
                isError() → s'=inError()
                [] isState(p) → IF exceptions(data(p)) THEN [[Catch]](e)(p,s')
                                ELSE s' = inStateU(p)
                                END //if-exception
        END //cases-s"


[[try Cseq1 Catch finally Cseq2 end]](e)(s,s') <=>
\except s" \in StateU: [[Cseq1]](e)(s,s") AND
        cases s" of
                isError() → s'=inError()
                [] isState(p) → IF exceptions(data(p)) THEN
                                        \exists s1 \in StateU: [[Catch]](e)(p,s1) AND
                                        cases s1 of
                                                isError() → s' = Error()
                                                [] isState(s2) → [[Cseq2]](e)(s2,s')
```

```
                                        END //cases-s1
                                ELSE
                                        [[Cseq2]](e)(p,s')
                                END //if-exception
        END //cases-s"

[[ I(Eseq) ]](e)(s,s') <=>
LET vseq \in ValueU*, s1 \in StateU: [[ Eseq ]](e)(s,s1,vseq)
IN
cases s1 of
        isError() → s'=inError()
        [] isState(s2) →
                IF hasUndefinedValue(vseq) THEN
                        s'=inError()
                ELSE
                cases [[I]](e) of
                        isProcedure(p) → \exists s3 \in StateU, v1 \in ValueU: p(vseq, s2, s3, v1) AND
                                        cases s3 of
                                                isError() → s'=inError()
                                                [] isState(s4) →
                                                cases v1 of
                                                        isUndefined() → s'=inError()
                                                        [] isValue(v') → s'=inStateU(s4)
                                                END //cases-v1
                                        END //cases-s3
                        [] … → s'=inError()
                END //cases-[[I]]
                END //if-hasUndefinedValue
END //let-vseq
```

[[`**type**/I:=T`]](e)(s,s') <=>

```
\exists tag \in Type-Tag: [[T]](e)(tag) AND
cases tag of
        isError-Tag() → s'=inError()
        [] … → \exists e1 \in Environment: e1 = push(e, I, tag) AND s'=inStateU(s)
END //cases-tag
```

CASE: Elif

```
[[ elif E then Cseq; Elif]](e)(s,s',tr) <=>
\exists v' \in ValueU, s" \in StateU: [[E]](e)(s,s",v') AND
        cases v' of
                isUndefined() → s' = inError()
                [] isValue(v1) → cases s" of
                                        isError() → s' = inError()
                                        [] isState(p)
                                        → IF v1=inTr(True) THEN
```

\exists s1 \in StateU: [[Cseq]](e)(p,s1) AND
cases s1 of
isError() → s' = inError() AND t = inTr(True)
[] isState(p2) → s' = inStateU(p2) AND t = inTr(True)
END //cases-s1
ELSE [[Elif]](e)(s,s',tr)
END //if-v1
END //cases-s"
END //cases-v'

CASE: Catch

[[catch “I”:Cseq; Catch]](e)(s,s') <=>
IF exceptions(data(s)) AND substring(“I”, ide(exception(data(s)))) THEN
LET s" = state(state(s), data(execute, exception(“no-exception”,inUndefined()), inUndefined()))
IN [[Cseq]](e)(s",s')
END //let-s"
ELSE
[[Catch]](e)(s,s')
END

CASE: E

[[**proc**(Pseq) S;R **end proc**]](e)(s,s',v) <=>
LET p \in Procedure, p(valseq, s0, s1, v', tag, s-tag) <=>
LET e' = push(e, identifiers(Pseq))
\exists varseq \in Variable*, s", s3 \in StateU, e", e"' \in Environment:
isTypeSeq(s-tag, valseq) AND
[[Pseq]](e')(e",varseq) AND
[[S]](e")(s0, s", e"') AND
cases s" of
isError() → s3=inError()
[] isState(s4) → \exists s2 \in State, v" \in ValueU:
s2 = update(s4,varseq,valseq) AND [[R]](e"')(s2,s3,v")
END //cases-s"
IN cases s3 of
isError() → s' = inError() AND v=inUndefined()
[] isState(p1) →
cases v" of
isUndefined() → s1=inError() AND v'=inUndefined()
[] isValue(v1) → s1 = inStateU(p1) AND v' = inValueU(v1)
END //cases-v"
END //let-e'
IN s' = inStateU(s) AND v = inValueU(p) END //let-p

[[**proc**(Pseq)::T S;R **end proc**]](e)(s,s',v) <=>
LET p \in Procedure, p(valseq, s0, s1, v', tag, s-tag) <=>

```
        LET   e' = push(e, identifiers(Pseq))
              [[T]](e')(tag)
              \exists varseq \in Variable*, s'', s3 \in StateU, e'', e''' \in Environment:
              isTypeSeq(s-tag, valseq) AND
              [[Pseq]](e')(e'',varseq) AND
              [[S]](e'')(s0, s'', e''') AND
              cases s'' of
                      isError() → s3=inError()
                      [] isState(s4) →  \exists s2 \in State, v'' \in ValueU:
                                        s2 = update(s4,varseq,valseq) AND [[R]](e''')(s2,s3,v'')
              END //cases-s''
        IN cases s3 of
              isError() → s' = inError() AND v=inUndefined()
              [] isState(p1) →
                      cases v'' of
                              isUndefined() → s1=inError() AND v'=inUndefined()
                              [] isValue(v1) →
                                      IF isType(tag, v1) THEN
                                              s1 = inStateU(p1) AND v' = inValueU(v1)
                                      ELSE
                                              s1=inError() AND v'=inUndefined()
                                      END //if-isType
                      END //cases-v''
        END //let-e'
IN s' = inStateU(s) AND v = inValueU(p) END //let-p

[[ module() S;R end module ]](e)(s,s',v) <=>
LET     m \in Module, m(iseq, v') <=>
        LET
              iseq = getExported(S)
              \exists s0, s1 \in StateU, v1 \in ValueU, e' \in Environment: [[S]](e)(s, s0, e') AND
              cases s0 of
                      isError() → s1=inError() AND v1=inUndefined()
                      [] isState(s11) → [[R]](e')(s11, s1, v1)
              END //cases-s0
        IN cases s1 of
              isError() → v'=inUndefined()
              [] isState(p) →
                      cases v1 of
                              isUndefined() → v'=inUndefined()
                              [] isValue(v') →
                                      LET
                                              v'' = getExportValues(e', p, iseq)
                                      IN
                                              cases v'' of
                                                      isUndefined() → v'=inUndefined()
                                                      [] isValue(v) →
```

```
                                        IF length(v) = length(iseq) THEN
                                             v'=inValueU(v)
                                        ELSE
                                             v'=inUndefined()
                                        END //if-length
                                        END //cases-v''
                                   END //let-p
                         END //cases-v1
                 END //cases-s1
           END //let-iseq
IN s'=inStateU(s) AND v=inValueU(m)
END //let-m

[[ E1 Bop E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
     cases s1 of
          isError() → s'=inError() AND v=inUndefined()
          [] isState(p) → cases v1 of
                              isUndefined() → s'=inError() AND v=inUndefined()
                              [] isValue(v') → \exists s2 \in StateU, v2 \in ValueU:
                                             [[E2]](e)(p,s2,v2) AND
     cases s2 of
          isError() → s'=inErrror() AND v=inUndefined()
          [] isState(q) → cases v2 of
                              isUndefined() → s'=inError() AND v=inUndefined()
                              [] isValue(v'') →
                                   cases v' of
                                        isInteger(i) → cases v'' of
                                                       isInteger(j) →
                                                            s'=inStateU(q) AND
                                                            [[Bop]](v',v'')(v)
                                                       [] … →s'=inError() AND
                                                             v=inUndefined()
                                                  END //cases-v''
                                        [] isFloat(i) → cases v'' of
                                                       isFloat(j) →
                                                            s'=inStateU(q) AND
                                                            [[Bop]](v',v'')(v)
                                                       [] … →s'=inError() AND
                                                             v=inUndefined()
                                                  END //cases-v''
                                        [] isRational(i) → cases v'' of
                                                       isRational(j) →
                                                            s'=inStateU(q) AND
                                                            [[Bop]](v',v'')(v)
                                                       [] … →s'=inError() AND
```

```
                                                                                    v=inUndefined()
                                                                        END //cases-v''
                                                    [] … → s'=inError() AND v=inUndefined()
                                            END //cases-v'
                                END //cases-v2
        END //cases-s2
                                END //cases-v1
        END //cases-s1

[[ I(Eseq) ]](e)(s,s',v) <=>
LET vseq \in ValueU*, s1 \in StateU: [[ Eseq ]](e)(s,s1,vseq)
IN
cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(s2) →
                IF hasUndefinedValue(vseq) THEN
                        s'=inError() AND v=inUndefined()
                ELSE
                cases [[I]](e) of
                        isProcedure(p) → \exists s3 \in StateU, v1 \in ValueU: p(vseq, s2, s3, v1) AND
                                        cases s3 of
                                                isError() → s'=inError() AND v=inUndefined()
                                                [] isState(s4) →
                                                cases v1 of
                                                        isUndefined() → s'=inError() AND v=inUndefind()
                                                        [] isValue(v') → s'=inStateU(s4)
                                                                                AND v=inValueU(v')
                                                END //cases-v1
                                        END //cases-s3
                        [] … → s'=inError() AND v=inUndefined()
                END //cases-[[I]]
                END //if-hasUndefinedValue
END //let-vseq

[[ I1:-I2 ]](e)(s,s',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() → s'=inError() AND
                                                                                v=inUndefined()
```

```
                                        [] isValue(mv) →
                                             cases evalMValue(mv) of
                                                  isUndefined() → s'=inError() AND
                                                            v=inUndefined()
                                                  [] isValue(v1) →
                                                       s'=inStateU(s) AND
                                                            v=inValueU(v1)
                                             END //cases-evalMValue
                                   END //cases-mv
                              ELSE
                                   s'=inError() AND v=inUndefined()
                              END //if-hasIdentifier
                         END //let-index
               [] … → s'=inError() AND v=inUndefined()
          END //cases-v'
     END //let-v'

[[ I1:-I2(Eseq) ]](e)(s,s',v) <=>
\exists s'' \in StateU, vs \in Value*: [[Eseq]](e)(s, s'', vs) AND
cases s'' of
     inError() → s'=inError() AND v=inUndefined()
     [] inState(s1) →     IF hasUndefinedValue(vseq) THEN
                              s'=inError() AND 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() → s'=inError() AND v=inUndefined()
                              [] isValue(mv) →
                                   LET    sm \in StateU, vm \in ValueU
                                        (sm, vm) = evalMProc(mv)
                                   IN
                                        s'=sm AND v=vm
                                   END //let-
                              END //cases-mv
                    ELSE
                         s'=inError() AND v=inUndefined()
                    END //if-hasIdentifier
               END //let-index
          [] … → s'=inError() AND v=inUndefined()
```

```
    END //cases-v'
    END //let-v'
                END //if-hasUndefinedValue
END //cases-s"

[[ E1 and E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
    isError() → s'=inError() AND v=inUndefined()
    [] isState(s11) →
        cases v1 of
            isUndefined() → s'=inError() AND v=inUndefined()
            [] isValue(v11) →
                cases v11 of
                    isBoolean(b1) →
                \exists s2 \in StateU, v2 \in ValueU: [[E2]](e)(s11, s2, v2) AND
            cases s2 of
                isError() → s'=inError() AND v=inUndefined()
                [] isState(s22) →
                    cases v2 of
                        isUndefined() → s'=inError() AND v=inUndefined()
                        [] isValue(v22) →
                            cases v22 of
                                isBoolean(b2) →
                              s'=inStateU(s22) AND v=inValueU(and(b1,b2))
                                [] … → s'=inError() AND v=inUndefined()
                    [] … → s'=inError() AND v=inUndefined()
                            END //cases-v22
                    END //cases-v2
            END //cases-s2
                    [] … → s'=inError() AND v=inUndefined()
                END //cases-v11
        END //cases-v1
END //cases-s1

[[ E1 or E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
    isError() → s'=inError() AND v=inUndefined()
    [] isState(s11) →
        cases v1 of
            isUndefined() → s'=inError() AND v=inUndefined()
            [] isValue(v11) →
                cases v11 of
                    isBoolean(b1) →
                \exists s2 \in StateU, v2 \in ValueU: [[E2]](e)(s11, s2, v2) AND
            cases s2 of
```

```
                        isError() → s'=inError() AND v=inUndefined()
                        [] isState(s22) →
                                cases v2 of
                                        isUndefined() → s'=inError() AND v=inUndefined()
                                        [] isValue(v22) →
                                                cases v22 of
                                                        isBoolean(b2) →
                                                   s'=inStateU(s22) AND v=inValueU(or(b1,b2))
                                                        [] … → s'=inError() AND v=inUndefined()
                                [] … → s'=inError() AND v=inUndefined()
                                                END //cases-v22
                                END //cases-v2
                END //cases-s2
                                [] … → s'=inError() AND v=inUndefined()
                        END //cases-v11
        END //cases-v1
END //cases-s1

[[ E1 = E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(s11) →
                cases v1 of
                        isUndefined() → s'=inError() AND v=inUndefined()
                        [] isValue(v11) →
                                \exists s2 \in StateU, v2 \in ValueU: [[E2]](e)(s11, s2, v2) AND
                        cases s2 of
                                isError() → s'=inError() AND v=inUndefined()
                                [] isState(s22) →
                                        cases v2 of
                                                isUndefined() → s'=inError() AND v=inUndefined()
                                                [] isValue(v22) → s'=inStateU(s22) AND
                                                                        v=inValueU(equals(v11,v22))
                                        END //cases-v2
                        END //cases-s2
                                        [] … → s'=inError() AND v=inUndefined()
                END //cases-v1
END //cases-s1

[[ E1 <> E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(s11) →
                cases v1 of
                        isUndefined() → s'=inError() AND v=inUndefined()
```

```
                        [] isValue(v11) →
                                \exists s2 \in StateU, v2 \in ValueU: [[E2]](e)(s11, s2, v2) AND
                        cases s2 of
                                isError() → s'=inError() AND v=inUndefined()
                                [] isState(s22) →
                                        cases v2 of
                                                isUndefined() → s'=inError() AND v=inUndefined()
                                                [] isValue(v22) → s'=inStateU(s22) AND
                                                                        v=inValueU(notequals(v11,v22))
                                        END //cases-v2
                        END //cases-s2
                                        [] … → s'=inError() AND v=inUndefined()
                END //cases-v1
END //cases-s1
```

```
[[ type(I,T) ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU, tag \in Type-Tag: [[T]](e)(s)(s1, v1, tag) AND
cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(p) →
                cases v1 of
                        isUndefined() → s'=inError() AND v=inUndefined()
                        [] isValue(v2) →
                                cases tag of
                                        isError-Tag() → v=inUndefined() AND s'=inError()
                                        [] … → \exits val \in Value: val = store(p)([[I]](e)) AND
                                        s'=inStateU(p) AND v=inValueU(inBoolean(isType(tag, val)))
                                END //cases-tag
                END //cases-v1
END //cases-s1
```

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-+

[[ E1 Bop E2 ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](e)(s,s1,v1) AND
cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(s11) →
            cases v1 of
                isUndefined() → s'=inError() AND v=inUndefined()
                [] isValue(v11) →
                    \exists s2 \in StateU, v2 \in ValueU: [[E2]](e)(s11, s2, v2) AND
                cases s2 of
                    isError() → s'=inError() AND v=inUndefined()
                    [] isState(s22) →
                        cases v2 of
                            isUndefined() → s'=inError() AND v=inUndefined()
                            [] isValue(v22) →
                            \exists v' \in Value: [[Bop]](v11, v22)(v') AND
                            s'=inStateU(s22) AND v=inValueU(v')
                        END //cases-v2
                END //cases-s2
                        [] … → s'=inError() AND v=inUndefined()
            END //cases-v1
END //cases-s1
```

<u>Case: Uop</u>

```
[[ 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 E ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E]](e)(s,s1,v1) AND
        cases s1 of
            isError() → s'=inError() AND v=inUndefined()
            [] isState(p) → cases v1 of
                        isUndefined() → s'=inError() AND v=inUndefined()
                        [] isValue(v'') → \exists v' \in Value: [[Uop]](v'')(v') AND
                            s'=inStateU(p) AND v=inValueU(v')
```

```
                        END //cases-v1
        END //cases-s1
```

<u>Case: Special Expressions</u>

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

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

```
// set construction
[[ {Eseq} ]](e)(s,s',v) <=>
\exists s" \in StateU, v' \in ValueU: [[Eseq]](e)(s,s",v') AND
        cases s" of
                isError() → s'=inError() AND v=inUndefined()
                [] isState(p) →
                        cases v' of
                                isUndefined() → s'=inError() AND v=inUndefined()
                                [] isValue(v") →
                                        cases v" of
                                                isValue*(vs) →
                                                                IF hasUndefinedValue(vs) THEN
                                                                        s'=inError() AND v=inUndefined()
                                                                ELSE
                                                \exists pl \in List: permutation(cons(vs,emptyList), pl)
                                                                AND s'=inStateU(p) AND
                                                                v=inValue(list2Set(pl))
                                                                END //if-hasUndefinedValue
                                                [] … → s'=inError() AND v=inUndefined()
                        END //cases-v'
        END //cases-s"

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

[[ op(E1,E2) ]](e)(s,s',v) <=>
\exists s2 \in StateU, v' \in ValueU: [[E2]](e)(s,s2,v') AND
cases s2 of
        isError() → s' = inError() AND v = inUndefined()
        []isState(p) → cases v' of
                                isUndefined() → s'=inError() AND v=inUndefined()
                                [] isValue(v1) →
                                LET
                                        vseq = expValues(v1)
                                        k = length(vseq)
                                IN
                                \exists s3 \in StateU, v3 \in ValueU: [[E1]](e)(p, s3, v3) AND
                                cases s3 of
                                        isError() → s'=inError() AND v=inUndefined()
                                        [] isState(p1) →
                                        cases v3 of
                                                isUndefined() → s'=inError() AND v=inUndefined()
                                                [] isValue(v33) →
                                                cases v33 of
```

```
                                                isInteger(n) →
                                                IF n > 0 AND n <= k THEN
                                                      s'=inStateU(p1) AND
                                                            v=inValueU(access(n,vseq))
                                                ELSE s'=inError() AND v=inUndefined() END
                                                [] … → s'=inError() AND v=inUndefined()
                                          END //cases-v33
                                    END //cases-v3
                              END //cases-s3
                              END //let-vseq
                        END //cases-v'
END //cases-s2

[[ op(E) ]](e)(s,s',v) <=>
\exists s1 \in State, s2 \in StateU, v' \in ValueU: [[E]](e)(s1,s2,v') AND
cases s2 of
      isError() → s' = inError() AND v = inUndefined()
      []isState(p) → cases v' of
                              isUndefined() → s'=inError() AND v=inUndefined()
                              [] isValue(v1) →
                                    s' = inStateU(p) AND v' = expValues(v1)
                        END //cases-v'
END //cases-s2

[[ op(E1...E2,E3) ]](e)(s,s',v) <=>
\exists s2 \in StateU, v' \in ValueU: [[E3]](e)(s,s2,v') AND
cases s2 of
      isError() → s' = inError() AND v = inUndefined()
      []isState(p) → cases v' of
                              isUndefined() → s'=inError() AND v=inUndefined()
                              [] isValue(v1) →
                              LET
                                    vseq = expValues(v1)
                                    k = length(vseq)
                              IN
                              \exists s3 \in StateU, v3 \in ValueU: [[E1]](e)(p, s3, v3) AND
                              cases s3 of
                                    isError() → s'=inError() AND v=inUndefined()
                                    [] isState(p1) →
                                    cases v3 of
                                          isUndefined() → s'=inError() AND v=inUndefined()
                                          [] isValue(v33) →
                                          cases v33 of
                                                isInteger(n) →
                              \exists s4 \in StateU, v4 \in ValueU: [[E2]](e)(p1, s4, v4) AND
                              cases s4 of
                                    isError() → s'=inError() AND v=inUndefined()
```

```
                              [] isState(p2) →
                              cases v4 of
                                   isUndefined() → s'=inError() AND v=inUndefined()
                                   [] isValue(v44) →
                                        isInteger(m) →
                                        IF n <= m AND n > 0 AND m <= k THEN
                                             s'=inStateU(p1) AND
                                                  v=inValueU(access(n, m ,vseq))
                                        ELSE s'=inError() AND v=inUndefined() END
                                        [] … → s'=inError() AND v=inUndefined()
                              END //cases-v4
                         END //cases-s4
                                        [] … → s'=inError() AND v=inUndefined()
                                   END //cases-v33
                              END //cases-v3
                         END //cases-s3
                         END //let-vseq
                    END //cases-v'
END //cases-s2
```

[[**nops**(E)]](e)(s,s',v) <=>

```
\exists s1 \in State, s2 \in StateU, v' \in ValueU: [[E]](e)(s1,s2,v') AND
cases s2 of
     isError() → s' = inError() AND v = inUndefined()
     []isState(p) → cases v' of
                         isUndefined() → s'=inError() AND v=inUndefined()
                         [] isValue(v1) → s'=inStateU(p) AND v=inValueU(length(v1))
                    END //cases-v'
END //cases-s2
```

[[**subsop**(E1=E2,E3)]](e)(s,s',v) <=>

```
\exists s1 \in StateU, v1 \in ValueU: [[E3]](e)(s, s1, v1) AND
cases s1 of
     isError() → s'=inError() AND v=inUndefined()
     [] isState(s11) →
          cases v1 of
               isUndefined() → s'=inError() AND v=inUndefined()
               [] isValue(v11) →
                    \exists s2 \in StateU, v2 \in ValueU: [[E1]](e)(s11, s2, v2) AND
                    cases s2 of
                         isError() → s'=inError() AND v=inUndefined()
                         [] isState(s22) →
                         cases v2 of
                              isUndefined() → s'=inError() AND v=inUndefined()
                              [] isValue(v22) →
                              cases v22 of
                                   isInteger(j) →
```

```
                              \exists s3 \in StateU, v3 \in ValueU: [[E1]](e)(s22, s3, v3)
                              AND
                              cases s3 of
                                  isError() → s'=inError() AND v=inUndefined()
                                  [] isState(s33) →
                              cases v3 of
                                  isUndefined()→ s'=inError() AND v=inUndefined()
                                  [] isValue(v33) →
                                      LET    vs \in ValueU
                                             vs = subsop(j, v33, v11)
                                      IN
                                      cases vs of
                                      inUndefined() →
                                      s'=inError() AND v=inUndefined()
                                      [] isValue(vs1) →
                                      s'=inStateU(s33) AND v=inValueU(vs1)
                                      END //cases-vs
                                      END //let-vs
                              END //cases-v3
                              END //cases-s3
                                  [] … → s'=inError() AND v=inUndefined()
                              END //cases-v22
                          END //cases-v2
                      END //cases-s2
              END //cases-v1
END //cases-s1

[[ subs(I=E1,E2) ]](e)(s,s',v) <=>
\exists s1 \in StateU, v1 \in ValueU: [[E1]](s, s1, v1) AND
    cases s1 of
        isError() → s'=inError() AND v=inUndefined()
        [] isState(s11) →
        cases v1 of
            isUndefined() → s'=inError() AND v=inUndefined()
            [] isValue(v1) → \exists e1 \in Environment, s2 \in State: e1 = push(I, e1) AND
                              s2 = update([[I]](e1), s11) AND [[E2]](e1)(s11, s', v)
        END //cases-v1
    END //cases-s1

[[ "E" ]](e)(s,s',v) <=> s' = s AND v=inValueU(inUneval("E"))

[[ eval(I,1) ]](e)(s,s',v) <=>
        LET
            v' = store(s)([[I]](e))
        IN
            cases v' of
                isUneval(u) →
```

```
                        cases eval(u) of
                            isUneval(u')→s'=inStateU(s) AND v=inValueU(u')
                            [] … → s'=inError() AND v=inUndefined()
                        END //cases-eval
                    [] … → s'=inError() AND v=inUndefined()
                END //cases-v'
            END //let-in

[[ seq(E1,I=E2...E3) ]](e)(s,s',v) <=>
\exists v' \in ValueU, s'' \in StateU: [[E2]](e)(s,s'',v') AND
cases s'' of
isError() → s'=inError() AND v = inUndefined()
[] isState(p) → cases v' of
                isUndefined() → s'=Error() AND v = inUndefined()
                [] isValue(v'') →
                cases v'' of
                    isInteger(m)→\exists v'' \in ValueU, s''' \in StateU:[[E3]](e)(p,s''',v'') AND
                cases s''' of
                    isError() → s'=inError() AND v=inUndefined()
                    [] isState(p1) →
                cases v''' of
                    isUndefined() → s'=inError() AND v=inUndefined()
                    isValue(v4) →
                    cases v4 of
                    isInteger(n) →
                LET
                    vseq = expRangeValues(m,n)
                IN
                    \exists k, t, u \in StateU*, e1 \in Environment, vs \in Value*:
                    t(1)=inStateU(s) AND u(1)=inStateU(s) AND e1 = push(I,e) AND
                    ( \forall i \in Nat'_k: seq(i, t, u, I, e1, vseq, vs, [[E1]]) ) AND
                        cases u(k') of
                            isError() → s' = inError() AND v = inUndefined()
                            [] isState(m) →
                            ( k' < length(vseq) AND u(k')=inError() AND s'=u(k')
                                AND v=inUndefined()
                            ) OR
                            ( k' = length(vs) AND s'=t(k') AND v=inValueU(vs)
                            )
                        END //cases-u(k')
                END //let-vseq
                    [] .. → s'=inError() AND v=inUndefined()
                    END //cases-v4
                END //cases-v'''
                END //cases-s'''
                [] … → s'=inError() AND v=inUndefined()
                END //cases-v''
```

```
            END //cases-v'
END //cases-s"

[[ seq(E1, I in E2) ]](e)(s,s',v) <=>
\exists v' \in ValueU, s" \in StateU: [[E2]](e)(s,s",v') AND
cases s" of
isError() → s'=inError() AND v = inUndefined()
[] isState(p) → cases v' of
                isUndefined() → s'=Error() AND v = inUndefined()
                [] isValue(v") →
                LET
                    vseq = expValues(v")
                IN
                    IF hasUndefinedValue(vseq) THEN
                        s'=inError() AND v=inUndefined()
                    ELSE
                    \exists k, t, u \in StateU*, e1 \in Environment, vs \in Value*:
                    t(1)=inStateU(s) AND u(1)=inStateU(s) AND e1 = push(I,e) AND
                    ( \forall i \in Nat'_k: seq(i,t,u,I,e1,vseq,vs,[[E1]]) ) AND
                        cases u(k') of
                            isError() → s' = inError() AND v = inUndefined()
                            [] isState(m) →
                            ( k' < length(vseq) AND access(k', vseq)=isUndefined()
                                AND s'=inError() AND v=inUndefined()
                            ) OR
                            ( k' = length(vs) AND s'=t(k') AND v=inValueU(vs)
                            )
                        END //cases-u(k')
                    END //if-hasUndefinedValues
                END //let-vseq
            END //cases-v'
END //cases-s"
```

<u>Case: Type</u>

[[**integer**]](e)(tag) <=> tag = inType-TagU(Integer-Tag)

[[**rational**]](e)(tag) <=> tag = inType-TagU(Rational-Tag)

[[**float**]](e)(tag) <=> tag = inType-TagU(Float-Tag)

[[**boolean**]](e)(tag) <=> tag = inType-TagU(Boolean-Tag)

[[**string**]](e)(tag) <=> tag = inType-TagU(String-Tag)

[[**symbol**]](e)(tag) <=> tag = inType-TagU(Symbol-Tag)

```
[[ list(T) ]](e)(tag) <=>        LET [[T]](e)(s-tag)
                                 IN
                                     cases s-tag of
                                         isError-Tag() → tag = Error-Tag
                                         [] … → tag = inType-TagU(cons(emtpyList-Tag, s-tag))
                                     END //cases-s-tag
                                 END //let-tag1
[[ {T} ]](e)(tag) <=>            LET [[T]](e)(s-tag)
                                 IN
                                     cases s-tag of
                                         isError-Tag() → tag = Error-Tag
                                         [] … → tag = inType-TagU(cons(emtpySet-Tag, s-tag))
                                     END //cases-s-tag
                                 END //let-tag1

[[ [ Tseq ] ]](e)(tag) <=>       LET [[Tseq]](e)(s-tag)
                                 IN
                                     cases s-tag of
                                         isType-Tag*(s) →
                                             IF hasErrorTag(s-tag) THEN
                                                 tag = Error-Tag
                                             ELSE
                                             tag = inType-TagU(cons(emtpyRecord-Tag, s-tag))
                                             END
                                         [] … → tag = Error-Tag
                                     END //cases-s-tag
                                 END //let-tag1

[[ Or(Tseq) ]](e)(tag) <=>       LET [[Tseq]](e)(s-tag)
                                 IN
                                     cases s-tag of
                                         isType-Tag*(s) →
                                             IF hasErrorTag(s-tag) THEN
                                                 tag = Error-Tag
                                             ELSE
                                             tag = inType-TagU(cons(emtpyOr-Tag, s-tag))
                                             END
                                         [] … → tag = Error-Tag
                                     END //cases-s-tag
                                 END //let-tag1

[[ proc[T](Tseq) ]](e)(tag) <=>      LET
                                         [[ T ]](e)(tag1)
                                         [[ Tseq ]](e)(tag2)
                                 IN
                                     cases tag1 of
                                         isError-Tag() → tag = Error-Tag
```

```
                        [] … →
                cases tag2 of
                        isType-Tag*(s) →
                                IF hasErrorTag(tag2) THEN
                                        tag = Error-Tag
                                ELSE
                                tag=inType-TagU(cons(emtpyProc-Tag, tag1, tag2))
                                END
                        [] … → tag = Error-Tag
                END //cases-tag2
                END //cases-tag1
        END //let-tag1
```

```
[[ I ]](e)(tag) <=>     cases [[I]](e) of
                                isType-Tag(tag1) → tag = tag1
                                [] … → tag = Error-Tag
                        END //cases-tag
```

...

[[**Uneval**]](e)(tag) <=>   tag = Uneval-Tag

[[**anything**]](e)(tag) <=>   tag = inType-TagU(Anything-Tag)