A Type Checker for *MiniMaple*[*]

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

November 7, 2011

Abstract

In this paper, we present the syntactic definition and the formal type system for a substantial subset of the language of the computer algebra system Maple, which we call *MiniMaple*. The goal of the type system is to prevent runtime typing errors by static analysis of the source code of *MiniMaple* programs. The type system is implemented by a type checker, which verifies the type correctness of *MiniMaple* programs respectively reports typing errors.

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

Contents

1 Introduction

Computer algebra programs written in symbolic computation languages such as Maple [5] and Mathematica [6] sometimes do not behave as expected, e.g. by triggering runtime errors or computing wrong results. To prevent such errors we have started a project whose aim is to design and develop a tool for the static analysis of computer algebra programs [14]. The tool will find errors in programs annotated with extra information such as variable types and method contracts [12], in particular type inconsistencies and violations of method preconditions.

As a starting point, we have designed a type system for (a subset of) the symbolic computation language *Maple* [5] because type safety is a prerequisite of program correctness. To validate our results we have implemented a type checker for the language. The results can be applied e.g. to the Maple package *DifferenceDifferential* [10] developed at our institute for the computation of bivariate difference-differential dimension polynomials. In this report, we describe both the formalization of the type system and its implementation.

There are various dynamically type computer algebra languages; Mathematica and Maple being the most prominent ones. We have chosen Maple for the following reasons:

- Maple has an imperative style of programming while Mathematica has a rule-based programming style with more complex semantics.
- Maple has type annotations for runtime checking which can be directly applied for static analysis. (There are also parameter annotations in Mathematica but they are use used for selecting the appropriate rule at runtime).

Still the results we derive with type checking Maple can be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple. During our study, we found the following special features for type checking Maple programs (which are typical for most computer algebra languages):

- The language does support some non-standard objects, e.g. symbol and unevaluated expressions.
- There is no clear difference between declaration and assignment. A global variable is introduced by an assignment; a subsequent assignment may modify the type information for the variable.

- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-checks which makes the selection of the respective code-block for further execution. This makes type inference more complex.

- The language allows runtime type checking by type annotations but these annotations are optional which give rise to type ambiguities.

- Maple has a kind of polymorphic type system [7]; since Maple has a hierarchy of types in a sub-typing relationship, one variable can be assigned values of different types. This also makes type inference more complex.

Although there is no complete static type system for Maple; there have been various approaches to exploit the type information in Maple for various purposes.

For instance, the Maple package Gauss [13] introduced parameterized types in Maple. Gauss runs on top of Maple and allows to implement generic algorithms in Maple in an AXIOM-like [2] manner. The system supports parameterized types and parameterized abstract types, however these are checked at runtime only. The package was introduced in Maple V Release 2 and later evolved into the *domains* package [4].

In [9], partial evaluation for Maple is developed. The focus of the work is to exploit the available static type information for Maple program manipulation by generating specialized programs [15]. The system aims to support generic programming that becomes more complex for dynamically type checked interpreted languages like Maple. The language of the partial evaluator has similar syntactic constructs (but fewer expressions) as *MiniMaple* and supports very limited types e.g. integers, rationals, floats and strings.

The work [8] aims at finding a mathematical description of the interfaces between the Maple routines. The paper mainly presents the study of the actual contracts in use by Maple routines. The contracts are statements with certain (static and dynamic) logical properties.

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

Here is the list of our contributions described in this paper:

- We have defined a formal grammar for *MiniMaple*.

- We have defined a type system for *MiniMaple*. The type system consists of typing judgments, logical rules to derive the judgments and auxiliary functions and predicates which are used in the rules.
- We have implemented a type checker for the type system.

We have not yet proved the soundness of the typing judgments and formal semantics of the language. We aim to do it provided that there remains sufficient time during our PhD dissertation.

The rest of the paper is organized as follows: in Section 2, we discuss the syntax for *MiniMaple* by an example which we will type check. In Section 3, the type system designed for *MiniMaple* is explained. In Section 4, implementation of a type checker for the type system is discussed. Section 5 presents conclusions and future work.

2 *MiniMaple*

MiniMaple is a simple but substantial subset of Maple that covers all the syntactic domains of Maple but has fewer alternatives in each domain than Maple; in particular, Maple has many expressions which are not supported in our language. The complete syntactic definition of *MiniMaple* is given in Appendix A.

The following example gives an overview of the syntax of *MiniMaple*. We will use the same example in the following sections for type checking and for general discussion.

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

```
18.             end if;
19.         end do;
20.         status:=-1;
21.         return [si,sf];
22.         end proc;
23. result := prod[1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);
24. print(result);
25. print(status);
```

The above example is a *MiniMaple* program that is a command followed by a procedure definition and its application. The procedure takes a list of integers and floats and computes the product of these integers and floats separately; the procedure returns a tuple of integer and float as the product of respective integers and floats in the list. The procedure may also terminate prematurely for certain inputs, i.e. either for an integer value 0 or for a float value less than 0.5 in the list; in this case the procedure computes the respective products just before the index at which the aforementioned terminating input occurs.

In the above example, the variable *status* has no type information at the "global" declaration, because global clause cannot have type information in the language. In the body of loop variable x can be of type integer or float depending on the type of the selected element of the list; similarly after the loop x can have any of the two types (integer or float) depending upon the execution of the corresponding if-else branch.

In the following we discuss our approach in *MiniMaple* to address the features highlighted above:

- *MiniMaple* uses only Maple type annotations for *static type checking*, which Maple uses for *dynamic type checking*.
- We introduce *global* and *local* contexts to handle the different semantics of variables inside and outside of the body of a procedure respective loop.
  - In a *global* context new identifiers may be introduced by assignments and the types of identifiers may change arbitrarily by assignments.
  - In a *local* context identifiers can only be introduced by declarations. The types of identifiers can only be *specialized* i.e. the new value of an identifier should be a subtype of the declared identifier super-type.

- A sub-typing relation > is observed while specializing the type of a variable, e.g. anything > Or(integer,float) > float > integer.

Now we present the result of our static (type) analysis of the above example:

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

By the static analysis of the program, the program is annotated with the type environments of the form # π ={variable:type,...}.

The type environment at line *6* shows the types of the respective variables as determined by the static analysis of parameter and identifier declarations (*global* and *local*).

The static analysis of two branches of conditional command in the body of loop introduces the type environments at lines *12* and *18* respectively;

the type of variable x is determined as *integer* and *float* by the conditional type-expressions respectively.

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

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

The local type information introduced/modified by the analysis of body of loop doesn't effect the global type information. The type environment at line *6* and *26* reflects this fact for variables *status*, i and x. This is because of the fact that the number of loop iterations might have an effect on the type of the variable otherwise. For example in the following program (which is not correct according to our type system)

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

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

- After first iteration π = {..., x:list(anything), y:list(list(anything)) ,...}
- After second iteration π = {..., x:list(list(anything)), y:list(list(list(anything))),...}

The type environments after first and second iterations show how the types of x and y get influenced by the number of loop iterations.

In the next section we define our type system for *MiniMaple* formally.

3 A Type System for *MiniMaple*

A *type* is (an upper bound on) the range of values of a variable. A *type system* is a set of formal typing rules to determine the variables types from the text of a program. A type system prevents *forbidden errors* during the execution of the program [7]. It completely prevents the *untrapped errors* and also a large class of *trapped errors*. *Untrapped errors* may go unnoticed for a while

and later cause an arbitrary behavior during execution of a program, while *trapped errors* immediately stop execution.

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

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

can be read as "in the given type environment π, E is a well-typed expression of type τ".

A type system is *sound*, if the deduced types indeed capture the program values exhibited at runtime. For example, if we can derive the typing judgment

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

and e is an environment which is consistent with π, then

$$[[E]]e \in [[\tau]]$$

i.e. at runtime the expression E in environment e indeed denotes a value of type τ ($[[E]]$ describes the runtime value of E and $[[\tau]]$ describes the set of values specified by type τ).

We have defined a typing judgment for each syntactic domain of *Mini-Maple*. Logical rules are defined to derive the typing judgments by using auxiliary functions and predicates. In the following subsection we discuss the declarations for typing judgments.

3.1 Declarations

The typing judgments depend on the following kinds of objects:

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

- $asgnset_n \subseteq$ Identifier: An updated set of assignable identifiers introduced by type checking the declarations.
- $expidset \subseteq$ Identifier: A set of exported identifiers introduced by type checking the export declarations in procedure/module.
- $expidset_n \subseteq$ Identifier: An updated set of exported identifiers introduced by the export declarations in procedure/module.
- $\epsilon set \subseteq$ Identifier: A set of thrown exceptions introduced by type checking the corresponding syntactic phrase.
- $\epsilon cset \subseteq$ Identifier: A set of caught exceptions by type checking the catch syntactic phrase.
- $\tau set \subseteq$ Type: A set of return types introduced by type checking the corresponding syntactic phrase.
- $\tau seq \subseteq$ Type: A sequence of types introduced by type checking the corresponding syntactic phrase.
- $rflag \in$ {aret, not_aret}: A return flag to check if the last statement of every execution of the corresponding syntactic phrase is a *return* command.

In the following subsection we list the typing judgments for the various syntactic domains of *MiniMaple*.

3.2 Typing Judgments

A typing judgment has following four parts (the names will be explained below):

Input The "input" variables are the variables on the left side of the judgment.

Body The body represents any phrase for the corresponding syntactic domain of *MiniMaple*.

Output The "output" variables are the variables enclosed in parentheses on the right side of the judgment; they represent the type information generated by deriving the judgment.

Tag The tag indicates the syntactic domain to which the judgment is associated.

For example in the following typing judgment

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

we have the following pieces of information:

Input = π, c, *asgnset*

Body = C

Output = (π_1, τset, ϵset, *rflag*)

Tag = **comm**

In the following subsections we explain the typing judgments for corresponding syntactic domains.

3.2.1 Program

Typing Judgment

$$\vdash Prog{:}\mathbf{prog}$$

Prog is a well-typed program.

Example

- *Prog* = x:=10;

3.2.2 Command Sequence

Typing Judgment

$$\pi,\ c,\ asgnset \vdash Cseq{:}(\pi_1,\ \tau set,\ \epsilon set,\ rflag)\mathbf{cseq}$$

With the given π, c and *asgnset*, *Cseq* is a well-typed command sequence with type annotation (π_1, τset, ϵset, *rflag*).

Example

- π = {x:anything, y:integer}
- c = global
- *asgnset* = {x,y}

- *Cseq* = x:=y;y:=y+5;
- π_1 = {x:integer, y:integer}
- τset = {}
- ϵset = {}
- *rflag* = not_aret

3.2.3 Command

Typing Judgment

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

With the given π, c and *asgnset*, C is a well-typed command with type annotation (π_1, τset, ϵset, *rflag*).

Example

- π = {x:anything, y:integer}
- c = global
- *asgnset* = {x}
- C = x:=y;
- π_1 = {x:integer, y:integer}
- τset = {}
- ϵset = {}
- *rflag* = not_aret

3.2.4 Elif

Typing Judgment

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

With the given π, c and *asgnset*, *Elif* is a well-typed conditional guard (else-if) with type annotation (π_1, πset, τset, ϵset, *rflag*).

Example

- π = {x:Or(integer,boolean), y:integer}
- c = global
- *asgnset* = {x,y}
- *Elif* = **elif type**(x,integer) **then** x:=y*10;
- π_1 = {x:integer, y:integer}
- πset = {x:boolean}
- τset = {}
- ϵset = {}
- *rflag* = not_aret

3.2.5 Catch

Typing Judgment

$$\pi,\ asgnset \vdash Catch{:}(\tau set,\ \epsilon set,\ \epsilon cset,\ rflag)\textbf{catch}$$

With the given π and *asgnset*, *Catch* is a well-typed exception handler with type annotation (τset, ϵset, $\epsilon cset$, *rflag*).

Example

- π = {x:integer, y:integer}
- c = global
- *asgnset* = {x,y}
- *Catch* = **try** result:=divide(x,y); **catch** "notdefined":result:=-1; **end;**
- τset = {}
- ϵset = {}
- $\epsilon cset$ = {notdefined:string}
- *rflag* = not_aret

3.2.6 Expression Sequence

Typing Judgment

$$\pi \vdash Eseq{:}(\tau seq)\mathbf{eseq}$$

With the given π, *Eseq* is a well-typed expression sequence with type annotation τseq.

Example

- π = {x:float, y:integer}
- *Eseq* = y*5;x*y
- τseq = integer,float

3.2.7 Expression

Typing Judgment

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

With the given π, E is a well-typed expression with type τ.

Example

- π = {x:Or(float,integer), y:integer}
- E = **type**(x,integer)
- τ = integer

3.2.8 Sequence

Typing Judgment

$$\pi,\ asgnset,\ expidset \vdash S{:}(\pi_1,\ asgnset_1,\ expidset_1)\mathbf{seq}$$

With the given π, *asgnset* and *expidset*, S is a well-typed declaration with type annotation (π_1, $asgnset_1$, $expidset_1$).

Example

- $\pi = \{\}$
- $asgnset = \{\}$
- $expidset = \{\}$
- S = **global** x; **local** y::integer; **export** z;
- π_1 = {x:anything, y:integer, z:anything}
- $asgnset_1 = \{x,y,z\}$
- $expidset_1 = \{z\}$

3.2.9 Recurrence

Typing Judgment

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

With the given π, c and *asgnset*, *Recc* is a well-typed body of a module/procedure with type annotation (π_1, τset, ϵset, *rflag*).

Recc is like a command sequence, the only difference is that in *recc*, *Cseq* may be followed by an expression i.e. *Cseq;E*.

3.2.10 Parameter Sequence

Typing Judgment

$$\pi \vdash Pseq{:}(\pi_1)\textbf{pseq}$$

With the given π, *Pseq* is a well-typed parameter sequence with type annotation π_1.

Example

- $\pi = \{\}$
- *Pseq* = **proc**(x::integer, y::string)
- π_1 = {x:integer, y:string}

3.2.11 Parameter

Typing Judgment

$$\pi \vdash P{:}(\pi_1)\mathbf{param}$$

With the given π, P is a well-typed parameter with type annotation π_1.

Example

- $\pi = \{\}$
- $P =$ **proc**(x::integer)
- $\pi_1 = \{$x:integer$\}$

3.2.12 Modifier

Typing Judgment

$$\pi \vdash M{:}(\tau)\mathbf{mod}$$

With the given π, M is a well-typed variable modifier with type τ.

Example

- $\pi = \{$x:anything, ...$\}$
- $M =$ x::string
- $\tau =$ string

3.2.13 Identifier Sequence

Typing Judgment

$$\pi \vdash \mathit{Idseq}{:}(\tau \mathit{seq})\mathbf{idseq}$$

With the given π, *Idseq* is a well-typed identifier (variable) sequence with type τseq.

Example

- $\pi = \{$x:anything, y:Or(integer,boolean)$\}$
- *Iseq* = x,y;
- τseq = anything,Or(integer,boolean)

3.2.14 Identifier

Typing Judgment

$$\pi \vdash I{:}(\tau)\mathbf{id}$$

With the given π, I is a well-typed identifier (variable) with type τ.

Example

- π = {x:anything, ...}
- I = x;
- τ = anything

3.2.15 Identifier Typed Sequence

Typing Judgment

$$\pi,\ asgnset \vdash Idtseq{:}(\tau seq,\ asgnset_1)\mathbf{idtseq}$$

With the given π and *asgnset*, *Idtseq* is a well-typed typed-identifier sequence with type (τseq, $asgnset_1$).

Example

- π = {}
- *asgnset* = {}
- *Idtseq* = x::integer, y::string;
- τseq = integer,string
- $asgnset_1$ = {x,y}

3.2.16 Identifier Typed

Typing Judgment

$$\pi,\ asgnset \vdash Idt{:}(\tau seq,\ asgnset_1)\mathbf{idt}$$

With the given π and *asgnset*, *Idt* is a well-typed typed-identifier with type (τseq, $asgnset_1$).

Example

- $\pi = \{\}$
- $asgnset = \{\}$
- Idt = x::integer;
- τ = integer
- $asgnset_1 = \{x\}$

3.2.17 Binary Operators

Typing Judgment

$$\pi \vdash Bop{:}(\tau)\textbf{bop}$$

With the given π, Bop is a well-typed binary operator with type τ.

Example

- π = {x:float, y:integer}
- Bop = x*y;
- τ = float

3.2.18 Unary Operators

Typing Judgment

$$\pi \vdash Uop{:}(\tau)\textbf{uop}$$

With the given π, Uop is a well-typed unary operator with type τ.

Example

- π = {x:boolean, ...}
- Uop = not x;
- τ = boolean

3.2.19 Especial Operators

Typing Judgment

$$\pi \vdash Esop{:}(\tau)\mathbf{esop}$$

With the given π, *Esop* is a well-typed especial operator with type τ.

Example

- π = {x:list(string), ...}
- *Esop* = nops(x)
- τ = integer

3.2.20 Type Sequence

Typing Judgment

$$\pi \vdash Tseq{:}(\tau seq)\mathbf{tseq}$$

With the given π, *Tseq* is a well-typed type sequence with type τseq.

3.2.21 Type

Typing Judgment

$$\pi \vdash T{:}(\tau)\mathbf{type}$$

With the given π, T is a well-typed defined type τ.

Example

- π = {x:integer, y:string}
- T = **type**(x,integer);
- τ = integer

3.2.22 Numeral

It is a sequence of decimal digits.

4 A Type Checker for *MiniMaple*

In this section, we discuss the implementation of the type checker [11] and its application by an example.

The general work-flow of the type checker is shown in Figure1:

- The lexical analyzer translates a *MiniMaple* program into a sequence of tokens.
- The parser generates an abstract syntax tree (AST) from the tokens.
- The type checker annotates the AST generated by parser (respectively reports a typing error).

Additionally the lexical analyzer, parser and type checker may generate errors and warning messages. Later a verifier will use the annotated AST (generated by the type checker) for checking the correctness of the program.

We have used the parser generator ANTLR [1] for the implementation of the lexical analyzer and parser, and implemented type checker in Java [3]. In the current release, the type checker consists of 159 Java classes and approx. 15K+ lines of code.

The type checker is fully operational and has been tested with small examples (one is given below). As discussed above the Maple package *DifferenceDifferential* developed at RISC will be a more comprehensive test case for the *MiniMaple* type checker.

Currently the type checker has the following limitations:

- All the code must be contained in a single Maple file.
- Procedure and module definitions must precede their application. In the future, we may consider the following alternatives:
  - We can use forward declarations i.e. procedure/module prototypes embedded in comments in the *MiniMaple* program.
  - We can use two-pass type checking. In the first pass we can collect the procedure and module information and in second pass we can type check rest of the program with the given procedure/module definitions.
- Procedure parameter(s) and return types have to be explicitly given in the *MiniMaple* program. In the future, we may use type inference (to determine the parameter(s) and return types) by the applications of the parameter(s) and procedure(s).

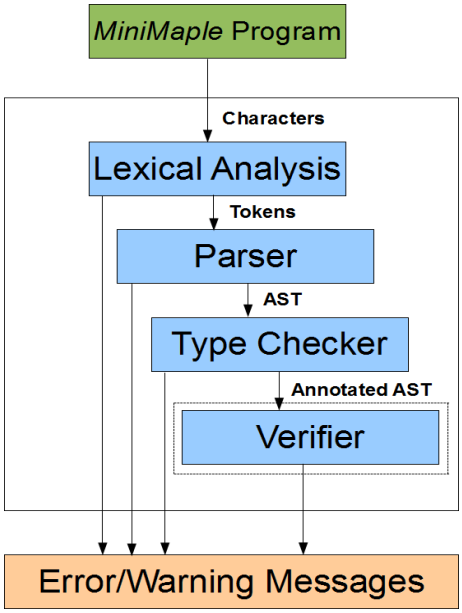

Figure 1: General work-flow of the Type Checker

- Type checking terminates at very first error message, so one cannot see all the type information flow if the type checking fails this way.

In the following we present the demonstration of an example *MiniMaple* program that is type checked with our type checker. The output of the type checker is an annotated abstract syntax tree (AST). The example *MiniMaple* file is shown in the following listing:

```
status:=0;
prod := proc(l::list(Or(integer,float)))::[integer,float];
        global status;
        local i,x::Or(integer,float), si::integer:=1, sf::float:=1.0;
        for i from 1 by 1 to nops(l) do
                x:=l[i];
                status:=i;
                if type(x,integer) then
                        if (x = 0) then
                                return [si,sf];
                        end if;
                        si:=si*x;
                elif type(x,float) then
                        if (x < 0.5) then
```

```
                                return [si,sf];
                            end if;
                            sf:=sf*x;
                    end if;
            end do;
            status:=-1;
            return [si,sf];
            end proc;
result:= prod([1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);
print(result);
print(status);
```

To type check the above program, we execute the following command;

```
java fmrisc/typechecker/MiniMapleTypeChecker -typecheck Test6.m
```

In the following we show by a sequence of screen-shots the output of the type checker for the above program. The output of the type checker in essence is an annotated abstract syntax tree (AST). So in this demonstration we show some parts of the program, which are type annotated by the type checker. The complete output of the type checker for this program is given in Appendix E.

Each screen-shot has in principle three major syntactic variations as follows:

Type annotation Type annotation starts and ends with a starred line containing the name of the corresponding syntactic domain as the head. The type annotations for the respective syntactic phrase is shown between the start and end line of the annotation. The body of the type annotations has the name-value pairs for the annotation parts.

***MiniMaple* source** This is the original text from *MiniMaple* program.

Abstract syntax The notations for the respective syntactic block are enclosed in #. This represents an abstract syntax tree of the corresponding syntactic phrase.

For example in the following output information

```
1  #globalseq#
2  global
3  #expression#
4  #idexp#
5  status
6  *************IDENTIFIER-ANNOTATION BEGIN*************
7  integer
8  *************IDENTIFIER-ANNOTATION END***************
```

```
9  ;
10 ************|GLOBAL| SEQUENCE-ANNOTATION BEGIN************
11 PI -> [
12 status:anything
13 ]
14 AsgnIDSet -> {status}
15 ExpIDSet -> {}
16 ************|GLOBAL| SEQUENCE-ANNOTATION END**************
```

we have the following elements:

Type annotation for the syntax of a global declaration sequence is between lines 10 and 16. The corresponding annotation values for PI, AsgnIDSet and ExpIDSet are given at lines 11, 14 and 15.

***MiniMaple* source** for global declaration sequence (`global status;`) is given at lines at 2,5 and 9.

Abstract syntax tree notations for global declaration is given on lines 1, 3 and 4. For an identifier *status* the tree is an expression (`#expression#`) of type identifier-expression (`#idexp#`).

Every other text that appears in the output is a message (warning or information).

Figure 2 shows that the file has successfully passed the grammar-check by the parser ANTLR. This also shows the type annotations for the first assignment command of the program.

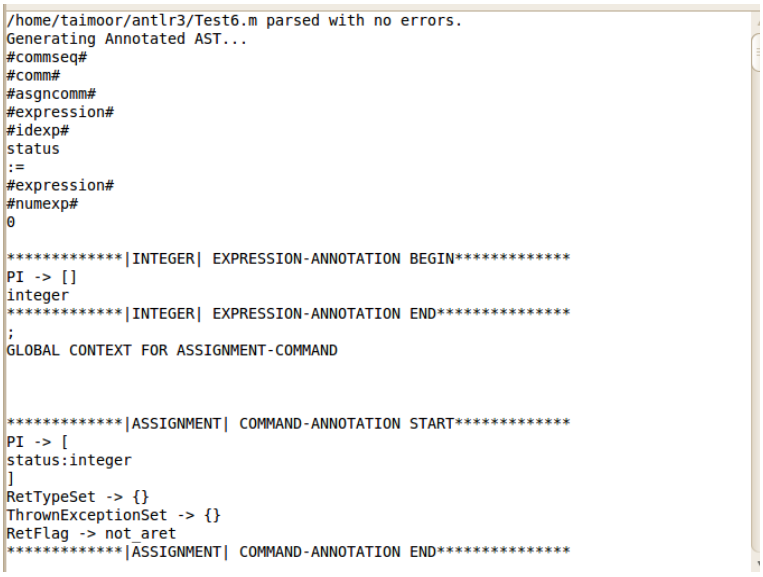

Figure 2: Parsing and generating annotated AST

Figure 3 shows the type annotations for global declarations. As global variables are untyped, so the variable *status* is assigned the more general type i.e. **anything** and is also put in the set of assignable identifiers. The set is populated In the *local* context only those variables are allowed to be assigned some value, which are declared.

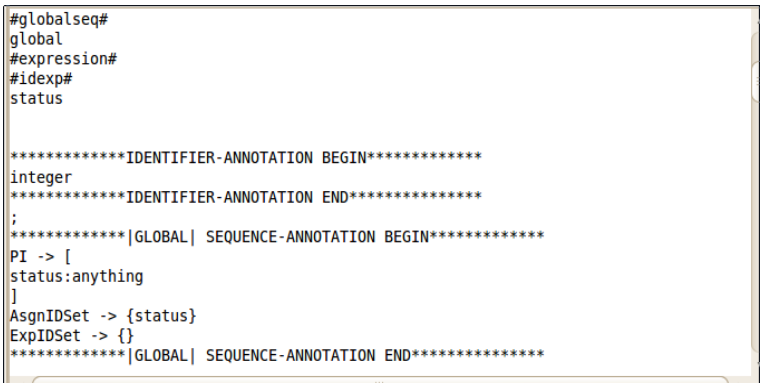

Figure 3: Global declarations type annotated

Figure 4 shows the type annotations for local declarations. The variable x is assigned the declared type while the variable i is assigned type **symbol**

and both the variables are put in the set of assignable identifiers.

```
*************|LOCAL| SEQUENCE-ANNOTATION BEGIN*************
PI -> [
i:symbol
x:Or(integer,float)
sf:float
si:integer
]
AsgnIDSet -> {si,x,sf,i}
ExpIDSet -> {}
*************|LOCAL| SEQUENCE-ANNOTATION END***************
#recc#
#commseq#
```

Figure 4: Local declarations type annotated

Figure 5 shows the type annotations for the *nested-if* branch of the if-conditional command.

```
*************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
status:integer
sf:float
i:integer
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************COMMAND-SEQUENCE-ANNOTATION END***************

end if;
*************|CONDITIONAL| COMMAND-ANNOTATION START*************
PI -> [
status:integer
i:integer
sf:float
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************|CONDITIONAL| COMMAND-ANNOTATION END***************
```

Figure 5: Conditional command (*if-if part*) type annotated

Figure 6 shows the type annotations for the *nested-if* branch of the elif-conditional command.

```
************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
x:float
status:integer
i:integer
sf:float
l:list(Or(integer,float))
si:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
************COMMAND-SEQUENCE-ANNOTATION END***************

end if;
************|CONDITIONAL| COMMAND-ANNOTATION START*************
PI -> [
status:integer
x:float
sf:float
i:integer
l:list(Or(integer,float))
si:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
************|CONDITIONAL| COMMAND-ANNOTATION END***************
```

Figure 6: Conditional command (*elif-if part*) type annotated

Figure 7 shows the type annotations for a for-loop. The global type information is not affected by the body of the loop; the global variables have the declared types and not the modified types while the executing of the body of the loop.

```
************|FOR-LOOP| COMMAND-ANNOTATION START*************
PI -> [
i:symbol
x:Or(integer,float)
sf:float
l:list(Or(integer,float))
si:integer
status:anything
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
************|FOR-LOOP| COMMAND-ANNOTATION END***************
```

Figure 7: For-loop type annotated

Figure 8 shows the type annotations for the body of the procedure. The body of the procedure always returns a tuple of integer and float, and the procedure doesn't throw any exception.

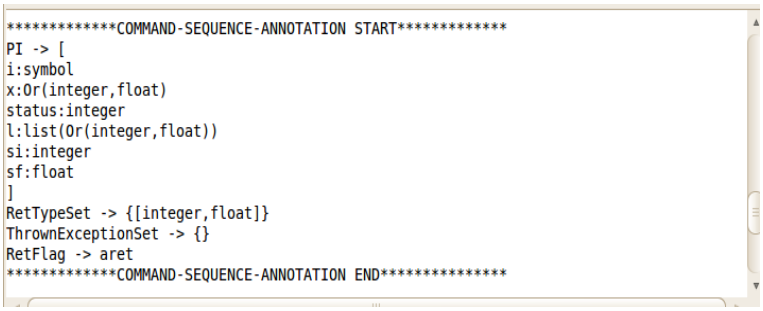

Figure 8: The body of procedure type annotated

Figure 9 shows the type annotations for the whole program, which is essentially a sequence of assignment commands. The type environment shows the associated identifiers and their respective types while type checking.

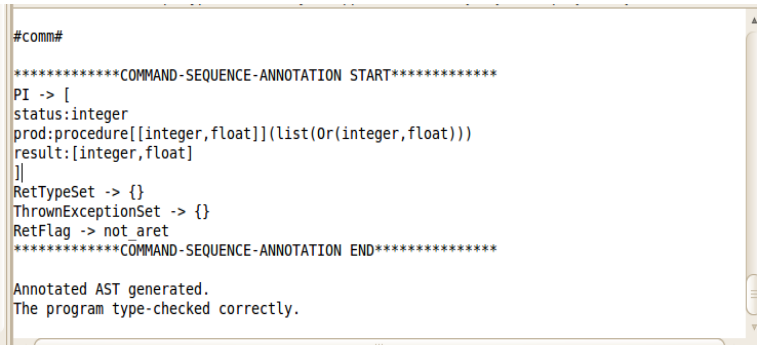

Figure 9: The program (as a sequence of commands) type annotated

In the next section we conclude and highlight our future work.

5 Conclusions and Future Work

In this paper we have defined a substantial subset of Maple called *MiniMaple*. *MiniMaple* has similar syntactic constructs as of Maple but with fewer expressions. We designed a formal type system for *MiniMaple* which consists of typing judgments (for syntactic domains of *MiniMaple*), logical rules to derive the judgments and auxiliary functions and predicates which are used in the rules. The type system makes use of Maple annotations

for static analysis (compile-time type checking). We have implemented the type system by a program for type-checking *MiniMaple* programs. We have validated our results with small examples; but still exhaustive testing of the type checker is required. A Maple package *DifferenceDifferential* developed at our institute will be a comprehensive test suite for *MiniMaple*.

As a next step, we are working on the design of a formal specification language for *MiniMaple*. A verifier will be developed which takes a formal specification of a *MiniMaple* program and the annotated syntax tree produced by the type checker to verify/falsify the correctness of the program with respect to the specification.

References

[1] ANTLR v3. http://www.antlr.org/.

[2] AXIOM. http://www.axiom-developer.org/.

[3] Java 1.5. http://www.oracle.com/technetwork/java/index.html.

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

[5] Maple Software. http://www.maplesoft.com/.

[6] Mathematica. http://www.wolfram.com/mathematica/.

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

[8] Jacques Carette and Stephen Forrest. Mining Maple Code for Contracts. In Silvio Ranise and Anna Bigatti, editors, *Calculemus*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[9] Jacques Carette and Michael Kucera. Partial Evaluation of Maple. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 41–50, New York, NY, USA, 2007.

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

[11] Muhammad Taimoor Khan. Software for *MiniMaple*. Doktoratskolleg (DK), Johannes Kepler University, Linz, Austria, http://www.risc.jku.at/people/mtkhan/dk10/.

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

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

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

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

Appendices

In the following subsections we show the complete definition of *Mini-Maple*, the typing rules for the syntactic domains of *MiniMaple* with auxiliary functions and predicates, and the output of a sample execution of the type checker.

A Syntactic Definition of *MiniMaple*

In this appendix we give the formal abstract syntax (language grammar) of *MiniMaple*.

Prog ∈ Program
Cseq ∈ Command Sequence
C ∈ Command
Elif ∈ ElseIf
Catch ∈ Catch
Eseq ∈ Expression Sequence
E ∈ Expression
S ∈ Sequence
R ∈ Recurrence
Pseq ∈ Parameter Sequence
P ∈ Parameter
M ∈ Modifiers
Iseq ∈ Identifier Sequence
I ∈ Identifier
Itseq ∈ Identifier Typed Sequence
It ∈ Identifier Typed
Bop ∈ Boolean Operators
Uop ∈ Unary Operators
Esop ∈ Especial Operators
Tseq ∈ Type Sequence
T ∈ Type
N ∈ Numeral

Prog ::= Cseq
Cseq ::= EMPTY | (C; | E;)Cseq
C ::= **if** E **then** Cseq Elif **end if**; | **if** E **then** Cseq Elif **else** Cseq **end if**;
| **while** E **do** Cseq **end do**;

| **for** I **in** E **do** Cseq **end do**;
| **for** I **in** E **while** E **do** Cseq **end do;**
| **for** I **from** E **by** E **to** E **do** Cseq **end do**;
| **for** I **from** E **by** E **to** E **while** E **do** Cseq **end do;**
| **return** E; | **return**; | **error;** | **error** I,Eseq;
| **try** Cseq Catch **end;** | **try** Cseq Catch **finally** Cseq **end;**
| I,Iseq := E,Eseq; | E(Eseq); | **‘type**/I‘ := T; | **print**(e);

Elif ::= EMPTY | **elif** E **then** Cseq;Elif
Catch ::= EMPTY | **catch “**I**” :**Cseq, Catch
Eseq ::= EMPTY | E,Eseq
E ::= I | N | **module()** S;R **end module**;
| **proc**(Pseq) S;R **end proc**;| **proc**(Pseq)::T; S;R **end proc**;
| E1 Bop E2 | Uop E | Esop | E1 **and** E2 | E1 **or** E2 | E(Eseq)
| I1:-I2 | E,E,Eseq | **type(** I,T **)** | E1 = E2 | E1 <> E2
S ::= EMPTY | **local** It,Itseq;S | **global** I,Iseq;S | **uses** I,Iseq;S
| **export** It,Itseq;S
R ::= Cseq | Cseq;E
Pseq ::= EMPTY | P,Pseq
P ::= I | I :: M
M ::= **seq(** T **)** | T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Itseq :: = EMPTY | It, Itseq
It ::= I | I :: T | I := E | I::T:=E
Bop ::= +| − | / | ∗ | **mod** | < | > | ≤ | ≥
Uop ::= **not** | − |+
Esop ::= **op(** E1, E2 **)** | **op(** E **)** | **op(** E..E, E **)** | **nops(** E **)**
| **subsop(** E1=E2, E3 **)** | **subs(** I=E1, E2 **)** | “ E “ | [Eseq]
| I[Eseq] | **seq**(E, I = E..E **)** | **seq**(E, I **in** E) | **eval(** I,1 **)**
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | **[** Tseq **]** | **procedure[** T **](** Tseq **)**
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **uneval** | I
N ::= a sequence of decimal digits

B Logical Rules

In the following subsections, we list the logical rules for each phrase/alternative of syntactic domain, which are used to derive typing judgments. The rules state the conditions under which the syntactic phrases are well typed.

B.1 Program

In this subsection we show the typing rules for all the syntactic phrases of the top level domain Program.

B.1.1 Cseq

A well typed command sequence *Cseq* is a well typed program.

$$\frac{\{\},\text{global},\{\} \vdash \mathit{Cseq}:(\pi,\tau set,\epsilon set,rflag)\mathbf{cseq}}{\vdash \mathit{Cseq}:\mathbf{prog}}$$

B.2 Command Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Command Sequence.

B.2.1 *EMPTY*

An *EMPTY* command sequence produces the same type environment as type annotation.

$$\pi,\ c,\ asgnset \vdash \mathit{EMPTY}:(\pi,\{\},\{\},\text{not\_aret})\mathbf{cseq}$$

B.2.2 *C;Cseq*

The phrase "*C;Cseq*" is a well typed command sequence with the union of the respective type annotations of C and *Cseq* and the extended type environment π_2 by the analysis of *Cseq*.

$$\frac{\begin{array}{c}\pi,\ c,\ asgnset \vdash C:(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm} \\ \pi_1,\ c,\ asgnset_1 \vdash \mathit{Cseq}:(\pi_2,\tau set_2,\epsilon set_2,rflag_2)\mathbf{cseq}\end{array}}{\begin{array}{c}\pi,\ c,\ asgnset \vdash C;\mathit{Cseq}:(\pi_2,\tau set_1 \cup \tau set_2,\epsilon set_1 \cup \epsilon set_2,\text{retCseq}(rflag_1,\\ rflag_2))\mathbf{cseq}\end{array}}$$

B.3 Command

In this subsection we show the typing rules for all the syntactic phrases of the domain Command.

B.3.1 if *E* then *Cseq Elif* end if

The phrase "**if** *E* **then** *Cseq Elif* **end if**" is a well typed conditional command that combines the type environment of its two conditional branches (*if* and *elif*) and union the other respective type annotations. With the static analysis as we are not sure which of the two branches will be executed runtime, so we need to combine the type information of the branches.

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

B.3.2 if *E* then *Cseq*$_1$ *Elif* else *Cseq*$_2$ end if

The phrase "**if** *E* **then** *Cseq*$_1$ *Elif* **else** *Cseq*$_2$ **end if**" is also a well typed conditional command that combines the type environment of its two conditional branches (*if* and *elif*) and an *else* branch. It also joins the other corresponding type annotations.

$$\frac{\begin{array}{c}\pi \vdash E\text{: }(\pi')\textbf{boolexp} \quad \text{canSpecialize}(\pi,\pi') \\ \text{specialize}(\pi,\pi'),\ c,\ asgnset \vdash Cseq_1\text{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\textbf{cseq} \\ \pi,\ c,\ asgnset \vdash Elif\text{:}(\pi_2,\pi set,\ \tau set_2,\epsilon set_2,rflag_2)\textbf{elif} \\ \pi,\ c,\ asgnset \vdash Cseq_2\text{:}(\pi_3,\tau set_3,\epsilon set_3,rflag_3)\textbf{cseq}\end{array}}{\begin{array}{c}\pi,\ c,\ asgnset \vdash \textbf{if}\ E\ \textbf{then}\ Cseq_1\ Elif\ \textbf{else}\ Cseq_2\ \textbf{end} \\ \textbf{if}\text{:}(\text{combine}(\text{combine}(\pi_1,\pi_2),\pi_3),\tau set_1 \cup \tau set_2 \cup \tau set_3,\epsilon set_1 \cup \epsilon set_2 \cup \\ \epsilon set_3,\text{ret}(\text{ret}(rflag_1,\ rflag_2),rflag_3))\textbf{comm}\end{array}}$$

B.3.3 while *E* do *Cseq* end do

The phrase "**while** *E* **do** *Cseq* **end do**" is a well typed loop command that leaves the input type environment π unchanged. In the *local* context we specialize the types, i.e. the new value of an identifier must be a subtype of its declared type. The type environment is not modified by the body of the loop because the number of loop iterations might otherwise influence the type information.

$$\frac{\pi \vdash E\text{: } (\pi')\textbf{boolexp} \quad \text{canSpecialize}(\pi,\pi') \quad \underline{\text{specialize}(\pi,\pi'),\ local,\ asgnset \vdash Cseq\text{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\textbf{cseq}}}{\pi,\ c,\ asgnset \vdash \textbf{while } E \textbf{ do } Cseq \textbf{ end do}\text{:}(\pi,\tau set_1,\epsilon set_1,rflag_1)\textbf{comm}}$$

B.3.4 for *I* in *E* do *Cseq* end do

The phrase "**for** I **in** E **do** *Cseq* **end do**" is a well typed loop command that leaves the input type environment π unchanged.

$$\frac{\pi \vdash E\text{: } (\tau)\textbf{iterator} \quad \underline{\text{override}(\pi,\{I\text{: }\tau\}),\ local,\ asgnset \vdash Cseq\text{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\textbf{cseq}}}{\pi,\ c,\ asgnset \vdash \textbf{for } I \textbf{ in } E \textbf{ do } Cseq \textbf{ end do}\text{:}(\pi,\tau set_1,\epsilon set_1,rflag_1)\textbf{comm}}$$

B.3.5 for *I* in E_1 while E_2 do *Cseq* end do

The phrase "**for** I **in** E_1 **while** E_2 **do** *Cseq* **end do**" is a well typed loop command that leaves the input type environment π unchanged.

$$\frac{\begin{array}{c}\pi \vdash E_1\text{: } (\tau)\textbf{iterator} \\ \text{override}(\pi,\{I\text{: }\tau\}) \vdash E_2\text{: } (\pi')\textbf{boolexp} \\ \text{canSpecialize}(\text{override}(\pi,\{I\text{: }\tau\}),\pi') \\ \text{specialize}(\text{override}(\pi,\{I\text{: }\tau\}),\pi'),local,asgnset \\ \underline{\vdash Cseq\text{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\textbf{cseq}}\end{array}}{\begin{array}{c}\pi,\ c,\ asgnset \vdash \textbf{for } I \textbf{ in } E_1 \textbf{ while } E_2 \textbf{ do } Cseq \textbf{ end} \\ \textbf{do}\text{:}(\pi,\tau set_1,\epsilon set_1,rflag_1)\textbf{comm}\end{array}}$$

B.3.6 for *I* from E_1 by E_2 to E_3 do *Cseq* end do

The phrase "**for** I **from** E_1 **by** E_2 **to** E_3 **do** *Cseq* **end do**" is a well typed loop command that leaves the input type environment π unchanged.

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

B.3.7 for *I* from E_1 by E_2 to E_3 while E_4 do *Cseq* end do

The phrase "**for** I **from** E_1 **by** E_2 **to** E_3 **while** E_4 **do** *Cseq* **end do**" is a well typed loop command that leaves the input type environment π unchanged.

$$\frac{\begin{array}{c}\pi \vdash E_1\text{: (integer)}\mathbf{exp} \\ \pi \vdash E_2\text{: (integer)}\mathbf{exp} \\ \pi \vdash E_3\text{: (integer)}\mathbf{exp} \\ \pi \vdash E_4\text{: }(\pi')\mathbf{boolexp} \\ \text{canSpecialize(override}(\pi,\{I{:}integer\}),\pi') \\ \text{specialize(override}(\pi,\{I{:}integer\}),\pi'),\ local,\ asgnset \\ \vdash Cseq{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{cseq}\end{array}}{\pi,\ c,\ asgnset \vdash \mathbf{for}\ I\ \mathbf{from}\ E_1\ \mathbf{by}\ E_2\ \mathbf{to}\ E_3\ \mathbf{while}\ E_4\ \mathbf{do}\ Cseq\ \mathbf{end}\ \mathbf{do}{:}(\pi,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm}}$$

B.3.8 return *E*

A well typed return command always returns (aret).

$$\frac{\pi \vdash E\text{: }(\tau)\mathbf{exp}}{\pi,\ local,\ asgnset \vdash \mathbf{return}\ E{:}(\pi,\{\tau\},\{\},aret)\mathbf{comm}}$$

B.3.9 return

The phrase "**return**" is a well typed void-return command.

$$\pi,\ local,\ asgnset \vdash \mathbf{return}{:}(\pi,\{\text{void}\},\{\},aret)\mathbf{comm}$$

B.3.10 error

The phrase "**error**" is a well typed command.

$$\pi,\ c,\ asgnset \vdash \mathbf{error}{:}(\pi,\{\text{annonymous}\},\{\},not\_\ aret)\mathbf{comm}$$

B.3.11 error *I,Eseq*

The phrase "**error** *I,Eseq*" is a well typed error command that raises an exception I with its associated type (τseq).

$$\frac{\pi \vdash Eseq\text{: }(\tau seq)\mathbf{expseq}}{\pi,\ c,\ asgnset \vdash \mathbf{error}\ I,Eseq{:}(\pi,\{\},\{\text{I:}\tau seq\},not\_\ aret)\mathbf{comm}}$$

B.3.12 try *Cseq Catch* end

The phrase "**try** *Cseq Catch* **end**" is a well typed try-catch command whose set of thrown exceptions is equal to the difference of exceptions thrown in try block to the exceptions caught in the catch block. The block leaves an input type environment π unchanged because at runtime any of the branch of this construct can execute.

$$\frac{\pi,\ local,\ asgnset \vdash Cseq{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm} \quad \pi \vdash Catch{:}(\tau set_2,\epsilon set_2,\epsilon cset,rflag_2)\mathbf{catch}}{\pi,\ c,\ asgnset \vdash \mathbf{try}\ Cseq\ Catch\ \mathbf{end}{:}(\pi,\tau set_1 \cup \tau set_2,(\epsilon set_1 \backslash \epsilon cset) \cup \epsilon set_2,\text{retCatch}(rflag_1,rflag_2))\mathbf{comm}}$$

B.3.13 try *Cseq*$_1$ *Catch* finally *Cseq*$_2$ end

The phrase "**try** $Cseq_1$ *Catch* **finally** $Cseq_2$ **end**" is a well typed try-catch-finally command whose type annotations are mainly determined by the annotations of the finally block; because the finally block is often executed in the try-catch-finally construct.

$$\frac{\begin{array}{c}\pi,\ local,\ asgnset \vdash Cseq_1{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm} \\ \pi \vdash Catch{:}(\tau set_2,\epsilon set_2,\epsilon cset,rflag_2)\mathbf{catch} \\ \pi,\ c,\ asgnset \vdash Cseq_2{:}(\pi_3,\tau set_3,\epsilon set_3,rflag_3)\mathbf{comm}\end{array}}{\begin{array}{c}\pi,\ c,\ asgnset \vdash \mathbf{try}\ Cseq_1\ Catch\ \mathbf{finally}\ Cseq_2 \\ \mathbf{end}{:}(\pi,\tau set_1 \cup \tau set_2 \cup \tau set_3,(\epsilon set_1 \backslash \epsilon cset) \cup \epsilon set_2 \cup \\ \epsilon set_3,\text{retCseq}(\text{retCatch}(rflag_1,rflag_2),rflag_3))\mathbf{comm}\end{array}}$$

B.3.14 *I,Iseq* := *E,Eseq*

The phrase "*I,Iseq* := *E,Eseq*" is a well typed assignment command which updates the types of the identifiers only if the types of expressions (E and $Eseq$) are the subtypes of the declared identifiers types (I and $Iseq$).

for local context

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

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

for global context

$$\begin{array}{c}\text{isNotRepeated(I,Iseq)} \\ \pi \vdash E{:}(\tau)\mathbf{exp}\end{array}$$

$$\frac{\pi \vdash \mathit{Eseq}{:}(\tau seq)\mathbf{expseq}}{\pi,\ \mathit{global},\ \mathit{asgnset} \vdash \mathit{I,Iseq}\ :=\ \mathit{E,Eseq}{:}(\text{update}(\pi,(\text{I,Iseq}),(\tau,\tau seq)),\{\},\{\},\mathit{not\_}\ \mathit{aret})\mathbf{comm}}$$

B.3.15 *E(Eseq)*

The phrase "*E(Eseq)*" is a well typed procedure-call command, where E is a well typed procedure defined over general parameters (types) than appeared parameters (types) in *Eseq*.

$$\frac{\pi \vdash E{:}(\text{procedure}[\tau](\tau seq_1))\mathbf{exp} \quad \pi \vdash \mathit{Eseq}{:}(\tau seq_2)\mathbf{expseq} \quad \text{superTypeSeq}(\tau seq_1,\tau seq_2)}{\pi,\ c,\ \mathit{asgnset} \vdash \mathit{E(Eseq)}{:}(\pi,\tau,\{\},\mathit{not\_}\ \mathit{aret})\mathbf{comm}}$$

B.3.16 *`type/I`:=*T

An identifier I represents the new user-defined type T; it is tagged with *type* in type environment to distinguish between an identifier name and identifier type.

$$\frac{\text{I is a valid identifier} \quad \pi \vdash T{:}(\tau)\mathbf{type}}{\pi,\ c,\ \mathit{asgnset} \vdash \grave{}\mathit{type/I}\acute{}{:=}\text{T}{:}(\{\text{I}{:}\mathit{type}(\text{T})\},\{\},\{\},\mathit{not\_}\ \mathit{aret})\mathbf{comm}}$$

B.4 Iterator

In this subsection we show the typing rules for the iterator, which determines the type of conditional expression in for-loop.

B.4.1 *E is a list(τ) expression*

A well typed expression list(τ) is a well typed τ iterator.

$$\frac{\pi \vdash E{:}(\text{list}(\tau))\mathbf{exp}}{\pi \vdash E{:}(\tau)\mathbf{iterator}}$$

B.4.2 *E is a string expression*

A well typed expression string is a well typed string iterator.

$$\frac{\pi \vdash E{:}(\text{string})\mathbf{exp}}{\pi \vdash E{:}(\text{string})\mathbf{iterator}}$$

B.4.3 *E is a {τ} expression*

A well typed expression set(τ) is a well typed τ iterator.

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

B.4.4 *E is a [τseq] expression*

A well typed expression tuple is a well typed union-type iterator for the (types of the) components of the tuple.

$$\frac{\pi \vdash E{:}([\tau seq])\mathbf{exp}}{\pi \vdash E{:}(\mathrm{Or}(\tau seq))\mathbf{iterator}}$$

B.5 Elif

In this subsection we show the typing rules for all the syntactic phrases of the domain Elif.

B.5.1 *EMPTY*

An *EMPTY* elif-construct leaves the input type environment π unchanged.

$$\pi, c, asgnset \vdash EMPTY{:}(\pi,\{\},\{\},not\_aret)\mathbf{elif}$$

B.5.2 elif *E* then *Cseq;Elif*

The phrase "**elif** *E* **then** *Cseq;Elif*" is a well typed elif-construct that combines the type environment for its own block and the following *Elif* block. The combined environment captures the possible runtime type information of the construct.

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

B.6 Catch

In this subsection we show the typing rules for all the syntactic phrases of the domain Catch.

B.6.1 *EMPTY*

An *EMPTY* catch-construct does not always returns.

$$\pi, asgnset \vdash EMPTY{:}(\{\},\{\},\{\},not\_\ aret)\mathbf{catch}$$

B.6.2 catch *I:Cseq,Catch*

The phrase "**catch** *I:Cseq,Catch*" is a well typed catch-construct that joins the sets of exceptions thrown by its own block and the following *Catch* block.

$$\frac{\pi, c, asgnset \vdash Cseq{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\mathbf{comm} \quad \pi, asgnset \vdash Catch{:}(\tau set_2,\epsilon set_2,\epsilon cset,rflag_2)\mathbf{catch}}{\pi,\ asgnset \vdash \mathbf{catch}}$$

$$I{:}Cseq,Catch{:}(\tau set_1 \cup \tau set_2,\epsilon set_1 \cup \epsilon set_2,\epsilon cset,\text{retCatch}(rflag_1,rflag_2))\mathbf{catch}$$

B.7 Expression Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Expression Sequence.

B.7.1 *EMPTY*

An *EMPTY* expression sequence gives an empty type annotation, i.e. no or null type information.

$$\pi \vdash EMPTY{:}(\text{empty})\mathbf{expseq}$$

B.7.2 *E,Eseq*

The phrase "*E,Eseq*" is a well typed expression sequence with a syntactic sequence of the types by E and *Eseq*.

$$\frac{\pi \vdash E{:}(\tau)\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq)\mathbf{expseq}}{\pi \vdash E,Eseq{:}((\tau,\tau seq))\mathbf{expseq}}$$

B.8 Expression

In this subsection we show the typing rules for all the syntactic phrases of the domain Expression. As the domain Expression has two kinds of typing judgments, the two conversion rules are given at the end.

B.8.1 *I*

An identifier I can not be declared of type uneval.

$$\pi \vdash I{:}(\tau)\mathbf{exp}, \text{ if } (\mathrm{I}{:}\tau) \in \pi \wedge \tau \neq \text{“}\mathit{uneval}\text{”}$$

B.8.2 *N*

Any valid integer expression is a well typed integer.

$$\pi \vdash N{:}(\text{integer})\mathbf{exp}$$

B.8.3 module() *S;R* end module

The phrase “**module()** *S;R* **end module**“ is a well typed expression of type *module* with a set of exported identifiers and their respective defined types (type environment).

$$\frac{\begin{array}{c}\pi, \mathit{local}, \{\}, \{\} \vdash S{:}(\pi_1, \mathit{asgnset}, \mathit{expidset})\mathbf{seq} \\ \text{override}(\pi,\pi'), \mathit{local}, \mathit{asgnset} \vdash R{:}(\pi_2, \{\}, \epsilon\mathit{set}, \mathit{rflag})\mathbf{recc} \\ \text{checkExpIds}(\text{expidset}, \pi_2)\end{array}}{\pi \vdash \mathbf{module()}\ S;R\ \mathbf{end\ module}{:}(\text{module}(\text{restrict}(\pi_2, \text{expidset})))\mathbf{exp}}$$

B.8.4 proc(*Pseq) S;R* end proc

The phrase ”**proc(***Pseq) S;R* **end proc**” is a well typed expression of type *procedure* with a return type void and a sequence of types of the parameters.

$$\frac{\begin{array}{c}\pi \vdash \mathit{Pseq}{:}(\pi_1)\mathbf{paramseq} \\ \pi, \mathit{local}, \{\}, \{\} \vdash S{:}(\pi_2, \mathit{asgnset}, \mathit{expidset})\mathbf{seq} \\ \pi_2, \mathit{local}, \mathit{asgnset} \vdash R{:}(\pi_3, \tau\mathit{set}, \epsilon\mathit{set}, \mathit{aret})\mathbf{recc} \\ \text{checkTypes}(\tau\mathit{set}, \text{void})\end{array}}{\pi \vdash \mathbf{proc(}\mathit{Pseq)}\ S;R\ \mathbf{end\ proc}{:}(\text{procedure}[\text{void}](\text{getParamTypes}(\pi_1)))\mathbf{exp}}$$

B.8.5 proc(*Pseq)::T; S;R* end proc

The phrase “**proc(***Pseq)::T; S;R* **end proc**“ is a well typed expression of type *procedure* with a return type τ and a sequence of types of the parameters.

$$\frac{\begin{array}{c}\pi \vdash \mathit{Pseq}{:}(\pi_1)\mathbf{paramseq} \\ \pi \vdash T{:}(\tau)\mathbf{type} \\ \pi, \mathit{local}, \{\}, \{\} \vdash S{:}(\pi_2, \mathit{asgnset}, \mathit{expidset})\mathbf{seq} \\ \pi_2, \mathit{local}, \mathit{asgnset} \vdash R{:}(\pi_3, \tau\mathit{set}, \epsilon\mathit{set}, \mathit{aret})\mathbf{recc} \\ \text{checkTypes}(\tau\mathit{set}, \text{void})\end{array}}{\pi \vdash \mathbf{proc(}\mathit{Pseq)::T;}\ S;R\ \mathbf{end\ proc}{:}(\text{procedure}[\tau](\text{getParamTypes}(\pi_1)))\mathbf{exp}}$$

B.8.6 E_1 *Bop* E_2

The phrase "E_1 *Bop* E_2" is a well typed expression of type τ_3, where τ_3 is the computed result type of a well defined binary operator *Bop* over two super-types (τ_1' and τ_2') of the two expression types (τ_1 and τ_2).

$$\frac{\begin{array}{c}\pi \vdash E_1{:}(\tau_1)\mathbf{exp} \\ \pi \vdash E_2{:}(\tau_2)\mathbf{exp} \\ \pi \vdash Bop{:}(\mathrm{op}(\tau_1',\tau_2',\mathrm{Bop}){:}\tau_3)\mathbf{exp} \\ \mathrm{superType}(\tau_1',\tau_1) \\ \mathrm{superType}(\tau_2',\tau_2)\end{array}}{\pi \vdash E_1\ Bop\ E_2{:}(\tau_3)\mathbf{exp}}$$

B.8.7 *Uop E*

The phrase "*Uop E*" is a well typed expression of type τ_2, where τ_2 is the computed result type of a well defined unary operator *Uop* over a super-type (τ_1) of the expression type (τ).

$$\frac{\begin{array}{c}\pi \vdash E{:}(\tau)\mathbf{exp} \\ \pi \vdash Uop{:}(\mathrm{op}(\tau_1,\mathrm{Uop}){:}\tau_2)\mathbf{exp} \\ \mathrm{superType}(\tau_1,\tau)\end{array}}{\pi \vdash Uop\ E{:}(\tau_2)\mathbf{exp}}$$

B.8.8 *Esop*

A well typed special expression *Esop* is a well typed expression.

$$\frac{\pi \vdash Esop{:}(\tau)\mathbf{esop}}{\pi \vdash Esop{:}(\tau)\mathbf{exp}}$$

B.8.9 E_1 **and** E_2

The phrase "E_1 **and** E_2" is a well typed boolean expression with the set of identifiers (of E_1 and E_2) and their respective logical-and types.

$$\frac{\begin{array}{c}\pi \vdash E_1{:}(\pi_1)\mathbf{boolexp} \\ \mathrm{canSpecialize}(\pi,\pi_1) \\ \pi \vdash E_2{:}(\pi_2)\mathbf{boolexp} \\ \mathrm{andCombinable}(\pi_1,\pi_2)\end{array}}{\pi \vdash E_1\ \mathbf{and}\ E_2{:}(\mathrm{andCombine}(\pi_1,\pi_2))\mathbf{boolexp}}$$

B.8.10 E_1 or E_2

The phrase "E_1 **or** E_2" is a well typed boolean expression with the set of identifiers (of E_1 and E_2) and their respective union types.

$$\frac{\pi \vdash E_1{:}(\pi_1)\mathbf{boolexp} \quad \text{orCombine}(\pi,\pi_1) \vdash E_2{:}(\pi_2)\mathbf{boolexp} \quad \text{canSpecialize}(\pi_1,\pi_2)}{\pi \vdash E_1 \textbf{ and } E_2{:}(\text{orCombine}(\pi_1,\pi_2))\mathbf{boolexp}}$$

B.8.11 *E(Eseq)*

The phrase "*E(Eseq)*" is a well typed expression of type τ that is the return type of the procedure expression E.

$$\frac{\pi \vdash E{:}(\text{procedure}[\tau](\tau seq))\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq_1)\mathbf{exp} \quad \text{superTypeSeq}(\tau seq,\tau seq_1)}{\pi \vdash E(Eseq){:}(\tau)\mathbf{exp}}$$

B.8.12 *I(Eseq)*

The phrase "*I(Eseq)*" is a well typed expression of named type I with types of parameters τseq.

$$\frac{\pi \vdash I{:}(\text{symbol})\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq)\mathbf{exp}}{\pi \vdash I(Eseq){:}(\text{I}(\tau seq))\mathbf{exp}}$$

B.8.13 I_1:-I_2

The phrase "I_1:-I_2" is a well typed expression of type τ, i.e. the type of an exported identifier I_2 of a module I_1.

$$\frac{\pi \vdash I_1{:}(\text{module}(\pi_1))\mathbf{exp} \quad \pi_1 \vdash I_2{:}(\tau)\mathbf{id}}{\pi \vdash I_1\text{:-}I_2{:}(\tau)\mathbf{exp}}$$

B.8.14 type(I,T)

The phrase "**type**(I,T)" is a boolean expression with a set of a pair of an identifier and its type T; and the declared type (τ_1) is the super-type of T (τ_2).

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

B.8.15 not type(*I,T*)

The phrase "**not type**(I,T)" is a boolean expression with a set of a pair of an identifier and its type T; and the declared type (τ_1) is the super-type of T (τ_2).

$$\frac{\pi \vdash I{:}(\tau_1)\mathbf{id} \quad \pi \vdash T{:}(\tau_2)\mathbf{type} \quad \text{canNegateType}(\tau_2,\tau_1)}{\pi \vdash \mathbf{not\ type}(I,T){:}(\{\text{I}{:}\text{negateType}(\tau_2,\tau_1)\})\mathbf{boolexp}}$$

B.8.16 E_1 <> E_2

The phrase "E_1 <> E_2" is a boolean expression where the inequality of two types of expressions (E_1 and E_2) is well defined.

$$\frac{\pi \vdash E_1{:}(\tau_1)\mathbf{exp} \quad \pi \vdash E_2{:}(\tau_2)\mathbf{exp} \quad \text{equalTypes}(\tau_1,\tau_2)}{\pi \vdash E_1 <> E_2{:}(\text{boolean})\mathbf{exp}}$$

B.8.17 $E_1 = E_2$

The phrase "$E_1 = E_2$" is a boolean expression where the equality of two types of expressions (E_1 and E_2) is well defined.

$$\frac{\pi \vdash E_1{:}(\tau_1)\mathbf{exp} \quad \pi \vdash E_2{:}(\tau_2)\mathbf{exp} \quad \text{equalTypes}(\tau_1,\tau_2)}{\pi \vdash E_1 = E_2{:}(\text{boolean})\mathbf{exp}}$$

B.8.18 Conversion Rule I

If E is a well typed expression of type boolean, then E is a well typed boolexp with an empty type environment.

$$\frac{\pi \vdash E{:}(\text{boolean})\mathbf{exp}}{\pi \vdash E{:}(\{\})\mathbf{boolexp}}$$

B.8.19 Conversion Rule II

If E is a well typed expression of type boolexp and with new type environment π_1, then E is just a well typed expression of type boolean.

$$\frac{\pi \vdash E{:}(\pi_1)\mathbf{boolexp}}{\pi \vdash E{:}(\text{boolean})\mathbf{exp}}$$

B.9 Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain (declaration) Sequence.

B.9.1 *EMPTY*

An *EMPTY* declaration sequence results in the same type annotation as was given as an input.

$$\pi,\ asgnset, expidset \vdash \mathbf{EMPTY}{:}(\pi, asgnset, expidset)\mathbf{seq}$$

B.9.2 local *Idt,Idtseq;S*

The phrase "**local** *Idt,Idtseq;S*" is a well typed declaration sequence with overridden types of the local identifier.

$$\frac{\begin{array}{c}\pi, asgnset \vdash Idt{:}(\tau, asgnset_1)\mathbf{idt} \\ \pi, asgnset_1 \vdash Idtseq{:}(\tau seq, asgnset_2)\mathbf{idtseq} \\ \text{isNotRepeatedTypedId}(Idt,\ Idtseq) \\ \text{overrideLocal}(\pi,(Idt,Idtseq),(\tau,\tau seq)),\ asgnset_2, expidset \\ \vdash S{:}(\pi_1, asgnset_3, expidset_1)\mathbf{seq}\end{array}}{\pi,\ asgnset, expidset \vdash \mathbf{local}\ Idt,Idtseq;S{:}(\pi_1, asgnset_3, expidset_1)\mathbf{seq}}$$

B.9.3 global *I,Iseq;S*

The phrase "**global** *I,Iseq;S*" is a well typed declaration sequence with the assigned most general type (anything) to the identifiers. No type information is allowed in the global declarations of identifiers.

$$\frac{\begin{array}{c}\pi \vdash I{:}(\tau)\mathbf{id} \\ \pi \vdash Iseq{:}(\tau seq)\mathbf{idseq} \\ \text{isNotRepeatedId}(I,Iseq) \\ \pi \cup \{I : anything\} \cup seqToSetTyped(Iseq), \\ \{I\} \cup seqToSet(Iseq) \cup asgnset, expidset\end{array}}{}$$

$$\frac{\vdash S{:}(\pi_1, asgnset_1, expidset_1)\mathbf{seq}}{\pi,\, asgnset, expidset \vdash \mathbf{global}\ I,Iseq;S{:}(\pi_1, asgnset_1, expidset_1)\mathbf{seq}}$$

B.9.4 uses *I,Iseq;S*

The phrase "**uses** *I,Iseq;S*" is a well typed declaration sequence (like import-statement in Java) of the identifiers of type module. The type information (of exported identifier) of each module is combined with the global type information.

$$\frac{\begin{array}{c}\pi \vdash I{:}(\text{module}(\pi_1))\mathbf{id} \\ \pi \vdash Iseq{:}(\tau seq)\mathbf{idseq} \\ \text{isModuleType}(\tau seq) \\ \text{isNotRepeatedId}(I,Iseq) \\ \pi seq{=}\text{getModuleEnvsSeq}(\tau seq) \\ \text{combine}(\pi,\, \text{override}(\pi_1, \pi seq)),\, asgnset, \{I\} \cup seqToSet(Iseq) \cup expidset \\ \vdash S{:}(\pi_1, asgnset_1, expidset_1)\mathbf{seq}\end{array}}{\pi,\, asgnset, expidset \vdash \mathbf{uses}\ I,Iseq;S{:}(\pi_1, asgnset_1, expidset_1)\mathbf{seq}}$$

B.9.5 export *Idt,Idtseq;S*

The phrase "**export** *Idt,Idtseq;S*" is a well typed declaration sequence of the identifiers with their declared types.

$$\frac{\begin{array}{c}\pi, asgnset \vdash Idt{:}(\tau, asgnset_1)\mathbf{idt} \\ \pi, asgnset_1 \vdash Idtseq{:}(\tau seq, asgnset_2)\mathbf{idtseq} \\ \text{isNotRepeatedTypedId}(Idt,\ Idtseq) \\ \text{overrideLocal}(\pi,(Idt,Idtseq),(\tau,\tau seq)),\, asgnset_2, ((idt, idtseq), expidset) \\ \vdash S{:}(\pi_1, asgnset_3, expidset_1)\mathbf{seq}\end{array}}{\begin{array}{l}\pi,\, asgnset, expidset \vdash \mathbf{export}\ Idt,Idtseq;S{:}(\text{override}(\pi, \pi_1), \\ asgnset_3, expidset_1)\mathbf{seq}\end{array}}$$

B.10 Recurrence

In this subsection we show the typing rules for all the syntactic phrases of the domain Recurrence, which represents the body of module/procedure.

B.10.1 *Cseq*

A well typed command sequence *Cseq* is a well typed recurrence.

$$\frac{\pi, local, asgnset \vdash Cseq{:}(\pi_1, \tau set, \epsilon set, rflag)\mathbf{cseq}}{\pi, local, asgnset \vdash Cseq{:}(\pi_1, \tau set, \epsilon set, not\_aret)\mathbf{recc}}$$

B.10.2 *Cseq;E*

A well typed command sequence *Cseq* followed by an expression is also a well typed recurrence. The type of an expression E will help in inferring the return type of a procedure.

$$\frac{\pi, local, asgnset \vdash Cseq{:}(\pi_1, \tau set, \epsilon set, rflag)\mathbf{cseq} \quad \pi \vdash E{:}(\tau)\mathbf{exp}}{\pi, local, asgnset \vdash Cseq{:}(\pi_1, \{\tau\}, \epsilon set, aret)\mathbf{recc}}$$

B.11 Parameter Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Parameter Sequence.

B.11.1 *EMPTY*

An *EMPTY* parameter sequence gives an empty type annotation.

$$\pi \vdash EMPTY{:}(\text{empty})\mathbf{paramseq}$$

B.11.2 *P,Pseq*

The phrase "*P,Pseq*" is a well typed parameter sequence with a union of the two type environments. No parameter is repeated.

$$\frac{\pi \vdash P{:}(\pi_1)\mathbf{param} \quad \pi_1 \vdash Pseq{:}(\pi_2)\mathbf{paramseq} \quad \text{isNotRepeatedParam}((P,Pseq))}{\pi \vdash P,Pseq{:}(\pi_1 \cup \pi_2)\mathbf{paramseq}}$$

B.12 Parameter

In this subsection we show the typing rules for all the syntactic phrases of the domain Parameter.

B.12.1 *I*

The phrase "*I*" is a well typed parameter with type anything.

$$\pi \vdash I{:}(\{\text{I:anything}\})\mathbf{param}$$

B.12.2 *I::M*

The phrase "*I::M*" is a well typed parameter with type of its modifier.

$$\frac{\pi \vdash I{:}(\pi_1)\mathbf{param} \quad \pi_1 \vdash M{:}(\tau)\mathbf{mod}}{\pi \vdash I{::}M{:}(\{\mathrm{I}{:}\tau\})\mathbf{param}}$$

B.13 Modifier

In this subsection we show the typing rules for all the syntactic phrases of the domain (type) Modifier for parameters.

B.13.1 seq(*T*)

The phrase "**seq**(T)" is a well typed modifier with type seq(τ).

$$\frac{\pi \vdash T{:}(\tau)\mathbf{type}}{\pi \vdash \mathbf{seq}(T){:}(\mathbf{seq}(\tau))\mathbf{mod}}$$

B.13.2 *T*

A well typed modifier T is a well typed modifier.

$$\frac{\pi \vdash T{:}(\tau)\mathbf{type}}{\pi \vdash T{:}(\tau)\mathbf{mod}}$$

B.14 Identifier Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Identifier Sequence.

B.14.1 *EMPTY*

An *EMPTY* identifier sequence gives no type information.

$$\pi \vdash EMPTY{:}(\text{empty})\mathbf{idseq}$$

B.14.2 *I,Iseq*

The phrase "*I,Iseq*" is a well typed identifier sequence with a syntactic sequence of I and *Iseq*, i.e. (*I,Iseq*). Any identifier must not be repeated in an identifier sequence.

$$\pi \vdash I{:}(\tau)\mathbf{id}$$
$$\pi \vdash Iseq{:}(\tau seq)\mathbf{idseq}$$
$$\frac{\text{isNotRepeatedId}(I,Iseq)}{\pi \vdash I,Iseq{:}((\tau,\tau seq))\mathbf{idseq}}$$

B.15 Identifier

In this subsection we show the typing rules for the syntactic domain Identifier.

B.15.1 *I*

An identifier is not allowed to be declared with type "uneval".

$$\pi \vdash I{:}(\tau)\mathbf{id}, \text{ where } (\text{I}{:}\tau) \in \pi \wedge \tau \neq \{uneval\}$$

B.16 Identifier Typed Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Identifier Typed Sequence.

B.16.1 *EMPTY*

An *EMPTY* identifier sequence gives empty type annotation.

$$\pi, asgnset \vdash EMPTY{:}(\text{empty},\ asgnset)\mathbf{idt}$$

B.16.2 *Idt,Idtseq*

The phrase "*Idt,Idtseq*" is a well typed identifier typed sequence with a syntactic sequence of *Idt* and *Idtseq*, i.e. (*Idt,Idtseq*). Any identifier typed must not be repeated in an identifier typed sequence.

$$\pi, asgnset \vdash Idt{:}(\tau, asgnset_1)\mathbf{idt}$$
$$\pi, asgnset_1 \vdash Idtseq{:}(\tau seq, asgnset_2)\mathbf{idtseq}$$
$$\frac{\text{isNotRepeatedTypedId}(Idt,Idtseq)}{\pi, asgnset \vdash Idt,Idtseq{:}((\tau,\tau seq),\ asgnset_2)\mathbf{idtseq}}$$

B.17 Identifier Typed

In this subsection we show the typing rules for all the syntactic phrases of the domain Identifier typed.

B.17.1 *I*

An identifier is a well typed identifier typed with a "symbol" type.

$$\pi, asgnset \vdash I{:}(\text{symbol}, asgnset \cup \{I\})\mathbf{idt}$$

B.17.2 *I::T*

A type-qualified-identifier is a well typed identifier typed with a type T.

$$\frac{\pi \vdash T{:}(\tau)\mathbf{type}}{\pi, asgnset \vdash I{::}T{:}(\tau, asgnset \cup \{I\})\mathbf{idt}}$$

B.17.3 *I:=E*

An expression-qualified-identifier is a well typed identifier typed with a type of expression E.

$$\frac{\pi \vdash E{:}(\tau)\mathbf{exp}}{\pi, asgnset \vdash I{:=}E{:}(\tau, asgnset \cup \{I\})\mathbf{idt}}$$

B.17.4 *I::T:=E*

A fully qualified identifier is a well typed identifier typed with a type of T, where the type T is more general than the type of expression E.

$$\frac{\pi \vdash T{:}(\tau_1)\mathbf{type} \quad \pi \vdash E{:}(\tau_2)\mathbf{exp} \quad \text{superType}(\tau_1, \tau_2)}{\pi, asgnset \vdash I{::}T{:=}E{:}(\tau_1, asgnset \cup \{I\})\mathbf{idt}}$$

B.18 Binary Operators

In this subsection we show the typing rules for all the syntactic phrases of the domain Binary Operators.

B.18.1 +

The phrase "+" is a well typed binary operator with type τ_3, where τ_3 determines the type of output of this operator when applied to the compatible types τ_1 and τ_2.

$$\pi \vdash +{:}(\tau_3)\mathbf{bop}, \text{ if } \exists \tau_3 : \tau_3 = \text{op}(\tau_1, \tau_2, +)$$

B.18.2 -

The phrase "-" is a well typed binary operator with type τ_3, where τ_3 determines the type of output of this operator when applied to the compatible types τ_1 and τ_2.

$$\pi \vdash \text{-}:(\tau_3)\mathbf{bop}, \text{ if } \exists \tau_3 : \tau_3 = \text{op}(\tau_1, \tau_2, \text{-})$$

B.18.3 /

The phrase "/" is a well typed binary operator with type τ_3, where τ_3 determines the type of output of this operator when applied to the compatible types τ_1 and τ_2.

$$\pi \vdash /:(\tau_3)\mathbf{bop}, \text{ if } \exists \tau_3 : \tau_3 = \text{op}(\tau_1, \tau_2, /)$$

B.18.4 *

The phrase "*" is a well typed binary operator with type τ_3, where τ_3 determines the type of output of this operator when applied to the compatible types τ_1 and τ_2.

$$\pi \vdash *:(\tau_3)\mathbf{bop}, \text{ if } \exists \tau_3 : \tau_3 = \text{op}(\tau_1, \tau_2, *)$$

B.18.5 mod

The phrase "**mod**" is a well typed binary operator with type τ_3, where τ_3 determines the type of output of this operator when applied to the compatible types τ_1 and τ_2.

$$\pi \vdash \mathbf{mod}:(\tau_3)\mathbf{bop}, \text{ if } \exists \tau_3 : \tau_3 = \text{op}(\tau_1, \tau_2, \mathbf{mod})$$

B.18.6 <

The phrase "<" is a well typed boolean operator such that the operator is well defined for the compatible types τ_1 and τ_2.

$$\pi \vdash <:(\text{boolean})\mathbf{bop}, \text{ if } \exists \mathit{boolean} : \mathit{boolean} = \text{op}(\tau_1, \tau_2, <)$$

B.18.7 >

The phrase ">" is a well typed boolean operator such that the operator is well defined for the compatible types τ_1 and τ_2.

$$\pi \vdash >:(\text{boolean})\mathbf{bop}, \text{ if } \exists \mathit{boolean} : \mathit{boolean} = \text{op}(\tau_1, \tau_2, >)$$

B.18.8 <=

The phrase "<=" is a well typed boolean operator such that the operator is well defined for the compatible types τ_1 and τ_2.

$$\pi \vdash <=:(\text{boolean})\textbf{bop}, \text{ if } \exists \textit{boolean} : \textit{boolean} = \text{op}(\tau_1, \tau_2, <=)$$

B.18.9 >=

The phrase ">=" is a well typed boolean operator such that the operator is well defined for the compatible types τ_1 and τ_2.

$$\pi \vdash >=:(\text{boolean})\textbf{bop}, \text{ if } \exists \textit{boolean} : \textit{boolean} = \text{op}(\tau_1, \tau_2, >=)$$

B.19 Unary Operators

In this subsection we show the typing rules for all the syntactic phrases of the domain Unary Operators.

B.19.1 not

The phrase "**not**" is a well typed unary boolean operator defined over a type boolean.

$$\pi \vdash \textbf{not}:(\text{boolean})\textbf{uop}, \text{ if } \exists \textit{boolean} : \textit{boolean} = \text{op}(\text{boolean}, \textbf{not})$$

B.19.2 +

The phrase "+" is a well typed unary operator of type τ_2 defined over a compatible type τ_1.

$$\pi \vdash +:(\tau_2)\textbf{uop}, \text{ if } \exists \tau_1, \tau_2 : \tau_2 = \text{op}(\tau_1, +)$$

B.19.3 -

The phrase "-" is a well typed unary operator of type τ_2 defined over a compatible type τ_1.

$$\pi \vdash -:(\tau_2)\textbf{uop}, \text{ if } \exists \tau_1, \tau_2 : \tau_2 = \text{op}(\tau_1, -)$$

B.20 Especial Operators

In this subsection we show the typing rules for all the syntactic phrases of the domain Especial Operators.

B.20.1 op(E_1,E_2)

The phrase "**op**(E_1,E_2)" is a well typed special operator of type τ where E_1 indicates the positions of operands in an expression E_2.

$$\frac{\pi \vdash E_1{:}(\text{integer})\mathbf{exp} \quad \pi \vdash E_2{:}(\tau)\mathbf{esubexp}}{\pi \vdash \mathbf{op}(E_1,E_2){:}(\tau)\mathbf{esop}}$$

B.20.2 op(E)

The phrase "**op**(E)" is a well typed special operator of type seq(τ) where seq(τ) is the sequence of the type of the operands of an expression E.

$$\frac{\pi \vdash E{:}(\tau)\mathbf{esubexp}}{\pi \vdash \mathbf{op}(E){:}(\mathbf{seq}(\tau))\mathbf{esop}}$$

B.20.3 op($E_1 \ldots E_2$,E_3)

The phrase "**op**($E_1 \ldots E_2$,E_3)" is a well typed special operator of type seq(τ) where seq(τ) is the sequence of the type of the operands (from E_1 to E_2) of an expression E_3.

$$\frac{\pi \vdash E_1{:}(\text{integer})\mathbf{exp} \quad \pi \vdash E_2{:}(\text{integer})\mathbf{exp} \quad \pi \vdash E_3{:}(\tau)\mathbf{esubexp}}{\pi \vdash \mathbf{op}(E_1 \ldots E_2,E_3){:}(\mathbf{seq}(\tau))\mathbf{esop}}$$

B.20.4 nops(E)

The phrase "**nops**(E)" is a well typed special operator of type integer, which presents the number of operands of an expression E.

$$\frac{\pi \vdash E{:}(\tau)\mathbf{esubexp}}{\pi \vdash \mathbf{nops}(E){:}(\text{integer})\mathbf{esop}}$$

B.20.5 subsop(E_1=E_2,E_3)

The phrase "**subsop**(E_1=E_2,E_3)" is a well typed special operator of type τ_1, where τ_1 is the type of the substitutes of the expression positions E_1 with expression E_2 in expression E_3.

$$\frac{\pi \vdash E_1\text{:(integer)}\mathbf{exp} \quad \pi \vdash E_2\text{:}(\tau_1)\mathbf{exp} \quad \pi \vdash E_3\text{:}(\tau_2)\mathbf{esubexp} \quad \text{superType}(\tau_1, \tau_2)}{\pi \vdash \mathbf{subsop}(E_1{=}E_2, E_3)\text{:}(\tau_1)\mathbf{esop}}$$

B.20.6 subs(*I*=*E*$_1$,*E*$_2$)

The phrase "**subs**(I=E_1,E_2)" is a well typed special operator of type τ_2, where τ_2 is the type of an expression E_2 in which an identifier I is substituted with an expression E_1.

$$\frac{\pi \vdash I\text{:(symbol)}\mathbf{id} \quad \pi \vdash E_1\text{:}(\tau_1)\mathbf{exp} \quad \pi \vdash E_2\text{:}(\tau_2)\mathbf{esubexp} \quad \text{checkSubs}(I,\ E_2)}{\pi \vdash \mathbf{subs}(I{=}E_1, E_2)\text{:}(\tau_2)\mathbf{esop}}$$

B.20.7 *"E"*

The phrase "*"E"*" is a well typed special operator of type uneval, an expression is doubly single quoted as a general representation of unevaluated expression in the *MiniMaple*.

$$\frac{\pi \vdash E\text{:}(\tau)\mathbf{exp}}{\pi \vdash \text{"}E\text{"}\text{:(uneval)}\mathbf{esop}}$$

B.20.8 [*Eseq*]

The phrase "[*Eseq*]" is a well typed special operator of type tuple. This construct is also used to represent the list contents where this construct is of type list(τ).

$$\frac{\pi \vdash Eseq\text{:}(\tau seq)\mathbf{expseq}}{\pi \vdash [Eseq]\text{:}([\tau seq])\mathbf{esop}}$$

B.20.9 seq(*E*$_1$,*I*=*E*$_2$...*E*$_3$)

The phrase "**seq**(E_1,I=E_2...E_3)" is a well typed special operator of type seq(τ). The operator generates a sequence of expression E_1 with the range from E_2 to E_3.

$$\frac{\text{override}(\pi,\{\text{I:integer}\}) \vdash E_1{:}(\tau)\mathbf{esubexp} \quad \pi \vdash E_2{:}(\text{integer})\mathbf{exp} \quad \pi \vdash E_3{:}(\text{integer})\mathbf{exp}}{\pi \vdash \mathbf{seq}(E_1,I{=}E_1 \ldots E_2){:}(\mathbf{seq}(\tau))\mathbf{esop}}$$

B.20.10 seq(E_1,*I in* E_2)

The phrase "**seq**(E_1,*I in* E_2)" is a well typed special operator of type seq(τ_1). The operator generates a sequence of expression E_1 applied to the identifier I in expression E_2.

$$\frac{\pi \vdash E_2{:}(\tau_2)\mathbf{esubexp} \quad \text{override}(\pi,\{\text{I:}\tau_2\}) \vdash E_1{:}(\tau_1)\mathbf{exp}}{\pi \vdash \mathbf{seq}(E_1,I \textit{ in } E_2){:}(\mathbf{seq}(\tau_1))\mathbf{esop}}$$

B.20.11 eval(*I,1*)

The phrase "**eval**(*I,1*)" is a well typed special operator of type uneval. After the evaluation one level of single quote is stripped off in an identifier I.

$$\frac{\pi \vdash I{:}(\text{uneval})\mathbf{id}}{\pi \vdash \mathbf{eval}(I,1){:}(\text{uneval})\mathbf{esop}}$$

B.21 Especial Operators Sub Expressions

In this subsection we show the typing rules for especial operators subexpressions that appear in the especial operators.

B.21.1 *For integer sub expression*

A well typed integer expression E is a well typed special subexpression.

$$\frac{\pi \vdash E{:}(\text{integer})\mathbf{exp}}{\pi \vdash E{:}(\text{integer})\mathbf{esubexp}}$$

B.21.2 *For string sub expression*

A well typed string expression E is a well typed special subexpression.

$$\frac{\pi \vdash E{:}(\text{string})\mathbf{exp}}{\pi \vdash E{:}(\text{string})\mathbf{esubexp}}$$

B.21.3 *For boolean sub expression*

A well typed boolean expression E is a well typed special subexpression.

$$\frac{\pi \vdash E{:}(\text{boolean})\mathbf{exp}}{\pi \vdash E{:}(\text{boolean})\mathbf{esubexp}}$$

B.21.4 *For symbol sub expression*

A well typed symbol expression E is a well typed special subexpression.

$$\frac{\pi \vdash E{:}(\text{symbol})\mathbf{exp}}{\pi \vdash E{:}(\text{symbol})\mathbf{esubexp}}$$

B.21.5 *For uneval sub expression*

A well typed unevaluated expression E is a well typed special subexpression.

$$\frac{\pi \vdash E{:}(\text{uneval})\mathbf{exp}}{\pi \vdash E{:}(\text{uneval})\mathbf{esubexp}}$$

B.21.6 *For list sub expression*

A well typed special subexpression of the type of contents of a list expression E.

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

B.21.7 *For record sub expression*

A well typed special subexpression of union-type of the contents of a tuple expression E.

$$\frac{\pi \vdash E{:}([\tau seq])\mathbf{exp}}{\pi \vdash E{:}(\mathbf{Or}(\tau seq))\mathbf{esubexp}}$$

B.21.8 *For set sub expression*

A well typed special subexpression of type of the contents of a set expression E.

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

B.22 Type Sequence

In this subsection we show the typing rules for all the syntactic phrases of the domain Type Sequence.

B.22.1 *EMPTY*

An *EMPTY* type sequence produces an empty type annotation.

$$\pi \vdash \mathit{EMPTY}\text{:(empty)}\mathbf{typeseq}$$

B.22.2 *T,Tseq*

The phrase "*T,Tseq*" is a well typed type sequence with a syntactic sequence of *T* and *Tseq*. All the types in the sequence must be well defined.

$$\frac{\pi \vdash T\text{:}(\tau)\mathbf{type}\ \pi \vdash \mathit{Tseq}\text{:}(\tau\mathit{seq})\mathbf{typeseq}}{\pi \vdash T, \mathit{Tseq}\text{:}((\tau, \tau\mathit{seq}))\mathbf{typeseq}}$$

B.23 Type

In this subsection we show the typing rules for all the syntactic phrases of the domain Type.

B.23.1 integer

An integer literal is a well defined representative for type integer.

$$\pi \vdash \mathbf{integer}\text{:(integer)}\mathbf{type}$$

B.23.2 boolean

A boolean literal is a well defined representative for type boolean.

$$\pi \vdash \mathbf{boolean}\text{:(boolean)}\mathbf{type}$$

B.23.3 string

A string literal is a well defined representative for type string.

$$\pi \vdash \mathbf{string}\text{:(string)}\mathbf{type}$$

B.23.4 anything

An anything literal is a well defined representative for type anything.

$$\pi \vdash \textbf{anything}\text{:(anything)}\textbf{type}$$

B.23.5 symbol

A symbol literal is a well defined representative for type symbol.

$$\pi \vdash \textbf{symbol}\text{:(symbol)}\textbf{type}$$

B.23.6 void

A void literal is a well defined representative for type void.

$$\pi \vdash \textbf{void}\text{:(void)}\textbf{type}$$

B.23.7 uneval

An uneval literal is a well defined representative for type uneval.

$$\pi \vdash \textbf{uneval}\text{:(uneval)}\textbf{type}$$

B.23.8 {*T*}

A set construct is a well defined representative for type set.

$$\frac{\pi \vdash T\text{:}(\tau)\textbf{type}}{\pi \vdash \{T\}\text{:}(\tau set)\textbf{type}}$$

B.23.9 list(*T*)

The phrase "**list**(T)" is a well typed representative for type list.

$$\frac{\pi \vdash T\text{:}(\tau)\textbf{type}}{\pi \vdash \textbf{list}(T)\text{:}(\text{list}(\tau))\textbf{type}}$$

B.23.10 [*Tseq*]

A tuple construct is a well defined representative for type tuple.

$$\frac{\pi \vdash Tseq\text{:}(\tau seq)\textbf{typeseq}}{\pi \vdash [Tseq]\text{:}([\tau seq])\textbf{type}}$$

B.23.11 procedure[*T*](*Tseq*)

The phrase "procedure[T]($Tseq$)" is a well typed representative for type procedure, where τ is the return type of the procedure and τseq is the type sequence of its parameters.

$$\frac{\pi \vdash T{:}(\tau)\textbf{type} \quad \pi \vdash Tseq{:}(\tau seq)\textbf{type}}{\pi \vdash \text{procedure}[T](Tseq){:}(\text{procedure}[\tau](\tau seq))\textbf{type}}$$

B.23.12 *I*(*Tseq*)

The phrase "$I(Tseq)$" is a well typed representative for named-type with a sequence of types for its parameters.

$$\frac{\pi \vdash Tseq{:}(\tau seq)\textbf{type}}{\pi \vdash I(Tseq){:}(\text{I}(\tau seq))\textbf{type}}$$

B.23.13 Or(*Tseq*)

A union-type construct is a well defined representative for union-type, i.e. Or.

$$\frac{\pi \vdash Tseq{:}(\tau seq)\textbf{type}}{\pi \vdash \textbf{Or}(Tseq){:}(\text{Or}(\tau seq))\textbf{type}}$$

B.23.14 *I*

$\pi \vdash I{:}(\text{type}(\tau))\textbf{type}$, where τ is the name of user-defined type.

B.24 Numeral

$\pi \vdash N{:}(\text{integer})\textbf{num}$, where N is a valid sequence of decimal digits.

C Auxiliary Functions

In the following subsections we define the auxiliary functions used in logical rules to derive typing judgments. The auxiliary functions are defined over type environment and over syntactic domains type, identifier, typed identifier, parameter, expression and some utility functions over return flag and general syntactic domain sequences.

C.1 Functions over Type Environment

In this section we define the functions over type environment.

combine :**TypeEnvironment** $\times$ **TypeEnvironment**
$\rightarrow$ **TypeEnvironment**

The function combines the identifiers of the former type environment with the identifiers in latter type environment. The result type environment has the disjoint identifiers with their corresponding types and the common identifiers with an or-type τ_3 of the two corresponding types.

$$\begin{aligned} combine(\pi_1, \pi_2) = &\{(I : \tau_1) \in \pi_1 : \neg\exists (I : \tau_2) \in \pi_2)\} \\ &\cup \{(I : \tau_2) \in \pi_2 : \neg\exists (I : \tau_1) \in \pi_1)\} \\ &\cup \{(I : \tau_3) : \exists (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_2 \\ &\qquad \wedge \tau_3 = orCombine(\tau_1, \tau_2)\} \end{aligned}$$

orCombine :**TypeEnvironment** $\times$ **TypeEnvironment**
$\rightarrow$ **TypeEnvironment**

The function combines the common identifiers (in both the type environments) with an or-type τ_3 of the two corresponding types.

$$\begin{aligned} orCombine(\pi_1, \pi_2) = \{(I : \tau_3) : \exists \tau_1, \tau_2 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \\ \wedge \tau_3 = orCombine(\tau_1, \tau_2)\} \end{aligned}$$

andCombine :**TypeEnvironment** $\times$ **TypeEnvironment**
$\rightarrow$ **TypeEnvironment**

The function combines the identifiers of the former type environment with the identifiers in latter type environment. The result type environment has the disjoint identifiers with their corresponding types and the common identifiers with an and-type τ_3 of the two corresponding types.

$$\begin{aligned} andCombine(\pi_1, \pi_2) = &\{(I : \tau_1) \in \pi_1 : \neg\exists (I : \tau_2) \in \pi_2)\} \\ &\cup \{(I : \tau_2) \in \pi_2 : \neg\exists (I : \tau_1) \in \pi_1)\} \\ &\cup \{(I : \tau_3) : \exists (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_1 \\ &\qquad \wedge \tau_3 = andCombine(\tau_1, \tau_2)\} \end{aligned}$$

override :**TypeEnvironment** $\times$ **TypeEnvironment**
$\rightarrow$ **TypeEnvironment**

The function overrides the identifiers of former type environment with the identifiers in latter type environment. The function removes the common identifiers from the former type environment and unions with the identifiers in latter type environment to produce the result.

$$override(\pi_1, \pi_2) = (\pi_1 \setminus \{(I : \tau_1) \in \pi_1 : \exists \tau_2 : (I : \tau_2) \in \pi_2\}) \cup \pi_2$$

override :**TypeEnvironment** $\times$ **set**(**TypeEnvironment**)
$\rightarrow$ **TypeEnvironment**

The function overrides the identifiers of former type environment with the identifiers in the set of type environments. The function removes the common identifiers from the former type environment and unions with the identifiers in the set of type environments as a result.

$$\begin{aligned} override(\pi_1, \pi set) =& \pi_1 \setminus \{(I : \tau_1) \in \pi_1 : \exists \pi_2 \in \pi set \wedge \exists \tau_2 : (I : \tau_2) \in \pi_2\} \\ & \cup \{(I : \tau_2) \in \pi_2 : \forall \pi_2 \in \pi set\} \end{aligned}$$

override :**TypeEnvironment** $\times$ **sequence**(**TypeEnvironment**)
$\rightarrow$ **TypeEnvironment**

The function overrides the identifiers of former type environment with the identifiers in the sequence of type environments. The function removes the common identifiers from the former type environment and unions with the identifiers of the type environments from the sequence.

$$\begin{aligned} override(\pi_1, \pi seq) =& (\pi_1 \setminus \{(I : \tau_1) \in \pi_1 : \exists \pi_2 \in range(\pi seq) : \exists \tau_2 : (I : \tau_2) \in \pi_2\}) \\ & \cup \{(I : \tau_2) \in \pi_2 : \forall \pi_2 \in range(\pi seq)\} \end{aligned}$$

overrideLocal :**TypeEnvironment** $\times$ **TypedIdentifierSequence**
$\times$**TypeSequence** $\rightarrow$ **TypeEnvironment**

The function overrides the identifiers in type environment with the identifiers (from the sequence of typed identifier) and their corresponding types (from the sequence of type).

$$overrideLocal(\pi, empty, empty) = \pi$$

$$\begin{aligned} & overrideLocal(\pi, (It, Itseq), (\tau, \tau seq)) = \\ & \quad overrideLocal(\pi \setminus \{(TypedIdtoId(It) : \tau_1) \in \pi\} \\ & \qquad \cup \{(TypedIdtoId(It) : \tau)\}, (Itseq), (\tau seq))) \end{aligned}$$

restrict : **TypeEnvironment** $\times$ **set**(**Identifier**) $\rightarrow$ **TypeEnvironment**

The function restricts the type environment to only those identifiers which are also in the set of identifiers.

$restrict(\pi, idset) = (I : \tau) \in \pi : I \in idset$

specialize :**TypeEnvironment** $\times$ **TypeEnvironment**
$\rightarrow$ **TypeEnvironment**

The function specializes the identifiers of the former type environment with the identifiers in latter type environment. The result type environment has the disjoint identifiers with their corresponding types and the common identifiers with a subtype τ_2 between the two types, if so.

$$\begin{aligned} specialize(\pi_1, \pi_2) = & \{(I : \tau_1) \in \pi_1 : \neg\exists (I : \tau_2) \in \pi_2)\} \\ & \cup \{(I : \tau_2) \in \pi_2 : \neg\exists (I : \tau_1) \in \pi_1)\} \\ & \cup \{(I : \tau_2) : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \\ & \qquad \wedge superType(\tau_1, \tau_2)\} \end{aligned}$$

update :**TypeEnvironment** $\times$ **IdentifierSequence** $\times$ **sequence**(**Type**)
$\rightarrow$ **TypeEnvironment**

The function updates the type environment with the identifiers (in the sequence of identifiers) with their corresponding types (in the sequence of types).

$update(\pi, empty, empty) = \pi$

$$\begin{aligned} & update(\pi, (Id, Idseq), (\tau, \tau seq)) = \\ & \qquad update(\pi \setminus \{(Id : \tau_1) : (id : \tau_1) \in \pi\} \cup \{(Id : \tau)\}, (Idseq), (\tau seq)) \end{aligned}$$

C.2 Functions over Type

In this section we define the functions over type.

orCombine : **Type** $\times$ **Type** $\rightarrow$ **Type**

The function returns the general type (if there) between the two (former and later type), otherwise returns the union of the two types.

$$orCombine(integer, \tau) = \begin{cases} integer & \text{if } \tau = integer \\ anything & \text{if } \tau = anything \\ Or(integer, \tau) & \text{if } \tau \notin \{integer, anything\} \end{cases}$$

$$orCombine(rational, \tau) = \begin{cases} rational & \text{if } \tau = rational \vee \tau = integer \\ anything & \text{if } \tau = anything \\ Or(rational, \tau) & \text{if } \tau \notin \{rational, anything\} \end{cases}$$

$$orCombine(float, \tau) = \begin{cases} float & \text{if } \tau = float \\ anything & \text{if } \tau = anything \\ Or(float, \tau) & \text{if } \tau \notin \{float, anything\} \end{cases}$$

$$orCombine(boolean, \tau) = \begin{cases} boolean & \text{if } \tau = boolean \\ anything & \text{if } \tau = anything \\ Or(boolean, \tau) & \text{if } \tau \notin \{boolean, anything\} \end{cases}$$

$$orCombine(string, \tau) = \begin{cases} string & \text{if } \tau = string \\ anything & \text{if } \tau = anything \\ Or(string, \tau) & \text{if } \tau \notin \{string, anything\} \end{cases}$$

$$orCombine(anything, \tau) = anything$$

$$orCombine(\{\tau\}, \tau_1) = \begin{cases} \{\tau\} & \text{if } \exists \tau_2 : \tau_1 = \{\tau_2\} \\ anything & \text{if } \tau = anything \\ Or(\{\tau\}, \tau_1) & \text{if } \tau_1 \neq anything \wedge \neg \exists \tau_2 : \tau_1 = \{\tau_2\} \end{cases}$$

$$orCombine(list(\tau), \tau_1) = \begin{cases} list(\tau) & \text{if } \exists \tau_2 : \tau_1 = list(\tau_2) \\ anything & \text{if } \tau = anything \\ Or(list(\tau), \tau_1) & \text{if } \tau_1 \neq anything \wedge \neg \exists \tau_2 : \tau_1 = list(\tau_2) \end{cases}$$

$$orCombine([\tau seq], \tau_1) = \begin{cases} [orCombineSeq(\tau seq, \tau seq_1)] & \text{if } \exists \tau seq_1 : \tau_1 = [\tau seq_1] \\ anything & \text{if } \tau = anything \\ Or([\tau seq], \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg \exists \tau seq_1 : \tau_1 = [\tau seq_1] \end{cases}$$

$$orCombine(procedure[\tau](\tau seq), \tau_1) =$$

$$\begin{cases} procedure[\tau][orCombineSeq(\tau seq, \tau seq_1)] & \text{if } \exists \tau_2, \tau seq_1 \\ & : \tau_1 = procedure[\tau_2](\tau seq_1) \\ anything & \text{if } \tau = anything \\ Or(procedure[\tau](\tau seq), \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg \exists \tau_2, \tau seq_1 \\ & : \tau_1 = procedure[\tau_2](\tau seq_1) \end{cases}$$

$$orCombine(I(\tau seq), \tau_1) = \begin{cases} I(orCombineSeq(\tau seq, \tau seq_1)) & \text{if } \exists I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \\ anything & \text{if } \tau = anything \\ Or(I(\tau seq), \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg \exists I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \end{cases}$$

$$orCombine(Or(\tau seq), \tau_1) = \begin{cases} Or(orCombineSeq(\tau seq, \tau seq_1)) & \text{if } \exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq) \\ & \wedge hasTypeAnything(\tau seq_1) \\ anything & \text{if } \tau = anything \\ Or(\tau seq, \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg hasTypeAnything(\tau seq) \end{cases}$$

$$orCombine(symbol, \tau) = \begin{cases} symbol & \text{if } \tau = symbol \\ anything & \text{if } \tau = anything \\ Or(symbol, \tau) & \text{if } \tau \notin \{symbol, anything\} \end{cases}$$

$$orCombine(void, \tau) = \begin{cases} void & \text{if } \tau = void \\ anything & \text{if } \tau = anything \\ Or(void, \tau) & \text{if } \tau \notin \{void, anything\} \end{cases}$$

$$orCombine(uneval, \tau) = \begin{cases} uneval & \text{if } \tau = uneval \\ anything & \text{if } \tau = anything \\ Or(uneval, \tau) & \text{if } \tau \notin \{uneval, anything\} \end{cases}$$

orCombineSeq : **TypeSequence** $\times$ **TypeSequence** $\rightarrow$ **TypeSequence**

The function returns the sequence of general types (if there) of all the corresponding pairs of types from the sequences (former and later type), otherwise returns the sequence of union-types of the two sequences.

$orCombineSeq(empty, empty) \Leftrightarrow empty$

$orCombineSeq(\tau seq, empty) \Leftrightarrow empty$

$orCombineSeq(empty, \tau seq) \Leftrightarrow empty$

$$orCombineSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) \Leftrightarrow \\ (orCombine(\tau_1, \tau_2), orCombineSeq(\tau seq_1, \tau seq_2))$$

andCombine : **Type** $\times$ **Type** $\rightarrow$ **Type**

In principle the function returns the concrete type between the two types (former and later type).

$andCombine(integer, \tau) = integer$ if $\tau = integer$

$andCombine(integer, \tau) = rational$ if $\tau = rational$

$andCombine(integer, \tau) = float$ if $\tau = float$

$andCombine(boolean, \tau) = boolean$ if $\tau = boolean$

$andCombine(string, \tau) = string$ if $\tau = string$

$andCombine(anything, \tau) = \tau$

$$andCombine(\{\tau\}, \tau_1) = \begin{cases} \{\tau\} & \text{if } \tau_1 = anything \\ \{andCombine(\tau, \tau_2)\} & \text{if } \exists \tau_2 : \tau_1 = \{\tau_2\} \\ & \wedge andCombinable(\tau, \tau_2) \\ \{\tau\} & \text{otherwise} \end{cases}$$

$$andCombine(list(\tau), \tau_1) = \begin{cases} list(\tau) & \text{if } \tau_1 = anything \\ list(andCombine(\tau, \tau_2)) & \text{if } \exists \tau_2 : \tau_1 = list(\tau_2) \\ & \wedge andCombinable(\tau, \tau_2) \\ list(\tau) & \text{otherwise} \end{cases}$$

$andCombine([\tau seq], \tau_1) =$

$$\begin{cases} [\tau seq] & \text{if } \tau_1 = anything \\ [andCombineSeq(\tau seq, \tau seq_1) & \text{if } \exists \tau seq_1 : \tau_1 = [\tau seq_1] \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ [\tau seq] & \text{otherwise} \end{cases}$$

$$andCombine(procedure[\tau](\tau seq), \tau_1) =$$
$$\begin{cases} procedure[\tau](\tau seq) & \text{if } \tau_1 = anything \\ procedure[\tau'](\tau seq') & \text{if} \\ & \exists \tau', \tau seq', \tau_2, \tau seq_1 : \tau_1 \\ & = procedure[\tau_2](\tau seq_1) \\ & \wedge andCombinable(\tau, \tau_2) \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ & \wedge \tau' = andCombine(\tau, \tau_2) \\ & \wedge \tau seq' = andCombineSeq(\tau seq, \tau seq_1) \\ procedure[\tau](\tau seq) & \text{otherwise} \end{cases}$$

$$andCombine(I(\tau seq), \tau_1) =$$
$$\begin{cases} I(\tau seq) & \text{if } \tau_1 = anything \\ I(andCombineSeq(\tau seq, \tau seq_1)) & \text{if} \\ & \exists I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ I(\tau seq) & \text{otherwise} \end{cases}$$

$$andCombine(Or(\tau seq), \tau_1) =$$
$$\begin{cases} Or(\tau seq) & \text{if } \tau_1 = anything \\ & \wedge \neg hastTypeAnything(\tau seq) \\ Or(andCombineSeq(\tau seq, \tau seq_1)) & \text{if } \exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq) \\ & \wedge \neg hastTypeAnything(\tau_1) \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ Or(\tau seq) & \text{otherwise} \end{cases}$$

$$andCombine(symbol, \tau) = symbol$$

$$andCombine(void, \tau) = void$$

$andCombine(uneval, \tau) = uneval$

andCombineSeq : **TypeSequence** $\times$ **TypeSequence** $\rightarrow$ **TypeSequence**

The function returns the sequence of concrete types of all the corresponding pairs of types from the sequences (former and later type).

$andCombineSeq(empty, empty) = empty$

$andCombineSeq(empty, (\tau, \tau seq)) = empty$

$andCombineSeq((\tau, \tau seq), empty) = empty$

$$andCombineSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) = \\ (andCombine(\tau_1, \tau_2), andCombineSeq(\tau seq_1, \tau seq_2))$$

negateType : **Type** $\times$ **Type** $\rightarrow$ **Type**

The function returns the type (if there exists) obtained by logically subtracting the former from the latter type. In most cases, this type is the subtype of the latter (super) type excluding the former type. As a special case subtraction of any type from type *anything* is not allowed as it does not give any information about types.

$$negateType(integer, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - integer \end{cases}$$

$$negateType(rational, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - rational \end{cases}$$

$$negateType(float, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - float \end{cases}$$

$$negateType(boolean, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - boolean \end{cases}$$

$$negateType(string, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - string \end{cases}$$

$$negateType(anything, \tau) = anything \qquad if \tau = anything$$

$$negateType(\{\tau\}, \tau_1) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau_1 = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - \{\tau\} \end{cases}$$

$$negateType(list(\tau), \tau_1) = \\ \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau_1 = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - list(\tau) \end{cases}$$

$$negateType([\tau seq], \tau_1) = \begin{cases} Or(\tau seq_2) & \text{if } \exists \tau seq_1, \tau seq_2 : \tau_1 = Or(\tau seq_1) \\ & \wedge \tau seq_2 = \tau seq_1 - [\tau seq] \end{cases}$$

$$negateType(procedure[\tau](\tau seq), \tau_1) = \begin{cases} Or(\tau seq_2) & \text{if } \exists \tau seq_1, \tau seq_2 : \tau_1 = Or(\tau seq_1) \\ & \wedge \tau seq_2 = \tau seq_1 - procedure[\tau](\tau seq) \end{cases}$$

$$negateType(I(\tau seq), \tau_1) = \begin{cases} Or(\tau seq_2) & \text{if } \exists \tau seq_1, \tau seq_2 : \tau_1 = Or(\tau seq_1) \\ & \wedge \tau seq_2 = \tau seq_1 - I(\tau seq) \end{cases}$$

$$negateType(Or(\tau seq), \tau_1) = \begin{cases} Or(negateTypeSeq(\tau seq_1, \tau seq)) & \text{if } \exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ anything & \text{if } \tau_1 = anything \end{cases}$$

$$negateType(symbol, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - symbol \end{cases}$$

$$negateType(void, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - void \end{cases}$$

$$negateType(uneval, \tau) = \begin{cases} Or(\tau seq_1) & \text{if } \exists \tau seq, \tau seq_1 : \tau = Or(\tau seq) \\ & \wedge \tau seq_1 = \tau seq - uneval \end{cases}$$

negateTypeSeq : **TypeSequence** $\times$ **TypeSequence** $\rightarrow$ **TypeSequence**

The function returns the type sequence (if there exists) obtained by logically subtracting the former from the latter type sequence.

$$negateTypeSeq(empty, (\tau, \tau seq)) = (\tau, \tau seq)$$

$$negateTypeSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) = (negateType(\tau_1, \tau_2), negateTypeSeq(\tau seq_1, (\tau_2, \tau seq_2)))$$

op : **Type** $\times$ **Type** $\times$ **BinaryOperator** $\rightarrow$ **Type**

The function returns the result type of the binary operation (operator) on the given types.

$$op(\tau_1, \tau_2, +) = \begin{cases} integer & \text{if } \tau_1 = integer \wedge \tau_2 = integer \\ float & \text{if } \tau_1 = float \wedge \tau_2 = float \\ rational & \text{if } \tau_1 = rational \wedge \tau_2 = rational \\ rational & \text{if } (\tau_1 = integer \wedge \tau_2 = rational) \\ & \vee(\tau_1 = rational \wedge \tau_2 = integer) \end{cases}$$

$$op(\tau_1, \tau_2, -) = \begin{cases} integer & \text{if } \tau_1 = integer \wedge \tau_2 = integer \\ float & \text{if } \tau_1 = float \wedge \tau_2 = float \\ rational & \text{if } \tau_1 = rational \wedge \tau_2 = rational \\ rational & \text{if } (\tau_1 = integer \wedge \tau_2 = rational) \\ & \vee(\tau_1 = rational \wedge \tau_2 = integer) \end{cases}$$

$$op(\tau_1, \tau_2, /) = \begin{cases} float & \text{if } (\tau_1 = float \wedge \tau_2 = float) \\ & \vee(\tau_1 = integer \wedge \tau_2 = float) \\ & \vee(\tau_1 = float \wedge \tau_2 = integer) \\ rational & \text{if } \tau_1 = integer \wedge \tau_2 = integer \\ rational & \text{if } \tau_1 = rational \wedge \tau_2 = rational \\ rational & \text{if } (\tau_1 = integer \wedge \tau_2 = rational) \\ & \vee(\tau_1 = rational \wedge \tau_2 = integer) \end{cases}$$

$$op(\tau_1, \tau_2, *) = \begin{cases} integer & \text{if } \tau_1 = integer \wedge \tau_2 = integer \\ float & \text{if } (\tau_1 = float \wedge \tau_2 = float) \\ & \vee(\tau_1 = integer \wedge \tau_2 = float) \\ & \vee(\tau_1 = float \wedge \tau_2 = integer) \\ rational & \text{if } \tau_1 = rational \wedge \tau_2 = rational \\ rational & \text{if } (\tau_1 = integer \wedge \tau_2 = rational) \\ & \vee(\tau_1 = rational \wedge \tau_2 = integer) \end{cases}$$

$$op(\tau_1, \tau_2, mod) = \begin{cases} float & \text{if } (\tau_1 = float \wedge \tau_2 = float) \\ & \vee(\tau_1 = integer \wedge \tau_2 = float) \\ & \vee(\tau_1 = float \wedge \tau_2 = integer) \\ rational & \text{if } \tau_1 = integer \wedge \tau_2 = integer \\ rational & \text{if } \tau_1 = rational \wedge \tau_2 = rational \\ rational & \text{if } (\tau_1 = integer \wedge \tau_2 = rational) \\ & \vee(\tau_1 = rational \wedge \tau_2 = integer) \end{cases}$$

$$op(\tau_1, \tau_2, <) = \begin{cases} boolean & \text{if } equalTypes(\tau_1, \tau_2) \wedge (\tau_1 = integer \vee \tau_1 = float \\ & \vee \tau_1 = string \vee \tau_1 = rational \vee \exists(\tau_3 : \tau_1 = list(\tau_3)) \\ & \vee (\exists \tau_3 : \tau_1 = \{\tau_3\}) \vee (\exists \tau seq : \tau_1 = Or(\tau seq))) \end{cases}$$

$$op(\tau_1, \tau_2, >) = \begin{cases} boolean & \text{if } equalTypes(\tau_1, \tau_2) \wedge (\tau_1 = integer \vee \tau_1 = float \\ & \vee \tau_1 = string \vee \tau_1 = rational \vee \exists(\tau_3 : \tau_1 = list(\tau_3)) \\ & \vee (\exists \tau_3 : \tau_1 = \{\tau_3\}) \vee (\exists \tau seq : \tau_1 = Or(\tau seq))) \end{cases}$$

$$op(\tau_1, \tau_2, <=) = \begin{cases} boolean & \text{if } equalTypes(\tau_1, \tau_2) \wedge (\tau_1 = integer \vee \tau_1 = float \\ & \vee \tau_1 = string \vee \tau_1 = rational \vee \exists(\tau_3 : \tau_1 = list(\tau_3)) \\ & \vee (\exists \tau_3 : \tau_1 = \{\tau_3\}) \vee (\exists \tau seq : \tau_1 = Or(\tau seq))) \end{cases}$$

$$op(\tau_1, \tau_2, >=) = \begin{cases} boolean & \text{if } equalTypes(\tau_1, \tau_2) \wedge (\tau_1 = integer \vee \tau_1 = float \\ & \vee \tau_1 = string \vee \tau_1 = rational \vee \exists(\tau_3 : \tau_1 = list(\tau_3)) \\ & \vee (\exists \tau_3 : \tau_1 = \{\tau_3\}) \vee (\exists \tau seq : \tau_1 = Or(\tau seq))) \end{cases}$$

op : **Type** $\times$ **UnaryOperator** $\rightarrow$ **Type**

The function returns the result type of the unary operation (operator) on the given type.

$op(\tau, +) = \tau$ if $\tau = integer \vee \tau = rational \vee \tau = float$

$op(\tau, -) = \tau$ if $\tau = integer \vee \tau = rational \vee \tau = float$

$op(\tau, not) = \tau$ if $\tau = boolean$

getModTypeEnv : **Type** $\rightarrow$ **TypeEnvironment**

The function returns the type environment for the given module type. The type environment contains the exported identifiers and their corresponding types for the module.

$getModTypeEnv(module(\pi)) = \pi$

getModuleEnvsSeq : **TypeSequence** $\rightarrow$ **TypeEnvironmentSequence**

The function returns the sequence of type environments extracted from the given sequence of module types. Each type environment (for each module type) contains the exported identifiers and their corresponding types of that module.

$getModuleEnvsSeq(\tau, empty) = getModTypeEnv(\tau)$

$getModuleEnvsSeq(\tau, \tau seq) = getModTypeEnv(\tau), getModuleEnvsSeq(\tau seq)$

C.3 Functions over Identifier

In this section we define the functions over identifier.

IdSeqToSet : **IdentifierSequence** $\rightarrow$ **set**(**Identifier**)

The function converts the sequence of identifiers to a set of identifiers.

$IdSeqToSet(empty) = empty$

$IdSeqToSet((I, Iseq)) = \{I\} \cup IdSeqToSet(Iseq)$

C.4 Functions over Typed Identifier

In this section we define the functions over typed identifier.

TypedIdToId : **TypedIdentifier** $\rightarrow$ **Identifier**

The function extracts an identifier from the typed identifier.

$TypedIdToId(I) = I$

$TypedIdToId(I :: T) = I$

$TypedIdToId(I := E) = I$

$TypedIdToId(I :: T := E) = I$

TypedIdSeqToSet : **TypedIdentifierSequence** $\rightarrow$ **set**(**Identifier**)

The function extracts a set of identifiers from a sequence of typed identifiers.

$TypedIdSeqToSet(empty) = empty$

$TypedIdSeqToSet(It, Itseq) = TypedIdToId(It) \cup TypedIdSeqToSet(Itseq)$

C.5 Functions over Parameter

In this section we define the functions over parameter.

ParamToId : **Parameter** $\rightarrow$ **Identifier**

The function extracts an identifier from a given parameter.

$ParamToId(I) = I$

$ParamToId(I :: M) = I$

ParamSeqToSet : **ParameterSequence** $\rightarrow$ **set**(**Identifier**)

The function extracts a set of identifiers from a sequence of parameters.

$ParamSeqToSet(empty) = empty$

$ParamSeqToSet(P, Pseq) = ParamToId(P) \cup ParamSeqToSet(Pseq)$

C.6 Functions over Expression

In this section we define the functions over expression.

expToIdSet : **Expression** $\rightarrow$ **set**(**Identifier**)

The function returns a set of all identifiers that occur in the given expression.

$expToIdSet(expr) = \{I : expr = Identifier\} \cup expToIdSet(expr)$

C.7 Functions over Return Flag

In this section we define the functions over return flag.

ret : **ReturnFlag** $\times$ **ReturnFlag** $\rightarrow$ **ReturnFlag**

In general two syntactic constructs always return if every execution of the two constructs has the last statement (in either construct) as return command.

$$ret(rflag_1, rflag_2) = \begin{cases} not\_aret & \text{if } rflag_1 = not\_aret \vee rflag_2 = not\_aret \\ aret & \text{otherwise} \end{cases}$$

retCseq : **ReturnFlag** $\times$ **ReturnFlag** $\rightarrow$ **ReturnFlag**

In principle the two command sequences always return if every execution of the two command sequences has the last statement as return command.

$$retCseq(rflag_1, rflag_2) = \begin{cases} aret & \text{if } rflag_1 = aret \vee rflag_2 = aret \\ not\_aret & \text{otherwise} \end{cases}$$

retCatch : **ReturnFlag** $\times$ **ReturnFlag** $\rightarrow$ **ReturnFlag**

In principle the try-catch block always return if every execution of the try-catch block has the last statement as return command.

$$retCatch(rflag_1, rflag_2) = \begin{cases} aret & \text{if } rflag_1 = aret \wedge rflag_2 = aret \\ not\_aret & \text{otherwise} \end{cases}$$

C.8 Functions over Domain

In this section we define the general functions over domain.

seqToSet : **sequence**(**D**) $\rightarrow$ **set**(**D**)

The function returns a set of the individual elements of a sequence of any syntactic domain, e.g. to extract a set of identifiers from a syntactic domain identifier sequence.

$seqToSet(d) = \{d(j) : j \in domain(d)\}$

D Auxiliary Predicates

In the following subsections we give the auxiliary predicates used in logical rules to derive typing judgments. The auxiliary predicates are defined over type environment and over syntactic domains type, identifier, typed identifier and parameter.

D.1 Predicates over Type Environment

In this section we define the predicates over type environment.

canSpecialize $\subset$ **TypeEnvironment** $\times$ **TypeEnvironment**

The predicate returns true if all the common identifiers (in both the type environments) have a super-type in the former type environment or correspondingly a subtype in the latter type environment.

$$canSpecialize(\pi_1, \pi_2) \Leftrightarrow (\forall I, \tau_1, \tau_2 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \wedge superType(\tau_1, \tau_2))$$

andCombinable $\subset$ **TypeEnvironment** $\times$ **TypeEnvironment**

The predicate returns true if all the common identifiers (in both the type environments) have the two corresponding types that can be logically intersected.

$$andCombinable(\pi_1, \pi_2) \Leftrightarrow (\forall I, \tau_1, \tau_2 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \Rightarrow andCombinable(\tau_1, \tau_2))$$

D.2 Predicates over Type

In this section we define the predicates over type.

equalTypes $\subset$ **Type** $\times$ **Type**

The predicate returns true (in most cases) if both the types are general to each other, i.e. the equality of the types is defined.

$$equalTypes(integer, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = integer \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(rational, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = rational \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(float, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = float \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(boolean, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = boolean \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(string, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = string \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(anything, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = anything \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(\{\tau\}, \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2 : \tau_1 = \{\tau_2\} \wedge equalTypes(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(list(\tau), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2 : \tau_1 = list(\tau_2) \wedge equalTypes(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes([\tau seq], \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq_1 : \tau_1 = [\tau seq_1] \\ & \wedge equalTypesSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(procedure[\tau](\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2, \tau seq_1 : \tau_1 = procedure[\tau_2](\tau seq_1) \\ & \wedge equalTypes(\tau, \tau_1) \wedge equalTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(I(\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \\ & \wedge equalTypesSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(Or(\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } hasTypeAnything(\tau seq) \\ & \wedge(\exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge hasTypeAnything(\tau seq_1)) \\ & \vee \tau_1 = anything) \\ true & \text{if } \neg hasTypeAnything(\tau seq) \\ & \wedge(\exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq_1)) \\ & \wedge equalTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(symbol, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = symbol \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(void, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = void \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(uneval, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = uneval \\ false & \text{otherwise} \end{cases}$$

$$equalTypes(I, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = I \\ false & \text{otherwise} \end{cases}$$

equalTypeSeq $\subset$ **TypeSequence** $\times$ **TypeSequence**

The predicate returns true (in most cases) if every type from the former sequence of types is general to the corresponding type in the latter sequence of types and vice versa, i.e. the equality of the types is defined.

$$equalTypeSeq(empty, empty) \Leftrightarrow true$$

$$equalTypeSeq(empty, \tau seq) \Leftrightarrow \begin{cases} true & \text{if } \tau seq = empty \\ false & \text{otherwise} \end{cases}$$

$$equalTypeSeq(\tau seq, empty) \Leftrightarrow \begin{cases} true & \text{if } \tau seq = empty \\ false & \text{otherwise} \end{cases}$$

$$equalTypeSeq((\tau, \tau seq), (\tau_1, \tau seq_1)) \Leftrightarrow \begin{cases} true & \text{if } equalTypes(\tau, \tau_1) \\ & \wedge equalTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

superType $\subset$ **Type** $\times$ **Type**

The predicate returns true (in most cases) if the former type is general type than the latter type, i.e. former type is the super type of the latter. Type anything matches every type and returns always true.

$$superType(integer, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = integer \\ false & \text{otherwise} \end{cases}$$

$$superType(rational, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = integer \vee \tau = rational \\ false & \text{otherwise} \end{cases}$$

$$superType(float, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = float \\ false & \text{otherwise} \end{cases}$$

$$superType(boolean, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = boolean \\ false & \text{otherwise} \end{cases}$$

$$superType(string, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = string \\ false & \text{otherwise} \end{cases}$$

$$superType(anything, \tau) \Leftrightarrow true$$

$$superType(\{\tau\}, \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \tau_1 \neq anything \\ & \wedge \exists \tau_2 : \tau_1 = \{\tau_2\} \\ & \wedge superType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$superType(list(\tau), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \tau_1 \neq anything \\ & \wedge \exists \tau_2 : \tau_1 = list(\tau_2) \\ & \wedge superType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$superType([\tau seq], \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq_1 : \tau = [\tau seq_1] \\ & \wedge superTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$superType(procedure[\tau](\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2, \tau seq_1 : \tau_1 = procedure[\tau_2](\tau seq_1) \\ & \wedge superType(\tau, \tau_2) \wedge superTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$superType(I(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists I_1, \tau seq_1 : \tau = I_1(\tau seq_1) \\ & \wedge I = I_1 \\ & \wedge superTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$superType(Or(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } hasTypeAnything(\tau seq) \\ true & \text{if } \exists \tau seq_1 : \tau = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq_1) \\ & \wedge superTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$superType(symbol, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = symbol \\ false & \text{otherwise} \end{cases}$$

$$superType(void, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = void \\ false & \text{otherwise} \end{cases}$$

$$superType(uneval, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = uneval \\ false & \text{otherwise} \end{cases}$$

$$superType(I, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau : \tau = I \\ false & \text{otherwise} \end{cases}$$

superTypeSeq $\subset$ **TypeSequence** $\times$ **TypeSequence**

The predicate returns true (in most cases) if every type from the former sequence of types is general to the corresponding type in the latter sequence of types, i.e. the former type is a super type of the latter type.

$superTypeSeq(empty, empty) \Leftrightarrow true$

$superTypeSeq(empty, (\tau, \tau seq)) \Leftrightarrow false$

$$superTypeSeq((\tau, \tau seq), empty) \Leftrightarrow \begin{cases} true & \text{if } \tau seq = seq(\tau) \\ & \wedge superTypeSeq(\tau seq, empty) \\ false & \text{otherwise} \end{cases}$$

$superTypeSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) \Leftrightarrow$

$$\begin{cases} true & \text{if } \exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge superType(\tau', \tau'') \\ & \wedge superTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge \neg(superType(\tau', \tau'') \\ & \wedge superTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \exists \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg \exists \tau'' : \tau_2 = seq(\tau'') \\ & \wedge superType(\tau', \tau'') \\ & \wedge superTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg \exists \tau'' : \tau_2 = seq(\tau'') \\ & \wedge \neg(superType(\tau', \tau'') \\ & \wedge superTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \neg(\exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'')) \\ & \wedge superType(\tau', \tau'') \\ & \wedge superTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

canNegateType $\subset$ **Type** $\times$ **Type**

The predicate returns true (in most cases) if the former type can be logically negated from the latter type, i.e. former type can be logically subtracted from the latter (super) type.

$$canNegateType(integer, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \wedge integer \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(rational, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \wedge rational \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(float, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \wedge float \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(boolean, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \wedge boolean \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(string, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \wedge string \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(anything, \tau) \Leftrightarrow false$$

$$canNegateType(\{\tau\}, \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2, \tau seq : \tau_1 = Or(\tau seq) \\ & \wedge \{\tau_2\} \in range(\tau seq) \\ & \wedge canNegateType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(list(\tau), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2, \tau seq : \tau_1 = Or(\tau seq) \\ & \wedge list(\tau_2) \in range(\tau seq) \\ & \wedge canNegateType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType([\tau seq], \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq_1, \tau seq_2 : \tau = Or(\tau seq_1) \\ & \wedge [\tau seq_2] \in range(\tau seq_1) \\ & \wedge canNegateTypeSeq(\tau seq, \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(procedure[\tau](\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau_2, \tau seq_1, \tau seq_2 : \tau_1 = Or(\tau seq_1) \\ & \wedge procedure[\tau_2](\tau seq_2) \in range(\tau seq_1) \\ & \wedge canNegateType(\tau, \tau_2) \wedge canNegateTypeSeq(\tau seq, \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(I(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq_1, \tau seq_2 : \tau = Or(\tau seq_1) \\ & \wedge I(\tau seq_2) \in range(\tau seq_1) \\ & \wedge canNegateTypeSeq(\tau seq, \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(Or(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq_1 : \tau = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq_1) \\ & \wedge canNegateTypeSeq(\tau seq, \tau seq_1) \\ false & \text{if } hasTypeAnything(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(symbol, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \\ & \wedge symbol \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(void, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \\ & \wedge void \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(uneval, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \\ & \wedge uneval \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

$$canNegateType(I, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq : \tau = Or(\tau seq) \\ & \wedge I \in range(\tau seq) \\ false & \text{otherwise} \end{cases}$$

canNegateTypeSeq $\subset$ **TypeSequence** $\times$ **TypeSequence**

The predicate returns true (in most cases) if every type from the former sequence of types is general to the corresponding type in the latter sequence of types, i.e. the former type is a super type of the latter type.

$canNegateTypeSeq(empty, empty) \Leftrightarrow false$

$canNegateTypeSeq(empty, (\tau, \tau seq)) \Leftrightarrow true$

$canNegateTypeSeq((\tau, \tau seq), empty) \Leftrightarrow false$

$canNegateTypeSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) \Leftrightarrow$

$$\begin{cases} true & \text{if } \exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge canNegateType(\tau', \tau'') \\ & \wedge canNegateTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge \neg(canNegateType(\tau', \tau'') \\ & \wedge canNegateTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \exists \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg \exists \tau'' : \tau_2 = seq(\tau'') \\ & \wedge canNegateType(\tau', \tau'') \\ & \wedge canNegateTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg \exists \tau'' : \tau_2 = seq(\tau'') \\ & \wedge \neg(canNegateType(\tau', \tau'') \\ & \wedge canNegateTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \neg(\exists \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'')) \\ & \wedge canNegateType(\tau', \tau'') \\ & \wedge canNegateTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

andCombinable $\subset$ **Type** $\times$ **Type**

The predicate returns true (in most cases) if the former type can logically be intersected with the later type. Every type can be intersected with itself and with type anything.

$$andCombinable(integer, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{integer, rational, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(rational, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{rational, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(float, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{float, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(boolean, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{boolean, symbol, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(string, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{string, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(anything, \tau) \Leftrightarrow \{true$$

$$andCombinable(\{\tau\}, \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau' : \tau_1 = \{\tau'\} \\ & \wedge andCombinable(\tau, \tau') \\ true & \text{if } \tau_1 = anything \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(list(\tau), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau' : \tau_1 = list(\tau') \\ & \wedge andCombinable(\tau, \tau') \\ true & \text{if } \tau_1 = anything \\ false & \text{otherwise} \end{cases}$$

$$andCombinable([\tau], \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists \tau seq' : \tau_1 = [\tau seq'] \\ & \wedge andCombinableSeq(\tau seq, \tau seq') \\ true & \text{if } \tau_1 = anything \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(procedure[\tau](\tau seq), \tau_1) \Leftrightarrow$$
$$\begin{cases} true & \text{if } \exists \tau', \tau seq' : \tau_1 = procedure[\tau'](\tau seq') \\ & \wedge andCombinable(\tau, \tau') \\ & \wedge andCombinableSeq(\tau seq, \tau seq') \\ true & \text{if } \tau_1 = anything \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(I(\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists I', \tau seq' : \tau_1 = I'(\tau seq') \\ & \wedge I = I' \\ & \wedge andCombinableSeq(\tau seq, \tau seq') \\ true & \text{if } \tau_1 = anything \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(Or(\tau seq), \tau_1) \Leftrightarrow$$

$$\begin{cases} true & \text{if } (\tau_1 = anything \\ & \vee(\exists \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge hasTypeAnything(\tau seq_1)) \\ & \wedge hasTypeAnything(\tau seq) \\ true & \text{if } (\exists \tau seq_1 : \tau_1 = \tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq) \\ & \wedge \neg hasTypeAnything(\tau seq_1) \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ true & \text{if } \neg(\exists \tau seq_1 : \tau_1 = \tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq) \\ & \wedge \neg hasTypeAnything(\tau seq_1) \\ & \wedge andCombinableSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(symbol, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{symbol, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(void, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{void, anything\} \\ false & \text{otherwise} \end{cases}$$

$$andCombinable(uneval, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau \in \{uneval, anything\} \\ false & \text{otherwise} \end{cases}$$

andCombinableSeq $\subset$ **TypeSequence** $\times$ **TypeSequence**

The predicate returns true (in most cases) if every type from the former sequence of types can be logically intersected to the corresponding type in the latter sequence of types.

$andCombinableSeq(empty, empty) \Leftrightarrow true$

$andCombinableSeq((\tau, \tau seq), empty) \Leftrightarrow false$

$andCombinableSeq(empty, (\tau, \tau seq)) \Leftrightarrow false$

$$andCombinableSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) \Leftrightarrow \\ andCombinable(\tau_1, \tau_2) \wedge andCombinableSeq(\tau seq_1, \tau seq_2)$$

hasTypeAnything $\subset$ **TypeSequence**

The predicate returns true if any of the type is anything in the given sequence of types.

$hasTypeAnything(empty) \Leftrightarrow false$

$hasTypeAnything(\tau, \tau seq) \Leftrightarrow \tau = anything \vee hasTypeAnything(\tau seq)$

hasType $\subset$ **TypeSequence** $\times$ **Type**

The predicate returns true if the given type is equal to any of the type in the sequence of types.

$hasType(empty, \tau_1) \Leftrightarrow false$

$hasType((\tau, \tau seq), \tau_1) \Leftrightarrow \tau = \tau_1 \vee hasType(\tau seq, \tau_1)$

isTypeModule $\subset$ **TypeSequence**

The predicate returns true if any of the type is module in the given sequence of types.

$isTypeModule(empty) \Leftrightarrow true$

$$isTypeModule(\tau, \tau seq) \Leftrightarrow (\exists \pi \in TypeEnvironment : \tau = module(\pi)) \\ \wedge isTypeModule(\tau seq)$$

checkTypes $\subset$ **Type** $\times$ **set**(**Type**)

The predicate returns true if the declared return type in a procedure declaration doesn't conflict the set of return types appeared (as a return commands) in the body of the procedure.

$checkTypes(\tau, \tau set) \Leftrightarrow (\forall \tau_1 \in \tau set : superType(\tau, \tau_1))$

D.3 Predicates over Identifier

In this section we define the predicates over identifier.

isAssignable $\subset$ **IdentifierSequence** $\times$ **set**(**Identifier**)

The predicate returns true if all the identifiers in a sequence are in a set of assignable identifiers.

$isAssignable(empty, s) \Leftrightarrow true$

$isAssignable((I, Iseq), s) \Leftrightarrow I \in s \wedge isAssignable(Iseq, s)$

isNotRepeated $\subset$ **IdentifierSequence**

The predicate returns true if every identifier in the given sequence occurs only once.

$isNotRepeated(empty) \Leftrightarrow true$

$isNotRepeated(I, Iseq) \Leftrightarrow (I \notin IdSeqToSet(Iseq)) \wedge isNotRepeated(Iseq)$

checkExpIds $\subset$ **set**(**Identifier**) $\times$ **TypeEnvironment**

The predicate returns true if all the exported identifiers from the set have the corresponding defined type in the type environment. This type comes from the definition of the exported identifiers in the body of the module.

$checkExpIds(idset, \pi) \Leftrightarrow (\forall I \in idset : \exists \tau \in Type \wedge (I : \tau) \in \pi)$

checkSubs $\subset$ **Identifier** $\times$ **Expression**

The predicate returns true if an identifier has at least a single occurrence in a given expression.

$$checkSubs(Id, expr) \Leftrightarrow \begin{cases} true & \text{if } \exists Id \in expToIdSet(expr) \\ false & \text{otherwise} \end{cases}$$

D.4 Predicates over Typed Identifier

In this section we define the predicates over typed identifier.

isNotRepeatedTypedId $\subset$ **TypedIdentifierSequence**

The predicate returns true if every identifier in the given sequence of typed identifiers occurs only once.

$isNotRepeatedTypedId(empty) \Leftrightarrow true$

$$isNotRepeatedTypedId((It, Itseq)) \Leftrightarrow (TypedIdToId(It) \notin TypedIdSeqToSet(Itseq)) \wedge isNotRepeatedTypedId(Itseq)$$

D.5 Predicates over Parameter

In this section we define the predicates over parameter.

isNotRepeatedParam $\subset$ **ParameterSequence**

The predicate returns true if every identifier in the given sequence of parameters occurs only once.

$isNotRepeatedParam(empty) \Leftrightarrow true$

$$isNotRepeated((P, Pseq)) \Leftrightarrow (ParamToId(P) \notin ParamSeqToSet(Pseq)) \wedge isNotRepeatedParam(Pseq)$$

In the next section we will discuss the output of the type checker.

E Output

In this section we show the complete output of the type checker for the example given in Section 4.

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

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



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

#comm#
#asgncomm#
#expression#
#idexp#
prod
:=
#expression#
#procedureexp#
proc(#parameterseq#
#parameter#
#expression#
#idexp#
1

#type-modifier#
::#type#
#extended-type#
#list-type#
list(#type#
#extended-type#
```

```
#or-type#
Or(#typeseq#
#type#
#primitive-type#
#integer-type#
integer
*************TYPE-ANNOTATION BEGIN*************
integer
*************TYPE-ANNOTATION END***************

,
#type#
#primitive-type#
#float-type#
float
*************TYPE-ANNOTATION BEGIN*************
float
*************TYPE-ANNOTATION END***************



*************TYPE-SEQUENCE-ANNOTATION BEGIN*************
integer,float
*************TYPE-SEQUENCE-ANNOTATION END***************
)
*************TYPE-ANNOTATION BEGIN*************
Or(integer,float)
*************TYPE-ANNOTATION END***************

)
*************TYPE-ANNOTATION BEGIN*************
list(Or(integer,float))
*************TYPE-ANNOTATION END***************


*************MODIFIER-ANNOTATION BEGIN*************
list(Or(integer,float))
*************MODIFIER-ANNOTATION END***************


*************PARAMETER-ANNOTATION BEGIN*************
PI -> [
l:list(Or(integer,float))
]
*************PARAMETER-ANNOTATION END***************



*************PARAMETER-SEQUENCE-ANNOTATION BEGIN*************
PI -> [
l:list(Or(integer,float))
]
*************PARAMETER-SEQUENCE-ANNOTATION END***************
)
::#type#
#extended-type#
```

```
#record-type#
[#typeseq#
#type#
#primitive-type#
#integer-type#
integer
*************TYPE-ANNOTATION BEGIN*************
integer
*************TYPE-ANNOTATION END***************

,
#type#
#primitive-type#
#float-type#
float
*************TYPE-ANNOTATION BEGIN*************
float
*************TYPE-ANNOTATION END***************



*************TYPE-SEQUENCE-ANNOTATION BEGIN*************
integer,float
*************TYPE-SEQUENCE-ANNOTATION END***************
]
*************TYPE-ANNOTATION BEGIN*************
[integer,float]
*************TYPE-ANNOTATION END***************

#globalseq#
global
#expression#
#idexp#
status


*************IDENTIFIER-ANNOTATION BEGIN*************
integer
*************IDENTIFIER-ANNOTATION END***************
;
*************|GLOBAL| SEQUENCE-ANNOTATION BEGIN*************
PI -> [
status:anything
]
AsgnIDSet -> {status}
ExpIDSet -> {}
*************|GLOBAL| SEQUENCE-ANNOTATION END***************

#localseq#
local
#idtyped#
#expression#
#idexp#
i

*************IDTYPED-ANNOTATION START*************
```

```
Type -> symbol
AsgnIDSet -> {i}
*************IDTYPED-ANNOTATION END***************

#identifiertypedseq#
#idtyped#
#expression#
#idexp#
x
::
#type#
#extended-type#
#or-type#
Or(#typeseq#
#type#
#primitive-type#
#integer-type#
integer
*************TYPE-ANNOTATION BEGIN*************
integer
*************TYPE-ANNOTATION END***************

,
#type#
#primitive-type#
#float-type#
float
*************TYPE-ANNOTATION BEGIN*************
float
*************TYPE-ANNOTATION END***************



*************TYPE-SEQUENCE-ANNOTATION BEGIN*************
integer,float
*************TYPE-SEQUENCE-ANNOTATION END***************
)
*************TYPE-ANNOTATION BEGIN*************
Or(integer,float)
*************TYPE-ANNOTATION END***************


*************IDTYPED-ANNOTATION START*************

Type -> Or(integer,float)
AsgnIDSet -> {x}
*************IDTYPED-ANNOTATION END***************

,
#idtyped#
#expression#
#idexp#
si
::
#type#
```

```
#primitive-type#
#integer-type#
integer
*************TYPE-ANNOTATION BEGIN*************
integer
*************TYPE-ANNOTATION END**************

:=
#expression#
#numexp#
1

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END**************

*************IDTYPED-ANNOTATION START*************

Type -> integer
AsgnIDSet -> {si}
*************IDTYPED-ANNOTATION END**************

,
#idtyped#
#expression#
#idexp#
sf
::
#type#
#primitive-type#
#float-type#
float
*************TYPE-ANNOTATION BEGIN*************
float
*************TYPE-ANNOTATION END**************

:=
#expression#
#floatexp#
#expression#
#numexp#
1

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END**************
.#expression#
#numexp#
0

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END**************
```

```
*************|FLOAT| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
*************|FLOAT| EXPRESSION-ANNOTATION END**************

*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END**************

*************IDTYPED-ANNOTATION START*************

Type -> float
AsgnIDSet -> {sf}
*************IDTYPED-ANNOTATION END**************



*************IDTYPED-SEQUENCE-ANNOTATION START*************

Types -> [Or(integer,float),integer,float]
AsgnIDSet -> {si,x,sf,status,i}
*************IDTYPED-SEQUENCE-ANNOTATION END**************
;

*************|LOCAL| SEQUENCE-ANNOTATION BEGIN*************
PI -> [
i:symbol
x:Or(integer,float)
sf:float
si:integer
]
AsgnIDSet -> {si,x,sf,i}
ExpIDSet -> {}
*************|LOCAL| SEQUENCE-ANNOTATION END**************
#recc#
#commseq#
#comm#
#forloopcomm#
for
#expression#
#idexp#
i


*************IDENTIFIER-ANNOTATION BEGIN*************
symbol
*************IDENTIFIER-ANNOTATION END**************

from
#expression#
#numexp#
1
```

```
************|INTEGER| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************
by
#expression#
#numexp#
1

************|INTEGER| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************
to
#expression#
#special-expression#
#nops-expression#
nops(
#expression#
#idexp#
l


************IDENTIFIER-ANNOTATION BEGIN************
list(Or(integer,float))
************IDENTIFIER-ANNOTATION END**************
)
************|NOPS-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> integer
************|NOPS-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION END**************
do
#commseq#
#comm#
#asgncomm#
#expression#
#idexp#
x


************IDENTIFIER-ANNOTATION BEGIN************
Or(integer,float)
************IDENTIFIER-ANNOTATION END**************
:=
#expression#
#special-expression#
;
LOCAL CONTEXT FOR ASSIGNMNET COMMAND


************|ASSIGNMENT| COMMAND-ANNOTATION START************
PI -> [
x:Or(integer,float)
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
```

```
RetFlag -> not_aret
************|ASSIGNMENT| COMMAND-ANNOTATION END**************

#comm#
#asgncomm#
#expression#
#idexp#
status


************IDENTIFIER-ANNOTATION BEGIN*************
anything
************IDENTIFIER-ANNOTATION END**************
:=
#expression#
#idexp#
i


************IDENTIFIER-ANNOTATION BEGIN*************
integer
************IDENTIFIER-ANNOTATION END**************
;
LOCAL CONTEXT FOR ASSIGNMNET COMMAND


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

#comm#
#conditionalcomm#
if
#expression#
#type-expression#
type(#expression#
#idexp#
x


************IDENTIFIER-ANNOTATION BEGIN*************
Or(integer,float)
************IDENTIFIER-ANNOTATION END**************
,#type#
#primitive-type#
#integer-type#
integer
************TYPE-ANNOTATION BEGIN*************
integer
************TYPE-ANNOTATION END**************
```

```
)
*************|TYPE-EXP| EXPRESSION-ANNOTATION BEGIN*************
PI -> [
x:integer
]
Type -> boolean
*************|TYPE-EXP| EXPRESSION-ANNOTATION END***************

then
#commseq#
#comm#
#conditionalcomm#
if
#expression#
#parenthesized-expression#
(#expression#
#binary-expression#
#equal-expression#
#expression#
#idexp#
x


*************IDENTIFIER-ANNOTATION BEGIN*************
integer
*************IDENTIFIER-ANNOTATION END***************

=
#expression#
#numexp#
0

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END***************

*************|EQUAL| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> boolean
*************|EQUAL| EXPRESSION-ANNOTATION END***************
)
then
#commseq#
#comm#
#returncomm#
return
#expression#
#special-expression#
#list-expression#
[#expressionseq#
#expression#
#idexp#
si
```

```
**********IDENTIFIER-ANNOTATION BEGIN**********
integer
**********IDENTIFIER-ANNOTATION END**********
,
#expression#
#idexp#
sf


**********IDENTIFIER-ANNOTATION BEGIN**********
float
**********IDENTIFIER-ANNOTATION END**********


**********EXPRESSION-SEQUENCE-ANNOTATION BEGIN**********
Types -> [integer,float]
**********EXPRESSION-SEQUENCE-ANNOTATION END**********
]
**********|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> [integer,float]
**********|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION END**********

**********|RETURN| COMMAND-ANNOTATION START**********
PI -> [
i:integer
status:integer
sf:float
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
**********|RETURN| COMMAND-ANNOTATION END**********
;

**********COMMAND-SEQUENCE-ANNOTATION START**********
PI -> [
i:integer
status:integer
sf:float
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
**********COMMAND-SEQUENCE-ANNOTATION END**********

end if;
**********|CONDITIONAL| COMMAND-ANNOTATION START**********
PI -> [
```

```
i:integer
sf:float
status:integer
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
************|CONDITIONAL| COMMAND-ANNOTATION END**************

#comm#
#asgncomm#
#expression#
#idexp#
si


************IDENTIFIER-ANNOTATION BEGIN*************
integer
************IDENTIFIER-ANNOTATION END**************
:=
#expression#
#binary-expression#
#times-expression#
#expression#
#idexp#
si


************IDENTIFIER-ANNOTATION BEGIN*************
integer
************IDENTIFIER-ANNOTATION END**************

*
#expression#
#idexp#
x


************IDENTIFIER-ANNOTATION BEGIN*************
integer
************IDENTIFIER-ANNOTATION END**************

************|TIMES| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
************|TIMES| EXPRESSION-ANNOTATION END**************
;
LOCAL CONTEXT FOR ASSIGNMNET COMMAND


************|ASSIGNMENT| COMMAND-ANNOTATION START*************
PI -> [
si:integer
```

```
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|ASSIGNMENT| COMMAND-ANNOTATION END***************


*************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
i:integer
status:integer
sf:float
l:list(Or(integer,float))
si:integer
x:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************COMMAND-SEQUENCE-ANNOTATION END***************

#elif#
elif
#expression#
#type-expression#
type(#expression#
#idexp#
x


*************IDENTIFIER-ANNOTATION BEGIN*************
Or(integer,float)
*************IDENTIFIER-ANNOTATION END***************
,#type#
#primitive-type#
#float-type#
float
*************TYPE-ANNOTATION BEGIN*************
float
*************TYPE-ANNOTATION END***************


)
*************|TYPE-EXP| EXPRESSION-ANNOTATION BEGIN*************
PI -> [
x:float
]
Type -> boolean
*************|TYPE-EXP| EXPRESSION-ANNOTATION END***************
then
#commseq#
#comm#
#conditionalcomm#
if
#expression#
#parenthesized-expression#
```

```
(#expression#
#binary-expression#
#less-expression#
#expression#
#idexp#
x


**********IDENTIFIER-ANNOTATION BEGIN**********
float
**********IDENTIFIER-ANNOTATION END**********

<
#expression#
#floatexp#
#expression#
#numexp#
0

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********
.#expression#
#numexp#
5

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********


**********|FLOAT| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> float
**********|FLOAT| EXPRESSION-ANNOTATION END**********

**********|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> float
**********|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END**********

**********|LESS| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> boolean
**********|LESS| EXPRESSION-ANNOTATION END**********
)
then
#commseq#
#comm#
#returncomm#
return
#expression#
#special-expression#
#list-expression#
```

```
[#expressionseq#
#expression#
#idexp#
si


************IDENTIFIER-ANNOTATION BEGIN************
integer
************IDENTIFIER-ANNOTATION END**************
,
#expression#
#idexp#
sf


************IDENTIFIER-ANNOTATION BEGIN************
float
************IDENTIFIER-ANNOTATION END**************


************EXPRESSION-SEQUENCE-ANNOTATION BEGIN************
Types -> [integer,float]
************EXPRESSION-SEQUENCE-ANNOTATION END**************
]
************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> [integer,float]
************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION END**************

************|RETURN| COMMAND-ANNOTATION START************
PI -> [
i:integer
sf:float
status:integer
l:list(Or(integer,float))
si:integer
x:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
************|RETURN| COMMAND-ANNOTATION END**************
;

************COMMAND-SEQUENCE-ANNOTATION START************
PI -> [
i:integer
sf:float
status:integer
l:list(Or(integer,float))
si:integer
x:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
```

```
************COMMAND-SEQUENCE-ANNOTATION END**************

end if;
************|CONDITIONAL| COMMAND-ANNOTATION START************
PI -> [
i:integer
status:integer
sf:float
l:list(Or(integer,float))
si:integer
x:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
************|CONDITIONAL| COMMAND-ANNOTATION END**************

#comm#
#asgncomm#
#expression#
#idexp#
sf


************IDENTIFIER-ANNOTATION BEGIN************
float
************IDENTIFIER-ANNOTATION END**************
:=
#expression#
#binary-expression#
#times-expression#
#expression#
#idexp#
sf


************IDENTIFIER-ANNOTATION BEGIN************
float
************IDENTIFIER-ANNOTATION END**************

*
#expression#
#idexp#
x


************IDENTIFIER-ANNOTATION BEGIN************
float
************IDENTIFIER-ANNOTATION END**************

************|TIMES| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> float
************|TIMES| EXPRESSION-ANNOTATION END**************
;
LOCAL CONTEXT FOR ASSIGNMNET COMMAND
```

```
*************|ASSIGNMENT| COMMAND-ANNOTATION START*************
PI -> [
sf:float
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|ASSIGNMENT| COMMAND-ANNOTATION END***************


*************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
i:integer
sf:float
status:integer
l:list(Or(integer,float))
si:integer
x:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************COMMAND-SEQUENCE-ANNOTATION END***************


*************|ELIF|-ANNOTATION START*************
PI -> [
i:integer
status:integer
l:list(Or(integer,float))
si:integer
x:float
sf:float
]
 PI-Set -> [PI -> [
x:integer
]]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************|ELIF|-ANNOTATION END***************

end if;
*************|CONDITIONAL| COMMAND-ANNOTATION START*************
PI -> [
i:integer
status:integer
l:list(Or(integer,float))
si:integer
x:Or(integer,float)
sf:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
```

```
RetFlag -> aret
************|CONDITIONAL| COMMAND-ANNOTATION END**************


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

end do;

************|FOR-LOOP| COMMAND-ANNOTATION START************
PI -> [
i:symbol
x:Or(integer,float)
sf:float
l:list(Or(integer,float))
si:integer
status:anything
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
************|FOR-LOOP| COMMAND-ANNOTATION END**************
#comm#
#asgncomm#
#expression#
#idexp#
status


************IDENTIFIER-ANNOTATION BEGIN************
anything
************IDENTIFIER-ANNOTATION END**************
:=
#expression#
#unary-expression#
#minus-expression#
-
#expression#
#numexp#
1

************|INTEGER| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************

************|MINUS| EXPRESSION-ANNOTATION BEGIN************
PI -> []
```

```
Type -> integer
*************|MINUS| EXPRESSION-ANNOTATION END***************
;
GLOBAL CONTEXT FOR ASSIGNMENT-COMMAND



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

#comm#
#returncomm#
return
#expression#
#special-expression#
#list-expression#
[#expressionseq#
#expression#
#idexp#
si


*************IDENTIFIER-ANNOTATION BEGIN*************
integer
*************IDENTIFIER-ANNOTATION END***************
,
#expression#
#idexp#
sf


*************IDENTIFIER-ANNOTATION BEGIN*************
float
*************IDENTIFIER-ANNOTATION END***************


*************EXPRESSION-SEQUENCE-ANNOTATION BEGIN*************
Types -> [integer,float]
*************EXPRESSION-SEQUENCE-ANNOTATION END***************
]
*************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> [integer,float]
*************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION END***************

*************|RETURN| COMMAND-ANNOTATION START*************
PI -> [
i:symbol
status:integer
x:Or(integer,float)
```

```
sf:float
l:list(Or(integer,float))
si:integer
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************|RETURN| COMMAND-ANNOTATION END***************
;

*************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
i:symbol
status:integer
x:Or(integer,float)
l:list(Or(integer,float))
si:integer
sf:float
]
RetTypeSet -> {[integer,float]}
ThrownExceptionSet -> {}
RetFlag -> aret
*************COMMAND-SEQUENCE-ANNOTATION END***************


end proc;
*************|PROCEDURE| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> procedure[[integer,float]](list(Or(integer,float)))
*************|PROCEDURE| EXPRESSION-ANNOTATION END***************


;
GLOBAL CONTEXT FOR ASSIGNMENT-COMMAND



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

#comm#
#asgncomm#
#expression#
#idexp#
result
:=
#expression#
#call-expression#
#expression#
#idexp#
```

```
prod


************IDENTIFIER-ANNOTATION BEGIN*************
procedure[[integer,float]](list(Or(integer,float)))
************IDENTIFIER-ANNOTATION END**************

(
#expressionseq#
#expression#
#special-expression#
#list-expression#
[#expressionseq#
#expression#
#numexp#
1

************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************
,
#expression#
#floatexp#
#expression#
#numexp#
8

************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************
.#expression#
#numexp#
54

************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
************|INTEGER| EXPRESSION-ANNOTATION END**************


************|FLOAT| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
************|FLOAT| EXPRESSION-ANNOTATION END**************

************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END**************
,
#expression#
#floatexp#
#expression#
#numexp#
```

```
34

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********
.#expression#
#numexp#
4

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********


**********|FLOAT| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> float
**********|FLOAT| EXPRESSION-ANNOTATION END**********

**********|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> float
**********|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END**********
,
#expression#
#numexp#
6

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********
,
#expression#
#floatexp#
#expression#
#numexp#
8

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********
.#expression#
#numexp#
1

**********|INTEGER| EXPRESSION-ANNOTATION BEGIN**********
PI -> []
Type -> integer
**********|INTEGER| EXPRESSION-ANNOTATION END**********


**********|FLOAT| EXPRESSION-ANNOTATION BEGIN**********
```

```
PI -> []
Type -> float
*************|FLOAT| EXPRESSION-ANNOTATION END***************

*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END***************
,
#expression#
#numexp#
10

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END***************
,
#expression#
#numexp#
12

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END***************
,
#expression#
#floatexp#
#expression#
#numexp#
5

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END***************
.#expression#
#numexp#
4

*************|INTEGER| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> integer
*************|INTEGER| EXPRESSION-ANNOTATION END***************


*************|FLOAT| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
*************|FLOAT| EXPRESSION-ANNOTATION END***************

*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION BEGIN*************
PI -> []
Type -> float
*************|FLOAT-EXPRESSION| EXPRESSION-ANNOTATION END***************
```

```
************EXPRESSION-SEQUENCE-ANNOTATION BEGIN************
Types -> [integer,float,float,integer,float,integer,integer,float]
************EXPRESSION-SEQUENCE-ANNOTATION END**************
]
************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> [integer,float,float,integer,float,integer,integer,float]
************|RECORD-SPECIAL-EXPRESSION| EXPRESSION-ANNOTATION END**************


************EXPRESSION-SEQUENCE-ANNOTATION BEGIN************
Types -> [[integer,float,float,integer,float,integer,integer,float]]
************EXPRESSION-SEQUENCE-ANNOTATION END**************
)
************|PROCEDURE-CALL| EXPRESSION-ANNOTATION BEGIN************
PI -> []
Type -> [integer,float]
************|PROCEDURE-CALL| EXPRESSION-ANNOTATION END**************
;
GLOBAL CONTEXT FOR ASSIGNMENT-COMMAND



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

#comm#

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

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