The DeCCo project papers I

Z Specification of Pasp

Susan Stepney and Ian T. Nabney

University of York Technical Report YCS-2002-358

June 2003

Contents

Preface

Historical background of the DeCCo project

In 1990 Logica's Formal Methods Team performed a study for RSRE (now QinetiQ) into how to develop a compiler for high integrity applications that is itself of high integrity. In that study, the source language was Spark, a subset of Ada designed for safety critical applications, and the target was Viper, a high integrity processor. Logica' Formal Methods Team developed a mathematical technique for specifying a compiler and proving it correct, and developed a small proof of concept prototype. The study is described in [Stepney *et al.* 1991], and the small case study is worked up in full, including all the proofs, in [Stepney 1993]. Experience of using the PVS tool to prove the small case study is reported in [Stringer-Calvert *et al.* 1997]. Futher developments to the method to allow separate compilation are described in [Stepney 1998].

Engineers at AWE read about the study and realised the technique could be used to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). They contacted Logica, and between 1992 and 2001 Logica used these techniques to deliver a high integrity compiler, integrated in a development and test environment, for progressively larger subsets of Pascal.

The full specifications of the final version of the DeCCo compiler are reproduced in these technical reports. These are written in the Z specification language. The variant of Z used is that supported by the *Z Specific Formaliser* tool [Formaliser], which was used to prepare and type-check all the DeCCo specifications. This variant is essentially the Z described in the *Z Reference Manual* [Spivey 1992] augmented with a few new constructs from *ISO Standard Z* [ISO-Z]. Additions to ZRM are noted as they occur in the text.

The DeCCo Reports

The DeCCo Project case study is detailed in the following technical reports (this preface is common to all the reports):

I. **Z Specification of Pasp**
The denotational semantics of the high level source language, Pasp. The definition is split into several static semantics (such as type checking) and a dynamic semantics (the meaningof executing a program). Later smeantics are not defined for those programs where the result of earlier semantics is error.

II. **Z Specification of Asp, AspAL and XAspAL**
The denotational semantics of the low level target assembly languages. XAspAL is the target of compilation of an individual Pasp module; it is AspAL extended with some cross-module instructions that are resolved at link time. The meaning of these extra instructions is given implicitly by the specification of the linker and hexer. AspAL is the target of linking a set of XAspAL modules, and also the target of compilation of a complete Pasp program. Asp is the non-relocatable assembly language of the chip, with AspAL's labels replaced by absolute program addresses. The semantics of programs with errors is not defined, because these defintions will only ever be used to define the meaning of correct, compiled programs.

III. **Z Specification of Compiler Templates**
The operational semantics of the Pasp source language, in the form of a set of XAspAL target language templates.

IV. **Z Specification of Linker and Hexer**
The linker combines compiled XAspAL modules into a single compiled AspAL program. The hexer converts a relocatable AspAL program into an Asp program located at a fixed place in memory.

V. **Compiler Correctness Proofs**
The compiler's operational semantics are demonstrated to be equivalent to the source language's denotational semantics, by calculating the meaning of each Pasp construct, and the corresponding meaning

of the AspAL template, and showing them to be equivalent. Thus the compiler transformation is *meaning preserving*, and hence the compiler is correct.

VI. **Z to Prolog DCTG translation guidelines**
The Z specifications of the Pasp semantics and compiler templates are translated into an executable Prolog DCTG implementation of a Pasp interpreter and Pasp-to-Asp compiler. The translation is done manually, following the stated guidelines.

Acknowledgements

We would like to thank the client team at AWE – Dave Thomas, Wilson Ifill, Alun Lewis, Tracy Bourne – for providing such an interesting development project to work on.

We would like to thank the rest of the development team at Logica: Tim Wentford, John Taylor, Roger Eatwell, Kwasi Ametewee.

1 Introduction

This document introduces the abstract syntax, concrete syntax, and various semantics of Pasp, the Pascal-like language used on the DeCCo project.

2 Abstract syntax

Pasp's abstract syntax is defined in this chapter, and is used as the basis of the semantics definitions. Its concrete syntax is specified in the next chapter, as a mapping from abstract syntax to the corresponding strings of characters.

2.1 Identifiers

Abstract identifiers are used to represent names of constants, types, variables, procedures, functions and modules. They are introduced as a given set, because their internal structure is important.

$[ID]$

2.2 Values

A Pasp expression denote a *value*. Pasp values are unsigneds, bytes, enumerated values, booleans, and abstract locations. (Pasp has array variables, but an expression cannot denote a whole array, only an element of an array.)

2.2.1 Unsigned

Pasp's unsigned values are 16-bit numbers.

$$UNSIGNED == 0 \ldots (2 \uparrow 16 - 1)$$

2.2.2 Byte

The byte type is defined to take advantage of the greater speed offered by using single register operations in the compiled code. As such, its definition is guided by the capabilities of the Asp chip.

The byte is an unsigned type, and allows efficient accessing of arrays. Arrays indexed by byte variables may have at most 256 elements.

*BYTE*s are defined in the appendix.

2.2.3 Enumerated value

The values of enumerated types are modelled by pairs of bytes: the first being the maximum value an element of this type is allowed[1], the second being the current value[2].

2.2.4 Boolean

BOOLEAN is defined in the appendix.

2.2.5 Streams

Input and output are modelled as streams: sequences of values. They are used in the dynamic semantics, but have no concrete form, and so are not available directly to the Pasp programmer.

2.2.6 Location

Pasp abstract values include *abstract* memory locations. These are used to model call-by-reference parameters to functions and procedures, memory mapped i/o, and array offsets. (An abstract location may be mapped to more than one physical data addresses.)

$$LOCN == \mathbb{N}$$

[1]The maximum value allowed is used only in the dynamic check of *succ* overflow, which check is not implemented in the compiler. So the compiler implementation of enumerated values takes only one byte.

[2]This means that an enumerated type can have at most 256 values, more than any programmer is likely to need.

2.2.7 Name

Pasp abstract values include identifiers, used only in initialisation values, as enumerated values, or constants.

2.2.8 Value

$VALUE$ is the disjoint union of the different primitive values.

$$\begin{aligned} VALUE ::= {} & vunsgn\langle\langle UNSIGNED\rangle\rangle \mid vbyte\langle\langle BYTE\rangle\rangle \\ & \mid venum\langle\langle BYTE \times BYTE\rangle\rangle \mid vbool\langle\langle BOOLEAN\rangle\rangle \\ & \mid stream\langle\langle \text{seq } VALUE\rangle\rangle \mid pointer\langle\langle LOCN\rangle\rangle \\ & \mid name\langle\langle ID\rangle\rangle \end{aligned}$$

2.2.9 Converting Pasp values to numbers

It is convenient to have a function to convert a numerical $VALUE$ into its natural number equivalent.

$$\begin{array}{|l} number : VALUE \pfun \mathbb{N} \\ \hline number = \\ \quad vunsgn^\sim \\ \quad \cup\, vbyte^\sim \\ \quad \cup \{\ v : \operatorname{ran} venum \bullet v \mapsto (venum^\sim\ v).2\ \} \end{array}$$

2.3 Types

When variables are declared, they are given a type, which restricts the kinds of values they can hold: unsigned (integer), (unsigned) byte, subrange type, enumerated value, or boolean.

2.3.1 Subrange types

A bound is a constant literal number, or an enumerated value, or a named constant. (Syntactically, the bounds are required only to be values; checking of their types is carried out in the type checking semantics.)

$$BOUND == VALUE$$

A subrange consists of a lower bound and an upper bound.

$$Subrange \mathrel{\widehat=} [\ lb, ub : BOUND\ ]$$

The values of the subrange bounds are determined in the type checking semantics.

2.3.2 Enumerated types

Enumerated types are defined with a type definition (they may not be defined implicitly in a variable declaration as in Pascal). So an enumerated type comprises the type name.

2.3.3 Scoped types

The existence of user defined types means that an identifier may represent a type. Types are scoped: they may be defined at the outermost level of a module, or within a procedure or function definition. Two types are the same if and only if they are defined in the same place. Thus, for the purposes of type checking, the name of the block in which a type is defined is added to the type name. This cannot be done in the parser: a type name is converted to a scoped type name in the type checking of type definitions (see section 9.3.2).

$$ScopedType \mathrel{\widehat=} [\ b, \xi : ID\ ]$$

2.3.4 Other internal types

A dummy type, *typeWrong*, is needed for the purposes of specification: it is used only in the static type checking semantics, and so is described there.

Two other types, *indirectAddr* and *returnAddr*, are needed for the purposes of demonstrating the compiler's correctness.

2.3.5 Type

TYPE is the disjoint union of these types.

$$\begin{aligned} TYPE ::= & \; unsigned \mid pbyte \mid boolean \mid subrange\langle\langle Subrange\rangle\rangle \\ & \mid typeName\langle\langle ID\rangle\rangle \mid scopedTypeName\langle\langle ScopedType\rangle\rangle \\ & \mid typeValue\langle\langle ID\rangle\rangle \mid scopedTypeValue\langle\langle ScopedType\rangle\rangle \\ & \mid typeWrong \mid indirectAddr \mid returnAddr \end{aligned}$$

Note: the Asp specification uses the keyword *byte* in the Asp value free type definition, so the name cannot also be used here.

2.4 Operators

Pasp has two kinds of operator: unary (that take one argument and return a result), and binary (that take two arguments and return a result).

2.4.1 Unary operators

Unsigned, byte and boolean values may be inverted. Unsigned and byte values may be shifted, left or right. There are various casting operators: converting between bytes and unsigneds, between bytes and booleans, and converting an enumerated type to its equivalent byte value. Since enumerated types have ordinal values, there are defined successor and predecessor operators.

The complete list of unary operators is:

$$
\begin{aligned}
UNY\_OP ::= & \ unot \mid bnot \mid not \mid bleft \mid bright \mid uleft \mid uright \\
& \mid boolToByte \mid byteToBool \mid byteToUnsgn \mid unsgnToByte \\
& \mid loByte \mid hiByte \mid ord \mid pred \mid succ
\end{aligned}
$$

2.4.2 Binary operators

The binary arithmetic operators comprise the four basic arithmetic operations together with the modulus operator, in unsigned and byte versions. The binary comparison operators are the usual tests for equality, inequality, and relative magnitude, again divided into unsigned and byte; in addition equality and inequality operators are defined for enumerated types. The binary logic operators take two unsigned, byte or boolean arguments and return a value of the same type. There are two casting operators: *join* joins two bytes into an unsigned; *byteToEnum* converts the given byte to the given enumerated type.

The complete list of binary operators is:

$$
\begin{aligned}
BIN\_OP ::= & \ uplus \mid uminus \mid umul \mid udiv \mid umod \\
& \mid bplus \mid bminus \mid bmul \mid bdiv \mid bmod \\
& \mid ueq \mid une \mid ult \mid ule \mid ugt \mid uge \\
& \mid beq \mid bne \mid blt \mid ble \mid bgt \mid bge \\
& \mid eeq \mid ene \mid uand \mid uor \mid uxor \mid band \mid bor \mid bxor \mid and \mid or \\
& \mid join \mid byteToEnum
\end{aligned}
$$

2.5 Expressions

A Pasp expression denotes a value. The expressions are: a constant value, a reference to a named constant or a variable array, a unary expression, a binary expression, or a function call.

$$[EXPR]$$

$$ValueRefExpr \mathrel{\widehat=} [\ \xi : ID;\ E : \mathrm{seq}\, EXPR\ ]$$
$$UnyExpr \mathrel{\widehat=} [\ \Psi : UNY\_OP;\ \epsilon : EXPR\ ]$$
$$BinExpr \mathrel{\widehat=} [\ \Omega : BIN\_OP;\ \epsilon 1, \epsilon 2 : EXPR\ ]$$
$$FunCallExpr \mathrel{\widehat=} [\ \xi : ID;\ E : \mathrm{seq}\, EXPR\ ]$$

$$\begin{aligned} EXPR ::= {} & constant\langle\langle VALUE\rangle\rangle \mid valueRef\langle\langle ValueRefExpr\rangle\rangle \\ & \mid unyExpr\langle\langle UnyExpr\rangle\rangle \mid binExpr\langle\langle BinExpr\rangle\rangle \\ & \mid funCall\langle\langle FunCallExpr\rangle\rangle \end{aligned}$$

A value reference is either a named constant, a variable, or an array element referenced by an expression sequence. These are combined into one term. A value reference with a zero-length sequence of expressions denotes either a variable or a named constant. These alternatives are not distinguished at the level of the abstract syntax, but are separated during the static semantic checks. (These checks take on some of the roles of the syntactic checks in a conventional compiler.)

2.6 Statements

A Pasp statement is: a skip, a block (a non-empty sequence of statements), an assignment, an if statement, a case statement, a while statement, or a procedure call. These expressions are all self explanatory, with the exception of the case statement.

2.6.1 Case statements

A case statement consists of an expression on which the switch is made, a non-empty sequence of branches, and an abstract identifier (which does not appear in the concrete language).

A branch consists of a non-empty sequence of enumerated values, and the statement to be executed if the expression is equal to one of the values.

$$[STMT]$$

$$Branch \mathrel{\widehat=} [\ \Xi : \mathrm{seq}_1\ ID;\ \gamma : STMT\ ]$$

Case statements are allowed only with enumerated type expressions. To make the proof simpler, a case statement is transformed to an equivalent set of nested 'if then else' statements; a temporary abstract variable, given by the *IDENTIFIER*, is used to store the value of the expression.

2.6.2 Statement

STMT is the disjoint union of the different statement types.

$$AssignStmt \mathrel{\widehat=} [\ \xi : ID;\ E : \mathrm{seq}\ EXPR;\ \epsilon : EXPR\ ]$$
$$IfStmt \mathrel{\widehat=} [\ \epsilon : EXPR;\ \gamma 1, \gamma 2 : STMT\ ]$$
$$CaseStmt \mathrel{\widehat=} [\ \epsilon : EXPR;\ K : \mathrm{seq}\ Branch;\ \xi : ID\ ]$$
$$WhileStmt \mathrel{\widehat=} [\ \epsilon : EXPR;\ \gamma : STMT\ ]$$
$$ProcCallStmt \mathrel{\widehat=} [\ \xi : ID;\ E : \mathrm{seq}\ EXPR\ ]$$

$$\begin{aligned} STMT ::= {} & skip \mid block\langle\!\langle \mathrm{seq}_1\ STMT \rangle\!\rangle \mid assign\langle\!\langle AssignStmt \rangle\!\rangle \\ & \mid ifStmt\langle\!\langle IfStmt \rangle\!\rangle \mid caseStmt\langle\!\langle CaseStmt \rangle\!\rangle \\ & \mid whileStmt\langle\!\langle WhileStmt \rangle\!\rangle \mid procCall\langle\!\langle ProcCallStmt \rangle\!\rangle \end{aligned}$$

2.7 Simple Declarations

There are five principal kinds of declaration: three simple declarations (named constants, variables – including arrays, and types – currently enumerated types only); and two command declarations (procedures and functions), defined in the following section.

2.7.1 Named constant declarations

A named constant declaration consists of its name and its value.

$$ConstDecl \mathrel{\widehat=} [\ \xi : ID;\ \kappa : VALUE\ ]$$

2.7.2 Type declarations

User defined types must be introduced in a type declaration (unlike in Pascal, they cannot be defined implicitly in a variable declaration). Currently the only variety of user defined type is the enumerated type.

An enumerated type declaration consists of the type name (this is the identifier used in the *typeName* branch of $TYPE$) and a non-empty sequence of component names. The order of names in the sequence is significant.

$$EnumDecl \mathrel{\widehat=} [\ \xi : ID;\ \Xi : \mathrm{seq}_1\ ID\ ]$$

We use a Z free type definition for the Pasp type definition, to allow for possible future user defined types (such as records).

$$TYPE\_DEF ::= enumDecl\langle\langle EnumDecl\rangle\rangle$$

2.7.3 Variable declarations

Variables can have attributes: they can be declared as *readOnly* inputs or *writeOnly* outputs, as *nvram* variables, and they can be placed *at* a particular physical data address.

A Pasp program consists of a sequence of modules, which are compiled, and it is not until linking that the complete program is placed at a particular program address. So, at the compilation stage we do not need to know the program address.

The data address attribute has no semantics in Pasp itself, but it is required by the compiler, so is included in the abstract syntax.

$$ADDR == \mathbb{N}$$

$$ATTR ::= dataAt\langle\langle ADDR\rangle\rangle \mid readOnly \mid writeOnly \mid nvram$$

The type checking semantics checks that no variable is declared to be more than one of *readOnly*, *writeOnly*, and *nvram*.

It would also be advisable to check that there is no conflict in the allocation of variables to data addresses; this could be part of a check during compilation where addresses are assigned to values, but has not been included in the scope of this project.

Abstractly, all variables are considered to be arrays (simple variables being arrays of zero dimension) so the range of allowable indices must also be defined.

A variable declaration consists of the variable's name, a sequence of attributes (which may be empty), a sequence of array dimension subranges (which will be empty for simple variables), its type, and a sequence of initial values (which is empty for non-initialised variables, is a singleton for simple variables and block initialised arrays, and is a sequence with the same number of elements as the array otherwise):

$$VarDecl \mathrel{\widehat=} [\ \xi : ID;\ A : \operatorname{seq} ATTR;\ SR : \operatorname{seq} Subrange;\ \tau : TYPE;\ V : \operatorname{seq} VALUE\ ]$$

2.7.4 Simple declarations

We bundle up the simple declarations, for use in procedure and function bodies.

$$SIMPLE\_DECL ::= constDecl\langle\langle ConstDecl\rangle\rangle \mid typeDecl\langle\langle TYPE\_DEF\rangle\rangle \mid varDecl\langle\langle VarDecl\rangle\rangle$$

2.8 Command declarations

2.8.1 Formal parameters

Formal parameters have a call type associated with them. They can be call by value or call by reference.

$$CALL\_TYPE ::= val \mid ref$$

We distinguish parameter declarations from variable declarations because the call type augments the attributes. A parameter may be specified as read only or write only.

A parameter declaration consists of the parameter's name, its call type, its attributes, a sequence of subranges (empty for zero-dimensional arrays) and its type:

$$ParamDecl \mathrel{\widehat=} [\ \xi : ID;\ c : CALL\_TYPE;\ A : \operatorname{seq} ATTR;\ SR : \operatorname{seq} Subrange;\ \tau : TYPE\ ]$$

2.8.2 Body

The body of a command consists of local simple declarations of constants, types and variables (but no command declarations), and a statement that defines the actions of the command (this is a 'block' in Pascal).

$$Body \mathrel{\widehat=} [\ \Delta SD : \operatorname{seq} SIMPLE\_DECL;\ \gamma : STMT\ ]$$

2.8.3 Procedure declarations

A procedure header consists of the procedure name and parameters.

$$ProcHdr \mathrel{\widehat=} [\ \xi : ID;\ \Pi : \operatorname{seq} ParamDecl\ ]$$

A procedure declaration consists of the procedure header and the procedure body.

$$ProcDecl \mathrel{\widehat=} [\ ProcHdr;\ Body\ ]$$

2.8.4 Function declarations

A function header is like a procedure header, with the addition of a return type.

$$FunHdr \mathrel{\widehat=} [\ ProcHdr;\ \tau : TYPE\ ]$$

A function declaration consists of the function header and the function body.

$$FunDecl \mathrel{\widehat=} [\ FunHdr;\ Body\ ]$$

2.8.5 Command declaration

We bundle up the procedure and function declarations, for use in module bodies.

$$PROC\_FUN\_DECL ::= procDecl\langle\langle ProcDecl\rangle\rangle \mid funDecl\langle\langle FunDecl\rangle\rangle$$

2.9 Module declarations

2.9.1 Import declaration

An import declaration is a constant import, or variable declaration, a procedure header, or a function header. The abstract syntax is

$$\begin{aligned} IMPORT\_DECL ::= {} & constHdr\langle\langle ID \times TYPE\rangle\rangle \mid varHdr\langle\langle VarDecl\rangle\rangle \\ & \mid procHdr\langle\langle ProcHdr\rangle\rangle \mid funcHdr\langle\langle FunHdr\rangle\rangle \end{aligned}$$

- The type in the constant import is required to be boolean, byte or unsigned; this is enforced by the type checking.
- The sequence of attributes in the variable import declaration is required to be a single 'read only' attribute; this is enforced by the concrete syntax.

2.9.2 Export declaration

An export declaration is a list of the names of the exported constants, variables, procedures, and functions.

$$EXPORT\_DECL == \mathrm{seq}\, ID$$

This sequence could be empty, but this would mean that none of the component items could be called from another module. This would make such a module rather useless, but is allowed. The concrete syntax ensures that the export keyword still has to be defined.

2.9.3 Module Header

A module header has a name, import declarations, simple declarations, and command declarations.

$$\begin{aligned} ModuleHdr \mathrel{\widehat=} [\ & \xi : ID;\ \Delta I : \operatorname{seq} IMPORT\_DECL; \\ & \Delta S : \operatorname{seq} SIMPLE\_DECL; \\ & \Delta PF : \operatorname{seq} PROC\_FUN\_DECL\ ] \end{aligned}$$

2.9.4 Module

A module has a module header, and an export declaration.

$$Module \mathrel{\widehat=} [\ ModuleHdr;\ e : EXPORT\_DECL\ ]$$

2.9.5 Main module

The main module is similar to the other modules in the program, except that it has the main body of the code, that defines the action of the main program, rather than an export declaration.

The global declarations are not accessible from the import declarations since they are declared after the import declarations.

$$MainModule \mathrel{\widehat=} [\ ModuleHdr;\ \gamma : STMT\ ]$$

2.10 Program

2.10.1 Modularised program

A modularised Pasp program consists of sequence of modules and a main module.

$$MPROG == \text{seq}\, Module \times MainModule$$

2.10.2 Flattened program

An unmodularised (flattened) Pasp program consists of its name, an attribute (the start address of the code), simple declarations, command declarations, and the body statement.

$$Prog \mathrel{\widehat=} [\, \xi : ID;\ a : ATTR;\ \Delta S : \text{seq}\, SIMPLE\_DECL; \\ \Delta PF : \text{seq}\, PROC\_FUN\_DECL;\ \gamma : STMT\,]$$

This syntax is provided so that the dynamic semantics of the *MPROG* can be related to the *Prog* semantics.

3 Concrete syntax

The abstract syntax is not cluttered with the necessary disambiguation mechanisms such as brackets needed to make it possible to parse the concrete form. It is more convenient to define the mapping from abstract to concrete syntax, as the mapping in the other direction involves specifying unnecessary details about the parser. Although the mapping from concrete to abstract syntax is a function (the concrete syntax is unambiguous), it is not injective (different concrete forms may have the same abstract form, for example, representation of numbers), so the mapping from abstract to concrete syntax is one-to-many. This means that relations, rather than functions, between abstract and concrete syntax are defined.

The concrete syntax is in free format (white space, including newlines, may be placed between any tokens but not within them). It is assumed that all comments are stripped from the program at the lexical analysis stage.

3.1 Identifiers

3.1.1 Characters

The case used is not significant, but the expected convention is that keywords are capitalised, and identifiers are in lower case.

We first introduce names for various sets of characters. The set $CHAR$ contains all ASCII characters.

$$[CHAR]$$

We (loosely) define three subsets of $CHAR$ as follows:

$$\begin{array}{|l} ALPHA, ALPHANUM, IDCHAR : \mathbb{P}\, CHAR \\ \hline ALPHA \subseteq ALPHANUM \subseteq IDCHAR \end{array}$$

$ALPHA$ is the set of alphabetic characters. (Strictly speaking, we need to define both an abstract set which contains one copy of each letter, and a

map between this set and the two concrete forms of lower and upper case. This just adds some unilluminating complications.) The set *ALPHANUM* contains the alphabetic characters and the digits. The set *IDCHAR* contains the alphanumeric characters and the underscore character '_'.

Z has no pre-defined syntax for characters: here we use single quoted characters to represent the printable ascii characters. For example, '**a**'.

3.1.2 Strings

A string is a sequence of *CHAR*:

$$String == \text{seq}\, CHAR$$

An *IdString* (used for identifiers) is a non-empty sequence of *IDCHAR* with an initial *ALPHA* character.

$$IdString == \{\ s : \text{seq}_1\, IDCHAR \mid s\ 1\ \in ALPHA\ \}$$

Note that we do not use conventional Z sequence notation for strings. Strings are enclosed in double quotes, rather than being expressed as comma separated lists of characters in sequence brackets:

"**a string**"

rather than

⟨'**a**', ' ', '**s**', '**t**', '**r**', '**i**', '**n**', '**g**'⟩

We use " " to stand for the empty string.

3.1.3 Identifiers

The following relation is assumed to be the natural one.

$$\mid\ \mathcal{C}_{ID} : ID\ \leftrightarrow IdString$$

The concrete syntax for the reserved name *maxunsigned* is the string "**MAXUNSIGNED**".

3.1.4 Sequences of items

To facilitate the formal definition of repeated parts of the Pasp language, it is helpful to define a generic relation to form lists of strings from sequences of abstract syntactical constructs; elements of the list are separated by some character from *CHAR*.

$$
\begin{array}{|l}
\hline
[X] \\
sepList : \operatorname{seq} X \times String \times (X \leftrightarrow String) \leftrightarrow String \\
\hline
\forall\, sep : String;\ r : X \leftrightarrow String \bullet (\langle\,\rangle, sep, r) \mapsto \text{“ ”} \in sepList \\
\forall\, \xi : X;\ sep : String;\ r : X \leftrightarrow String;\ str : String \mid \\
\qquad \xi \mapsto str \in r \bullet \\
\qquad (\langle \xi \rangle, sep, r) \mapsto str \in sepList \\
\forall\, \xi : X;\ \Xi, \Xi' : \operatorname{seq}_1 X;\ sep : String;\ r : X \leftrightarrow String; \\
\qquad str, str' : String \mid \\
\qquad \{(\Xi, sep, r) \mapsto str, (\Xi', sep, r) \mapsto str'\} \subseteq sepList \bullet \\
\qquad (\Xi \frown \Xi', sep, r) \mapsto str \frown sep \frown str' \in sepList \\
\hline
\end{array}
$$

Many of the concrete syntax of sequences of items can be derived from the relevant concrete syntax of single items in a generic manner.

A concrete syntax $\mathcal{C}$ is a relation between an item ξ and a string str. The concrete syntax of a sequence of such items is often the concatenation of the strings of the individual items. This concrete syntax is $\mathcal{C}_*$, where

function$(\_*)$

$$
\begin{array}{|l}
\hline
[X] \\
\_* : (X \leftrightarrow String) \rightarrow (\operatorname{seq} X \leftrightarrow String) \\
\hline
\forall\, c : X \leftrightarrow String \bullet \langle\,\rangle \mapsto \text{“ ”} \in c\,* \\
\forall\, c : X \leftrightarrow String;\ \xi : X;\ \Xi : \operatorname{seq} X;\ str, str' : String \mid \\
\qquad \xi \mapsto str \in c \land \Xi \mapsto str' \in c\,* \bullet \\
\qquad \langle \xi \rangle \frown \Xi \mapsto str \frown str' \in c\,* \\
\hline
\end{array}
$$

3.2 Values

3.2.1 Number literals

Number literals can be written in different bases, so we define a map from the natural numbers to these notations.

$$\mathcal{C}_{BASE} : 2 \ldots 36 \times \mathbb{N} \rightarrow \mathrm{seq}_1 \, ALPHANUM$$

$\mathcal{C}_{BASE}$ is the obvious function from the natural numbers to a representation given by the base. The fact that this *is* a function means that leading zeroes are not allowed. (The value of 36 for the maximum base allowed is because the symbols used are alphanumerics without case distinction.) Note that $\mathcal{C}_{BASE}(10, n)$ denotes the normal base ten representation of the number n.

The concrete syntax for numbers is a relation between the number and any of the concrete syntaxes it could have, depending on the base. The number can be written in a 'based' form, with the base b written in base ten, then a hash, then the value written in base b. There is also a special form for base ten, with the explicit base and hash omitted.

$$\mathcal{C}_{NUM} : \mathbb{N} \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{NUM} = \\
&\quad \{ \, n : \mathbb{N}; \; b : 2 \ldots 36 \bullet \\
&\qquad n \mapsto \mathcal{C}_{BASE}(10, b) \frown \text{“}\#\text{”} \frown \mathcal{C}_{BASE}(b, n) \} \\
&\quad \cup \{ \, n : \mathbb{N} \bullet n \mapsto \mathcal{C}_{BASE}(10, n) \, \}
\end{aligned}$$

3.2.2 Number values

Byte values are written as numbers. (Note: this definition stops values larger than a byte being written this way.)

$$\mathcal{C}_{CByte} : VALUE \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{CByte} = \\
&\quad \{ \, n : BYTE; \; str : String \mid n \mapsto str \in \mathcal{C}_{NUM} \bullet \\
&\qquad vbyte \; n \mapsto str \, \}
\end{aligned}$$

Unsigned values are written with an obligatory leading zero followed by a number. (Note: this definition stops values larger than an unsigned being written this way.)

$$\mathcal{C}_{CU} : VALUE \leftrightarrow String$$

$$\mathcal{C}_{CU} = \{\ n : UNSIGNED;\ str : String \mid n \mapsto str \in \mathcal{C}_{NUM} \bullet vunsgn\ n \mapsto \text{“}\mathbf{0}\text{”} \frown str\ \}$$

(Note: the conversions defined above are actually carried out in the lexical analysis phase, before parsing.)

So byte zero is written **0**, and unsigned zero is written **00**. The value ‘42’ can be written:

	byte	unsigned
normal denary	**42**	**042**
based denary	**10#42**	**010#42**
hexadecimal	**16#2A**	**016#2A**
binary	**2#101010**	**02#101010**

3.2.3 Boolean values

Booleans are written in the obvious textual form.

$$\mathcal{C}_{CB} : VALUE \leftrightarrow String$$

$$\mathcal{C}_{CB} = \{vbool\ ptrue \mapsto \text{“}\mathbf{TRUE}\text{”}, vbool\ pfalse \mapsto \text{“}\mathbf{FALSE}\text{”}\}$$

3.2.4 Values

The concrete syntax for values is the disjoint union of each kind.

$$\mathcal{C}_C : VALUE \leftrightarrow String$$

$$\mathcal{C}_C = \mathcal{C}_{CU} \cup \mathcal{C}_{CB} \cup \mathcal{C}_{CByte} \cup \{\ \xi : ID;\ str : String \mid \xi \mapsto str \in \mathcal{C}_{ID} \bullet name\ \xi \mapsto str\ \}$$

Note that there is no concrete syntax for the other kinds of $VALUE$.

The $\mathcal{C}_{ID}$ term supplies concrete syntax for constants and enumerated values occurring in initialisation statements. A constants or enumerated value in an expression is parsed as a *valueRef*.

3.3 Types

3.3.1 Bound

The concrete form of a subrange bound is a value string:

$$\mathcal{C}_B == \mathcal{C}_C$$

3.3.2 Subrange

The concrete form of a subrange has the two bounds separated by a pair of dots:

$$\mathcal{C}_{SR} : Subrange \leftrightarrow String$$

$$\mathcal{C}_{SR} = \{\ Subrange;\ bound, bound' : String \mid \{lb \mapsto bound, ub \mapsto bound'\} \subseteq \mathcal{C}_B \bullet \theta Subrange \mapsto bound \frown \text{“..”} \frown bound'\ \}$$

3.3.3 Types

The *typeWrong* has no corresponding concrete form as it is used only in the type checking semantics. Defined types are parsed as a *typeName* and then converted to an abstract *scopedTypeName* (which has no concrete syntax) in the type checking semantics.

The identifiers for built in types are reserved words and are not translated to a defined type.

$$
\begin{array}{|l}
\mathcal{C}_{ST} : TYPE \leftrightarrow String \\
\hline
\mathcal{C}_{ST} = \\
\quad \{unsigned \mapsto \text{“}\mathbf{UNSIGNED}\text{”}, pbyte \mapsto \text{“}\mathbf{BYTE}\text{”}, \\
\qquad boolean \mapsto \text{“}\mathbf{BOOLEAN}\text{”}\} \\
\quad \cup \{\ Subrange;\ ids : IdString \mid \theta Subrange \mapsto ids \in \mathcal{C}_{SR} \bullet \\
\qquad subrange\ \theta Subrange \mapsto ids\ \} \\
\quad \cup \{\ \xi : ID;\ ids : IdString \mid \xi \mapsto ids \in \mathcal{C}_{ID} \bullet \\
\qquad typeName\ \xi \mapsto ids\ \}
\end{array}
$$

3.4 Operators

3.4.1 Unary operators

The concrete syntax for unary operators is[3]

$$
\begin{array}{|l}
\mathcal{C}_{UO} : UNY\_OP \leftrightarrow String \\
\hline
\mathcal{C}_{UO} = \\
\quad \{unot \mapsto \text{“}\mathbf{UNOT}\text{”}, bnot \mapsto \text{“}\mathbf{BNOT}\text{”}, not \mapsto \text{“}\mathbf{NOT}\text{”}, \\
\quad bleft \mapsto \text{“}<<\text{”}, bleft \mapsto \text{“}\mathbf{BLEFT}\text{”}, \\
\quad bright \mapsto \text{“}>>\text{”}, bright \mapsto \text{“}\mathbf{BRIGHT}\text{”}, \\
\quad uleft \mapsto \text{“}\mathbf{ULEFT}\text{”}, uright \mapsto \text{“}\mathbf{URIGHT}\text{”}, \\
\quad byteToBool \mapsto \text{“}\mathbf{B2BOOL}\text{”}, \\
\quad boolToByte \mapsto \text{“}\mathbf{BOOL2B}\text{”}, \\
\quad byteToUnsgn \mapsto \text{“}\mathbf{B2U}\text{”}, unsgnToByte \mapsto \text{“}\mathbf{U2B}\text{”}, \\
\quad loByte \mapsto \text{“}\mathbf{LO}\text{”}, hiByte \mapsto \text{“}\mathbf{HI}\text{”}, \\
\quad ord \mapsto \text{“}\mathbf{E2B}\text{”}, pred \mapsto \text{“}\mathbf{PRED}\text{”}, succ \mapsto \text{“}\mathbf{SUCC}\text{”}\}
\end{array}
$$

[3]Earlier notation of **USGNB** for **U2B**, of **BYT** for **B2U**, and of **ORD** for **E2B**, is retained in the implementation for backwards compatibility, but is deprecated. It will be removed in some future version.

Non-symbolic forms of the byte shift operators are provided, intended for use with the B tool.

3.4.2 Binary operators

Binary operators are grouped into infix and prefix operators. Some operators can take sequences of expression arguments. Byte binary expressions can be written in infix notation.

$$\mathcal{C}_{IN}, \mathcal{C}_{INSEQ} : BIN\_OP \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{INSEQ} = \\
&\quad \{bplus \mapsto \text{“}+\text{”}, bminus \mapsto \text{“}-\text{”}, \\
&\qquad bmul \mapsto \text{“}*\text{”}, bdiv \mapsto \text{“}\mathbf{DIV}\text{”}, bmod \mapsto \text{“}\mathbf{MOD}\text{”}, \\
&\qquad band \mapsto \text{“}\&\text{”}, bor \mapsto \text{“}|\text{”}, bxor \mapsto \text{“}\hat{}\ \text{”}\} \\
&\quad \cup \{and \mapsto \text{“}\mathbf{AND}\text{”}, or \mapsto \text{“}\mathbf{OR}\text{”}\} \\
&\mathcal{C}_{IN} = \\
&\quad \{beq \mapsto \text{“}=\text{”}, blt \mapsto \text{“}<\text{”}, ble \mapsto \text{“}<=\text{”}, \\
&\qquad bgt \mapsto \text{“}>\text{”}, bge \mapsto \text{“}>=\text{”}, bne \mapsto \text{“}\backslash =\text{”}\} \\
&\quad \cup \mathcal{C}_{INSEQ}
\end{aligned}$$

All other binary expressions are written in prefix notation (also, a prefix form for byte binary expressions is supported, intended for use by the B tool).

$$\mathcal{C}_{PRE}, \mathcal{C}_{PRESEQ} : BIN\_OP \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{PRESEQ} = \\
&\quad \{uplus \mapsto \text{“}\mathbf{UADD}\text{”}, uminus \mapsto \text{“}\mathbf{USUB}\text{”}, \\
&\qquad umul \mapsto \text{“}\mathbf{UMUL}\text{”}, udiv \mapsto \text{“}\mathbf{UDIV}\text{”}, \\
&\qquad umod \mapsto \text{“}\mathbf{UMOD}\text{”}, \\
&\qquad uand \mapsto \text{“}\mathbf{UAND}\text{”}, uor \mapsto \text{“}\mathbf{UOR}\text{”}, uxor \mapsto \text{“}\mathbf{UXOR}\text{”}\} \\
&\quad \cup \{bplus \mapsto \text{“}\mathbf{BADD}\text{”}, bminus \mapsto \text{“}\mathbf{BSUB}\text{”}, \\
&\qquad bmul \mapsto \text{“}\mathbf{BMUL}\text{”}, bdiv \mapsto \text{“}\mathbf{BDIV}\text{”}, \\
&\qquad bmod \mapsto \text{“}\mathbf{BMOD}\text{”}, \\
&\qquad band \mapsto \text{“}\mathbf{BAND}\text{”}, bor \mapsto \text{“}\mathbf{BOR}\text{”}, bxor \mapsto \text{“}\mathbf{BXOR}\text{”}\} \\
&\mathcal{C}_{PRE} = \\
&\quad \{ueq \mapsto \text{“}\mathbf{UEQ}\text{”}, ult \mapsto \text{“}\mathbf{ULT}\text{”}, ule \mapsto \text{“}\mathbf{ULE}\text{”}, \\
&\qquad ugt \mapsto \text{“}\mathbf{UGT}\text{”}, uge \mapsto \text{“}\mathbf{UGE}\text{”}, une \mapsto \text{“}\mathbf{UNE}\text{”}\} \\
&\quad \cup \{eeq \mapsto \text{“}\mathbf{EEQ}\text{”}, ene \mapsto \text{“}\mathbf{ENE}\text{”}\} \\
&\quad \cup \{join \mapsto \text{“}\mathbf{JOIN}\text{”}, byteToEnum \mapsto \text{“}\mathbf{B2E}\text{”}\} \\
&\quad \cup \mathcal{C}_{PRESEQ}
\end{aligned}$$

3.5 Expressions

$$\mathcal{C}_E : EXPR \leftrightarrow String$$

3.5.1 Array index list

The concrete form of an empty array index list is empty. The concrete form of a non-empty array index list is a comma separated list of the expressions, enclosed in square brackets.

$$\mathcal{C}_{AI^*} : \operatorname{seq} EXPR \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{AI^*} = \\
&\quad \{\langle\,\rangle \mapsto \text{“ ”}\} \\
&\quad \cup \{\ E : \operatorname{seq}_1 EXPR;\ exprlist : String \mid \\
&\qquad\qquad (E, \text{“,”}, \mathcal{C}_E) \mapsto exprlist \in sepList \bullet \\
&\qquad E \mapsto \text{“[”} \frown exprlist \frown \text{“]”}\ \}
\end{aligned}$$

3.5.2 Actual parameter list

The concrete form of an empty actual parameter list is empty. The concrete form of a non-empty list is a comma separated list of the expressions, enclosed in parentheses.

$$\mathcal{C}_{AP^*} : \operatorname{seq} EXPR \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{AP^*} = \\
&\quad \{\langle\,\rangle \mapsto \text{“ ”}\} \\
&\quad \cup \{\ E : \operatorname{seq}_1 EXPR;\ exprlist : String \mid \\
&\qquad\qquad (E, \text{“,”}, \mathcal{C}_E) \mapsto exprlist \in sepList \bullet \\
&\qquad E \mapsto \text{“(”} \frown exprlist \frown \text{“)”}\ \}
\end{aligned}$$

3.5.3 Expression

$$
\begin{array}{|l}
\mathcal{C}_E : EXPR \leftrightarrow String \\
\hline
\mathcal{C}_E = \\
\quad \{\ \kappa : VALUE;\ str : String \mid \kappa \mapsto str \in \mathcal{C}_C \bullet constant\ \kappa \mapsto str\ \} \\
\quad \cup \{\ ValueRefExpr;\ ids : IdString;\ indexlist : String \mid \\
\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \land E \mapsto indexlist \in \mathcal{C}_{AI^*} \bullet \\
\qquad valueRef\ \theta ValueRefExpr \mapsto ids \frown indexlist\ \} \\
\quad \cup \{\ UnyExpr;\ op, expr : String \mid \\
\qquad \Psi \mapsto op \in \mathcal{C}_{UO} \land \epsilon \mapsto expr \in \mathcal{C}_E \bullet \\
\qquad unyExpr\ \theta UnyExpr \mapsto op \frown \text{“(”} \frown expr \frown \text{“)”}\ \} \\
\quad \cup \{\ BinExpr;\ op, expr1, expr2 : String \mid \\
\qquad \Omega \mapsto op \in \mathcal{C}_{IN} \\
\qquad \land \{\epsilon 1 \mapsto expr1, \epsilon 2 \mapsto expr2\} \subseteq \mathcal{C}_E \bullet \\
\qquad binExpr\ \theta BinExpr \mapsto \\
\qquad \text{“(”} \frown expr1 \frown op \frown expr2 \frown \text{“)”}\ \} \\
\quad \cup \{\ BinExpr;\ E : \mathrm{seq}\, EXPR;\ op, exprlist, expr : String \mid \\
\qquad \Omega \mapsto op \in \mathcal{C}_{INSEQ} \\
\qquad \land (E, op, \mathcal{C}_E) \mapsto exprlist \in sepList \\
\qquad \land \epsilon 1 \mapsto \text{“(”} \frown exprlist \frown \text{“)”} \in \mathcal{C}_E \\
\qquad \land \epsilon 2 \mapsto expr \in \mathcal{C}_E \bullet \\
\qquad binExpr\ \theta BinExpr \mapsto \\
\qquad \text{“(”} \frown exprlist \frown op \frown expr \frown \text{“)”}\} \\
\quad \cup \{\ BinExpr;\ op, expr1, expr2 : String \mid \\
\qquad \Omega \mapsto op \in \mathcal{C}_{PRE} \\
\qquad \land \{\epsilon 1 \mapsto expr1, \epsilon 2 \mapsto expr2\} \subseteq \mathcal{C}_E \bullet \\
\qquad binExpr\ \theta BinExpr \\
\qquad \mapsto op \frown \text{“(”} \frown expr1 \frown \text{“,”} \frown expr2 \frown \text{“)”}\ \} \\
\quad \cup \{\ BinExpr;\ E : \mathrm{seq}\, EXPR;\ op, exprlist, expr : String \mid \\
\qquad \Omega \mapsto op \in \mathcal{C}_{PRESEQ} \\
\qquad \land (E, \text{“,”}, \mathcal{C}_E) \mapsto exprlist \in sepList \\
\qquad \land \epsilon 1 \mapsto op \frown \text{“(”} \frown exprlist \frown \text{“)”} \in \mathcal{C}_E \\
\qquad \land \epsilon 2 \mapsto expr \in \mathcal{C}_E \bullet \\
\qquad binExpr\theta BinExpr \\
\qquad \mapsto op \frown \text{“(”} \frown exprlist \frown \text{“,”} \frown expr \frown \text{“)”}\ \} \\
\quad \cup \{\ FunCallExpr;\ ids : IdString;\ exprlist : String \mid \\
\qquad E \neq \langle\,\rangle \land \xi \mapsto ids \in \mathcal{C}_{ID} \land E \mapsto exprlist \in \mathcal{C}_{AP^*} \bullet \\
\qquad funCall\ \theta FunCallExpr \mapsto ids \frown exprlist\ \} \\
\quad \cup \{\ \epsilon : EXPR;\ expr : String \mid \epsilon \mapsto expr \in \mathcal{C}_E \bullet \\
\qquad \epsilon \mapsto \text{“(”} \frown expr \frown \text{“)”}\ \}
\end{array}
$$

- A constant is the relevant value string.
- An value reference is the name, then the list of index expressions. For example:

  a [(num_cells + 3), 12]

- A unary expressions is the unary operator followed by the parenthesised expression.
- A binary expression involving an infix operator is the first argument expression, then the infix operator, then the second argument expression, with the whole binary expression parenthesised, (so no precedence for operators need be defined).
- A nested binary expression involving an infix operator that can take a sequence of arguments, is the parenthesised, operator separated, flattened list of the nested argument expressions.
- A binary expression involving prefix operators is the operator, then the parenthesised, comma separated list of the two argument expressions.
- A nested binary expression involving prefix operators that can take a sequence of arguments, is the operator, then the parenthesised, comma separated flattened list of the nested argument expressions.
- A function call, which must have at least one parameter, is the function name then the parenthesised, comma separated list of argument expression. (The concrete restriction to non-empty parameter lists enables the parser to distinguish between a variable and a function with no arguments while building up the symbol table, and check whether the identifier was declared as a variable or a function.)
- Any expression may be supplied with an arbitrary number of additional brackets.

3.6 Statements

$$\mathcal{C}_S : STMT \leftrightarrow String$$

3.6.1 Branch

First we define the concrete syntax of a *Branch*.

$$\mathcal{C}_{CL} : Branch \leftrightarrow String$$

$$\begin{aligned}
&\mathcal{C}_{CL} = \\
&\quad \{\ Branch;\ idlist, stmt : String \mid \\
&\qquad \gamma \mapsto stmt \in \mathcal{C}_S \\
&\qquad \land\ (\Xi, \text{“,”}, \mathcal{C}_{ID}) \mapsto idlist \in sepList \bullet \\
&\quad \theta Branch \mapsto idlist \frown \text{“:”} \frown stmt\ \}
\end{aligned}$$

3.6.2 Statement

$$
\begin{array}{|l}
\mathcal{C}_S : STMT \leftrightarrow String \\
\hline
\mathcal{C}_S = \\
\quad \{skip \mapsto \text{“ ”}\} \\
\quad \cup \{\ \Gamma : \mathrm{seq}_1\, STMT;\ stmtlist : String \mid \\
\qquad\qquad (\Gamma, \text{“}\mathbf{;}\text{”}, \mathcal{C}_S) \mapsto stmtlist \in sepList \bullet \\
\qquad block\ \Gamma \mapsto \text{“}\mathbf{BEGIN}\text{”} \frown stmtlist \frown \text{“}\mathbf{END}\text{”}\} \\
\quad \cup \{\ AssignStmt;\ ids : IdString;\ indexlist, expr : String \mid \\
\qquad\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \land E \mapsto indexlist \in \mathcal{C}_{AI^*} \\
\qquad\qquad \land\ \epsilon \mapsto expr \in \mathcal{C}_E \bullet \\
\qquad assign\ \theta AssignStmt \mapsto ids \frown indexlist \frown \text{“}\mathbf{:=}\text{”} \frown expr\ \} \\
\quad \cup \{\ IfStmt;\ expr, stmt : String \mid \\
\qquad\qquad \epsilon \mapsto expr \in \mathcal{C}_E \land \gamma 1 \mapsto stmt \in \mathcal{C}_S \bullet \\
\qquad ifStmt\ \theta IfStmt \mapsto \text{“}\mathbf{IF}\text{”} \frown expr \frown \text{“}\mathbf{THEN}\text{”} \frown stmt\ \} \\
\quad \cup \{\ IfStmt;\ expr, stmt1, stmt2 : String \mid \\
\qquad\qquad \epsilon \mapsto expr \in \mathcal{C}_E \\
\qquad\qquad \land\ \{\gamma 1 \mapsto stmt1, \gamma 2 \mapsto stmt2\} \subseteq \mathcal{C}_S \bullet \\
\qquad ifStmt\ \theta IfStmt \mapsto \\
\qquad\qquad \text{“}\mathbf{IF}\text{”} \frown expr \frown \text{“}\mathbf{THEN}\text{”} \frown stmt1 \frown \text{“}\mathbf{ELSE}\text{”} \frown stmt2\ \} \\
\quad \cup \{\ CaseStmt;\ expr, caselist : String \mid \\
\qquad\qquad \epsilon \mapsto expr \in \mathcal{C}_E \\
\qquad\qquad \land\ (K, \text{“}\mathbf{;}\text{”}, \mathcal{C}_{CL}) \mapsto caselist \in sepList \bullet \\
\qquad caseStmt\ \theta CaseStmt \mapsto \\
\qquad\qquad \text{“}\mathbf{CASE}\text{”} \frown expr \frown \text{“}\mathbf{OF}\text{”} \frown caselist \frown \text{“}\mathbf{END}\text{”}\ \} \\
\quad \cup \{\ WhileStmt;\ expr, stmt : String \mid \\
\qquad\qquad \epsilon \mapsto expr \in \mathcal{C}_E \land \gamma \mapsto stmt \in \mathcal{C}_S \bullet \\
\qquad whileStmt\ \theta WhileStmt \mapsto \\
\qquad\qquad \text{“}\mathbf{WHILE}\text{”} \frown expr \frown \text{“}\mathbf{DO}\text{”} \frown stmt\ \} \\
\quad \cup \{\ ProcCallStmt;\ ids : IdString;\ paramlist : String \mid \\
\qquad\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \land E \mapsto paramlist \in \mathcal{C}_{AP^*} \bullet \\
\qquad procCall\ \theta ProcCallStmt \mapsto ids \frown paramlist\ \}
\end{array}
$$

- The skip statement becomes an empty string, written as “ ”. This allows a semicolon to be used to terminate the last statement in a block if so desired. Although the concrete syntax for this statement is just a null string, there is no ambiguity over the number of skips

in a block, since a block consists of a semicolon separated sequence of statements.

- A block is surrounded by begin-end keywords, and its list of statements are separated by semicolons.
- Assignment uses the "**:=**"symbol.
- There are two forms for the if statement: the else clause may be omitted if the else statement is a 'skip'.
- A procedure call consists of the procedure name followed by the actual arguments enclosed in parentheses.

3.7 Simple declarations

3.7.1 Named constant declarations

A named constant declaration consists of the name, an equals sign, the value that is denoted by the name, and a terminating semicolon:

$$\mathcal{C}_{NC} : \mathit{ConstDecl} \leftrightarrow \mathit{String}$$

$$\begin{aligned}
&\mathcal{C}_{NC} = \\
&\quad \{\ \mathit{ConstDecl};\ \mathit{ids} : \mathit{IdString};\ \mathit{value} : \mathit{String} \mid \\
&\qquad \xi \mapsto \mathit{ids} \in \mathcal{C}_{ID} \land \kappa \mapsto \mathit{value} \in \mathcal{C}_C \bullet \\
&\qquad \theta \mathit{ConstDecl} \mapsto \\
&\qquad\quad \text{“}\mathbf{CONST}\text{”} \frown \mathit{ids} \frown \text{“}=\text{”} \frown \mathit{value} \frown \text{“}\mathbf{;}\text{”}\ \}
\end{aligned}$$

For example, the following fragment is a well-formed definition of a named constant declaration (with byte type):

CONST num_cells = 10;

3.7.2 Type declarations

User defined types are introduced in a type declaration. Currently the only user defined types are enumerated types.

An enumerated type declaration consists of the type name, an equals sign, a bracketed comma separated list of names for the enumerated values, and a terminating semicolon.

$$
\begin{array}{|l}
\mathcal{C}_{TD} : TYPE\_DEF \leftrightarrow String \\
\hline
\mathcal{C}_{TD} = \\
\quad \{\ EnumDecl;\ ids : IdString;\ idlist : String \mid \\
\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \\
\qquad \land\ (\Xi, \text{“,”}, \mathcal{C}_{ID}) \mapsto idlist \in sepList \bullet \\
\qquad enumDecl\ \theta EnumDecl \mapsto \\
\qquad \text{“}\mathbf{TYPE}\text{”} \frown ids \frown \text{“=”} \frown \text{“(”} \frown idlist \frown \text{“)”} \frown \text{“}\mathbf{;}\text{”}\ \}
\end{array}
$$

For example

TYPE colour = (red, green, blue);

3.7.3 Variable declarations

3.7.3.1 Attributes

The *readOnly*, *writeOnly*, and *nvram* attributes are written as simple strings. The *dataAt* attribute has the same concrete form: the keyword “**AT**” followed by a parenthesized concrete form of the location value.

$$
\begin{array}{|l}
\mathcal{C}_A : ATTR \leftrightarrow String \\
\hline
\mathcal{C}_A = \\
\quad \{readOnly \mapsto \text{“}\mathbf{READONLY}\text{”}, \\
\qquad writeOnly \mapsto \text{“}\mathbf{WRITEONLY}\text{”}, \\
\qquad nvram \mapsto \text{“}\mathbf{NVRAM}\text{”}\} \\
\quad \cup \{\ n : ADDR;\ num : String \mid n \mapsto num \in \mathcal{C}_{NUM} \bullet \\
\qquad dataAt\ n \mapsto \text{“}\mathbf{AT}\ \mathbf{(}\text{”} \frown num \frown \text{“}\mathbf{)}\text{”}\ \}
\end{array}
$$

For example,

AT (16#100)

The concrete form of an empty attribute list is the empty string. The concrete form of a non-empty attribute list is a comma separated list of the attributes, enclosed in special attribute brackets.

$$
\begin{array}{|l}
\mathcal{C}_{A^*} : \operatorname{seq} ATTR \leftrightarrow String \\
\hline
\mathcal{C}_{A^*} = \\
\quad \{\langle\,\rangle \mapsto \text{“ ”}\} \\
\quad \cup \{\ A : \operatorname{seq}_1 ATTR;\ attrlist : String \mid \\
\qquad\qquad (A, \text{“,”}, \mathcal{C}_A) \mapsto attrlist\ \in sepList \bullet \\
\qquad A\ \mapsto \text{“\{>”}\ \frown attrlist\ \frown \text{“<\}”}\ \}
\end{array}
$$

For example,

{> **READONLY, AT (16#100)** <}

3.7.3.2 Array subrange list

The concrete form of an empty subrange list is the empty string. The concrete form of a non-empty subrange list is a comma separated list of the subranges, enclosed in square brackets, plus two keywords.

$$
\begin{array}{|l}
\mathcal{C}_{ST^*} : \operatorname{seq} Subrange \leftrightarrow String \\
\hline
\mathcal{C}_{ST^*} = \\
\quad \{\langle\,\rangle \mapsto \text{“ ”}\} \\
\quad \cup \{\ SR : \operatorname{seq}_1 Subrange;\ srlist : String \mid \\
\qquad\qquad (SR, \text{“,”}, \mathcal{C}_{SR}) \mapsto srlist\ \in sepList \bullet \\
\qquad SR\ \mapsto \text{“}\mathbf{ARRAY}\ \text{[”}\ \frown srlist\ \frown \text{“]}\ \mathbf{OF}\text{”}\ \}
\end{array}
$$

For example,

ARRAY [0 . . num_cells, 0 . . num_books] OF

3.7.3.3 Initialisation value

The concrete form of an empty initialisation is the empty string. The concrete form of a simple initialisation is an equals sign followed by the value. The concrete form of an array initialisation is an equals sign followed by a square bracketed, comma separated list of the values.

$$
\begin{array}{|l}
\mathcal{C}_I : \mathrm{seq}\ VALUE \leftrightarrow String \\
\hline
\mathcal{C}_I = \\
\quad \{\langle\,\rangle \mapsto \text{“ ”}\} \\
\quad \cup \{\ \kappa : VALUE;\ value : String \mid \kappa\ \mapsto value\ \in \mathcal{C}_C \bullet \\
\qquad \langle \kappa \rangle\ \mapsto \text{“=”}\ \frown value\ \} \\
\quad \cup \{\ V : \mathrm{seq}_1\ VALUE;\ vlist : String \mid \\
\qquad\quad (V, \text{“,”}, \mathcal{C}_C) \mapsto vlist\ \in sepList \bullet \\
\qquad V\ \mapsto (\text{“=”}\ \frown \text{“[”}\ \frown vlist\ \frown \text{“]”})\ \}
\end{array}
$$

For example,

= 00
= [1 , 2, 4]

3.7.3.4 Variable declaration

A variable declaration starts with the **VAR** keyword; followed by the name or the variable; a colon; a bracketed comma separated attribute list (or empty unbracketed list); keyword “**ARRAY**”, a bracketed comma separated list of subranges, and keyword “**OF**” (or no keywords and an empty list); the type; the initialisation, and a terminating semicolon. (Only certain combinations of attributes and initialisations are allowed: these are checked in the type-checking semantics, not constrained by the syntax.)

$$
\begin{array}{|l}
\mathcal{C}_V : \mathit{VarDecl} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_V = \\
\quad \{\ \mathit{VarDecl};\ \mathit{ids} : \mathit{IdString};\ \mathit{attrlist}, \mathit{srlist}, \mathit{type}, \mathit{vlist} : \mathit{String} \mid \\
\qquad \xi \mapsto \mathit{ids} \in \mathcal{C}_{ID} \land A \mapsto \mathit{attrlist} \in \mathcal{C}_{A^*} \\
\qquad \land\ SR \mapsto \mathit{srlist} \in \mathcal{C}_{ST^*} \\
\qquad \land\ \tau \mapsto \mathit{type} \in \mathcal{C}_{ST} \land V \mapsto \mathit{vlist} \in \mathcal{C}_I \bullet \\
\quad \theta\, \mathit{VarDecl} \mapsto \\
\qquad (\text{“}\mathbf{VAR}\text{”} \frown \mathit{ids} \frown \text{“}\mathbf{:}\text{”} \\
\qquad\quad \frown \mathit{attrlist} \frown \mathit{srlist} \frown \mathit{type} \frown \mathit{vlist} \frown \text{“}\mathbf{;}\text{”})\ \} \\
\end{array}
$$

For example:

```
VAR x1 : BOOLEAN = TRUE ;
VAR x2 : {> AT (42) <} UNSIGNED = 02 ;
VAR x3 : ARRAY [0 .. 3] OF BYTE = [ 0 , 1, 2, 4 ];
VAR x4 : {> AT (16#100) <} ARRAY [0 .. 3] OF UNSIGNED
    = [ 00 , 01, 02, 04 ];
VAR x5 : ARRAY [0 .. 3] OF BOOLEAN = FALSE ;
VAR x6 : {> AT (16#100) <} ARRAY [0 .. 3] OF UNSIGNED
    = 04 ;
VAR in1 : {> READONLY , AT (42) <} BYTE ;
VAR in2 : {> READONLY , AT (42) <}
    ARRAY [0 .. 3] OF BYTE ;
VAR out1 : {> WRITEONLY , AT (42) <} BOOLEAN;
VAR out2 : {> WRITEONLY , AT (42) <}
    ARRAY [0 .. 3] OF UNSIGNED;
VAR n1 : {> NVRAM , AT (42) <} BYTE ;
VAR n2 : {> NVRAM , AT (42) <} ARRAY [0 .. 3] OF BYTE ;
```

3.7.4 Simple declaration

$$
\begin{array}{|l}
\mathcal{C}_{SD} : SIMPLE\_DECL \leftrightarrow String \\
\hline
\mathcal{C}_{SD} = \\
\quad \{\ \delta : ConstDecl;\ decl : String \mid \delta\ \mapsto decl\ \in \mathcal{C}_{NC} \bullet \\
\qquad\qquad constDecl\ \delta\ \mapsto decl\ \} \\
\quad \cup \{\ \delta : TYPE\_DEF;\ decl : String \mid \delta\ \mapsto decl\ \in \mathcal{C}_{TD} \bullet \\
\qquad\qquad typeDecl\ \delta\ \mapsto decl\ \} \\
\quad \cup \{\ \delta : VarDecl;\ decl : String \mid \delta\ \mapsto decl\ \in \mathcal{C}_{V} \bullet \\
\qquad\qquad varDecl\ \delta\ \mapsto decl\ \}
\end{array}
$$

3.8 Command declarations

3.8.1 Formal parameters

A formal parameter declaration has a similar concrete syntax to a variable declaration. The most significant difference is the use of a call type to modify the variable.

$$
\begin{array}{|l}
\mathcal{C}_{CT} : CALL\_TYPE \leftrightarrow String \\
\hline
\mathcal{C}_{CT} = \{ref\ \mapsto \text{“}\mathbf{VAR}\text{”}, val\ \mapsto \text{“ ”}\}
\end{array}
$$

This definition means that, as in standard Pascal, the string

VAR

means that a parameter is treated as call by reference, while the absence of any modifier means that the parameter is call by value. (Since a null string can always be found, the parser must check for the call by reference case first.)

$$
\begin{array}{|l}
\mathcal{C}_{PMD} : \mathit{ParamDecl} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{PMD} = \\
\quad \{\ \mathit{ParamDecl};\ \mathit{ids} : \mathit{IdString};\ \mathit{call}, \mathit{attrlist}, \mathit{srlist}, \mathit{type} : \mathit{String} \mid \\
\qquad \xi \mapsto \mathit{ids} \in \mathcal{C}_{ID} \land c \mapsto \mathit{call} \in \mathcal{C}_{CT} \\
\qquad \land\ A \mapsto \mathit{attrlist} \in \mathcal{C}_{A^*} \land \mathit{SR} \mapsto \mathit{srlist} \in \mathcal{C}_{ST^*} \\
\qquad \land\ \tau \mapsto \mathit{type} \in \mathcal{C}_{ST} \bullet \\
\quad \theta \mathit{ParamDecl} \mapsto \\
\qquad \mathit{call} \frown \mathit{ids} \frown \text{“}\mathtt{:}\text{”} \frown \mathit{attrlist} \frown \mathit{srlist} \frown \mathit{type}\ \}
\end{array}
$$

The concrete syntax for sequences of parameter declarations is a semi-colon separated list. An empty list of parameters is an empty string, otherwise parameters are parenthesised.

$$
\begin{array}{|l}
\mathcal{C}_{PMD^*} : \mathrm{seq}\, \mathit{ParamDecl} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{PMD^*} = \\
\quad \{\langle\ \rangle \mapsto \text{“ ”}\} \\
\quad \cup \{\ \Pi : \mathrm{seq}_1\, \mathit{ParamDecl};\ \mathit{paramlist} : \mathit{String} \mid \\
\qquad (\Pi, \text{“}\mathtt{;}\text{”}, \mathcal{C}_{PMD}) \mapsto \mathit{paramlist} \in \mathit{sepList} \bullet \\
\quad \Pi \mapsto \text{“}\mathtt{(}\text{”} \frown \mathit{paramlist} \frown \text{“}\mathtt{)}\text{”}\ \}
\end{array}
$$

3.8.2 Body

A command body syntax is the list of simple declarations and the body, concatenated together.

$$
\begin{array}{|l}
\mathcal{C}_{BODY} : \mathit{Body} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{BODY} = \\
\quad \{\ \mathit{Body};\ \mathit{decllist}, \mathit{stmt} : \mathit{String} \mid \\
\qquad \Delta \mathit{SD} \mapsto \mathit{decllist} \in \mathcal{C}_{SD^*} \land \gamma \mapsto \mathit{stmt} \in \mathcal{C}_S \bullet \\
\quad \theta \mathit{Body} \mapsto \mathit{decllist} \frown \mathit{stmt}\ \}
\end{array}
$$

3.8.3 Procedure declarations

The concrete syntax of a procedure header is the keyword “**PROCEDURE**”, followed by the procedure name, the parameter list, and a terminating semi-

colon.

$$
\begin{array}{|l}
\mathcal{C}_{PH} : \mathit{ProcHdr} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{PH} = \\
\quad \{\ \mathit{ProcHdr};\ \mathit{ids} : \mathit{IdString};\ \mathit{paramlist} : \mathit{String} \mid \\
\qquad \xi \mapsto \mathit{ids} \in \mathcal{C}_{ID} \land \Pi \mapsto \mathit{paramlist} \in \mathcal{C}_{PMD^*} \bullet \\
\qquad \theta \mathit{ProcHdr} \mapsto \text{“}\mathbf{PROCEDURE}\text{”} \frown \mathit{ids} \frown \mathit{paramlist} \frown \text{“}\mathbf{;}\text{”}\ \}
\end{array}
$$

The concrete syntax of a procedure declaration is the procedure header, followed by the body statement, and a terminating semicolon.

$$
\begin{array}{|l}
\mathcal{C}_{PD} : \mathit{ProcDecl} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{PD} = \\
\quad \{\ \mathit{ProcDecl};\ \mathit{hdr}, \mathit{body} : \mathit{String} \mid \\
\qquad \theta \mathit{ProcHdr} \mapsto \mathit{hdr} \in \mathcal{C}_{PH} \\
\qquad \land\ \theta \mathit{Body} \mapsto \mathit{body} \in \mathcal{C}_{BODY} \bullet \\
\qquad \theta \mathit{ProcDecl} \mapsto \mathit{hdr} \frown \mathit{body} \frown \text{“}\mathbf{;}\text{”}\ \}
\end{array}
$$

3.8.4 Function declarations

The concrete syntax of a function header is the keyword “**FUNCTION**”, followed by the procedure name, a non-empty parameter list, a colon, the return type, and a terminating semicolon.

$$
\begin{array}{|l}
\mathcal{C}_{FH} : \mathit{FunHdr} \leftrightarrow \mathit{String} \\
\hline
\mathcal{C}_{FH} = \\
\quad \{\ \mathit{FunHdr};\ \mathit{ids} : \mathit{IdString};\ \mathit{paramlist}, \mathit{type} : \mathit{String} \mid \\
\qquad \xi \mapsto \mathit{ids} \in \mathcal{C}_{ID} \land \Pi \mapsto \mathit{paramlist} \in \mathcal{C}_{PMD^*} \\
\qquad \land\ \tau \mapsto \mathit{type} \in \mathcal{C}_{ST} \bullet \\
\qquad \theta \mathit{FunHdr} \mapsto \\
\qquad\quad \text{“}\mathbf{FUNCTION}\text{”} \frown \mathit{ids} \frown \mathit{paramlist} \frown \text{“}\mathbf{:}\text{”} \frown \mathit{type} \frown \text{“}\mathbf{;}\text{”}\ \}
\end{array}
$$

The concrete syntax of a function declaration is the function header, followed by the body statement, and a terminating semicolon.

$$\mathcal{C}_{FD} : FunDecl \leftrightarrow String$$

$$\mathcal{C}_{FD} = \{\ FunDecl;\ hdr, body : String \mid \theta FunHdr \mapsto hdr \in \mathcal{C}_{FH} \land \theta Body \mapsto body \in \mathcal{C}_{BODY} \bullet \theta FunDecl \mapsto hdr \frown body \frown \text{“;”}\ \}$$

Note that the concrete syntax enforces the restriction that functions must take at least one parameter. This is to avoid confusion with named constants and variables. This restriction is necessary because of the simple nature of the parser; if it built up a symbol table during parsing, then the problem would not arise.

3.8.5 Command declarations

$$\mathcal{C}_{PFD} : PROC\_FUN\_DECL \leftrightarrow String$$

$$\mathcal{C}_{PFD} = \{\ \delta : ProcDecl;\ decl : String \mid \delta \mapsto decl \in \mathcal{C}_{PD} \bullet procDecl\ \delta \mapsto decl\ \} \cup \{\ \delta : FunDecl;\ decl : String \mid \delta \mapsto decl \in \mathcal{C}_{FD} \bullet funDecl\ \delta \mapsto decl\ \}$$

3.9 Module declarations

3.9.1 Import declarations

An import declaration is syntactically like a named constant declaration (with a type instead of a value), or a variable declaration (with a read-only attribute, and hence no initialisation), a procedure header or a function header, preceded by the “**IMPORT**” keyword. For example

IMPORT CONST c = BYTE;
IMPORT VAR b : {> READONLY <} BYTE;
IMPORT VAR arr : {> READONLY <} ARRAY [0..b]
OF UNSIGNED;
IMPORT PROCEDURE some_proc (VAR first:UNSIGNED,
second:BOOLEAN);
IMPORT FUNCTION a_function (VAR first:BYTE,
second:BYTE):UNSIGNED;

The import declarations are included at the beginning of the module. This allows correct semantic checking before use within the module.

$$\mathcal{C}_{IM} : IMPORT\_DECL \leftrightarrow String$$

$$\begin{aligned}
\mathcal{C}_{IM} = \\
&\{\ \xi : ID;\ \tau : TYPE;\ ids, type : String \mid \\
&\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \land \tau \mapsto type \in \mathcal{C}_{ST} \bullet \\
&\quad constHdr(\xi, \tau) \mapsto \\
&\qquad (\text{“}\mathbf{IMPORT}\text{”} \frown \text{“}\mathbf{CONST}\text{”} \frown ids \frown \text{“}=\text{”} \frown type)\ \} \\
&\cup \{\ VarDecl;\ var : String \mid \\
&\qquad A = \langle readOnly \rangle \land V = \langle\ \rangle \\
&\qquad \land\ \theta VarDecl \mapsto var \in \mathcal{C}_V \bullet \\
&\quad varHdr\ \theta VarDecl \mapsto \text{“}\mathbf{IMPORT}\text{”} \frown var\ \} \\
&\cup \{\ ProcHdr;\ hdr : String \mid \theta ProcHdr \mapsto hdr \in \mathcal{C}_{PH} \bullet \\
&\quad procHdr\ \theta ProcHdr \mapsto \text{“}\mathbf{IMPORT}\text{”} \frown hdr\ \} \\
&\cup \{\ FunHdr;\ hdr : String \mid \theta FunHdr \mapsto hdr \in \mathcal{C}_{FH} \bullet \\
&\quad funcHdr\ \theta FunHdr \mapsto \text{“}\mathbf{IMPORT}\text{”} \frown hdr\ \}
\end{aligned}$$

Note that the attribute list in a variable import must be precisely *readOnly*, and that no initialisation is allowed.

3.9.2 Export declarations

An export declaration for a module is a list of all the constant, variable, procedure, and function names to be exported from the module. Each module

has exactly one export declaration (whereas it can have zero or multiple import declarations). For example:

EXPORT some_proc, var1, a_func, const2, ano_func, var3

The export declaration is included at the end of the module. This allows correct semantic checking to ensure declaration before export within the module. The syntax for an export declaration is the keyword, followed by a comma separated list of identifiers.

$$
\begin{array}{|l}
\mathcal{C}_{EX} : EXPORT\_DECL \leftrightarrow String \\
\hline
\mathcal{C}_{EX} = \\
\quad \{\ \Xi : \mathrm{seq}\, ID;\ namelist : String \mid \\
\qquad (\Xi, \text{“,”}, \mathcal{C}_{ID}) \mapsto namelist \in sepList \bullet \\
\qquad \Xi \mapsto \text{“}\mathbf{EXPORT}\text{”} \frown namelist\ \}
\end{array}
$$

3.9.3 Module Header

A Module Header is a name, a list of import statements, simple declarations, and command declarations.

$$
\begin{array}{|l}
\mathcal{C}_{MH} : ModuleHdr \leftrightarrow String \\
\hline
\mathcal{C}_{MH} = \\
\quad \{\ ModuleHdr;\ ids : IdString;\ importlist, sdlist, pflist : String \mid \\
\qquad \xi \mapsto ids \in \mathcal{C}_{ID} \\
\qquad \land\ \Delta I \mapsto importlist \in \mathcal{C}_{IM}\ ^{*} \\
\qquad \land\ \Delta S \mapsto sdlist \in \mathcal{C}_{SD}\ ^{*} \\
\qquad \land\ \Delta PF \mapsto pflist \in \mathcal{C}_{PFD}\ ^{*} \bullet \\
\quad \theta ModuleHdr \mapsto \\
\qquad \text{“}\mathbf{MODULE}\text{”} \frown ids \frown \text{“}\mathbf{;}\text{”} \\
\qquad\quad \frown importlist \frown sdlist \frown pflist\ \}
\end{array}
$$

3.9.4 Module

A Module is a header, an export declaration, and a terminating full stop.

$$\mathcal{C}_M : Module \leftrightarrow String$$

$$\mathcal{C}_M = \{\ Module;\ e : EXPORT\_DECL;\ hdr, exportlist : String \mid \theta ModuleHdr \mapsto hdr \in \mathcal{C}_{MH} \land e \mapsto exportlist \in \mathcal{C}_{EX} \bullet \theta Module \mapsto hdr \frown exportlist \frown \text{“.”}\ \}$$

3.9.5 Main module

The main module text starts with the “**MAIN**” keyword, followed by the header, the body statement, and a terminating full stop.

Pasp follows Pascal in insisting that the program statement be a block. As there is no other reason to restrict Pasp in this way (though a program consisting of a single statement that is not a block is not very interesting), the restriction is enforced in the concrete syntax.

$$\mathcal{C}_{MA} : MainModule \leftrightarrow String$$

$$\mathcal{C}_{MA} = \{\ MainModule;\ hdr, stmt : String \mid \gamma \in \operatorname{ran} block \land \theta ModuleHdr \mapsto hdr \in \mathcal{C}_{MH} \land \gamma \mapsto stmt \in \mathcal{C}_S \bullet \theta MainModule \mapsto hdr \frown stmt \frown \text{“.”}\ \}$$

3.10 Program

There is no concrete syntax for programs in this version Pasp: all component files of the program are individual modules, and which modules comprise a program is determined by the linker.

(If there are no import declarations then the main module behaves as a flattened “**PROGRAM**” would.)

4 Overview of Pasp semantics

4.1 Introduction

There are four different semantics defined for a Pasp program. Three of these are static semantics and correspond to checks that can be performed at compile time. These are 'symbol declaration', 'type checking', and 'use after declaration'. The dynamic semantics defines the meaning of executing a program. Each semantics is defined only if all the previous ones (with the exception of the use after declaration semantics) have checked successfully. If a program fails to check successfully for the use after declaration semantics, a warning is generated, but the dynamic semantics are still defined. In addition, the syntax must be correct for any semantics to be defined.

Note that some of the semantics are undefined when certain conditions are not met. For example, the dynamic semantics of an assignment to a variable element is undefined if the element is out of range. This means that the behaviour in such circumstances can be defined to be whatever is appropriate to the use being made of the semantics. For the sake of proving equivalence of semantics it is more efficient to leave the behaviour undefined. However, an interpreter based on the dynamic semantics could check for such behaviour and signal it as an error. If such a check was included in the semantics, it would have to appear in the compiled code, which would have a severe impact on the execution speed.

The semantics allow the compiler to give the user a limited amount of information about the nature of errors in a program. Typically, the identifiers that are incorrectly declared, or variables whose type is incorrectly used, together with the program block in which the error occurred, are available. However, the precise position in the program at which the errors occurred is not available.

4.2 Syntax

If a program is not syntactically correct in its concrete form, then none of its semantics are defined. Any program that is successfully parsed to produce

an (abstract) *PROG* must be syntactically correct.

4.3 Symbol declaration semantics

The first check made on a syntactically correct program is a declaration check, to make sure that named constants, variables, arrays, procedures, and functions are declared before they are used. The semantics ensure that the identifier declaration is in the correct scope. The check also ensures that each identifier is declared at most once in any given scope. For example, the meaning function for an assignment statement is (informally):

$\mathcal{D}_E(assignment) =$
if (target identifier is declared and in scope)
$\wedge$ (source expression has no undeclared identifiers)
then check function unchanged
else map undeclared identifiers to *checkWrong*

Only if the whole program passes this check are the other semantics defined. Because this is the first semantic check all meaning functions are total on the domain of syntactically correct Pasp programs.

The output of this semantics is a check value for every (symbol identifier, program block) pair used in the program.

Note that the usage of identifiers is only checked in statements. Identifiers that are used in defining subrange types and array bounds are not checked in this semantics. If undeclared variables are used, then this is trapped in the type checking semantics.

4.4 Type checking semantics

The type checking semantics checks that types of symbols are compatible with their use in the program: for example, named constants and *readOnly* variables cannot be assigned to; procedures and functions must have the correct number and type of actual parameters. The meaning function $\mathcal{T}_P$ is defined only on programs that have been declaration checked.

The output of the semantics is a type for each symbol per block. If any symbol is *typeWrong* then an error is flagged, and none of the subsequent semantics is defined.

Named constants are treated in the same manner as *readOnly* variables for the purposes of this semantics. This is because both can be referenced, but neither can be assigned to. Variable declarations are checked to ensure that only named constants are used to define subrange types and array bounds. This traps both the use of variables and undeclared identifiers in such definitions.

Generally the meaning function for sequences is extended inductively from the meaning function for a sequence element. The type checking of sequences of expressions is a counterexample to this. This is because expression sequences are always used to reference array elements, and thus the type checking required is more complex.

4.5 Use after declaration semantics

This semantics checks each declared identifier to ensure that it is actually used in a program statement.

Variable names, procedure names, function names, type neames, and named constants are also checked. (Note that in the current definition of the semantics, the use of named constants in the declaration of variables and formal parameters is not noted.)

The checks are carried out optimistically in the sense that provided a variable is used somewhere in a program, then it checks alright, even if the branch might not be executed.

4.6 Dynamic semantics

The dynamic semantics defines the execution meaning of the program and thus yields an interpreter for the Pasp language. To make the interpreter usable in practice, some additional functionality is required for input and

output, an initial NVRAM values. A suitable mechanism would be to query the user for input and read the value whenever a *readOnly* variable is referenced. Similarly, a value would be output whenever a *writeOnly* variable is assigned to. This functionality is not included in the semantics, as then it would have to be reproduced in the compiled code.

5 States and Environments

5.1 Introduction

In this chapter we introduce some auxiliary definitions that are needed in order to define the semantics of Pasp in the following chapters.

5.2 Semantics — general

5.2.1 Environment and state

Rather than map variable names to their values directly, we introduce intermediate *locations*, and use an *environment* to map names to locations, and a *store* to map locations to associated values. A declaration alters the environment by adding a new location (or set of locations, in the case of arrays) to the environment, and the initial value to the store. (The initial value is given either on declaration, or by the initial state of NVRAM, or by the initial value of the input stream.) The changing state of a computation consists of this store together with the input and output. Input and output are modelled as *streams* at specific locations.

In the case of the first two static semantics – symbol declaration and type checking – the denotation of a variable is constant throughout the program statement: once declared it remains so, and once assigned a type it retains that type. So for these semantics, an environment, but no store, is required. With the other semantics – use after declaration, and dynamic a variable's state can change (when it is first used, and when it is assigned to, respectively). So each of these is modelled using an environment and a store.

5.2.2 Environment and block structuring

Pasp is a block structured language: it has local declarations in procedures and functions. It is relatively easy to dynamically alter the dynamic environment when entering and leaving blocks; however the static semantics need

to leave a record of the status of variables even after they have gone out of scope, in order to give sensible error reports. For example, if an undeclared variable is used in a procedure, then the program should not pass the symbol declaration check. This makes the definition of environments rather complex.[4]

Each scope has an associated environment. A generic environment is a relation between identifiers and generic values. (Environments are conventionally single valued, or functional. Here we allow some environments to be multi-valued. For example, in the use after declaration semantics, a variable could be both *unread* and *unwritten*.) We instantiate a different generic environment for each different semantics.

$$Env[X] == ID \leftrightarrow X$$

The full 'trace' environment[5] is a mapping from a block to the environment belonging to that block. Each block is labelled with the name of the surrounding procedure or function; the outermost (global) block is labelled with the module name.

$$EnvTr[X] == ID \pfun Env[X]$$

So blocks in a trace environment do not go out of scope. For example, in the body statement of a main module, a trace environment might look something like:

$$\begin{aligned}\rho = \{&main \mapsto \{v_1 \mapsto x_1, v_2 \mapsto x_2, f_1 \mapsto x_3, f_2 \mapsto x_4\},\\ &f_1 \mapsto \{v_1 \mapsto x_5, v_3 \mapsto x_6\},\\ &f_2 \mapsto \{v_3 \mapsto x_7, v_4 \mapsto x_8\}\}\end{aligned}$$

To model scope, we use a separate stack of block names, to keep track of those declarations that are currently in scope.

[4] Although Pasp allows only one level of nesting, the definitions of environments and static semantics have been made more general, so that if deeper nesting is added later, there will be as little impact on the compiler as possible. The most serious impact would probably be on the operational semantics (the code templates) and proof. In addition, the convention for labelling blocks would also have to change. At present, each block is labelled with the name of the procedure/function that contains it. With full block structuring, procedure identifiers need no longer be unique, but a sequence of names could be used instead.

[5] so called because it allows the tracing of errors in all blocks

$$STACK == \mathrm{seq}_1\ ID$$

In Pasp, this sequence has length 1 in the body statement of the main module, where it is $\langle mainModuleName \rangle$, and length 2 everywhere else, in a function or procedure, where it is $\langle moduleName, cmdName \rangle$. However, the utility definitions below are applicable to arbitrary length block stacks.

Applying a trace environment $\rho t : ID \pfun EnvX$ to a block stack B yields a sequence of environments $\rho t \circ B : seqEnvX$, where the later the entry in the sequence, the deeper the nesting (and the higher the priority of the definition). To extract the correct environment from the sequence, we use distributed function overriding, to makes later environments override earlier ones. Thus to extract the current environment, we use

$$\oplus/\ (\rho t \circ B)$$

findBlock returns the innermost block in which an identifier is declared, by searching for the identifier from the end of the block stack.

$$[X]$$

$$findBlock : ID \times STACK \times EnvTr[X] \pfun ID$$

$$\begin{aligned}
&\forall \xi : ID;\ B : STACK;\ \rho t : EnvTr[X] \mid \xi \in \mathrm{dom}(\oplus/\ (\rho t \circ B)) \bullet \\
&\quad \exists\, b == last\ B \bullet \\
&\qquad findBlock(\xi, B, \rho t) \\
&\qquad\quad = \mathbf{if}\ \xi \in \mathrm{dom}(\rho t\ b) \\
&\qquad\qquad \mathbf{then}\ b\ \mathbf{else}\ findBlock(\xi, front\ B, \rho t)
\end{aligned}$$

For example, in the environment ρ given above,

$$findBlock(v_1, \langle main, f_1 \rangle, \rho) = f_1$$
$$findBlock(v_1, \langle main, f_2 \rangle, \rho) = main$$

lookup returns the value of an identifier in the current environment.

$$[X]$$

$$lookup : ID \times STACK \times EnvTr[X] \pfun X$$

$$\begin{aligned}
&\forall \xi : ID;\ B : STACK;\ \rho t : EnvTr[X] \mid \xi \in \mathrm{dom}(\oplus/\ (\rho t \circ B)) \bullet \\
&\quad lookup(\xi, B, \rho t) = \rho t(findBlock(\xi, B, \rho t))\xi
\end{aligned}$$

For example, in the environment ρ given above,

$$lookup(v_1, \langle main, f_1 \rangle, \rho) = x_5$$
$$lookup(v_1, \langle main, f_2 \rangle, \rho) = x_1$$

update updates a single value of the environment corresponding to the last block of a block stack.

$$[X]$$
$$update : EnvTr[X] \times STACK \times Env[X] \pfun EnvTr[X]$$
$$\forall \rho t : EnvTr[X];\ B : STACK;\ \rho : Env[X] \bullet$$
$$\exists b == last\ B \bullet$$
$$update(\rho t, B, \rho) = \rho t \oplus \{b \mapsto (\rho t\ b \oplus \rho)\}$$

For example, in the environment ρ given above,

$$update(\rho, \langle main, f_1 \rangle, \{v_1 \mapsto x_9, v_5 \mapsto x_{10}\})$$
$$= \{main \mapsto \{v_1 \mapsto x_1, v_2 \mapsto x_2, f_1 \mapsto x_3, f_2 \mapsto x_4\},$$
$$f_1 \mapsto \{v_1 \mapsto x_9, v_3 \mapsto x_6, v_5 \mapsto x_{10}\},$$
$$f_2 \mapsto \{v_3 \mapsto x_7, v_4 \mapsto x_8\}\}$$

5.2.3 Static semantics — check status

The purpose of a static semantics is to say that a construct either conforms to the check or does not. One check value, *checkOK*, is defined for all semantics. Other values are defined as needed. For example, it may be useful to define a warning value to indicate that there may be an error, but not to stop the subsequent semantics from being defined.

$$CHECK == \mathbb{Z}$$
$$checkOK == 0$$

Check results need to be combined, for example when the check status of a binary expression is calculated. We define $\bowtie$ to be a function that combines check arguments 'pessimistically'; the result satisfies the check only if both arguments do.

function 30 **leftassoc**($\_ \bowtie \_$)

$$\_ \bowtie \_ : CHECK \times CHECK \rightarrow CHECK$$
$$\forall\, c, d : CHECK \bullet c \bowtie d = max\ \{c, d\}$$

Environments are combined when two execution branches are possible, and different environments are generated by each branch. We define $\square$ to be a function that combines maps to *CHECK* pessimistically.

function 30 **leftassoc**($\_ \square \_$)

$$[X]$$
$$\_ \square \_ : (X \pfun CHECK) \times (X \pfun CHECK) \rightarrow (X \pfun CHECK)$$
$$\forall\, f, g : X \pfun CHECK \bullet$$
$$f \square g = ((f \cup g) \rhd \{checkOK\}) \oplus \{\, x : \mathrm{dom}\, f \cap \mathrm{dom}\, g \bullet x \mapsto (f\ x \bowtie g\ x)\}$$

$\square$ is the union of the two environments everywhere except where they are *checkOK*, overridden with the pessimistic value of their commonly defined elements. So if one environment defines no check value for an element, then the combined environment has no check value for that element if the second environment has the value *checkOK*, but has the value of the second environment if this is not *checkOK*. For example

$$\rho = \{x_1 \mapsto checkOK, x_2 \mapsto 2, x_3 \mapsto 1\}$$
$$\rho' = \{x_1 \mapsto checkOK, x_3 \mapsto 3, x_4 \mapsto checkOK\}$$
$$\rho \square \rho' = \{x_1 \mapsto checkOK, x_2 \mapsto 2, x_3 \mapsto 3\}$$

Note that x_1 is in the combined environment, but x_4 is not.

(As $\square$ is used only for the use semantics, the *ImportOK* value required for the symbol declaration semantics does not affect this definition.)

$\square$ is associative.

$$[X]\ f, g, h : X \pfun CHECK \vdash (f \square g) \square h = f \square (g \square h)$$

For the cases where identifiers map to a set of check values, simple union (pessimistic) combination of the environments suffices.

function 30 **leftassoc**($\_ \diamond \_$)

$$
\begin{array}{|l}
\hline
[X] \\
\_ \diamond \_ : EnvTr[X] \times EnvTr[X] \pfun EnvTr[X] \\
\hline
\forall \rho, \rho' : EnvTr[X] \bullet \\
\quad \exists \rho\rho == \rho \cup \rho' \bullet \\
\quad\quad \rho \ \diamond \ \rho' = \{ \ b : \dom \rho\rho \bullet b \mapsto \bigcup(\rho\rho(\!| \ \{b\} \ |\!)) \ \} \\
\hline
\end{array}
$$

5.3 Symbol declaration semantics

5.3.1 Symbol declaration environment

The declaration environment maps identifiers to their declaration check status.

$EnvD == Env[CHECK]$

The declaration trace environment is a map from block names to declaration environments.

$EnvDTrace == EnvTr[CHECK]$

5.3.2 Symbol declaration check values

The declaration check values (in addition to *checkOK*) are:

$ImportOK == 1$
$Undecl == 2$
$MultiDecl == 3$
$ImportExport == 4$
$ImportUndecl == 5$

- *ImportOK*: used to tag identifiers declared by import, so that they can be differentiated from those declared within the module (only those declared within a module may be exported)
- *Undecl*: an identifier is referenced, but had not been declared
- *MultiDecl*: an identifier is declared more than once
- *ImportExport*: an imported identifier occurs in a export declaration
- *ImportUndecl*: an imported identifier has not been declared in another module (which is checked on linking, or on flattening modules to a program)

When combining declaration environments from different sources pessimistically, it would be possible to use $\Box$ defined above. However, this is unnecessarily complicated for this semantics. Declaration environments are combined only in the body of the program. In the body, the only change to the environment is to mark previously unmarked identifiers as *Undecl* (a bad value). Thus taking the union of the two environments is sufficient (and is equivalent to $\Box$ in this case).

5.3.3 Initial symbol declaration environment

The initial declaration environment contains the reserved word **MAXUNSIGNED**.

$$maxunsigned : ID$$

$$\rho\delta 0 : EnvD$$
$$\rho\delta 0 = \{maxunsigned \mapsto checkOK\}$$

(Aside: An alternative language semantics could allow multiple declarations with the same type, but disallow those with different types. There would be no check for multiple declarations in such a semantics, but an extra one would be needed in the corresponding type checking.)

5.3.4 Auxiliary functions

addSymbol adds a symbol to a declaration trace environment. This is used whenever a non-imported identifier is declared.

$$
\begin{array}{|l}
addSymbol : ID \to ID \to EnvDTrace \to EnvDTrace \\
\hline
\forall \xi, b : ID;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists c == \textbf{if } \xi \in \mathrm{dom}(\rho\delta t\ b) \textbf{ then } MultiDecl \textbf{ else } checkOK \bullet \\
\quad\quad addSymbol\ \xi\ b\ \rho\delta t = update(\rho\delta t, \langle b \rangle, \{\xi \mapsto c\}) \\
\end{array}
$$

If the identifier is not already present in the block b, it is added to the trace environment as *checkOK*. If it is already present, it is marked as multiply declared.

An import declaration has a different check OK state, because an export declaration should not include any identifiers that have been imported to the module. Therefore adding an imported identifier is slightly different.

$$
\begin{array}{|l}
addImportSymbol : ID \to ID \to EnvDTrace \to EnvDTrace \\
\hline
\forall \xi, b : ID;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists c == \textbf{if } \xi \in \mathrm{dom}(\rho\delta t\ b) \textbf{ then } MultiDecl \textbf{ else } ImportOK \bullet \\
\quad\quad addImportSymbol\ \xi\ b\ \rho\delta t = update(\rho\delta t, \langle b \rangle, \{\xi \mapsto c\}) \\
\end{array}
$$

If the identifier is not already present in the block b, it is added to the trace environment as *ImportOK*. If it is already present, it is marked as multiply declared.

checkSymbol checks that a symbol being referenced has been declared.

$$
\begin{array}{|l}
checkSymbol : ID \to STACK \to EnvDTrace \to EnvDTrace \\
\hline
\forall \xi : ID;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad checkSymbol\ \xi\ B\ \rho\delta t = \\
\quad\quad \textbf{if } \xi \in \mathrm{dom}(\oplus/\ (\rho\delta t \circ B)) \\
\quad\quad \textbf{then } \rho\delta t \textbf{ else } update(\rho\delta t, B, \{\xi \mapsto Undecl\}) \\
\end{array}
$$

If the identifier is in scope, *checkSymbol* has no effect. Otherwise it updates the environment to mark the identifier as undeclared.

5.4 Type checking semantics

5.4.1 Type checking environment

IDTYPE is used to store type and associated information for identifiers in a program. It includes more than just type information because it is also used by later semantics. For example, the interpreter needs to determine whether a variable is *readOnly* or *writeOnly* in order to carry out input and output; the dynamic semantics needs to know upper and lower bounds of array ranges in order to allocation the correct number of locations; the operational semantics needs to know the call type of formal parameters in order to reference the correct data address.

Three types use subranges for their values: variables, formal parameters, and function calls. A *SubrangeType* comprises a pair of values (the values of the lower and upper bounds of the subrange) and the type of the subrange (either byte or unsigned).

$$SubrangeN \mathrel{\widehat=} [\ lb, ub : \mathbb{N} \mid lb \ \leq ub\ ]$$

$$SubrangeType \mathrel{\widehat=} [\ SubrangeN;\ \tau : TYPE\ ]$$

Two types use array bounds: variables and formal parameters. An *ArrayType* comprises a sequence of pairs of values (the values of the lower and upper bounds of each dimension of the array) and the type of the array indexes (either byte or unsigned).

$$ArrayType \mathrel{\widehat=} [\ SR : \operatorname{seq} SubrangeN;\ \tau a : TYPE\ ]$$

$$[IDTYPE]$$

$$VarType \mathrel{\widehat=} [\ A : \mathbb{P}\, ATTR;\ ArrayType;\ SubrangeType\ ]$$
$$ValueType \mathrel{\widehat=} [\ v : VALUE;\ \tau : TYPE\ ]$$
$$FormalType \mathrel{\widehat=} [\ \xi : ID;\ c : CALL\_TYPE;\ A : \mathbb{P}\, ATTR;\ ArrayType;\ SubrangeType\ ]$$
$$EnumType \mathrel{\widehat=} [\ \tau : TYPE;\ \Xi : \operatorname{seq}_1 ID\ ]$$
$$ProcType \mathrel{\widehat=} [\ B : STACK;\ I : \operatorname{seq} IDTYPE\ ]$$
$$FunType \mathrel{\widehat=} [\ B : STACK;\ I : \operatorname{seq} IDTYPE;\ SubrangeType\ ]$$

$$IDTYPE ::= variable\langle\langle VarType\rangle\rangle \mid const\langle\langle ValueType\rangle\rangle$$
$$\mid formalParam\langle\langle FormalType\rangle\rangle \mid tempVar\langle\langle TYPE\rangle\rangle$$
$$\mid enumType\langle\langle EnumType\rangle\rangle \mid enumValue\langle\langle ValueType\rangle\rangle$$
$$\mid procedure\langle\langle ProcType\rangle\rangle \mid function\langle\langle FunType\rangle\rangle$$

- *variable*: a variable's attributes, array bounds and their type, and the variable's bounds and type.
- *const*: a named constant's value and type.
- *formalParam*: a formal parameter's name, call type, attributes, array bounds and their type, and the parameter's bounds and type.
- *tempVar*: a temporary variable used in the construction of case statements. They are generated automatically, and so are used correctly, and need not be checked in static semantics. They are always of an enumerated type.
- *enumType*: an enumerated type's type (its *typeName* or *scopedTypeName*, or *typeWrong*) and the list of enumerated value names (needed in type checking case statements).
- *enumValue*: an enumerated value's value (a byte) and its type (the name of the enumerated type in *scopedTypeValue*).
- *procedure*: $\langle modName, procName\rangle$ (a procedure's block stack at declaration time), and the $IDTYPE$s of the formal parameters.
- *function*: $\langle modName, funName\rangle$ (a function's block stack at declaration time), the $IDTYPE$s of the formal parameters, and the bounds and type of its return value.

The numerical values of the upper and lower bounds in subranges need to be calculated at some point. It might seem more appropriate to do this in the dynamic semantics; however these values are also required in the compiler, which may be run independently of the interpreter (which is based on the dynamic semantics). Hence it is better to calculate them in the type checking semantics. The type could still remain seq *Subrange*, but it is clearer to use the seq *SubrangeN* type.

Defined types have a scope. Two variables with user defined types (enumerated types) have the same type if the two type definitions are the same. This is the case if and only if the types have the same name and are defined in the same block. This information is stored as a *scoped type name*.

The environment for the type checking semantics is a map from identifiers to their $IDTYPE$s.

$$EnvT == Env[IDTYPE]$$

The type trace environment is

$$EnvTTrace == EnvTr[IDTYPE]$$

The type environment associates types with identifiers. The type checking semantics also determines whether expressions and statements are well-typed. There is no unique label for these syntactic constructs, so we associate a check status with each block, using an $Env[CHECK]$.

5.4.2 Type check values

The type check values (in addition to $checkOK$) are:

$$attributeWrong == 1$$
$$checkTypeWrong == 3$$

- $attributeWrong$: an erroneous set of attributes.
- $checkTypeWrong$: type error.

5.4.3 Auxiliary functions

$constType$ returns the type of a constant value.

$$constType : VALUE \pfun TYPE$$

$$\forall \kappa : \operatorname{ran} vunsgn \bullet constType\ \kappa = unsigned$$
$$\forall \kappa : \operatorname{ran} vbyte \bullet constType\ \kappa = pbyte$$
$$\forall \kappa : \operatorname{ran} vbool \bullet constType\ \kappa = boolean$$

There is no need for the definition to cover *stream* or *locn*, since these are not concrete Pasp values.

We define various 'extractor' functions for accessing information from the $IDTYPE$. (It is not necessary to define *arrayOf* and *typeOf* as total functions; the information is sometimes extracted in a different way.)

$$
\begin{array}{l}
attributeOf : IDTYPE \rightarrow \mathbb{P}\, ATTR \\
arrayOf : IDTYPE \nrightarrow \mathrm{seq}\, SubrangeN \\
typeOf : IDTYPE \nrightarrow TYPE \\
valueOf : IDTYPE \nrightarrow VALUE \\
arrayIndexType : IDTYPE \nrightarrow TYPE \\
\hline
\forall\, VarType \bullet \\
\quad attributeOf(variable\ \theta VarType) = A \\
\quad \land\ arrayOf(variable\ \theta VarType) = SR \\
\quad \land\ typeOf(variable\ \theta VarType) = \tau \\
\quad \land\ arrayIndexType(variable\ \theta VarType) = \tau a \\
\forall\, ValueType \bullet \\
\quad attributeOf(const\ \theta ValueType) = \{readOnly\} \\
\quad \land\ arrayOf(const\ \theta ValueType) = \langle\ \rangle \\
\quad \land\ typeOf(const\ \theta ValueType) = \tau \\
\quad \land\ valueOf(const\ \theta ValueType) = v \\
\forall\, FormalType \bullet \\
\quad attributeOf(formalParam\ \theta FormalType) = A \\
\quad \land\ arrayIndexType(formalParam\ \theta FormalType) = \tau a \\
\forall \tau : TYPE \bullet \\
\quad attributeOf(tempVar\ \tau) = \varnothing \\
\quad \land\ arrayOf(tempVar\ \tau) = \langle\ \rangle \\
\forall\, EnumType \bullet \\
\quad attributeOf(enumType\ \theta EnumType) = \varnothing \\
\quad \land\ arrayOf(enumType\ \theta EnumType) = \langle\ \rangle \\
\quad \land\ typeOf(enumType\ \theta EnumType) = \tau \\
\forall\, ValueType \bullet \\
\quad attributeOf(enumValue\ \theta ValueType) = \{readOnly\} \\
\quad \land\ arrayOf(enumValue\ \theta ValueType) = \langle\ \rangle \\
\quad \land\ typeOf(enumValue\ \theta ValueType) = \tau \\
\quad \land\ valueOf(enumValue\ \theta ValueType) = v \\
\forall\, ProcType \bullet attributeOf(procedure\ \theta ProcType) = \varnothing \\
\forall\, FunType \bullet attributeOf(function\ \theta FunType) = \varnothing
\end{array}
$$

In order to check the type of expression lists, we define an auxiliary function that merges two types, by setting the result to their common value if they

are equal, and giving *typeWrong* if they are not.

$$mergeTypes : TYPE \times TYPE \rightarrow TYPE$$

$$\forall \tau, \tau' : TYPE \bullet$$
$$mergeTypes(\tau, \tau') = \textbf{if } \tau = \tau' \textbf{ then } \tau \textbf{ else } typeWrong$$

5.4.4 Initial and final type environments

The type checking semantics is a convenient place to hold a 'constant table', which links named constants with their values and types. A special initial type environment is defined to contain the value and type of the reserved constant name.

$$\rho\tau 0 : EnvT$$

$$\rho\tau 0 = \{maxunsigned \mapsto const(\!| \ v == vunsgn(2 \uparrow 16), \tau == unsigned \ |\!)\}$$

The type environment defined for a Pasp program starts from this initial environment at the program level.

The final type trace environment is made global, for use in later semantics. Its value is defined by the type checking semantics of a module.

$$\rho\tau t 0 : EnvTTrace$$

5.5 Use semantics

5.5.1 Use checking environment

The environment for the use semantics is a map from identifiers to their *USECHECK* status.

$$USECHECK ::= unread \mid unwritten \mid unused \mid uncalled$$

- *unread*: readable variables (RAM, NVRAM and input), and constants, that are never referenced.

- *unwritten*: writable variables (RAM, NVRAM and output) that are never assigned to.
- *unused*: user defined types that are never used in variable declarations.
- *uncalled*: procedures and functions that are never called.

It is impossible to check statically whether a given enumerated value is used in a program, because of the *pred* and *succ* operators. Hence the use of such identifiers cannot be checked even in principle.

The use environment maps identifiers to their use check status. (Note that an identifier may have two use check statuses, *unread* and *unwritten*, if it has not been used at all, or may have no use check status, if it has been used properly.)

$$EnvU == Env[USECHECK]$$

The use trace environment is a map from block names to use environments.

$$EnvUTrace == EnvTr[USECHECK]$$

The definition of the use semantics is essentially straightforward. Essentially, only two constructs affect the status of the use environment: value reference, and assignment. All the other constructs just pass the checking down the syntax tree. The presence of function and procedure parameters complicates the use after declaration semantics slightly. If a formal parameter is call by reference, the parameter's use inside the procedure can affect the use of the referenced variable.

5.6 Dynamic semantics

In this section we define the environment and state for the dynamic semantics. We also define the mapping from array indices to memory locations.

5.6.1 Dynamic state and store

The store maps abstract locations to the values stored there (which dynamically change as a program executes):

$$Store == LOCN \pfun VALUE$$

The input and output from a program are modelled as streams attached to particular unchanging locations. Because a stream type is included in $VALUE$, the store as defined can model this. It is important to ensure that the locations that have streams attached correspond to *readOnly* (input) and *writeOnly* (output) variables, and that the values in the streams have the correct type (see the semantics of Program).

$$Input == \mathbb{P}\, LOCN$$
$$Output == \mathbb{P}\, LOCN$$

The state of computation is given by the store together with the input and output locations.

$$State \mathrel{\widehat=} [\, \Sigma : Store;\ i : Input;\ o : Output \,]$$

There is no semantics for constant declarations, as the constant part of the type environment is used. The *Input* and *Output* sets are determined once locations have been assigned by the variable declaration semantics. These locations are not altered during the program run, and thus the store is the only component of the state that changes during execution. This means that updates to the state could be carried out by a function on the state that altered the store only. Instead, the state components are explicitly written out to clarify the relationship with the interpreter.

Referencing an input value changes the store by altering the content of the input stream, and assigning to an output value changes the store by altering the content of the output stream, so expressions have side effects. Thus it is necessary to define an order of expression evaluation. It is strictly left to right (as in Pascal), with precedence established by bracketing. Thus the arguments for binary expressions are evaluated left to right; sequences of array index expressions are evaluated left to right; in an assignment statement, the

array index expressions on the left hand side are evaluated before the expression on the right hand side; actual parameters to functions and procedures are evaluated left to right.

It is necessary to extract the output streams from a store, because the meaning of a program is just the value of its output streams.

$$outOf : State \rightarrow Store$$

$$\forall State \bullet outOf\ \theta State = o \lhd \Sigma$$

5.6.2 Dynamic environment

Denotable values (values that an identifier can stand for) are:

$$ExprValue \mathrel{\widehat=} [\ State;\ v : VALUE\ ]$$

$$\begin{aligned} DENVALUE ::= {} & loc\langle\langle \mathrm{seq}\,\mathbb{N} \rightarrowtail LOCN \rangle\rangle \mid cval \\ & \mid pval\langle\langle State \nrightarrow State \rangle\rangle \\ & \mid fval\langle\langle State \nrightarrow ExprValue \rangle\rangle \end{aligned}$$

- *loc*: variables and formal parameters. A function from array index values to their locations. The injective nature of the function ensures that every distinct sequence of array indices maps to a distinct location. (That these locations do not overlap locations of other variables is ensured by the meaning of variable declaration.) The length of the sequence of values corresponds to the dimension of the array. A zero dimensional array corresponds to an ordinary scalar variable.
- *cval*: constants. A single value, its sole function is to ensure that dynamic environments are total functions on all the identifiers in scope. This ensures that the dynamic environment accurately reflects the actual variables in scope, although the semantics are correctly defined without this, because the type trace environment is used first to determine whether an identifier is a named constant, and to store its value.

- *pval*: procedures. The state transformation denoted by the procedure body.
- *fval*: functions. The state transformation denoted by the function body, and the return value.

The dynamic environment is a mapping from identifiers to the values they denote.

$$EnvM == Env[DENVALUE]$$

The strategy used for array allocation is discussed in the following subsection 'Calculation of location offsets'.

The dynamic trace environment associates an environment with each block in a program.

$$EnvMTrace == EnvTr[DENVALUE]$$

5.6.3 Calculation of location offsets

To simplify the correctness proofs, the dynamic semantics have been defined so that the location of a call by reference parameter is stored at a specific location, and references to the formal parameter inside the procedure indirect to this location. This implies that the location of every array element (should the variable be an array) must be computable from the start address and array index. Hence the dynamic environment allocates locations for arrays in the same fixed way as the operational environment. All the array elements are in contiguous locations, and they are indexed so that the last index varies the fastest.

For example, in the array $a[1..4, 2..4, 0..1]$ the elements $a[1, 2, 0]$, $a[1, 2, 1]$, and $a[1, 3, 0]$ are stored at location offsets from the location of the first element of the array (that is $a[1, 2, 0]$) of 0, 1, and 2, respectively.

numElts returns the number of elements in an array given a sequence defining the values of the upper and lower bounds on each dimension. (There is no assumption that array bounds start at zero.)

$$
\begin{array}{|l}
numElts : \operatorname{seq} SubrangeN \rightarrow \mathbb{N} \\
\hline
numElts\langle\,\rangle = 1 \\
\forall\, s : \operatorname{seq} SubrangeN;\ SubrangeN \bullet \\
\qquad numElts(\langle \theta Subrange \rangle \frown s) = (ub - lb + 1) * numElts\ s
\end{array}
$$

For example, the size of the array $a[1..4, 2..4, 0..1]$ is
$(4 - 1 + 1) * (4 - 2 + 1) * (1 - 0 + 1) = 4 * 3 * 2 = 24$.

locationOffset gives the offsets of array elements.

$$
\begin{array}{|l}
locationOffset : \operatorname{seq}\mathbb{N} \times \operatorname{seq} SubrangeN \nrightarrow \mathbb{N} \\
\hline
\forall\, si : \operatorname{seq}\mathbb{N};\ ssr : \operatorname{seq} SubrangeN \mid \\
\qquad \#si = \#ssr \\
\qquad \land\ (\ \forall\, n : 1\,..\,\#si \bullet (ssr\ n).lb \leq si\ n \leq (ssr\ n).ub\ ) \bullet \\
\quad \exists\, ss : \operatorname{seq}\mathbb{N};\ normi : \operatorname{seq}\mathbb{N} \mid \\
\qquad\quad \#ss = \#si = \#normi \\
\qquad\quad \land\ (\ \forall\, n : 1\,..\,\#si \bullet \\
\qquad\qquad ss\ n = numElts(((n + 1)\,..\,\#ssr) \upharpoonright ssr)\ ) \\
\qquad\quad \land\ (\ \forall\, n : 1\,..\,\#si \bullet normi\ n = si\ n - (ssr\ n).lb\ ) \bullet \\
\qquad locationOffset(si, ssr) = dotProduct(ss, normi)
\end{array}
$$

Here *si* is the sequence of array indices for this element and *ssr* is the same-length sequence of subranges (upper and lower bounds) for the array.

Partial specification: Pasp's array indexing is checked dynamically, and the consequence of an index outside a subrange is *undefined*.

The effect of the $\upharpoonright$ function is to produce a new sequence of upper and lower bounds by including only the specified elements of the old sequence, so *ss* is a sequence of sub-array sizes. *normi* is a sequence of normalised array indices, the indices the array element would have if the array lower bounds were zero.

For example, the offset of element $a[3, 4, 1]$ in the array $a[1..4, 2..4, 0..1]$ is

$$
\begin{aligned}
&locnOffset\langle 3, 4, 1\rangle\langle (1, 4), (2, 4), (0, 1)\rangle \\
&\quad = dotProduct(\langle numElts\langle (2, 4), (0, 1)\rangle, numElts\langle (0, 1)\rangle, numElts\langle\ \rangle\rangle, \\
&\qquad\qquad \langle 3 - 1, 4 - 2, 1 - 0\rangle) \\
&\quad = dotProduct(\langle 6, 2, 1\rangle, \langle 2, 2, 1\rangle) \\
&\quad = 17
\end{aligned}
$$

6 Operators

6.1 Binary Operators

6.1.1 Binary Operators – Symbol declaration semantics

Binary operators are built into the language, so are all implicitly declared. No checking is necessary.

6.1.2 Binary Operators – Type checking semantics

The meaning function $\mathcal{T}_{BO}$ maps binary operators to functions from the types of the arguments to the type of the result.

Arithmetic operators take numbers and return numbers.

$$
\begin{array}{|l}
\mathcal{T}_{BO} : BIN\_OP \to TYPE \times TYPE \to TYPE \\
\hline
\forall \tau, \tau' : TYPE;\ \Omega : \{uplus, uminus, umul, udiv, umod\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\quad\quad \textbf{if}\ \tau = \tau' = unsigned\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE;\ \Omega : \{bplus, bminus, bmul, bdiv, bmod\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\quad\quad \textbf{if}\ \tau = \tau' = pbyte\ \textbf{then}\ pbyte\ \textbf{else}\ typeWrong
\end{array}
$$

Comparison operators take numbers and return *booleans*.

$$
\begin{array}{l}
\forall \tau, \tau' : TYPE;\ \Omega : \{ueq, une, ult, ule, ugt, uge\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = unsigned\ \textbf{then}\ boolean\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE;\ \Omega : \{beq, bne, blt, ble, bgt, bge\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = pbyte\ \textbf{then}\ boolean\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE;\ \Omega : \{eeq, ene\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \{\tau, \tau'\} \subseteq \operatorname{ran} scopedTypeValue \land \tau = \tau' \\
\qquad \textbf{then}\ boolean\ \textbf{else}\ typeWrong
\end{array}
$$

Logical operators take *unsigneds*, *bytes* and *booleans* and return values of the same type.

$$
\begin{array}{l}
\forall \tau, \tau' : TYPE;\ \Omega : \{uand, uor, uxor\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = unsigned\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE;\ \Omega : \{band, bor, bxor\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = pbyte\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE;\ \Omega : \{and, or\} \bullet \\
\quad \mathcal{T}_{BO}\ \Omega(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = boolean\ \textbf{then}\ boolean\ \textbf{else}\ typeWrong
\end{array}
$$

Casting operators change types.

$$
\begin{array}{l}
\forall \tau, \tau' : TYPE \bullet \\
\quad \mathcal{T}_{BO}\ join(\tau, \tau') = \\
\qquad \textbf{if}\ \tau = \tau' = pbyte\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\forall \tau, \tau' : TYPE \bullet \\
\quad \mathcal{T}_{BO}\ byteToEnum(\tau, \tau') = \\
\qquad \textbf{if}\ \tau \in \operatorname{ran} scopedTypeName \land \tau' = pbyte \\
\qquad \textbf{then}\ \tau\ \textbf{else}\ typeWrong
\end{array}
$$

6.1.3 Binary Operators – Use semantics

Binary operators are built into the language, so are all implicitly declared, but do not have to be used. No checking is necessary.

6.1.4 Binary Operators – Dynamic semantics

The dynamic meaning of a binary operator is the corresponding mathematical operator, as given below.

The unsigned arithmetic operators map to the same Z mathematical operators, preserving the unsigned type of the final result.

$$\mathcal{M}_{BO} : BIN\_OP \rightarrow VALUE \times VALUE \nrightarrow VALUE$$

$$\begin{aligned}
&\forall\, n, m : UNSIGNED \mid n + m \in UNSIGNED \bullet \\
&\quad \mathcal{M}_{BO}\ uplus(vunsgn\ n, vunsgn\ m) = vunsgn(n + m) \\
&\forall\, n, m : UNSIGNED \mid n - m \in UNSIGNED \bullet \\
&\quad \mathcal{M}_{BO}\ uminus(vunsgn\ n, vunsgn\ m) = vunsgn(n - m) \\
&\forall\, n, m : UNSIGNED \mid n * m \in UNSIGNED \bullet \\
&\quad \mathcal{M}_{BO}\ umul(vunsgn\ n, vunsgn\ m) = vunsgn(n * m) \\
&\forall\, n, m : UNSIGNED \mid m \neq 0 \bullet \\
&\quad \mathcal{M}_{BO}\ udiv(vunsgn\ n, vunsgn\ m) = vunsgn(n \operatorname{div} m) \\
&\quad \land\ \mathcal{M}_{BO}\ umod(vunsgn\ n, vunsgn\ m) = vunsgn(n \bmod m)
\end{aligned}$$

Partial specification: Pasp's unsigneds have constrained ranges, and the consequence of overflow or underflow is *undefined*. For example, if $n + m > \#WORD$, the definition of *uplus* is undefined at this point. Also undefined is the effect of division by zero. Where the semantics is undefined in this manner, an implementation is allowed to do anything. The implemented interpreter raises an error. The compiled code just silently does something wrong (because checking is too expensive to implement there).

The byte arithmetic operators map to the same Z mathematical operators.

$$
\begin{array}{l}
\forall\, b, c : BYTE \mid b + c \in BYTE \bullet \\
\quad \mathcal{M}_{BO}\ bplus(vbyte\ b, vbyte\ c) = vbyte(b + c) \\
\forall\, b, c : BYTE \mid b - c \in BYTE \bullet \\
\quad \mathcal{M}_{BO}\ bminus(vbyte\ b, vbyte\ c) = vbyte(b - c) \\
\forall\, b, c : BYTE \mid b * c \in BYTE \bullet \\
\quad \mathcal{M}_{BO}\ bmul(vbyte\ b, vbyte\ c) = vbyte(b * c) \\
\forall\, b, c : BYTE \mid c \neq 0 \bullet \\
\quad \mathcal{M}_{BO}\ bdiv(vbyte\ b, vbyte\ c) = vbyte(b \operatorname{div} c) \\
\quad \land\ \mathcal{M}_{BO}\ bmod(vbyte\ b, vbyte\ c) = vbyte(b \bmod c)
\end{array}
$$

Partial specification: Pasp's bytes have constrained ranges, and the consequence of overflow or underflow is *undefined*. Also undefined is the effect of division by zero.

Pasp's unsigned comparison operators map to the corresponding Z operations, returning a boolean result.

$$
\begin{array}{l}
\forall\, n, m : UNSIGNED \bullet \\
\quad \mathcal{M}_{BO}\ ueq(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n = m \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ une(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n \neq m \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ ult(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n < m \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ ule(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n \leq m \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ ugt(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n > m \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ uge(vunsgn\ n, vunsgn\ m) = \\
\qquad vbool(\textbf{if } n \geq m \textbf{ then } ptrue \textbf{ else } pfalse)
\end{array}
$$

Similarly for byte comparison operators:

$$
\begin{array}{l}
\forall\, b, c : BYTE \bullet \\
\quad \mathcal{M}_{BO}\ beq(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b = c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ bne(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b \neq c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ blt(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b < c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ ble(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b \leq c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ bgt(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b > c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ bge(vbyte\ b, vbyte\ c) = \\
\qquad vbool(\textbf{if } b \geq c \textbf{ then } ptrue \textbf{ else } pfalse)
\end{array}
$$

Enumerated values are represented by their (byte) position in the sequence of values for a type. (We know from type checking that $bmax = cmax$.)

$$
\begin{array}{l}
\forall\, bmax, b, cmax, c : BYTE \bullet \\
\quad \mathcal{M}_{BO}\ eeq(venum(bmax, b), venum(cmax, c)) = \\
\qquad vbool(\textbf{if } b = c \textbf{ then } ptrue \textbf{ else } pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ ene(venum(bmax, b), venum(cmax, c)) = \\
\qquad vbool(\textbf{if } b \neq c \textbf{ then } ptrue \textbf{ else } pfalse)
\end{array}
$$

Pasp's unsigned logical operators perform 'bitwise' operations. The form of the formal definitions is designed to be compatible with that required for the Asp specification.

$$
\begin{array}{l}
\forall\, n, m : UNSIGNED \bullet \\
\quad \mathcal{M}_{BO}\ uand(vunsgn\ n, vunsgn\ m) = \\
\qquad vunsgn(BitsToNat(NatToBits\ n\ AND\ NatToBits\ m)) \\
\quad \land\ \mathcal{M}_{BO}\ uor(vunsgn\ n, vunsgn\ m) = \\
\qquad vunsgn(BitsToNat(NatToBits\ n\ OR\ NatToBits\ m)) \\
\quad \land\ \mathcal{M}_{BO}\ uxor(vunsgn\ n, vunsgn\ m) = \\
\qquad vunsgn(BitsToNat(NatToBits\ n\ XOR\ NatToBits\ m))
\end{array}
$$

Similarly for byte logical operators:

$$
\begin{array}{l}
\forall\, b, c : BYTE \bullet \\
\quad \mathcal{M}_{BO}\ band(vbyte\ b, vbyte\ c) = \\
\qquad vbyte(BitsToNat(NatToBits\ b\ AND\ NatToBits\ c)) \\
\quad \land\ \mathcal{M}_{BO}\ bor(vbyte\ b, vbyte\ c) = \\
\qquad vbyte(BitsToNat(NatToBits\ b\ OR\ NatToBits\ c)) \\
\quad \land\ \mathcal{M}_{BO}\ bxor(vbyte\ b, vbyte\ c) = \\
\qquad vbyte(BitsToNat(NatToBits\ b\ XOR\ NatToBits\ c))
\end{array}
$$

Pasp's boolean logical operators map to the corresponding Z operations, returning a boolean result.

$$
\begin{array}{l}
\forall\, b, c : BOOLEAN \bullet \\
\quad \mathcal{M}_{BO}\ and(vbool\ b, vbool\ c) = \\
\qquad vbool(\textbf{if}\ b = ptrue \land c = ptrue\ \textbf{then}\ ptrue\ \textbf{else}\ pfalse) \\
\quad \land\ \mathcal{M}_{BO}\ or(vbool\ b, vbool\ c) = \\
\qquad vbool(\textbf{if}\ b = ptrue \lor c = ptrue\ \textbf{then}\ ptrue\ \textbf{else}\ pfalse)
\end{array}
$$

Pasp's cast operators:

$$
\begin{array}{l}
\forall\, bhi, blo : BYTE \bullet \\
\quad \mathcal{M}_{BO}\ join(vbyte\ bhi, vbyte\ blo) = vunsgn(bhi * 256 + blo) \\
\forall\, bmax, b : BYTE \mid b \leq bmax \bullet \\
\quad \mathcal{M}_{BO}\ byteToEnum(vbyte\ bmax, vbyte\ b) = venum(bmax, b)
\end{array}
$$

Partial specification: Pasp's enumerated types have constrained ranges, and the consequence of casting too large a byte is *undefined*.

6.2 Unary Operators

6.2.1 Unary Operators – Symbol declaration semantics

Unary operators are built into the language, so are all implicitly declared. No checking is necessary.

6.2.2 Unary Operators – Type checking semantics

The meaning function $\mathcal{T}_{UO}$ maps unary operators to functions from the type of the arguments to the type of the result.

The casting operators map between the types.

$$
\begin{array}{|l}
\mathcal{T}_{UO} : UNY\_OP \to TYPE \to TYPE \\
\hline
\forall \tau : TYPE;\ \Psi : \{unot, uleft, uright\} \bullet \\
\quad \mathcal{T}_{UO}\ \Psi\ \tau = \textbf{if}\ \tau = unsigned\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\forall \tau : TYPE;\ \Psi : \{bnot, bleft, bright\} \bullet \\
\quad \mathcal{T}_{UO}\ \Psi\ \tau = \textbf{if}\ \tau = pbyte\ \textbf{then}\ pbyte\ \textbf{else}\ typeWrong \\
\forall \tau : TYPE \bullet \\
\quad \mathcal{T}_{UO}\ not\ \tau = \textbf{if}\ \tau = boolean\ \textbf{then}\ boolean\ \textbf{else}\ typeWrong \\
\forall \tau : TYPE;\ \Psi : \{unsgnToByte, loByte, hiByte\} \bullet \\
\quad \mathcal{T}_{UO}\ \Psi\ \tau = \textbf{if}\ \tau = unsigned\ \textbf{then}\ pbyte\ \textbf{else}\ typeWrong \\
\forall \tau : TYPE \bullet \\
\quad \mathcal{T}_{UO}\ byteToBool\ \tau = \\
\qquad \textbf{if}\ \tau = pbyte\ \textbf{then}\ boolean\ \textbf{else}\ typeWrong \\
\quad \land\ \mathcal{T}_{UO}\ boolToByte\ \tau = \\
\qquad \textbf{if}\ \tau = boolean\ \textbf{then}\ pbyte\ \textbf{else}\ typeWrong \\
\quad \land\ \mathcal{T}_{UO}\ byteToUnsgn\ \tau = \\
\qquad \textbf{if}\ \tau = pbyte\ \textbf{then}\ unsigned\ \textbf{else}\ typeWrong \\
\quad \land\ \mathcal{T}_{UO}\ ord\ \tau = \\
\qquad \textbf{if}\ \tau \in \mathrm{ran}\ scopedTypeValue\ \textbf{then}\ pbyte\ \textbf{else}\ typeWrong \\
\forall \tau : TYPE;\ \Psi : \{succ, pred\} \bullet \\
\quad \mathcal{T}_{UO}\ \Psi\ \tau = \textbf{if}\ \tau \in \mathrm{ran}\ scopedTypeValue\ \textbf{then}\ \tau\ \textbf{else}\ typeWrong
\end{array}
$$

6.2.3 Unary Operators – Use semantics

Binary operators are built into the language, so are all implicitly declared, but do not have to be used. No checking is necessary.

6.2.4 Unary Operators – Dynamic semantics

Pasp's unary operators similarly map to the corresponding Z operators. Note that shift left provides a (fast) *non-overflowing* multiplication by 2.

$$\mathcal{M}_{UO} : UNY\_OP \rightarrow VALUE \nrightarrow VALUE$$

$$\begin{aligned}
&\forall\, n : UNSIGNED \bullet \\
&\quad \mathcal{M}_{UO}\ unot(vunsgn\ n) = vunsgn(\#UNSIGNED - n) \\
&\quad \land\ \mathcal{M}_{UO}\ uleft(vunsgn\ n) = vunsgn((n * 2) \bmod \#UNSIGNED) \\
&\quad \land\ \mathcal{M}_{UO}\ uright(vunsgn\ n) = vunsgn(n \operatorname{div} 2) \\
&\forall\, b : BYTE \bullet \\
&\quad \mathcal{M}_{UO}\ bnot(vbyte\ b) = vbyte(\#BYTE - b) \\
&\quad \land\ \mathcal{M}_{UO}\ bleft(vbyte\ b) = vbyte((b * 2) \bmod \#BYTE) \\
&\quad \land\ \mathcal{M}_{UO}\ bright(vbyte\ b) = vbyte(b \operatorname{div} 2) \\
&\forall\, b : BOOLEAN \bullet \\
&\quad \mathcal{M}_{UO}\ not(vbool\ b) = vbool(\textbf{if}\ b = ptrue\ \textbf{then}\ pfalse\ \textbf{else}\ ptrue)
\end{aligned}$$

The only tricky definition is unsigned logical not, *unot*. We use the result that $n + not\ n = \#UNSIGNED$.

$$\begin{aligned}
&\forall\, n : UNSIGNED \mid n \in BYTE \bullet \\
&\quad \mathcal{M}_{UO}\ unsgnToByte(vunsgn\ n) = vbyte\ n \\
&\forall\, n : UNSIGNED \bullet \\
&\quad \mathcal{M}_{UO}\ loByte(vunsgn\ n) = vbyte(n \bmod 256) \\
&\quad \land\ \mathcal{M}_{UO}\ hiByte(vunsgn\ n) = vbyte(n \operatorname{div} 256) \\
&\forall\, b : BOOLEAN \bullet \\
&\quad \mathcal{M}_{UO}\ boolToByte(vbool\ b) = vbyte(\textbf{if}\ b = pfalse\ \textbf{then}\ 0\ \textbf{else}\ 1) \\
&\forall\, b : BYTE \bullet \\
&\quad \mathcal{M}_{UO}\ byteToBool(vbyte\ b) = vbool(\textbf{if}\ b = 0\ \textbf{then}\ pfalse\ \textbf{else}\ ptrue) \\
&\quad \land\ \mathcal{M}_{UO}\ byteToUnsgn(vbyte\ b) = vunsgn\ b \\
&\forall\, bmax, b : BYTE \bullet \\
&\quad \mathcal{M}_{UO}\ ord(venum(bmax, b)) = vbyte\ b
\end{aligned}$$

Partial specification: Pasp's bytes have constrained ranges, and the consequence of casting too large an unsigned is *undefined*.

$$
\begin{array}{l}
\forall\, bamx, b : BYTE \mid 0 < b \bullet \\
\quad \mathcal{M}_{UO}\ pred(venum(bmax, b)) = venum(bmax, b-1) \\
\forall\, bmax, b : BYTE \mid b < bmax \bullet \\
\quad \mathcal{M}_{UO}\ succ(venum(bmax, b)) = venum(bmax, b+1)
\end{array}
$$

Partial specification: Pasp's enumerated types have constrained ranges, and the consequence of overflow or underflow is *undefined*.

7 Expressions

7.1 Meaning function declarations

7.1.1 Symbol declaration semantics

The declaration trace environment is updated to reflect the status of identifiers used in the expression.

$$\mathcal{D}_E : EXPR \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

7.1.2 Type checking semantics

The meaning function $\mathcal{T}_E$ maps an expression to its $TYPE$ and its $ATTR$s, in the context of the type environment.

$$ExprType \mathrel{\widehat=} [\ \tau : TYPE;\ A : \mathbb{P}\, ATTR\ ]$$

$$\mathcal{T}_E : EXPR \pfun EnvT \pfun ExprType$$

The function is partial because it is defined only for expressions containing (singly) declared identifiers.

The attribute set is also needed because the read/write status of the expression is checked in statements (and expression lists), as well as its $TYPE$.

7.1.3 Use semantics

The use trace environment is updated to reflect the status of any variables with relevant attributes that are used in the expression.

$$\mathcal{U}_E : EXPR \pfun STACK \pfun EnvUTrace \pfun EnvUTrace$$

7.1.4 Dynamic semantics

Given a dynamic environment and state, the meaning of an expression is a value together with an updated state. The state may be altered by referencing a *readOnly* variable.

$$\mathcal{M}_E : EXPR \pfun STACK \pfun EnvMTrace \pfun State \fun ExprValue$$

7.2 Multiple expressions, as array index lists

An array index list occurs as **a[e1, . . . , en]**. The semantics of each individual index is evaluated as an expression, and the array index list semantics combines these individual semantics suitably.

7.2.1 Array index list – Symbol declaration semantics

The symbol declaration semantics for an array index list is derived from the one for a single expression using $\mathcal{D}_{E*}$.

7.2.2 Array index list – Type checking semantics

The type semantics merges the types and union the attributes of the component index expressions.

$$\mathcal{T}_{E^*} : \mathrm{seq}_1\, EXPR \pfun EnvT \pfun ExprType$$

$$\begin{aligned}
&\forall \epsilon : EXPR;\ \rho\tau : EnvT \bullet \mathcal{T}_{E^*}\langle\epsilon\rangle\rho\tau = \mathcal{T}_E\ \epsilon\ \rho\tau \\
&\forall E, E' : \mathrm{seq}_1\, EXPR;\ \rho\tau : EnvT \bullet \\
&\quad \exists\, ExprType;\ ExprType';\ ExprType'' \mid \\
&\qquad \theta ExprType = \mathcal{T}_{E^*}\ E\ \rho\tau \\
&\qquad \land\ \theta ExprType' = \mathcal{T}_{E^*}\ E'\ \rho\tau \\
&\qquad \land\ \tau'' = mergeTypes(\tau, \tau') \\
&\qquad \land\ A'' = A \cup A' \bullet \\
&\quad \mathcal{T}_{E^*}(E \frown E')\rho\tau = \theta ExprType''
\end{aligned}$$

The type and attribute of the index list is checked where the array reference occurs.

7.2.3 Array indexes – Use semantics

The symbol declaration semantics for an array index list is derived from the one for a single expression using $\mathcal{U}_{E*}$.

7.2.4 Array index list – Dynamic semantics

The dynamic meaning of an array index list is the corresponding sequence of values, evaluated left to right, together with a possibly altered state.

$$ExprSeqValue \mathrel{\widehat=} [\ State;\ V : \mathrm{seq}\ VALUE\ ]$$

$$\mathcal{M}_{E^*} : \mathrm{seq}\, EXPR \pfun STACK \pfun EnvMTrace \pfun State \fun ExprSeqValue$$

$$\begin{aligned}
&\forall B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
&\quad \mathcal{M}_{E^*} \langle\ \rangle B\ \rho t\ \sigma = (\!|\ \sigma == \sigma, V == \langle\ \rangle\ |\!) \\
&\forall \epsilon : EXPR;\ E : \mathrm{seq}\, EXPR;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
&\quad \exists ExprValue';\ ExprSeqValue'';\ ExprSeqValue''' \mid \\
&\qquad \theta ExprValue' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \sigma \\
&\qquad \land\ \theta ExprSeqValue'' = \mathcal{M}_{E^*}\ E\ B\ \rho t\ \theta State' \\
&\qquad \land\ \theta State''' = \theta State'' \\
&\qquad \land\ V''' = \langle v' \rangle \frown V'' \bullet \\
&\quad \mathcal{M}_{E^*}(\langle \epsilon \rangle \frown E) B\ \rho t\ \sigma = \theta ExprSeqValue'''
\end{aligned}$$

7.3 Multiple expressions, as actual parameter lists

Some semantics of expressions used as actual parameters differ from those of other uses of expressions. An actual parameter list occurs as **fp(e1, . . . , en)**. The semantics of each parameter expression is found, and the actual parameter list semantics combines these individual semantics suitably.

7.3.1 Actual parameter list – Symbol declaration semantics

The symbol declaration semantics of actual parameter lists are the same as those for array index lists.

$$\mathcal{D}_{AP} == \mathcal{D}_E$$

7.3.2 Actual parameter list – Type checking semantics

An actual parameter type check takes the actual parameter expression an *IdType* (that of the corresponding formal parameter declaration) and a type environment (that of the call where the actual parameter occurs). It returns a *CHECK* value.

$$\mathcal{T}_{AP} : EXPR \pfun IDTYPE \pfun EnvT \pfun CHECK$$

Type checking a call-by-value actual parameter (which is known to have no attributes, from declaration type checking):

$$\begin{array}{l} \forall \epsilon : EXPR;\ FormalType;\ \rho\tau : EnvT \mid c = val \land A = \varnothing \bullet \\ \quad \exists\, ExprType' \mid \theta ExprType' = \mathcal{T}_E\ \epsilon\ \rho\tau \bullet \\ \quad\quad \mathcal{T}_{AP}\ \epsilon(formalParam\ \theta FormalType)\rho\tau = \\ \quad\quad\quad \textbf{if}\ \tau' = \tau \land writeOnly \notin A' \\ \quad\quad\quad \textbf{then}\ checkOK\ \textbf{else}\ checkTypeWrong \end{array}$$

- $\theta ExprType'$: the type and attributes of a call-by-value actual parameter expression are calculated using the expression type semantics.
- $\tau' = \tau$: the actual parameter must have the same type as the corresponding formal parameter (found from the *IdType*). (That the value of the actual parameter is consistent with any formal parameter subrange is checked dynamically.)
- $writeOnly \notin A'$: the actual parameter must not be a *writeOnly* variable.

Type checking a call-by-reference actual parameter:

$$
\begin{array}{l}
\forall\, \epsilon : EXPR;\ FormalType;\ \rho\tau : EnvT \mid \\
\qquad c = ref \\
\qquad \land \neg\, (\epsilon \in \operran\, valueRef \land (valueRef^{\sim}\ \epsilon).E = \langle\ \rangle) \bullet \\
\quad \mathcal{T}_{AP}\ \epsilon(formalParam\ \theta FormalType)\rho\tau = checkTypeWrong \\
\forall\, \epsilon : EXPR;\ FormalType;\ \rho\tau : EnvT \mid \\
\qquad c = ref \\
\qquad \land\, \epsilon \in \operran\, valueRef \land (valueRef^{\sim}\ \epsilon).E = \langle\ \rangle \bullet \\
\quad \exists\, \xi' : ID;\ FormalType';\ \chi : CHECK \mid \\
\qquad\quad \xi' = (valueRef^{\sim}\ \epsilon).\xi \\
\qquad\quad \land\, \rho\tau\ \xi' \in \{variable\ \theta VarType', formalParam\ \theta FormalType'\} \\
\qquad\quad \land\, \chi = \textbf{if}\ A' \cap \{readOnly, writeOnly\} \in \{\varnothing, A\} \\
\qquad\qquad\qquad \land\, \theta ArrayType' = \theta ArrayType \\
\qquad\qquad\qquad \land\, \theta SubrangeType' = \theta SubrangeType \\
\qquad\qquad\quad \textbf{then}\ checkOK\ \textbf{else}\ checkTypeWrong \bullet \\
\qquad \mathcal{T}_{AP}\ \epsilon(formalParam\ \theta FormalType)\rho\tau = \chi
\end{array}
$$

In addition to the call-by-value requirements, a call-by-reference parameter has further constraints.

- $\epsilon \in \operran\, valueRef \ldots$: the actual parameter cannot be a general expression, but must be a scalar variable reference (a *valueRef* with an empty expression list).
- $\xi' = \ldots$: the name of the variable reference
- $\rho\tau\ \xi' \in \ldots$: the name must be either a variable name or a formal parameter name (so not a function, procedure, constant or enumerated name)
- $(Aa, asra, sra)$: hence the type and attributes of a call-by-reference actual parameter can be looked up in the type environment.
- $A \cap \ldots$: the actual parameter must have either no read/write attributes, or the same one as the formal parameter.

- $asra = asr$: the array bounds of the actual and formal parameters must match, and the type of the array indexing must be the same for both actual and formal parameters.
- $sra = sr$: the subrange values of the actual and formal parameters must match, and the type of the subrange must be the same for both actual and formal parameters.

To check lists of actual parameters the individual check values are combined pessimistically.

$$
\begin{array}{|l}
\mathcal{T}_{AP^*} : \operatorname{seq} EXPR \pfun \operatorname{seq} IDTYPE \pfun EnvT \pfun CHECK \\
\hline
\forall\, \rho\tau : EnvT \bullet \mathcal{T}_{AP^*} \langle\,\rangle \langle\,\rangle\ \rho\tau = checkOK \\
\forall\, E : \operatorname{seq} EXPR;\ \epsilon : EXPR;\ I : \operatorname{seq} IDTYPE;\ i : IDTYPE;\ \rho\tau : EnvT \mid \\
\qquad \#E = \#I \bullet \\
\qquad \exists\, c, c', c'' : CHECK \mid \\
\qquad\qquad c = \mathcal{T}_{AP^*}\ E\ I\ \rho\tau \\
\qquad\qquad \land\ c' = \mathcal{T}_{AP}\ \epsilon\ i\ \rho\tau \\
\qquad\qquad \land\ c'' = c \bowtie c' \bullet \\
\qquad \mathcal{T}_{AP^*}(\langle\epsilon\rangle \frown E)(\langle i\rangle \frown I)\rho\tau = c''
\end{array}
$$

7.3.3 Actual parameter list – Use semantics

The use semantics of an actual parameter depends on the current status of the expression and on the call type of the formal parameter.

An actual parameter use check takes the actual parameter expression, the *IDTYPE* of the corresponding formal parameter declaration, the call-time blockstack, the block that is the called procedure or function name, and the call-time use environment. It returns an updated use environment.

A call-by-value parameter is checked like an expression. A call-by-reference parameter takes the use status of the formal parameter into account. Recall that the arguments include

- ξ: the name of the formal parameter

- b: the block where the formal parameter is declared (that is, the name of the function or procedure)

$$
\begin{array}{|l}
\mathcal{U}_{AP} : EXPR \pfun IDTYPE \pfun \\
\qquad STACK \pfun ID \pfun EnvUTrace \pfun EnvUTrace \\
\hline
\forall \epsilon : EXPR;\ FormalType;\ B : STACK;\ b : ID;\ \rho\upsilon t : EnvUTrace \mid \\
\qquad c = val \bullet \\
\quad \exists \rho\upsilon t' : EnvUTrace \mid \rho\upsilon t' = \mathcal{U}_E\ \epsilon\ B\ \rho\upsilon t \bullet \\
\qquad \mathcal{U}_{AP}\ \epsilon(formalParam\ \theta FormalType)B\ b\ \rho\upsilon t = \rho\upsilon t' \\
\forall \epsilon : EXPR;\ FormalType;\ B : STACK;\ b : ID;\ \rho\upsilon t : EnvUTrace \mid \\
\qquad c = ref \bullet \\
\quad \exists \xi', b' : ID;\ \chi, \chi' : \mathbb{P}\ USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad\quad \xi' = (valueRef^{\sim}\ \epsilon).\xi \\
\qquad\quad \land\ b' = findBlock(\xi', B, \rho\upsilon t) \\
\qquad\quad \land\ \chi' = (\rho\upsilon t\ b')(\!|\ \{\xi'\}\ |\!) \land \chi = (\rho\upsilon t\ b)(\!|\ \{\xi\}\ |\!) \\
\qquad\quad \land\ \rho\upsilon = \{\xi'\} \times \chi \cap \chi' \bullet \\
\qquad \mathcal{U}_{AP}\ \epsilon(formalParam\ \theta FormalType)B\ b\ \rho\upsilon t = \\
\qquad\quad update(\rho\upsilon t, \langle b' \rangle, \rho\upsilon)
\end{array}
$$

- ξa: the name of the variable that is the actual parameter.
- ba: the block where the actual parameter is declared (a call-by-reference parameter can only be a variable name: this is checked in the type checking semantics).
- χa: the initial use check status of the actual reference parameter.
- χ: the use check status of the formal reference parameter.
- χ': the final use check status of the actual reference parameter, optimistically combining (intersecting) the use statuses of the two cases.
- *update*: the use environment is updated so that the use check of the actual parameter is χ.

To move from single actual parameters to lists of actual parameters we use $\mathcal{U}_{AP*}$. Note that, as with sequences of expressions, the order is important, and parameter lists are evaluated left to right.

7.3.4 Actual parameter – Dynamic semantics

The dynamic semantics of an actual parameter consist of binding the correct value to the formal parameter. This is performed by storing the actual parameter's value at the correct location in the store.

The actual parameter dynamic semantics takes the actual parameter expression, the *IDTYPE* of the corresponding formal parameter declaration, the call-time blockstack, the block that is the called procedure or function name, and the call-time environment and state. It returns an updated state.

Dynamic meaning of a call-by-value actual parameter:

$$
\begin{array}{|l}
\mathcal{M}_{AP} : EXPR \pfun IDTYPE \pfun \\
\qquad STACK \pfun ID \pfun EnvMTrace \pfun State \pfun State \\
\hline
\forall \epsilon : EXPR;\ FormalType;\ b : ID;\ B : STACK; \\
\qquad\qquad \rho t : EnvMTrace;\ \sigma : State \mid \\
\qquad c = val \bullet \\
\quad \exists l : LOCN;\ ExprValue';\ State'' \mid \\
\qquad\qquad l = loc^{\sim}(\rho t\ b\ \xi)\langle\ \rangle \\
\qquad\qquad \land\ \theta ExprValue' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \sigma \\
\qquad\qquad \land\ \Sigma'' = \Sigma' \oplus \{l \mapsto v'\} \\
\qquad\qquad \land\ i'' = i' \land o'' = o' \bullet \\
\qquad lb \leq number\ v' \leq ub \Rightarrow \\
\qquad\qquad \mathcal{M}_{AP}\ \epsilon(formalParam\ \theta FormalType)B\ b\ \rho t\ \sigma = \theta State''
\end{array}
$$

For call by value, the stored value is the value of the actual parameter.

- b: the block where the formal parameter is declared (that is, the name of the function or procedure).
- ξ: the name of the formal parameter.
- $\rho t\ b\ \xi$: the meaning of the formal parameter: a mapping from indexes to locations. The parameter is call-by-value, so is a simple value not an array; the mapping is $\{\langle\ \rangle \mapsto l\}$.
- (σ', v): the after state and value corresponding to the expression meaning of the actual parameter.

- $\Sigma' \oplus \{l \mapsto v\}$: the after store, potentially changed to Σ' by evaluating the parameter expression, and overridden with the relevant location now containing the value of that expression.
- $nlo \leq number\ v \leq nhi$: if the actual value lies in the subrange, the dynamic semantics is as defined.

Dynamic meaning of a call-by-reference actual parameter:

$$
\begin{array}{l}
\forall \epsilon : EXPR;\ FormalType;\ B : STACK;\ b : ID; \\
\quad\quad \rho t : EnvMTrace;\ \sigma : State \mid \\
\quad c = ref \bullet \\
\quad \exists\, l : LOCN;\ \xi' : ID;\ idt : IDTYPE;\ f : \mathrm{seq}\,\mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow LOCN \\
\quad\quad v : VALUE;\ State;\ State' \mid \\
\quad\quad l = loc^{\sim}(\rho t\ b\ \xi)\langle\,\rangle \\
\quad\quad \land\ \xi' = (valueRef^{\sim}\ \epsilon).\xi \\
\quad\quad \land\ idt = lookup(\xi', B, \rho\tau t0) \\
\quad\quad \land\ f = loc^{\sim}(lookup(\xi', B, \rho t)) \\
\quad\quad \land\ \theta State = \sigma \\
\quad\quad \land\ v = \mathbf{if}\ idt \in \mathrm{ran}\, formalParam \\
\quad\quad\quad \mathbf{then}\ \Sigma(f\langle\,\rangle) \\
\quad\quad\quad \mathbf{else}\ pointer(f\{\ n : \mathrm{dom}\, SR \bullet n \mapsto (SR\ n).lb\ \}) \\
\quad\quad \land\ \Sigma' = \Sigma \oplus \{l \mapsto v\} \\
\quad\quad \land\ i' = i \land o' = o \bullet \\
\quad \mathcal{M}_{AP}\ \epsilon(formalParam\ \theta FormalType)B\ \ b\ \rho t\ \sigma = \theta State'
\end{array}
$$

For call by reference, the stored value is the start address of the variable.

- b: the block where the formal parameter is declared (that is, the name of the function or procedure).
- ξ: the name of the formal parameter.
- $\rho t\ b\ \xi$: the meaning of the formal parameter: a mapping from indexes to locations. The parameter is call-by-value, so is a simple value not an array; the mapping is $\{\langle\,\rangle \mapsto l\}$.
- ξa: the name of the variable that is the actual parameter.

- $\rho\tau\ \xi a$: the *IdType* of the actual parameter: either itself a call-by-reference formal parameter, or a variable.
- If the actual parameter is itself a formal parameter, the value to be stored is the location of the formal parameter, $\Sigma(f\ \langle\ \rangle)$.
- If the actual parameter is not itself a formal parameter, the value to be stored is a 'pointer' value containing the location of the first element of the referenced array, $f(\mathit{first} \circ \mathit{NN})$.
- $\Sigma \oplus \{l \mapsto v\}$: the store, overridden with the relevant location now containing the location of the reference expression.

7.4 Constant

7.4.1 Constant – Symbol declaration semantics

$$\forall \kappa : \mathit{VALUE};\ B : \mathit{STACK} \bullet \mathcal{D}_E(\mathit{constant}\ \kappa)B = \mathrm{id}\ \mathit{EnvDTrace}$$

A constant expression has no effect on the declaration environment.

7.4.2 Constant – Type checking semantics

$$\forall \kappa : \mathit{VALUE};\ \rho\tau : \mathit{EnvT} \bullet \\ \mathcal{T}_E(\mathit{constant}\ \kappa)\rho\tau = \langlebar\ \tau == \mathit{constType}\ \kappa, A == \varnothing\ \rangle\!\rangle$$

The type of a constant expression is given by the type of the value. Note that this does not depend on the environment.

7.4.3 Constant – Use semantics

$$\forall \kappa : \mathit{VALUE};\ B : \mathit{STACK} \bullet \mathcal{U}_E(\mathit{constant}\ \kappa)B = \mathrm{id}\ \mathit{EnvUTrace}$$

A constant expression has no effect on the use environment.

7.4.4 Constant – Dynamic semantics

$$\begin{array}{l}\forall\, \kappa : VALUE;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\ \quad \exists\, ExprValue \mid \theta State = \sigma \land v = \kappa \bullet \\ \quad\quad \mathcal{M}_E(constant\ \kappa) B\ \rho t\ \sigma = \theta ExprValue\end{array}$$

The value of a constant expression is the value of the constant. The state is unchanged.

7.5 Value reference

7.5.1 Value reference – Symbol declaration semantics

$$\begin{array}{l}\forall\, ValueRefExpr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\ \quad \exists\, \rho\delta t' : EnvDTrace \mid \rho\delta t' = \mathcal{D}_E\ {}_*\ E\ B\ \rho\delta t \bullet \\ \quad\quad \mathcal{D}_E(valueRef\ \theta ValueRefExpr) B\ \rho\delta t = \\ \quad\quad\quad checkSymbol\ \xi\ B\ \rho\delta t'\end{array}$$

The sequence of expressions comprising the array index list is checked, resulting in a possibly updated environment. If the identifier has been declared in the current environment, then the updated environment is left unchanged, otherwise the identifier is added to the environment as *Undecl*.

7.5.2 Value reference – Type checking semantics

$$
\begin{array}{l}
\forall\ ValueRefExpr;\ \rho\tau : EnvT \bullet \\
\quad \exists\, idt : IDTYPE;\ ExprType \mid \\
\qquad idt = \rho\tau\ \xi \\
\qquad \land\ \theta ExprType = \\
\qquad\quad \textbf{if}\ idt \in \operatorname{ran} variable \cup \operatorname{ran} const \\
\qquad\qquad\qquad \cup \operatorname{ran} formalParam \\
\qquad\qquad\qquad \cup \operatorname{ran} enumValue \cup \operatorname{ran} enumType \\
\qquad\qquad \land\ \#E = \#(arrayOf\ \ idt) \\
\qquad\qquad \land\ (\#E\ \ \geq 1 \Rightarrow \\
\qquad\qquad\quad (\ \exists\, ExprType \mid \theta ExprType = \mathcal{T}_{E^*}\ E\ \rho\tau \bullet \\
\qquad\qquad\qquad writeOnly \notin A \\
\qquad\qquad\qquad \land\ \tau = arrayIndexType\ \ idt \\
\qquad\qquad\qquad \land\ \tau \neq typeWrong)\ ) \\
\qquad\quad \textbf{then}\ \langle\!|\ \tau == typeOf\ \ idt, \\
\qquad\qquad A == attributeOf\ \ idts \\
\qquad\qquad\qquad \cap \{writeOnly, readOnly\}\ |\!\rangle \\
\qquad\quad \textbf{else}\ \langle\!|\ \tau == typeWrong, A == \varnothing\ |\!\rangle \bullet \\
\quad \mathcal{T}_E(valueRef\ \ \theta ValueRefExpr)\rho\tau = \theta ExprType
\end{array}
$$

The sequence of expressions comprising the array index list is checked. The value reference is mapped to the type and attribute of the identifier.

- ξ: the name of the value reference.
- $i = \rho\tau\ \xi$: the $IDTYPE$ of the value reference.
- $i \in \operatorname{ran} \ldots$: the identifier is allowed to be a value reference: a variable, named constant, formal parameter, enumerated value, or enumerated type (in cast expressions only).
- $\#E = \#(arrayOf\ \ I)$: the array dimension matches its declaration.
- $\#E\ \geq\ 1$: a non-zero dimensional array reference: the array index expressions are checked. None of them can refer to a write only variable; they are all of the same type as the subranges that define the array; they are all type correct.

- If all the checks are passed, the type of the value reference is its declared type, and the attributes are the declared attributes after discarding location attributes (which are of no use in type checking).
- If any checks is failed, the type of the value reference is *typeWrong*, with no attributes needed.

7.5.3 Value reference – Use semantics

$$
\begin{array}{l}
\forall\, ValueRefExpr;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t' : EnvUTrace;\ b : ID;\ \chi : \mathbb{P}\, USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_{E\ *}\ E\ B\ \rho\upsilon t \\
\qquad \land\ b = findBlock\ (\xi, B, \rho\upsilon t) \\
\qquad \land\ \chi = (\rho\upsilon t'\ b)(\!|\ \{\xi\}\ |\!) \setminus \{unread\} \\
\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
\quad \mathcal{U}_E(valueRef\ \theta\, ValueRefExpr)B\ \rho\upsilon t = update(\rho\upsilon t', \langle b \rangle, \rho\upsilon)
\end{array}
$$

- $\rho\upsilon t'$: the use check environment, after any array indexes have been checked.
- b: the block where the identifier is declared (that is, the name of the function or procedure, or module if it is global).
- χ: the use check status of the identifier after being read.
- *update*: the use environment is updated to say that ξ has been read.

7.5.4 Value reference – Dynamic semantics

The meaning of *valueRef* depends on whether it is a constant, an enumerated value, a variable, or a formal parameter.

If the name corresponds to a named constant or enumerated value its value is found from the type environment.

If the name corresponds to an enumerated type its value is the maximum allowed value of an element of the type, that is, one less than the number of elements in the enumeration.

$$
\begin{array}{l}
\forall \mathit{ValueRefExpr};\ B : \mathit{STACK};\ \rho t : \mathit{EnvMTrace};\ \sigma : \mathit{State} \bullet \\
\quad \forall \mathit{idt} : \mathit{IDTYPE} \mid \mathit{idt} = \mathit{lookup}(\xi, B, \rho\tau t0) \wedge \mathit{idt} \in \operatorname{ran} \mathit{const} \bullet \\
\qquad \exists \mathit{ExprValue} \mid \theta \mathit{State} = \sigma \wedge v = (\mathit{const}^{\sim}\ \mathit{idt}).v \bullet \\
\qquad\quad \mathcal{M}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr}) B\ \rho t\ \sigma = \theta \mathit{ExprVal} \\
\forall \mathit{ValueRefExpr};\ B : \mathit{STACK};\ \rho t : \mathit{EnvMTrace};\ \sigma : \mathit{State} \bullet \\
\quad \forall \mathit{idt} : \mathit{IDTYPE} \mid \\
\qquad\quad \mathit{idt} = \mathit{lookup}(\xi, B, \rho\tau t0) \wedge \mathit{idt} \in \operatorname{ran} \mathit{enumValue} \bullet \\
\qquad \exists \mathit{ExprValue} \mid \theta \mathit{State} = \sigma \wedge v = (\mathit{enumValue}^{\sim}\ \mathit{idt}).v \bullet \\
\qquad\quad \mathcal{M}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr}) B\ \rho t\ \sigma = \theta \mathit{ExprVal} \\
\forall \mathit{ValueRefExpr};\ B : \mathit{STACK};\ \rho t : \mathit{EnvMTrace};\ \sigma : \mathit{State} \bullet \\
\quad \forall \mathit{idt} : \mathit{IDTYPE} \mid \\
\qquad\quad \mathit{idt} = \mathit{lookup}(\xi, B, \rho\tau t0) \wedge \mathit{idt} \in \operatorname{ran} \mathit{enumType} \bullet \\
\qquad \exists \mathit{ExprValue} \mid \\
\qquad\qquad \theta \mathit{State} = \sigma \\
\qquad\qquad \wedge\ v = \mathit{vbyte}(\#(\mathit{enumType}^{\sim}\ \mathit{idt}).\Xi - 1) \bullet \\
\qquad\quad \mathcal{M}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr}) B\ \rho t\ \sigma = \theta \mathit{ExprVal}
\end{array}
$$

If the name corresponds to a variable its location is retrieved from the environment; if it is a call-by-reference formal parameter its location is retrieved by indirection from the environment and the store.

$$
\begin{array}{l}
\mathit{refLocation} : \mathit{IDTYPE} \times (\operatorname{seq} \mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{LOCN}) \times \mathit{Store} \times \operatorname{seq} \mathit{VALUE} \nrightarrow \\
\qquad \mathit{LOCN} \\
\hline
\forall \mathit{idt} : \mathit{IDTYPE};\ f : \operatorname{seq} \mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{LOCN};\ \Sigma : \mathit{Store};\ V : \operatorname{seq} \mathit{VALUE} \bullet \\
\quad \mathit{refLocation}(\mathit{idt}, f, \Sigma, V) = \\
\qquad \mathbf{if}\ \mathit{idt} \in \operatorname{ran} \mathit{formalParam} \wedge (\mathit{formalParam}^{\sim}\ \mathit{idt}).c = \mathit{ref} \\
\qquad \mathbf{then}\ \mathit{pointer}^{\sim}(\Sigma(f\langle\,\rangle)) \\
\qquad\qquad + \mathit{locationOffset}(\mathit{number} \circ V, \mathit{arrayOf}\ \mathit{idt}) \\
\qquad \mathbf{else}\ f(\mathit{number} \circ V)
\end{array}
$$

- f: the locations of the array ξ's elements

- V: the values V of the array indexes
- l: if ξ is a call-by-reference formal parameter, the location l is found by the array index offset from the location of the formal parameter. Otherwise l is looked up directly in the environment.

If the location is not an input stream, then the value is found from the relevant memory location in the current state; if it is an input stream, then the value is read from the head of the input stream.

$$
\begin{array}{l}
\forall\, ValueRefExpr;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \forall\, idt : IDTYPE \mid \\
\qquad idt = lookup(\xi, B, \rho\tau t0) \\
\qquad \land\ idt \in \operatorname{ran} variable \cup \operatorname{ran} formalParam \bullet \\
\quad \exists\, ExprSeqValue';\ f : \operatorname{seq}\mathbb{N} \rightarrowtail LOCN;\ l : LOCN;\ ExprValue'' \mid \\
\qquad \theta ExprSeqValue' = \mathcal{M}_{E^*}\ E\ B\ \rho t\ \sigma \\
\qquad \land\ f = loc^{\sim}(lookup(\xi, B, \rho t)) \\
\qquad \land\ l = refLocation(idt, f, \sigma'.\Sigma, V') \\
\qquad \land\ (l \in \sigma.i \Rightarrow \\
\qquad\quad (\ \exists\, I : \operatorname{seq}_1 VALUE \mid stream\ I = \Sigma'\ l \bullet \\
\qquad\qquad \Sigma'' = \Sigma' \oplus \{l \mapsto stream(tail\ I)\} \\
\qquad\qquad \land\ i'' = i' \land o'' = o' \\
\qquad\qquad \land\ v'' = head\ I\ )) \\
\qquad \land\ (l \notin \sigma.i \Rightarrow \theta State'' = \theta State' \land v'' = \Sigma'\ l) \bullet \\
\quad \mathcal{M}_E(valueRef\ \theta ValueRefExpr) B\ \rho t\ \sigma = \theta ExprValue''
\end{array}
$$

- f: the locations of the array ξ's elements
- $\theta ExprSeqValue'$: the values V of the array indexes, and the possibly changed state σ' from evaluating them
- l: the location of the referenced variable.
- $l \in \sigma.i$: if l is an input location, then location l contains a stream of values (the input stream). The value is the head of the stream, and the state is updated to have this read value removed.
- $l \notin \sigma.i$: if l is not an input location, the value is that stored in location l, and the final state is σ'.

7.6 Unary expression

7.6.1 Unary expression – Symbol declaration semantics

$$\begin{array}{l}\forall\, UnyExpr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\ \quad \exists\, \rho\delta t' : EnvDTrace \mid \rho\delta t' = \mathcal{D}_E\ \epsilon\ B\ \rho\delta t \bullet \\ \quad\quad \mathcal{D}_E(unyExpr\ \theta\, UnyExpr) B\ \rho\delta t = \rho\delta t'\end{array}$$

Unary operator names are considered to be declared implicitly, so the whole unary expression has the same declaration semantics as its sub-expression.

7.6.2 Unary expression – Type checking semantics

$$\begin{array}{l}\forall\, UnyExpr;\ \rho\tau : EnvT \bullet \\ \quad \exists\, ExprType;\ \tau' : TYPE \mid \\ \quad\quad \theta ExprType = \mathcal{T}_E\ \epsilon\ \rho\tau \\ \quad\quad \wedge\ \tau' = \mathcal{T}_{UO}\ \Psi\ \tau \bullet \\ \quad\quad \mathcal{T}_E(unyExpr\ \theta\, UnyExpr)\rho\tau = \langle\!|\ \tau == \tau', A == A\ |\!\rangle\end{array}$$

The type of a unary expression depends on the type of the operator and the type of the sub-expression. The attribute set is that of the sub-expression.

7.6.3 Unary expression – Use semantics

$$\begin{array}{l}\forall\, UnyExpr;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\ \quad \exists\, \rho\upsilon t' : EnvUTrace \mid \rho\upsilon t' = \mathcal{U}_E\ \epsilon\ B\ \rho\upsilon t \bullet \\ \quad\quad \mathcal{U}_E(unyExpr\ \theta\, UnyExpr) B\ \rho\upsilon t = \rho\upsilon t'\end{array}$$

Unary operator names are considered to be declared implicitly, so the whole unary expression has the same use semantics as its sub-expression.

7.6.4 Unary expression – Dynamic semantics

$$\begin{array}{l}\forall\, UnyExpr;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\ \quad \exists\, ExprValue';\ ExprValue'' \mid \\ \qquad \theta ExprValue' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \sigma \\ \qquad \land\ \theta State'' = \theta State' \\ \qquad \land\ v'' = \mathcal{M}_{UO}\ \Psi\ v' \bullet \\ \quad \mathcal{M}_E(unyExpr\ \theta UnyExpr)B\ \rho t\ \sigma = \theta ExprValue''\end{array}$$

The value of a unary expression is found by applying the unary operator meaning function to the value of the sub-expression. The state may be altered by evaluating the sub-expression.

7.7 Binary expression

7.7.1 Binary expression – Symbol declaration semantics

$$\begin{array}{l}\forall\, BinExpr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\ \quad \exists\, \rho\delta t';\ \rho\delta t'' : EnvDTrace \mid \\ \qquad \rho\delta t' = \mathcal{D}_E\ \epsilon 1\ B\ \rho\delta t \\ \qquad \land\ \rho\delta t'' = \mathcal{D}_E\ \epsilon 2\ B\ \rho\delta t' \bullet \\ \quad \mathcal{D}_E(binExpr\ \theta BinExpr)B\ \rho\delta t = \rho\delta t' \cup \rho\delta t''\end{array}$$

Binary operator names are considered to be declared implicitly, so the whole binary expression has a declaration semantics found by combining the separate semantics of its sub-expression.

The union function $\cup$ is used instead of the more complicated function $\Box$. This is because the only bad event that can occur when checking the sub-expressions is that an identifier may be added to the domain of the meaning function when a previously undeclared symbol is referenced.

7.7.2 Binary expression – Type checking semantics

$$
\begin{array}{l}
\forall\, BinExpr;\ \rho\tau : EnvT \bullet \\
\quad \exists\, ExprType;\ ExprType';\ A'' : \mathbb{P}\, ATTR;\ ExprType''' \mid \\
\qquad \theta ExprType = \mathcal{T}_E\ \epsilon 1\ \rho\tau \land \theta ExprType' = \mathcal{T}_E\ \epsilon 2\ \rho\tau \\
\qquad \land\ A'' = A \cup A' \bullet \\
\qquad \land\ \theta ExprType''' = \\
\qquad\quad \textbf{if}\ \#A'' \leq 1 \\
\qquad\quad \textbf{then}\ \langle\!|\ \tau == \mathcal{T}_{BO}\ \Omega(\tau, \tau'), A == A''\ |\!\rangle \\
\qquad\quad \textbf{else}\ \langle\!|\ \tau == typeWrong, A == \varnothing\ |\!\rangle \bullet \\
\quad \mathcal{T}_E(binExpr\ \theta BinExpr)\rho\tau = \theta ExprType'''
\end{array}
$$

The types of the sub-expressions are combined using the rules defined by the type checking semantics of the binary operator.

The attributes are combined by taking their union. The two sub-expressions are checked to ensure that their attributes are compatible: any use of an expression is legal for at most one of *readOnly* and *writeOnly*. (Recall that type checking of value references does not return any *at* attributes.)

7.7.3 Binary expression – Use semantics

$$
\begin{array}{l}
\forall\, BinExpr;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t';\ \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_E\ \epsilon 1\ B\rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_E\ \epsilon 2\ B\rho\upsilon t' \bullet \\
\quad \mathcal{U}_E(binExpr\ \theta BinExpr)B\ \rho\upsilon t = \rho\upsilon t''
\end{array}
$$

Binary operator names are considered to be declared implicitly, so the whole binary expression has a use semantics found by combining the separate semantics of its sub-expression.

The order of evaluation of expressions is important, because variables may be initialised (by being passed by reference to a function) in the first sub-expression.

7.7.4 Binary expression – Dynamic semantics

$$
\begin{array}{l}
\forall\, BinExpr;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, ExprValue';\ ExprValue'';\ ExprValue''' \mid \\
\qquad \theta ExprValue' = \mathcal{M}_E\ \epsilon 1\ B\ \rho t\ \sigma \\
\qquad \land\ \theta ExprValue'' = \mathcal{M}_E\ \epsilon 2\ B\ \rho t\ \theta State' \\
\qquad \land\ \theta State''' = \theta State'' \\
\qquad \land\ v''' = \mathcal{M}_{BO}\ \Omega(v', v'') \bullet \\
\quad \mathcal{M}_E(binExpr\ \theta BinExpr)B\ \rho t\ \sigma = \theta ExprValue'''
\end{array}
$$

The binary operator meaning function is applied to the values of the sub-expressions. The left expression is evaluated first, and the resulting state is used in evaluating the right expression.

7.8 Function call

7.8.1 Function call – Symbol declaration semantics

$$
\begin{array}{l}
\forall\, FunCallExpr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t' : EnvDTrace \mid \rho\delta t' = \mathcal{D}_{AP\ *}\ E\ B\ \rho\delta t \bullet \\
\qquad \mathcal{D}_E(funCall\ \theta FunCallExpr)B\ \rho\delta t = checkSymbol\ \xi\ B\ \rho\delta t'
\end{array}
$$

The list of actual parameters is checked, and the result of this is combined with the result of checking the function identifier.

7.8.2 Function call – Type checking semantics

$$\mathcal{T}_E : EXPR \pfun EnvT \pfun ExprType$$

$$\begin{aligned}
&\forall\, FunCallExpr;\ \rho\tau : EnvT \mid \rho\tau\ \xi \in \operatorname{ran} function \bullet \\
&\quad \exists\, FunType;\ \tau' : TYPE \mid \\
&\qquad function\ \theta FunType = \rho\tau\ \xi \\
&\qquad \land\ \tau' = \textbf{if}\ \#E = \#I \land \mathcal{T}_{AP^*}\ E\ I\ \rho\tau = checkOK \\
&\qquad\qquad \textbf{then}\ \tau\ \textbf{else}\ typeWrong \bullet \\
&\quad \mathcal{T}_E(funCall\ \theta FunCallExpr)\rho\tau = \langle\!|\ \tau == \tau', A == \varnothing\ |\!\rangle \\
&\forall\, FunCallExpr;\ \rho\tau : EnvT \mid \rho\tau\ \xi \notin \operatorname{ran} function \bullet \\
&\quad \mathcal{T}_E(funCall\ \theta FunCallExpr)\rho\tau = \langle\!|\ \tau == typeWrong, A == \varnothing\ |\!\rangle
\end{aligned}$$

A function call type checks correctly only if

- The name ξ is a function name
- The number of actual and formal parameters match.
- The actual parameters are type correct

Then the type of the function call is retrieved from the environment. Otherwise, it is *typeWrong*.

There is no attribute associated with a function call.

7.8.3 Function call – Use semantics

$$\begin{aligned}
&\forall\, FunCallExpr;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
&\quad \exists\, I : \operatorname{seq} IDTYPE;\ b : ID;\ \rho\upsilon t' : EnvUTrace; \\
&\qquad \chi : \mathbb{P}\ USECHECK;\ \rho\upsilon : EnvU \mid \\
&\qquad I = (function^{\sim}(lookup(\xi, B, \rho\tau t0))).I \\
&\qquad \land\ b = findBlock(\xi, B, \rho\upsilon t) \\
&\qquad \land\ \rho\upsilon t' = \mathcal{U}_{AP\ *}\ E\ I\ B\ \xi\ \rho\upsilon t \\
&\qquad \land\ \chi = (\rho\upsilon t'\ b)(\!|\ \{\xi\}\ |\!) \setminus \{uncalled\} \\
&\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
&\quad \mathcal{U}_E(funCall\ \theta FunCallExpr)B\ \rho\upsilon t = update(\rho\upsilon t', \langle b \rangle, \rho\upsilon)
\end{aligned}$$

The parameters to the function are checked. Then the function itself is known to be called.

- I: the formal parameter $IDTYPE$s.
- b: the block where the function is declared. With no nested functions and procedures, this must be the module name.
- $\rho v t'$: the call-by-value actual parameters are checked, with the calling blockstack.
- χ: the use check status of the identifier after being called.
- *update*: then this use of ξ is updated.

7.8.4 Function call – Dynamic semantics

$$
\begin{array}{l}
\forall\, FunCallExpr;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, FunType';\ \sigma' : State;\ f : State \pfun ExprValue;\ ExprValue'' \mid \\
\qquad \theta FunType' = function^{\sim}(lookup(\xi, B, \rho\tau t0)) \\
\qquad \land\ \sigma' = \mathcal{M}_{AP}\ *\ E\ I'\ B\ \xi\ \rho t\ \sigma \\
\qquad \land\ f = fval^{\sim}(lookup(\xi, B, \rho t)) \\
\qquad \land\ \theta ExprValue'' = f\ \sigma' \bullet \\
\quad lb' \leq number\ v'' \leq ub' \Rightarrow \\
\qquad \mathcal{M}_E(funCall\ \theta FunCallExpr)B\ \rho t\ \sigma = \theta ExprValue''
\end{array}
$$

- σ': The dynamic meanings of the actual parameters are evaluated, resulting in a state σ' that has their values bound to the appropriate formal parameters.
- the function is applied to this state, resulting in a possibly modified state and a return value.
- $lb' \leq number\ v'' \leq ub'$: if the return value lies in the subrange of the function's return type, the dynamic semantics is as defined.

  Partial specification: Pasp's subranges are checked dynamically, and the consequence of overflow or underflow is *undefined*.

8 Statements

8.1 Meaning function declarations

8.1.1 Symbol declaration semantics

The declaration environment is updated to reflect the status of any identifiers used in the statement.

$$\mathcal{D}_S : STMT \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

8.1.2 Type checking semantics

The statement is mapped to an updated type environment and a check status for that statement.

$$StmtType \mathrel{\widehat=} [\ \rho\tau t : EnvTTrace;\ c : CHECK\ ]$$

$$\mathcal{T}_S : STMT \pfun STACK \pfun EnvTTrace \pfun StmtType$$

In the symbol declaration semantics it is possible to associate an error with an identifier, and so it makes sense to update the environment in order to supply the user with that information. However, when type checking, two possible errors can occur: a variable or named constant may be improperly declared, or may be improperly used. In the latter case, it is the statement that needs to be flagged, rather than the variable.

The trace environment is updated when checking case statements to add a temporary variable for translating that statement.

8.1.3 Use semantics

The use environment is updated to reflect the status of any variables used in the statement.

$$\mathcal{U}_S : STMT \pfun STACK \pfun EnvUTrace \pfun EnvUTrace$$

8.1.4 Dynamic semantics

The meaning of a statement, given the context of a block stack, a dynamic environment, and a current state, is a new state.

$$\mathcal{M}_S : STMT \pfun STACK \pfun EnvMTrace \pfun State \pfun State$$

8.2 Multiple statements

8.2.1 Multiple statements – Symbol declaration semantics

The meaning function for a single statement is extended to a list of statements as $\mathcal{D}_{S*}$.

8.2.2 Multiple statements – Type checking semantics

The meaning function for a single statement is extended to a list of statements inductively.

$$\mathcal{T}_{S^*} : \operatorname{seq} STMT \pfun STACK \pfun EnvTTrace \pfun StmtType$$

$$\begin{array}{l}
\forall\, B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \mathcal{T}_{S^*}\langle\,\rangle\ B\ \rho\tau t = \llparenthesis\, \rho\tau t == \rho\tau t, c == checkOK \,\rrparenthesis \\
\forall\, \gamma : STMT;\ \Gamma : \operatorname{seq} STMT;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, StmtType';\ StmtType'';\ c''' : CHECK \mid \\
\qquad \theta StmtType' = \mathcal{T}_S\ \gamma\ B\ \rho\tau t \\
\qquad \land\ \theta StmtType'' = \mathcal{T}_{S^*}\ \Gamma\ B\ \rho\tau t' \\
\qquad \land\ c''' = c' \bowtie c'' \bullet \\
\qquad\quad \mathcal{T}_{S^*}(\langle\gamma\rangle \frown \Gamma) B\ \rho\tau t = \llparenthesis\, \rho\tau t == \rho\tau t'', c == c''' \,\rrparenthesis
\end{array}$$

8.2.3 Multiple statements – Use semantics

The meaning function for a single statement is extended to a list of statements as $\mathcal{U}_{S*}$.

8.2.4 Multiple statements – Dynamic semantics

The meaning function for a single statement is extended to a list of statements inductively.

$$\mathcal{M}_{S^*} : \operatorname{seq} STMT \pfun STACK \pfun EnvMTrace \pfun State \pfun State$$

$$\begin{aligned}
&\forall B : STACK;\ \rho t : EnvMTrace \bullet \mathcal{M}_{S^*} \langle\,\rangle\ B\ \rho t = \operatorname{id} State \\
&\forall \gamma : STMT;\ \Gamma : \operatorname{seq} STMT;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
&\quad \exists \sigma', \sigma'' : State \mid \\
&\qquad \sigma' = \mathcal{M}_S\ \gamma\ B\ \rho t\ \sigma \\
&\qquad \land\ \sigma'' = \mathcal{M}_{S^*}\ \Gamma\ B\ \rho t\ \sigma' \bullet \\
&\quad \mathcal{M}_{S^*}(\langle\gamma\rangle\ \frown \Gamma) B\ \rho t\ \sigma = \sigma''
\end{aligned}$$

8.3 Block

8.3.1 Block – Symbol declaration semantics

$$\forall \Gamma : \operatorname{seq}_1 STMT \bullet \mathcal{D}_S(block\ \Gamma) = \mathcal{D}_S\ {}_*\ \Gamma$$

A block's status is the result of checking the sequence of body statements in the context of a trace environment and block stack.

8.3.2 Block – Type checking semantics

$$\forall \Gamma : \operatorname{seq}_1 STMT \bullet \mathcal{T}_S(block\ \Gamma) = \mathcal{T}_{S^*}\ \Gamma$$

A block's status is the result of checking the sequence of body statements in the context of an environment.

8.3.3 Block – Use semantics

$$\forall \Gamma : \operatorname{seq}_1 STMT \bullet \mathcal{U}_S(block\ \Gamma) = \mathcal{U}_S\ {}_*\ \Gamma$$

A block's status is the result of checking the sequence of body statements in the context of a trace environment.

8.3.4 Block – Dynamic semantics

$$\forall \Gamma : \operatorname{seq}_1 STMT \bullet \mathcal{M}_S(block\ \Gamma) = \mathcal{M}_{S^*}\ \Gamma$$

A block's meaning is the meaning of the sequence of body statements.

8.4 Skip

8.4.1 Skip – Symbol declaration semantics

$$\forall B : STACK \bullet \mathcal{D}_S\ skip\ B = \mathrm{id}\, EnvDTrace$$

The declaration environment is left unchanged by a skip statement.

8.4.2 Skip – Type checking semantics

$$\forall B : STACK;\ \rho\tau t : EnvTTrace \bullet \\ \quad \mathcal{T}_S\ skip\ B\ \rho\tau t = (\!|\ \rho\tau t == \rho\tau t, c == checkOK\ |\!)$$

A skip statement type checks OK.

8.4.3 Skip – Use semantics

$$\forall B : STACK \bullet \mathcal{U}_S\ skip\ B = \mathrm{id}\, EnvUTrace$$

The use environment is left unchanged by a skip statement.

8.4.4 Skip – Dynamic semantics

$$\forall B : STACK;\ \rho t : EnvMTrace \bullet \mathcal{M}_S\ skip\ B\ \rho t = \mathrm{id}\, State$$

The state is left unchanged by a skip statement.

8.5 Assignment

8.5.1 Assignment – Symbol declaration semantics

$$\begin{array}{l} \forall AssignStmt;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\ \quad \exists \rho\delta t', \rho\delta t'', \rho\delta t''' : EnvDTrace \mid \\ \qquad \rho\delta t' = \mathcal{D}_{E\ *}\ E\ B\ \rho\delta t \\ \qquad \land\ \rho\delta t'' = checkSymbol\ \xi\ B\ \rho\delta t' \\ \qquad \land\ \rho\delta t''' = \mathcal{D}_E\ \epsilon\ B\ \rho\delta t'' \bullet \\ \quad \mathcal{D}_S(assign\ \theta AssignStmt)B\ \rho\delta t = \rho\delta t''' \end{array}$$

The source variable's array index expressions are checked, then the source variable's name is checked, then the target expression is checked.

8.5.2 Assignment – Type checking semantics

$$
\begin{array}{l}
\forall\, AssignStmt;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, \rho\tau : EnvT;\ ExprType;\ idt : IDTYPE;\ c : CHECK \mid \\
\qquad \rho\tau = \oplus/\ (\rho\tau t \circ B) \\
\qquad \land\ \theta ExprType = \mathcal{T}_E\ \epsilon\ \rho\tau \\
\qquad \land\ idt = \rho\tau\ \xi \\
\qquad \land\ c = \textbf{if}\ idt \in \operatorname{ran} variable \cup \operatorname{ran} formalParam \\
\qquad\qquad \land\ readOnly \notin attributeOf\ idt \\
\qquad\qquad \land\ writeOnly \notin A \\
\qquad\qquad \land\ typeOf\ idt = \tau \neq typeWrong \\
\qquad\qquad \land\ (\#E \geq 1 \Rightarrow \\
\qquad\qquad\quad (\ \exists_1\ ExprType' \mid \theta ExprType' = \mathcal{T}_{E^*}\ E\ \rho\tau \bullet \\
\qquad\qquad\qquad \tau' = arrayIndexType\ idt \\
\qquad\qquad\qquad \land\ \tau' \neq typeWrong \\
\qquad\qquad\qquad \land\ writeOnly \notin A'\ )) \\
\qquad\qquad \textbf{then}\ c == checkOK\ \textbf{else}\ c == checkTypeWrong \bullet \\
\quad \mathcal{T}_S(assign\ \theta AssignStmt)B\ \rho\tau t = \langle\!|\ \rho\tau t == \rho\tau t, c == c\ |\!\rangle
\end{array}
$$

An assignment statement must pass the following type checks:

- The target ξ must be a variable name, or formal parameter name (so not a constant, function of procedure name).
- The target must not be a *readOnly* variable.
- The source must not contain any *writeOnly* variables.
- The types of source and target must be equal, and not wrong.
- The type of the target's array indexes must be match those used in the original declaration, and not be wrong. (That the source value lies within the target subrange is checked dynamically.)
- The target's array indexes must not contain any *writeOnly* variables.

If the statement passes all these checks, it is mapped to *checkOK*, and otherwise it is mapped to *checkTypeWrong*. The type environment is unchanged.

8.5.3 Assignment – Use semantics

$$
\begin{array}{l}
\forall\, AssignStmt;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t', \rho\upsilon t'' : EnvUTrace;\ b : ID;\ \chi : \mathbb{P}\, USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_{E\ *}\ E\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_{E}\ \epsilon\ B\ \rho\upsilon t' \\
\qquad \land\ b = findBlock(\xi, B, \rho\upsilon t) \\
\qquad \land\ \chi = (\rho\upsilon t''\ b)(\!|\ \{\xi\}\ |\!) \setminus \{unwritten\} \\
\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
\quad \mathcal{U}_S(assign\ \theta AssignStmt)B\ \rho\upsilon t = update(\rho\upsilon t'', \langle b \rangle, \rho\upsilon)
\end{array}
$$

- $\rho\upsilon t'$: the use environment formed by checking the array reference expression list.
- $\rho\upsilon t''$: the use environment formed by checking the source expression.
- b: the block where the identifier is declared.
- χ: the use check status of the target identifier, no longer unwritten
- *update*: the use after declaration semantics is updated

8.5.4 Assignment – Dynamic semantics

$$
\begin{array}{l}
\forall \mathit{AssignStmt};\ B : \mathit{STACK};\ \rho t : \mathit{EnvMTrace};\ \sigma : \mathit{State} \bullet \\
\quad \exists \mathit{State};\ \mathit{ExprSeqValue}';\ \mathit{ExprValue}'';\ \mathit{idt} : \mathit{IDTYPE}; \\
\qquad \mathit{VarType};\ f : \mathrm{seq}\,\mathbb{N} \rightarrowtail\!\!\!\!\!\!\;\rightarrow \mathit{LOCN};\ l : \mathit{LOCN} \\
\qquad v''' : \mathit{VALUE};\ \Sigma''' : \mathit{Store} \mid \\
\qquad\quad \theta \mathit{State} = \sigma \\
\qquad\quad \land\ \theta \mathit{ExprSeqValue}' = \mathcal{M}_{E^*}\ E\ B\ \rho t\ \sigma \\
\qquad\quad \land\ \theta \mathit{ExprValue}'' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \theta \mathit{State}' \\
\qquad\quad \land\ \mathit{idt} = \mathit{lookup}(\xi, B, \rho\tau t0) \\
\qquad\quad \land\ \theta \mathit{VarType} = \mathit{variable}^{\sim}\ \mathit{idt} \\
\qquad\quad \land\ f = \mathit{loc}^{\sim}(\mathit{lookup}(\xi, B, \rho t)) \\
\qquad\quad \land\ l = \mathit{refLocation}(\mathit{idt}, f, \Sigma', V') \\
\qquad\quad \land\ v''' = \mathbf{if}\ l\ \notin o \\
\qquad\qquad \mathbf{then}\ v''\ \mathbf{else}\ \mathit{stream}(\mathit{stream}^{\sim}(\Sigma''\ l) \frown \langle v'' \rangle) \\
\qquad\quad \land\ \Sigma''' = \Sigma'' \oplus \{ l \mapsto v''' \} \bullet \\
\qquad \tau \in \mathrm{ran}\, \mathit{subrange} \land \mathit{lb} \leq \mathit{number}\ v'' \leq \mathit{ub} \\
\qquad \lor\ \tau \notin \mathrm{ran}\, \mathit{subrange} \Rightarrow \\
\qquad\quad \mathcal{M}_S(\mathit{assign}\ \theta \mathit{AssignStmt}) B\ \rho t\ \sigma = \\
\qquad\qquad \langle\!|\ \Sigma == \Sigma''', i == i, o == o\ |\!\rangle
\end{array}
$$

Assignment updates the store with the value of the assigned expression and updates the state with the new status of any relevant streams.

- $\theta \mathit{ExrpSeqValue}'$: the state and values that result from evaluating the target's array index expressions.
- $\theta \mathit{ExrpSeqValue}''$: the state and value that then results from evaluating the source expression.
- l: the location of the target variable.
- $l \notin \sigma.o$: if l is not an output location, the value is that stored in location l. Otherwise location l contains a stream of values (the output stream), and the value is concatenated to the end of the stream, and the state is updated to have this written value.
- $\tau \in \mathrm{ran}\, \mathit{subrange} \ldots \Rightarrow \ldots$: if the target has a subrange type and the

source value lies within the type, or if the target is an ordinary type, then the dynamic semantics is as given.

Partial specification: Pasp's subranges are checked dynamically, and the consequence of overflow or underflow is *undefined*.

8.6 If statement

8.6.1 If – Symbol declaration semantics

$$
\begin{array}{l}
\forall\, IfStmt;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t', \rho\delta t'', \rho\delta t''' : EnvDTrace \mid \\
\qquad \rho\delta t' = \mathcal{D}_E\ \epsilon\ B\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = \mathcal{D}_S\ \gamma 1\ B\ \rho\delta t \\
\qquad \land\ \rho\delta t''' = \mathcal{D}_S\ \gamma 2\ B\ \rho\delta t \bullet \\
\quad \mathcal{D}_S(ifStmt\ \theta IfStmt)B\ \rho\delta t = \rho\delta t' \cup \rho\delta t'' \cup \rho\delta t'''
\end{array}
$$

The statement alters the declaration environment to have the worst properties of the environments of the expression and the two substatements.

The reason why $\cup$ can be used in place of $\square$ is explained in section 8.5.1.

8.6.2 If – Type checking semantics

$$
\begin{array}{l}
\forall\, IfStmt;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, \rho\tau : EnvT;\ ExprType;\ StmtType';\ StmtType'';\ c''' : CHECK \mid \\
\qquad \rho\tau = \oplus/\ (\rho\tau t \circ B) \land \theta ExprType = \mathcal{T}_E\ \epsilon\ \rho\tau \\
\qquad \land\ \theta StmtType' = \mathcal{T}_S\ \gamma 1\ B\ \rho\tau t \\
\qquad \land\ \theta StmtType'' = \mathcal{T}_S\ \gamma 2\ B\ \rho\tau t' \\
\qquad \land\ c''' = \textbf{if}\ \tau = boolean \land writeOnly \notin A \\
\qquad\qquad \textbf{then}\ c'\ \bowtie\ c''\ \textbf{else}\ checkTypeWrong \bullet \\
\quad \mathcal{T}_S(ifStmt\ \theta IfStmt)B\ \rho\tau t = (\!|\ \rho\tau t == \rho\tau t'', c == c'''\ |\!)
\end{array}
$$

The if statement checks *checkOK* only if the expression has type *boolean* and is not *writeOnly*, and the two substatements both check *checkOK*.

8.6.3 If – Use semantics

$$
\begin{array}{l}
\forall\, IfStmt;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t', \rho\upsilon t'', \rho\upsilon t''', : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_E\ \epsilon\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_S\ \gamma 1\ B\ \rho\upsilon t' \\
\qquad \land\ \rho\upsilon t''' = \mathcal{U}_S\ \gamma 2\ B\ \rho\upsilon t' \bullet \\
\quad \mathcal{U}_S(ifStmt\ \theta IfStmt)B\ \rho\upsilon t = \rho\upsilon t''\ \diamond\ \rho\upsilon t'''
\end{array}
$$

The statement alters the use environment to have the worst properties of the environments of the expression and the two substatements.

8.6.4 If – Dynamic semantics

$$
\begin{array}{l}
\forall\, IfStmt;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, ExprValue';\ State'' \mid \\
\qquad \theta ExprValue' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \sigma \\
\qquad \land\ \theta State'' = \\
\qquad\quad \textbf{if}\ v' = vbool\ ptrue \\
\qquad\quad \textbf{then}\ \mathcal{M}_S\ \gamma 1\ B\ \rho t\ \theta State' \\
\qquad\quad \textbf{else}\ \mathcal{M}_S\ \gamma 2\ B\ \rho t\ \theta State' \bullet \\
\quad \mathcal{M}_S(ifStmt\ \theta IfStmt)B\ \rho t\ \sigma\theta State''
\end{array}
$$

The choice between the meaning of the two branch statements is made on the basis of the value of the test expression in the current state.

8.7 Case statement

8.7.1 Case statement – Symbol declaration semantics

We first define the symbol declaration semantics of the branches.

$$\mathcal{D}_{CL} : Branch \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

$$\begin{aligned}
&\forall\, Branch;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t';\ \rho\delta t'' : EnvDTrace \\
&\qquad \rho\delta t' = checkSymbol\ {}_{*}\ \Xi\ B\ \rho\delta t \\
&\qquad \land\ \rho\delta t'' = \mathcal{D}_S\ \gamma\ B\ \rho\delta t \bullet \\
&\quad \mathcal{D}_{CL}\ \theta Branch\ B\ \rho\delta t = rho\delta t' \cup \rho\delta t''
\end{aligned}$$

A branch checks okay if all the case names have been declared, and if the branch statement checks okay.

The case statement is checked by combining the results of checking the switch expression and all branches.

$$\begin{aligned}
&\forall\, CaseStmt;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t';\ \rho\delta t'' : EnvDTrace \\
&\qquad \rho\delta t' = \mathcal{D}_E\ \epsilon\ B\ \rho\delta t \\
&\qquad \land\ \rho\delta t'' = \mathcal{D}_{CL}\ {}_{*}\ K\ B\ \rho\delta t \bullet \\
&\quad \mathcal{D}_S(caseStmt\ \theta CaseStmt)B\ \rho\delta t = rho\delta t' \cup \rho\delta t''
\end{aligned}$$

8.7.2 Case statement – Type checking semantics

First we define the type checking semantics of a single branch.

$$\mathcal{T}_{CL} : Branch \nrightarrow STACK \nrightarrow EnvTTrace \nrightarrow \operatorname{seq} ID \times StmtType$$

$$\begin{aligned}
&\forall\, Branch;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
&\quad \exists\, StmtType' \mid \theta StmtType' = \mathcal{T}_S\ \gamma\ B\ \rho\tau t \bullet \\
&\quad \mathcal{T}_{CL}\ \theta Branch\ B\ \rho\tau t = (\Xi, \theta StmtType')
\end{aligned}$$

This returns the sequence of labels attached to the branch, and the declaration environment and check status that result from checking the branch statement.

This definition is extended inductively to a sequence of branches.

$$\mathcal{T}_{CL^*} : \operatorname{seq} Branch \pfun STACK \pfun EnvTTrace \pfun \operatorname{seq} ID \times StmtType$$

$$\forall B : STACK;\ \rho\tau t : EnvTTrace \bullet$$
$$\quad \mathcal{T}_{CL^*}\langle\,\rangle\ B\ \rho\tau t = (\langle\,\rangle, \{\!|\ \rho\tau t == \rho\tau t, c == checkOK\ |\!\})$$
$$\forall k : Branch;\ K : \operatorname{seq} Branch;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet$$
$$\quad \exists \Xi', \Xi'' : \operatorname{seq} ID;\ StmtType';\ StmtType'' \mid$$
$$\qquad (\Xi', \theta StmtType') = \mathcal{T}_{CL^*}\ K\ B\ \rho\tau t$$
$$\qquad \land\ (\Xi'', \theta StmtType'') = \mathcal{T}_{CL}\ k\ B\ \rho\tau t' \bullet$$
$$\qquad \mathcal{T}_{CL^*}(\langle k\rangle\ ^\frown K)B\ \rho\tau t =$$
$$\qquad\quad (\Xi'\ ^\frown \Xi'', \{\!|\ \rho\tau t == \rho\tau t'', c == c'\ \bowtie\ c''\ |\!\})$$

The type checking semantics for case statements can now be completed. Recall that the dummy identifier in the abstract syntax of a case statement had no concrete syntax. This identifier is added to the type trace environment as a temporary variable.

$$\forall CaseStmt;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet$$
$$\quad \exists \rho\tau : EnvT;\ ExprType';\ \rho\tau t' : EnvTTrace;\ \Xi : \operatorname{seq}_1 ID;\ StmtType'' \mid$$
$$\qquad \rho\tau = \oplus/\ (\rho\tau t \circ B) \land \theta ExprType' = \mathcal{T}_E\ \epsilon\ \rho\tau$$
$$\qquad \land\ (\forall b : \operatorname{dom} \rho\tau t \bullet \xi \notin \operatorname{dom}(\rho\tau t\ b))$$
$$\qquad \land\ \rho\tau t' = update(\rho\tau t, B, \{\xi \mapsto tempVar\ \tau'\})$$
$$\qquad \land\ (\Xi, \theta StmtType'') = \mathcal{T}_{CL^*}\ K\ B\ \rho\tau t' \bullet$$
$$\qquad (\tau' \in \operatorname{ran} scopedTypeValue \Rightarrow$$
$$\qquad\quad (\ \exists ScopedType';\ \Xi' : \operatorname{seq}_1 ID;\ c''' : CHECK \mid$$
$$\qquad\qquad \tau' = scopedTypeValue\ \theta ScopedType'$$
$$\qquad\qquad \land\ c''' = \textbf{if}\ \rho\tau t\ b'\ \xi' = enumType\ \theta EnumType'$$
$$\qquad\qquad\qquad \land\ c'' = checkOK$$
$$\qquad\qquad\qquad \land\ writeOnly\ \notin A'$$
$$\qquad\qquad\qquad \land\ \Xi\ \in \operatorname{iseq} ID$$
$$\qquad\qquad\qquad \land\ \operatorname{ran}\Xi = \operatorname{ran}\Xi'$$
$$\qquad\qquad\quad \textbf{then}\ checkOK\ \textbf{else}\ checkTypeWrong\ ) \bullet$$
$$\qquad\qquad \mathcal{T}_S(caseStmt\ \theta CaseStmt)B\ \rho\tau t =$$
$$\qquad\qquad\quad \{\!|\ \rho\tau t == \rho\tau t'', c == c'''\ |\!\})$$
$$\qquad \land\ (\tau' \notin \operatorname{ran} scopedTypeValue \Rightarrow$$
$$\qquad\quad \mathcal{T}_S(caseStmt\ \theta CaseStmt)B\ \rho\tau t =$$
$$\qquad\qquad \{\!|\ \rho\tau t == \rho\tau t'', c == checkTypeWrong\ |\!\})$$

The case statement type checks okay only if

- the temporary identifier ξ has not been used already (in practice this can be achieved by including the line number of the statement in the identifier)
- the switch expression's type is an enumerated type
- the branches type check okay, and have no write only attributes
- there are no repeated branch labels
- every enumerated value of the switch type appears as a label in a branch, and no other label occurs

8.7.3 Case statement – Use semantics

Each branch is checked and the results are combined pessimistically.

$$
\begin{array}{|l}
\mathcal{U}_{CL^*} : \operatorname{seq} Branch \pfun STACK \pfun EnvUTrace \pfun EnvUTrace \\
\hline
\forall B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \mathcal{U}_{CL^*} \langle\ \rangle\ B\ \rho\upsilon t = \rho\upsilon t \\
\forall Branch;\ K : \operatorname{seq} Branch;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists \rho\upsilon t';\ \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_S\ \gamma\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_{CL^*}\ K\ B\ \rho\upsilon t \bullet \\
\qquad \mathcal{U}_{CL^*}(\langle \theta Branch \rangle\ ^\frown K) B\ \rho\upsilon t = \rho\upsilon t'\ \Diamond\ \rho\upsilon t''
\end{array}
$$

The use semantics of a case statement applies that of the branches to the use environment formed by checking the switch expression.

$$
\begin{array}{|l}
\hline
\forall CaseStmt;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists \rho\upsilon t';\ \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_E\ \epsilon\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_{CL^*}\ K\ B\ \rho\upsilon t' \bullet \\
\qquad \mathcal{U}_S(caseStmt\ \theta CaseStmt) B\ \rho\upsilon t = \rho\upsilon t''
\end{array}
$$

8.7.4 Case statement – Dynamic semantics

The dynamic semantics of case statements are defined by calculating the dynamic semantics of an equivalent fragment of Pasp that contains the case statement transformed to nested if-then-elses[6]. In order to evaluate the expression at the head of the case statement only once, a temporary variable is used to store the expression value. So the case statement

CASE ϵ **OF**
$\xi_{11}, \ldots, \xi_{1n_1} : \gamma_1$
$\ldots$
$\xi_{m1}, \ldots, \xi_{mn_m} : \gamma_m$

is (abstractly) transformed to

ξ **:=** ϵ
IF $\xi = \xi_{11} \vee \ldots \vee \xi = \xi_{1n_1}$
THEN γ_1
ELSE IF $\xi = \xi_{21} \vee \ldots \vee \xi = \xi_{2n_2}$
THEN γ_2
ELSE IF $\ldots$
ELSE γ_m

We define some functions that convert parts of the case statement into their Pasp equivalents. The first of these defines the test expression required at the head of each branch. (No 'empty expression' exists, so the function is defined only for non-empty sequences of labels, making its inductive definition somewhat different in form from usual.)

[6]The operational semantics are defined as the operational meaning of the same fragment, thus making the proof of the correctness immediate by structural induction.

$$
\begin{array}{|l}
limbExpression : \mathrm{seq}_1\ ID \pfun ID \pfun EXPR \\
\hline
\forall \xi, \xi' : ID \bullet \\
\quad limbExpression\langle \xi \rangle\ \xi' = \\
\qquad binExpr(\!|\ \Omega == eeq, \epsilon 1 == valueRef\ (\!|\ \xi == \xi', E == \langle\ \rangle\ |\!), \\
\qquad\quad \epsilon 2 == valueRef\ (\!|\ \xi == \xi, E == \langle\ \rangle\ |\!)\ |\!) \\
\forall \Xi : \mathrm{seq}_1\ ID;\ \xi, \xi' : ID \bullet \\
\quad \exists \epsilon 1, \epsilon 2 : EXPR \mid \\
\qquad\quad \epsilon 1 = limbExpression\langle \xi \rangle \xi' \\
\qquad\quad \land\ \epsilon 2 = limbExpression\ \Xi\ \xi' \bullet \\
\qquad limbExpression(\langle \xi \rangle\ ^\frown \Xi) \xi' = \\
\qquad\quad binExpr(\!|\ \Omega == or, \epsilon 1 == \epsilon 1, \epsilon 2 == \epsilon 2\ |\!)
\end{array}
$$

We define the nested if statement that switches between different branches.

$$
\begin{array}{|l}
limbSwitch : \mathrm{seq}_1\ Branch \pfun ID \pfun STMT \\
\hline
\forall\ Branch;\ \xi : ID \bullet limbSwitch\langle \theta Branch \rangle\ \xi = \gamma \\
\forall\ Branch;\ \xi : ID;\ K : \mathrm{seq}\ Branch \bullet \\
\quad \exists \epsilon : EXPR;\ \gamma 2 : STMT \mid \\
\qquad\quad \epsilon = limbExpression\ \Xi\ \xi \\
\qquad\quad \land\ \gamma 2 = limbSwitch\ K\ \xi \bullet \\
\qquad limbSwitch(\langle \theta Branch \rangle\ ^\frown K) \xi = \\
\qquad\quad ifStmt(\!|\ \epsilon == \epsilon, \gamma 1 == \gamma, \gamma 2 == \gamma 2\ |\!)
\end{array}
$$

The dynamic semantics of case statements is defined as follows. The location assigned to the temporary variable is not needed outside the block defined in this semantics. However, it is easier to define distinct locations for all the temporary variables in the trace environment, as this reduces the impact on the proof conditions to a minimum (although it is slightly less efficient on space).

$$
\begin{array}{|l}
\hline
\forall\ CaseStmt \bullet \\
\quad \exists \Gamma : \mathrm{seq}\ STMT \mid \\
\qquad\quad \Gamma = \langle assign(\!|\ \xi == \xi, E == \langle\ \rangle, \epsilon == \epsilon\ |\!), \\
\qquad\qquad limbSwitch\ K\ \xi \rangle \bullet \\
\qquad \mathcal{M}_S(caseStmt\ \theta CaseStmt) = \mathcal{M}_S(block\ \Gamma)
\end{array}
$$

8.8 Loop

8.8.1 Loop – Symbol declaration semantics

$$
\begin{array}{l}
\forall\, WhileStmt;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t';\ \rho\delta t'' : EnvDTrace \mid \\
\qquad \rho\delta t' = \mathcal{D}_E\ \epsilon\ B\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = \mathcal{D}_S\ \gamma\ B\ \rho\delta t \bullet \\
\quad \mathcal{D}_S(whileStmt\ \theta\, WhileStmt)B\ \rho\delta t = \rho\delta t' \cup \rho\delta t''
\end{array}
$$

The loop statement alters the declaration environment to be the worse of the environments of the expression and the body statement.

8.8.2 Loop – Type checking semantics

$$
\begin{array}{l}
\forall\, WhileStmt;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, \rho\tau : EnvT;\ ExprType;\ StmtType';\ c : CHECK \mid \\
\qquad \rho\tau = \oplus/\ (\rho\tau t \circ B) \\
\qquad \land\ \theta ExprType = \mathcal{T}_E\ \epsilon\ \rho\tau \\
\qquad \land\ \theta StmtType' = \mathcal{T}_S\ \gamma\ B\ \rho\tau t \\
\qquad \land\ c = \textbf{if}\ \tau = boolean \land writeOnly \notin A \\
\qquad\quad \textbf{then}\ c'\ \textbf{else}\ checkTypeWrong \bullet \\
\quad \mathcal{T}_S(whileStmt\ \theta\, WhileStmt)B\ \rho\tau t = \langle\!|\ \rho\tau t == \rho\tau t', c == c\ |\!\rangle
\end{array}
$$

The loop statement type checks *checkTypeWrong* if the expression is not *boolean* or is *writeOnly*. Otherwise, it type checks to the check status of the body statement.

8.8.3 Loop – Use semantics

$$
\begin{array}{l}
\forall\, WhileStmt;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t';\ \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_E\ \epsilon\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_S\ \gamma\ B\ \rho\upsilon t' \bullet \\
\quad \mathcal{U}_S(whileStmt\ \theta\, WhileStmt)B\ \rho\upsilon t = \rho\upsilon t''\ \diamond\ \rho\upsilon t'
\end{array}
$$

The loop statement alters the use environment by composing the checking of the expression and the body statement in turn (the expression could use variables if it includes a function call).

8.8.4 Loop – Dynamic semantics

If the meaning of the test expression is true in the current state, the meaning of the loop is that of the body, followed by the loop again. If the test is false, the state is unchanged.

$$
\begin{array}{l}
\forall\, WhileStmt;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, ExprValue';\ State'' \mid \\
\qquad \theta ExprValue' = \mathcal{M}_E\ \epsilon\ B\ \rho t\ \sigma \\
\qquad \land\ (v' = vbool\ ptrue \Rightarrow \\
\qquad\quad (\ \exists\, State''' \mid \theta State''' = \mathcal{M}_S\ \gamma\ B\ \rho t\ \theta State' \bullet \\
\qquad\qquad \theta State'' = \\
\qquad\qquad\quad \mathcal{M}_S(whileStmt\ \theta WhileStmt)B\ \rho t\ \theta State''' \ )) \\
\qquad \land\ (v' \neq vbool\ ptrue \Rightarrow \theta State'' = \theta State') \bullet \\
\quad \mathcal{M}_S(whileStmt\ \theta WhileStmt)B\ \rho t\ \sigma = \theta State''
\end{array}
$$

8.9 Procedure call

8.9.1 Procedure call – Symbol declaration semantics

$$
\begin{array}{l}
\forall\, ProcCallStmt;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t';\ \rho\delta t'' : EnvDTrace \mid \\
\qquad \rho\delta t' = checkSymbol\ \xi\ B\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = \mathcal{D}_{AP\ *}\ E\ B\ \rho\delta t \bullet \\
\quad \mathcal{D}_S(procCall\ \theta ProcCallStmt)B\ \rho\delta t = \rho\delta t' \cup \rho\delta t''
\end{array}
$$

The procedure call statement alters the declaration environment to be the worse of the environments of the procedure identifier and the actual parameters.

8.9.2 Procedure call – Type checking semantics

$$
\begin{array}{l}
\forall\, ProcCallStmt;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, \rho\tau : EnvT;\ idt : IDTYPE \mid \\
\qquad \rho\tau = \oplus/\ (\rho\tau t \circ B) \\
\qquad \land\ idt = \rho\tau\ \xi \bullet \\
\quad (idt \in \operatorname{ran} procedure \Rightarrow \\
\qquad (\ \exists\, ProcType';\ c : CHECK \mid \\
\qquad\qquad \theta ProcType' = procedure^{\sim} idt \\
\qquad\qquad \land\ c = \textbf{if}\ \#E = \#I' \land \mathcal{T}_{AP^*}\ E\ I'\ \rho\tau = checkOK \\
\qquad\qquad\qquad \textbf{then}\ checkOK\ \textbf{else}\ checkTypeWrong \bullet \\
\qquad\quad \mathcal{T}_S(procCall\ \theta ProcCallStmt)B\ \rho\tau t = \\
\qquad\qquad (\!|\ \rho\tau t == \rho\tau t, c == c\ |\!)\ )) \\
\quad \land\ (idt \notin \operatorname{ran} procedure \Rightarrow \\
\qquad \mathcal{T}_S(procCall\ \theta ProcCallStmt)B\ \rho\tau t = \\
\qquad\quad (\!|\ \rho\tau t == \rho\tau t, c == checkTypeWrong\ |\!))
\end{array}
$$

The check status of a procedure call is *checkOK* only if there are the correct number of actual parameters, and they all have the correct type.

8.9.3 Procedure call – Use semantics

$$
\begin{array}{l}
\forall\, ProcCallStmt;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, I : \operatorname{seq} IDTYPE;\ b : ID;\ \rho\upsilon t' : EnvUTrace; \\
\qquad\qquad \chi : \mathbb{P}\ USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad I = (procedure^{\sim}(lookup(\xi, B, \rho\tau t0))).I \\
\qquad \land\ b = findBlock(\xi, B, \rho\upsilon t) \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_{AP\ *}\ E\ I\ B\ \xi\ \rho\upsilon t \\
\qquad \land\ \chi = (\rho\upsilon t'\ b)(\!|\ \{\xi\}\ |\!) \setminus \{uncalled\} \\
\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
\quad \mathcal{U}_S(procCall\ \theta ProcCallStmt)B\ \rho\upsilon t = update(\rho\upsilon t', \langle b \rangle, \rho\upsilon)
\end{array}
$$

The list of actual parameters is checked, then the fact that the procedure has been called is recorded.

8.9.4 Procedure call – Dynamic semantics

$$
\begin{array}{l}
\forall\, ProcCallStmt;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, I : \mathrm{seq}\, IDTYPE;\ f : State \ \nrightarrow State;\ State';\ State'' \mid \\
\qquad I = (procedure^{\sim}(lookup(\xi, B, \rho\tau t0))).I \\
\qquad \land\ f = pval^{\sim}(lookup(\xi, B, \rho t)) \\
\qquad \land\ \theta State' = \mathcal{M}_{AP\ *}\ E\ I\ B\ \xi\ \rho t\ \sigma \\
\qquad \land\ \theta State'' = f\ \theta State' \bullet \\
\quad \mathcal{M}_S(procCall\ \theta ProcCallStmt) B\ \rho t\ \sigma = \theta State''
\end{array}
$$

The correct bindings relating actual to formal parameters are carried out, and then the procedure body under the environment is applied to the resulting state.

9 Declarations — Semantics

9.1 Introduction

In this section we specify the semantics attached to declarations.

9.2 Named constant

9.2.1 Named constant – Symbol declaration semantics

The meaning function takes a declaration, the current block, and a declaration trace environment, and produces a new environment.

$$\mathcal{D}_{NC} : ConstDecl \rightarrow ID \rightarrow EnvDTrace \rightarrow EnvDTrace$$

$$\forall\, ConstDecl \bullet \mathcal{D}_{NC}\ \theta ConstDecl = addSymbol\ \xi$$

If a new named constant is being declared, then it is added to the environment as *checkOK*. If an identifier with the same name has already been declared in this block then the name is flagged as *MultiDecl*. This is achieved with the use of the function *addSymbol*.

9.2.2 Named constant – Type checking semantics

The type environment is updated with the value and type of the constant.

$$
\begin{array}{|l}
\mathcal{T}_{NC} : ConstDecl \pfun ID \pfun EnvTTrace \fun EnvTTrace \\
\hline
\forall\, ConstDecl;\ b : ID;\ \rho\tau t : EnvTTrace \mid \\
\qquad \kappa\ \in \operatorname{ran} name \land (name^{\sim}\ \kappa, \langle b \rangle, \rho\tau t) \in \operatorname{dom} lookup \bullet \\
\qquad \exists\, idt, idt' : IDTYPE \mid \\
\qquad\qquad idt = lookup(name^{\sim}\ \kappa, \langle b \rangle, \rho\tau t) \\
\qquad\qquad \land\ idt' = \textbf{if}\ idt \in \operatorname{ran} const \cup \operatorname{ran} enumValue \\
\qquad\qquad\qquad \textbf{then}\ idt\ \textbf{else}\ const(\!|\ v == \kappa, \tau == typeWrong\ |\!) \bullet \\
\qquad\qquad \mathcal{T}_{NC}\ \theta ConstDecl\ b\ \rho\tau t = update(\rho\tau t, \langle b \rangle, \{\xi \mapsto idt'\}) \\
\forall\, ConstDecl;\ b : ID;\ \rho\tau t : EnvTTrace \mid \\
\qquad \kappa \in \operatorname{ran} name \land (name^{\sim}\ \kappa, \langle b \rangle, \rho\tau t) \notin \operatorname{dom} lookup \bullet \\
\qquad \exists\, idt : IDTYPE \mid idt = const(\!|\ v == \kappa, \tau == typeWrong\ |\!) \bullet \\
\qquad\qquad \mathcal{T}_{NC}\ \theta ConstDecl\ b\ \rho\tau t = update(\rho\tau t, \langle b \rangle, \{\xi \mapsto idt\}) \\
\forall\, ConstDecl;\ b : ID;\ \rho\tau t : EnvTTrace \mid \kappa^{\sim} \notin \operatorname{ran} name \bullet \\
\qquad \exists\, idt : IDTYPE \mid idt = const(\!|\ v == \kappa, \tau == constType\ \kappa\ |\!) \bullet \\
\qquad\qquad \mathcal{T}_{NC}\ \theta ConstDecl\ b\ \rho\tau t = update(\rho\tau t, \langle b \rangle, \{\xi \mapsto idt\})
\end{array}
$$

Note that we know that ξ has not been declared previously, from the symbol declaration semantics.

- If the constant value is a name in scope, and if that name is itself a constant or an enumerated value, the constant is given the value and type of the name.
- If the constant value is a name in scope, but that name is not a constant or an enumerated value, there is a type error.
- If the constant value is a name not in scope, there is a type error.
- If the constant value is a literal, the constant is given the value and type of the literal.

9.2.3 Named constant – Use semantics

The environment is updated with the constant name marked as *unread*.

$$\mathcal{U}_{NC} : ConstDecl \pfun STACK \pfun EnvUTrace \pfun EnvUTrace$$

$$\forall\, ConstDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet$$
$$\mathcal{U}_{NC}\ \theta ConstDecl\ B\ \rho\upsilon t = update(\rho\upsilon t, B, \{\xi \mapsto unread\})$$

9.2.4 Named constant – Dynamic semantics

Every named constant identifier is mapped to the *cval* denotable value.

$$\mathcal{M}_{NC} : ConstDecl \pfun STACK \pfun EnvMTrace \pfun EnvMTrace$$

$$\forall\, ConstDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet$$
$$\mathcal{M}_{NC}\ \theta ConstDecl\ B\ \rho t = update(\rho t, B, \{\xi \mapsto cval\})$$

9.3 Type definition

9.3.1 Type definition – Symbol declaration semantics

$$\mathcal{D}_{TD} : TYPE\_DEF \to ID \to EnvDTrace \to EnvDTrace$$

$$\forall\, EnumDecl;\ b : ID \bullet$$
$$\mathcal{D}_{TD}(enumDecl\ \theta EnumDecl)b = addSymbol\ {}_*(\langle\xi\rangle\ \frown \Xi)b$$

The enumerated type name and all its component names are added to the declaration environment.

9.3.2 Type definition – Type checking semantics

The type environment is augmented with the definition of the user defined enumerated type. (The entire block stack is passed as an argument: this is not necessary for enumerated types, but would be required for records, say.)

Checking a type definition adds the scoped type name and the enumerated values to the type environment. For example, checking the enumerated type

TYPE colour = (red, green, blue);

updates the environment as

$$\begin{array}{l}\rho\tau t \oplus \{b \mapsto \\ \quad \rho\tau t\ b \\ \quad \oplus \{colour \mapsto enumType(scopedTypeName(b, colour), \langle red, blue, green\rangle), \\ \qquad red \mapsto enumValue((2,0), scopedTypeValue(b, colour)), \\ \qquad green \mapsto enumValue((2,1), scopedTypeValue(b, colour)), \\ \qquad blue \mapsto enumValue((2,2), scopedTypeValue(b, colour))\}\}\end{array}$$

(with the addition of some casting functions to get the correct branches of free types).

$$\begin{array}{|l} \mathcal{T}_{TD} : TYPE\_DEF \pfun STACK \pfun EnvTTrace \pfun EnvTTrace \\ \hline \forall\, EnumDecl;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\ \quad \exists\, b : ID;\ \tau', \tau : TYPE;\ u : ID \rel IDTYPE \mid \\ \qquad b = last\ B \\ \qquad \land\ \tau' = scopedTypeValue\ \theta ScopedType \\ \qquad \land\ \tau = \textbf{if}\ \#\Xi \leq \#BYTE \\ \qquad\quad \textbf{then}\ scopedTypeName\ \theta ScopedType\ \textbf{else}\ typeWrong \\ \qquad \land\ u = \{\xi \mapsto enumType\ \theta EnumType\} \\ \qquad\quad \cup \{\ n : \mathrm{dom}\,\Xi \bullet \\ \qquad\qquad \Xi\ n \mapsto \\ \qquad\qquad\quad enumValue(\!|\ v == venum(\#\Xi - 1, n - 1), \\ \qquad\qquad\qquad\qquad \tau == \tau'\ |\!|\ \} \bullet \\ \qquad \mathcal{T}_{TD}(enumDecl\ \theta EnumDecl)B\ \rho\tau t = update(\rho\tau t, B, u) \end{array}$$

For an enumerated type, the number of value labels is checked; it must not be greater than 256.

As with variables, if a defined type is incorrectly defined but not used, then this appears in the trace environment, not in the check status.

9.3.3 Type definition – Use semantics

The environment is updated with the type name marked as *unused*.

$$\mathcal{U}_{TD} : TYPE\_DEF \pfun STACK \pfun EnvUTrace \pfun EnvUTrace$$

$$\begin{aligned}
&\forall\, EnumDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
&\quad \mathcal{U}_{TD}(enumDecl\ \theta EnumDecl)B\ \rho\upsilon t = \\
&\qquad update(\rho\upsilon t, B, \{\xi \mapsto unused\})
\end{aligned}$$

9.3.4 Type definition – Dynamic semantics

All information required is extracted from the type trace environment.

9.4 Variable declaration

9.4.1 Variable declaration – Symbol declaration semantics

The variable declaration meaning function takes a variable declaration, the current block stack, and a declaration trace environment, and gives an updated environment with the declaration added.

$$\mathcal{D}_V : VarDecl \to STACK \to EnvDTrace \to EnvDTrace$$

$$\begin{aligned}
&\forall\, VarDecl;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t' : EnvDTrace \mid \\
&\qquad \rho\delta t' = \mathbf{if}\ \tau \in \operatorname{ran} typeName \\
&\qquad\qquad \mathbf{then}\ checkSymbol(typeName^{\sim}\ \tau)B\ \rho\delta t\ \mathbf{else}\ \rho\delta t \bullet \\
&\quad \mathcal{D}_V\ \theta VarDecl\ B\ \rho\delta t = addSymbol\ \xi(last\ B)\rho\delta t'
\end{aligned}$$

If the type is a user defined enumerated type, it is checked. The identifier is added to the environment.

If any identifiers used in the type (for defining an array bound) are not declared, this fact is not detected here; it is trapped by the type checking semantics.

9.4.2 Variable declaration – Type checking semantics

The type checking semantics for variable declarations adds the variable to the type trace environment. The type semantics is complex due to the presence of attributes and array ranges. For the sake of clarity, it is broken down into checks on each component. If each part checks successfully, the variable type is set; the *typeWrong* value is used if any of the checks fail.

9.4.2.1 Attribute checking

$$
\begin{array}{|l}
\mathcal{T}_{A^*} : \operatorname{seq} ATTR \times TYPE \rightarrow CHECK \\
\hline
\forall\, A : \operatorname{seq} ATTR;\ \tau : TYPE \bullet \\
\quad \mathcal{T}_{A^*}(A, \tau) = \\
\qquad \textbf{if } A \in \operatorname{iseq} ATTR \\
\qquad\quad \wedge \operatorname{ran} A \in \bigcup \{\ n : LOCN \bullet \\
\qquad\qquad\qquad \{\varnothing, \{dataAt\ n\}, \{writeOnly, dataAt\ n\}, \\
\qquad\qquad\qquad\quad \{readOnly, dataAt\ n\}, \{nvram, dataAt\ n\}\}\ \} \\
\qquad\quad \wedge\ (\{readOnly, nvram\} \cap \operatorname{ran} A \neq \varnothing \Rightarrow \tau = pbyte) \\
\qquad \textbf{then } checkOK \textbf{ else } attributeWrong
\end{array}
$$

The attribute check ensures that

- the attributes are all different
- the attribute list is either empty, or contains a single *dataAt* attribute, or contains a *writeOnly* attribute with a *dataAt* attribute, or contains a *readOnly* attribute with a *dataAt* attribute, or contains an *nvram* attribute with a *dataAt* attribute. This ensures that read only, write only, and nvram variables are always at specific locations.
- If the variable is read only or nvram, it has byte type

9.4.2.2 Bound checking

This checks that the bound has a value that can be calculated statically, that is, involves only literals or named constants. Only unsigned and byte bounds

are allowed. The type trace environment is used to determine the type of any named constants. This function also returns the numerical value of the bound, so that it can be checked when the whole subrange is checked.

$$
\begin{array}{l}
\mathcal{T}_{Bnd} : BOUND \rightarrow STACK \rightarrow EnvTTrace \nrightarrow \mathbb{N} \times TYPE \\
\hline
\forall\, \kappa : VALUE;\ B : STACK;\ \rho\tau t : EnvTTrace \mid \\
\qquad \kappa \in \operatorname{ran} vunsgn \cup \operatorname{ran} vbyte \bullet \\
\quad \mathcal{T}_{Bnd}\ \kappa\ B\ \rho\tau t = (number\ \kappa, constType\ \kappa) \\
\forall\, \xi : ID;\ B : STACK;\ \rho\tau t : EnvTTrace \mid \\
\qquad (\xi, B, \rho\tau t) \in \operatorname{dom} lookup \bullet \\
\quad \exists\, idt : IDTYPE;\ n : \mathbb{N};\ \tau : TYPE \mid \\
\qquad\quad idt = lookup(\xi, B, \rho\tau t) \\
\qquad\quad \land\ (idt \in \operatorname{ran} const \Rightarrow \\
\qquad\qquad (\ \exists\, ValueType' \mid \theta\, ValueType' = const^{\sim} idt \bullet \\
\qquad\qquad\quad (n, \tau) = \textbf{if}\ \tau \in \{unsigned, pbyte\} \\
\qquad\qquad\qquad\quad \textbf{then}\ (number\ v', \tau')\ \textbf{else}\ (0, typeWrong)\ )) \\
\qquad\quad \land\ (idt \in \operatorname{ran} enumValue \Rightarrow \\
\qquad\qquad (\ \exists\, ValueType' \mid \theta\, ValueType' = enumValue^{\sim} idt \bullet \\
\qquad\qquad\quad (n, \tau) = (number\ v', \tau')\ )) \\
\qquad\quad \land\ (idt \notin \operatorname{ran} const \cup \operatorname{ran} enumValue \Rightarrow \\
\qquad\qquad (n, \tau) = (0, typeWrong)) \bullet \\
\qquad \mathcal{T}_{Bnd}(nameBound\ \xi) B\ \rho\tau t = (n, \tau) \\
\forall\, \xi : ID;\ B : STACK;\ \rho\tau t : EnvTTrace \mid \\
\qquad (\xi, B, \rho\tau t) \notin \operatorname{dom} lookup \bullet \\
\quad \mathcal{T}_{Bnd}(nameBound\ \xi) B\ \rho\tau t = (0, typeWrong) \\
\forall\, \kappa : VALUE;\ B : STACK;\ \rho\tau t : EnvTTrace \mid \\
\qquad \kappa \notin \operatorname{ran} vunsgn \cup \operatorname{ran} vbyte \cup \operatorname{ran} name \bullet \\
\quad \mathcal{T}_{Bnd}\ \kappa\ B\ \rho\tau t = (0, typeWrong)
\end{array}
$$

1. If the bound value is an unsigned or byte number literal, the bound's value and type are given by the literal

2. If the bound value is a name, and that name has been declared, then

   (a) if the name is a named constant, the bound's value and type are given by the constant

(b) if the name is an enumerated value, the bound's value and type are given by the enumeration

(c) otherwise, there is a type error

3. If the bound value is a name, but that name has not been declared, then that is a type error. (So undeclared identifiers used in subranges are trapped here by the type checking semantics.)

4. If the bound value is not a literal or a name, then that is a type error.

If the type is *typeWrong* then the value assigned does not matter; 0 has been chosen arbitrarily.

9.4.2.3 Subrange checking

The subrange type check also substitutes the values of any named constants. This check returns *typeWrong* if the two bounds have different types, and the type of the first bound if they are the same. So if either has type *typeWrong*, then *typeWrong* is returned. The return value is also *typeWrong* if the second bound is smaller than the first.

$$
\begin{array}{|l}
\mathcal{T}_{SR} : Subrange \rightarrow STACK \nrightarrow EnvTTrace \nrightarrow SubrangeType \\
\hline
\forall\, Subrange;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, n, n' : \mathbb{N};\ \tau, \tau' : TYPE;\ SubrangeType'' \mid \\
\qquad (n, \tau) = \mathcal{T}_{Bnd}\ lb\ B\ \rho\tau t \\
\qquad \land\ (n', \tau') = \mathcal{T}_{Bnd}\ ub\ B\ \rho\tau t \\
\qquad \land\ \theta SubrangeType'' = \\
\qquad\quad \textbf{if}\ \tau = \tau' \in \{unsigned, pbyte\} \land n \leq n' \\
\qquad\quad \textbf{then}\ \langle\!|\ lb == n, ub == n', \tau == \tau\ |\!\rangle \\
\qquad\quad \textbf{else}\ \langle\!|\ lb == 0, ub == 0, \tau == typeWrong\ |\!\rangle \bullet \\
\quad \mathcal{T}_{SR}\ \theta Subrange\ B\ \rho\tau t = \theta SubrangeType''
\end{array}
$$

9.4.2.4 Array range checking

The empty range (that is, simple variable rather than array) is arbitrarily given a type of *unsigned*. In a non-empty sequence, the subranges must all have the same type (checked by *typeMerge*).

$$
\begin{array}{|l}
\mathcal{T}_{SR^*} : \operatorname{seq} Subrange \pfun STACK \pfun EnvTTrace \pfun ArrayType \\
\hline
\forall B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \mathcal{T}_{SR^*} \langle\ \rangle\ B\ \rho\tau t = (\!|\ SR == \langle\ \rangle, \tau a == unsigned\ |\!) \\
\forall s : Subrange;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists SubrangeType \mid \theta SubrangeType = \mathcal{T}_{SR}\ s\ B\ \rho\tau t \bullet \\
\qquad \mathcal{T}_{SR^*} \langle s \rangle\ B\ \rho\tau t = (\!|\ SR == \langle \theta SubrangeN \rangle, \tau a == \tau\ |\!) \\
\forall s : Subrange;\ sr : \operatorname{seq}_1 Subrange;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists SubrangeType;\ ArrayType;\ ArrayType' \mid \\
\qquad \theta SubrangeType = \mathcal{T}_{SR}\ s\ B\ \rho\tau t \\
\qquad \land\ \theta ArrayType = \mathcal{T}_{SR^*}\ sr\ B\ \rho\tau t \\
\qquad \land\ SR' = \langle \theta SubrangeN \rangle\ \frown SR \\
\qquad \land\ \tau a' = mergeTypes(\tau, \tau a) \bullet \\
\quad \mathcal{T}_{SR^*}(\langle s \rangle \frown sr) B\ \rho\tau t = \theta ArrayType'
\end{array}
$$

9.4.2.5 Simple type checking

This returns the numerical value of the bounds.

$$\mathcal{T}_T : TYPE \pfun STACK \pfun EnvTTrace \pfun CHECK \times SubrangeType$$

$$\begin{array}{l}
\forall\, B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \mathcal{T}_T\ unsigned\ B\ \rho\tau t = \\
\qquad (checkOK, \langlebag\ lb == 0, ub == \#UNSIGNED - 1, \\
\qquad\qquad \tau == unsigned\ \ranglebag) \\
\quad \land\ \mathcal{T}_T\ pbyte\ B\ \rho\tau t = \\
\qquad (checkOK, \langlebag\ lb == 0, ub == 255, \tau == pbyte\ \ranglebag) \\
\quad \land\ \mathcal{T}_T\ boolean\ B\ \rho\tau t = \\
\qquad (checkOK, \langlebag\ lb == 0, ub == 1, \tau == boolean\ \ranglebag) \\
\quad \land\ \mathcal{T}_T\ typeWrong\ B\ \rho\tau t = \\
\qquad (checkTypeWrong, \langlebag\ lb == 0, ub == 0, \tau == typeWrong\ \ranglebag) \\
\forall\, sr : Subrange;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, SubrangeType;\ c : CHECK \mid \\
\qquad \theta SubrangeType = \mathcal{T}_{SR}\ sr\ B\ \rho\tau t \\
\qquad \land\ c = \textbf{if}\ \tau = typeWrong \\
\qquad\qquad \textbf{then}\ checkTypeWrong\ \textbf{else}\ checkOK \bullet \\
\quad \mathcal{T}_T(subrange\ sr) B\ \rho\tau t = (c, \theta SubrangeType) \\
\forall\, \xi : ID;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, idt : IDTYPE;\ EnumType;\ n : \mathbb{N};\ c : CHECK \mid \\
\qquad idt = lookup(\xi, B, \rho\tau t) \\
\qquad \land\ \theta EnumType = enumType^{\sim}\ idt \\
\qquad \land\ n = \#\Xi - 1 \\
\qquad \land\ c = \textbf{if}\ \tau = typeWrong \\
\qquad\qquad \textbf{then}\ checkTypeWrong\ \textbf{else}\ checkOK \bullet \\
\quad \mathcal{T}_T(typeName\ \xi) B\ \rho\tau t = (c, \langlebag\ lb == 0, ub == n, \tau == \tau\ \ranglebag)
\end{array}$$

9.4.2.6 Initialisation value type checking

initType returns the type of an initialisation value (which may be a literal value, the name of a constant, or the name of an enumerated value).

$$
\begin{array}{|l}
initType : VALUE \pfun STACK \pfun EnvTTrace \pfun TYPE \\
\hline
\forall\, v : VALUE;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad (v \in \operatorname{ran} name \Rightarrow \\
\quad\quad (\ \exists \xi : ID \mid \xi = name^{\sim} v \bullet \\
\quad\quad\quad initType\ v\ B\ \rho\tau t = \\
\quad\quad\quad\quad \mathbf{if} (\xi, B, \rho\tau t) \in \operatorname{dom} lookup \\
\quad\quad\quad\quad\quad \wedge\ lookup(\xi, B, \rho\tau t) \in \operatorname{ran} const \cup \operatorname{ran} enumValue \\
\quad\quad\quad\quad \mathbf{then}\ typeOf(lookup(\xi, B, \rho\tau t)) \\
\quad\quad\quad\quad \mathbf{else}\ typeWrong\ )) \\
\quad \wedge\ (v \notin \operatorname{ran} name \Rightarrow initType\ v\ B\ \rho\tau t = constType\ v)
\end{array}
$$

- If the value is a *name*, it is an error if has not been declared, or declared as something other than a constant or an enumerated value
- If the value is not a *name*, its type is that given by the literal constant

initTypeSeq returns the sequence of types of a sequence of initialisation values.

$$
\begin{array}{|l}
initTypeSeq : \operatorname{seq} VALUE \pfun STACK \pfun EnvTTrace \pfun \operatorname{seq} TYPE \\
\hline
\forall\, B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad initTypeSeq\langle\ \rangle B\ \rho\tau t = \langle\ \rangle \\
\forall\, v : VALUE;\ V : \operatorname{seq} VALUE;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad initTypeSeq(\langle v \rangle \frown V) B\ \rho\tau t = \\
\quad\quad \langle initType\ v\ B\ \rho\tau t \rangle \frown initTypeSeq\ V\ B\ \rho\tau t
\end{array}
$$

The initialisation value type checking function takes the variable declaration, the low and high bounds of the variable's allowed value, and the number of elements in the array being declared, and returns a check value.

$$
\begin{array}{|l}
\mathcal{T}_I : VarDecl \times SubrangeN \times \mathbb{N} \pfun STACK \pfun EnvTTrace \pfun CHECK \\
\hline
\forall\, VarDecl;\ SubrangeN;\ n : \mathbb{N};\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, T : \mathbb{P}\, TYPE;\ A' : \mathbb{P}\, ATTR;\ c : CHECK \mid \\
\qquad T = \mathrm{ran}(initTypeSeq\ V\ B\ \rho\tau t) \\
\qquad \land\ A' = \mathrm{ran}\, A \cap \{readOnly, writeOnly, nvram\} \\
\qquad \land\ c = \mathbf{if}(A' \neq \varnothing \land \# T = 0 \\
\qquad\qquad \lor\ A' = \varnothing \land T = \{\tau\} \land \# V \in \{1, n\}) \\
\qquad\qquad \land\ (\forall\, i : \mathrm{dom}\, V \bullet l \leq number(V\ i) \leq ub) \\
\qquad\quad \mathbf{then}\ checkOK\ \mathbf{else}\ checkTypeWrong \bullet \\
\quad \mathcal{T}_I(\theta\, VarDecl, \theta SubrangeN, n) B\ \rho\tau t = c
\end{array}
$$

- T : a set containing the types of the variable's initialisation values
- A' : a set containing the variable's non-at attributes (at most one of *readOnly*, *writeOnly* or *nvram*)
- $A' \neq \varnothing \land \# T = 0$: if the variable has non-at attributes, then it must not be initialised.
- $A' = \varnothing \land T = \{\tau\} \land \# V \in \{1, n\}$: if the variable has no non-at attributes, then it must be initialised. All the initialisation values must be the same type as the variable, and there must be either one of them (simple initialisation) or one per array element.
- $\forall\, i : \mathrm{dom}\, V \bullet \ldots$: all the initialisation values must lie within the bounds of the variable's allowed value.

9.4.2.7 Variable declaration type checking

Array ranges are either unsigned values or bytes.

$$
\begin{array}{|l}
\mathcal{T}_V : VarDecl \pfun STACK \pfun EnvTTrace \fun EnvTTrace \\
\hline
\forall\, VarDecl;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, VarType';\ c, c' : CHECK;\ idt : IDTYPE \mid \\
\qquad (c', \theta SubrangeType') = \mathcal{T}_T\ \tau\ B\ \rho\tau t \\
\qquad \land\ \theta ArrayType' = \mathcal{T}_{SR^*}\ SR\ B\ \rho\tau t \\
\qquad \land\ c = \mathcal{T}_I(\theta VarDecl, \theta SubrangeN', numElts\ SR')B\ \rho\tau t \\
\qquad \land\ A' = \operatorname{ran} A \\
\qquad \land\ idt = \textbf{if}\ \mathcal{T}_{A^*}(A, \tau) \bowtie c \bowtie c' = checkOK \\
\qquad\quad \textbf{then}\ variable\ \theta VarType' \\
\qquad\quad \textbf{else}\ variable(\!|\ A == \varnothing, SR == \langle\ \rangle, \tau a == \tau a', \\
\qquad\qquad lb == 0, ub == 0, \tau == typeWrong\ |\!) \bullet \\
\quad \mathcal{T}_V\ \theta VarDecl\ B\ \rho\tau t = update(\rho\tau t, B, \{\xi \mapsto idt\})
\end{array}
$$

We know that ξ is not already declared, because the symbol declaration semantics have checked correctly.

The last block in the stack is the current block, and so the type environment for this block is updated.

9.4.3 Variable declaration – Use semantics

First we type check the use of enumerated types in the type of the variable declaration.

$$
\begin{array}{|l}
\mathcal{U}_T : TYPE \pfun STACK \pfun EnvUTrace \pfun EnvUTrace \\
\hline
\forall \tau : TYPE;\ B : STACK;\ \rho\upsilon t : EnvUTrace \mid \tau \in \operatorname{ran} typeName \bullet \\
\quad \exists\, ScopedType;\ \Xi : \operatorname{seq}_1 ID;\ \chi : \mathbb{P}\, USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad \tau = typeName\ \xi \land b = last\ B \\
\qquad \land\ lookup(\xi, B, \rho\tau t0) = \\
\qquad\quad enumType(\!|\ \tau == scopedTypeName\ \theta ScopedType, \\
\qquad\qquad \Xi == \Xi\ |\!) \\
\qquad \land\ \chi = (\rho\upsilon t\ b)(\!|\ \{\xi\}\ |\!) \setminus \{unused\} \\
\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
\quad \mathcal{U}_T\ \tau\ B\ \rho\upsilon t = update(\rho\upsilon t, B, \rho\upsilon) \\
\forall \tau : TYPE;\ B : STACK;\ \rho\upsilon t : EnvUTrace \mid \tau \notin \operatorname{ran} typeName \bullet \\
\quad \mathcal{U}_T\ \tau\ B\ \rho\upsilon t = \rho\upsilon t
\end{array}
$$

The definition of the use semantics for variable declarations is:

$$
\begin{array}{|l}
\mathcal{U}_V : VarDecl \pfun STACK \pfun EnvUTrace \pfun EnvUTrace \\
\hline
\forall\, VarDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t' : EnvUTrace;\ A' : \mathbb{P}\, ATTR;\ \chi : \mathbb{P}\, USECHECK;\ \rho\upsilon : EnvU \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_T\ \tau\ B\ \rho\upsilon t \land A' = \operatorname{ran} A \\
\qquad \land\ \chi = \textbf{if}\ readOnly \in A' \\
\qquad\quad \textbf{then}\ \{unread\} \\
\qquad\quad \textbf{else if}\ writeOnly \in A' \\
\qquad\qquad \textbf{then}\ \{unwritten\}\ \textbf{else}\ \{unread, unwritten\} \\
\qquad \land\ \rho\upsilon = \{\xi\} \times \chi \bullet \\
\quad \mathcal{U}_V\ \theta VarDecl\ B\ \rho\upsilon t = update\rho\upsilon t', B, \rho\upsilon)
\end{array}
$$

The type of the declaration is checked for use of user defined types. If the variable is read only, it is set to *unread*; if it is write only, it is set to *unwritten*; otherwise it is set to both *unread* and *unwritten*.

9.4.4 Variable declaration – Dynamic semantics

We define an auxiliary function that returns the locations used by a dynamic environment.

$$
\begin{array}{|l}
usedLocations : EnvMTrace \pfun \mathbb{P}\, LOCN \\
\hline
\forall\, \rho t : EnvMTrace \bullet \\
\quad usedLocations\ \rho t = \\
\qquad \bigcup \{\ b : \operatorname{dom} \rho;\ \xi : ID;\ f : \operatorname{seq} \mathbb{N} \pinj LOCN \mid \\
\qquad\quad \xi \in \operatorname{dom}(\rho t\ b) \land \rho t\ b\ \xi = loc\ f \bullet \\
\qquad\quad \operatorname{ran} f\ \}
\end{array}
$$

The locations allocated on declaration are

$$
\begin{array}{|l}
allocate : IDTYPE \times \mathbb{P}\, LOCN \pfun \operatorname{seq}\mathbb{N} \pinj LOCN \\
\hline
\forall\, i : IDTYPE;\ used : \mathbb{P}\, LOCN \bullet \\
\quad \exists\, NN : \operatorname{seq} SubrangeN;\ l : LOCN;\ f : \operatorname{seq}\mathbb{N} \pinj LOCN \mid \\
\qquad NN = arrayOf\ i \\
\qquad \land\ \operatorname{dom} f = \\
\qquad\quad \{\ N : \operatorname{seq}\mathbb{N} \mid \\
\qquad\qquad \#N = \#NN \\
\qquad\qquad \land\ (\forall\, n : 1\ \ldots \#N \bullet \\
\qquad\qquad\quad N\ n\ \in (NN\ n).lb\ \ldots (NN\ n).ub)\ \} \\
\qquad \land\ (\ \forall\, N : \operatorname{dom} f \bullet f\ N = l + locationOffset(N, NN)\ ) \\
\qquad \land\ \mathsf{disjoint}\langle \operatorname{ran} f, used\rangle \bullet \\
\quad allocate(i, used) = f
\end{array}
$$

This (loosely) specifies the function f, mapping array indices to locations with the following properties:

- $\mathsf{disjoint}\langle \operatorname{ran} f, used\rangle$
  The locations in f are distinct from any already allocated.
- $\operatorname{dom} f = \{N : \operatorname{seq}\mathbb{N} \mid \#N = \#NN \ldots\}$
  All the sequences of values in f are the same length as the sequence of subranges (NN is the sequence of numerical value pairs defining the array bounds derived from the original subranges).
- $\operatorname{dom} f = \{N : \operatorname{seq}\mathbb{N} \mid \ldots\}$
  Precisely those sequences of values between all lower and upper bounds are in f.
- $(\ \forall\, N : \operatorname{dom} f \bullet f\ N = l + locationOffset(N, NN)\ )$
  Each array element is situated at *locationOffset* from the first element's location l.

Note that if ξ is a zero-dimensional variable, then $\#NN = 0$. So the empty sequence is the sole member of the domain of f, and a single new location is added to the environment, as required.

The dynamic semantics of variable declaration is split into two parts: the change to the environment due to locations being allocated, and the change in

the state due to initialisation. These are separated because the environment change occurs once (on declaration), whereas the initialisation may occur several times (on each call of a procedure or function).

The allocation change to the environment is

$$
\begin{array}{|l}
\mathcal{M}_V : VarDecl \pfun STACK \pfun EnvMTrace \pfun EnvMTrace \\
\hline
\forall\, VarDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
\quad \exists\, i : IDTYPE;\ used : \mathbb{P}\, LOCN;\ f : \operatorname{seq}\mathbb{N} \pinj LOCN \mid \\
\qquad i = lookup(\xi, B, \rho\tau t0) \\
\qquad \land\ used = usedLocations\ \rho t \\
\qquad \land\ f = allocate(i, used) \bullet \\
\quad \mathcal{M}_V\ \theta VarDecl\ B\ \rho t = update(\rho t, B, \{\xi \mapsto loc\ f\})
\end{array}
$$

initValue returns the numerical value of an initialisation value (which may be a literal value, the name of a constant, or the name of an enumerated value).

$$
\begin{array}{|l}
initValue : VALUE \pfun STACK \pfun TYPE \\
\hline
\forall\, v : VALUE;\ B : STACK \bullet \\
\quad (v \in \operatorname{ran} name \Rightarrow \\
\qquad (\ \exists \xi : ID \mid \xi = name^{\sim} v \bullet \\
\qquad\quad initValue\ v\ B = valueOf(lookup(\xi, B, \rho\tau t0))\ )) \\
\quad \land\ (v \notin \operatorname{ran} name \Rightarrow initValue\ v\ B = v)
\end{array}
$$

The initialisation change to the state is

$$
\begin{array}{|l}
\mathcal{M}_{VI} : VarDecl \pfun STACK \pfun EnvMTrace \pfun State \pfun State \\
\hline
\forall\, VarDecl;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, L : \mathbb{P}\, LOCN;\ Ls : \operatorname{seq} LOCN;\ State;\ \Sigma' : Store \mid \\
\qquad L = \operatorname{ran}(loc^{\sim}(lookup(\xi, B, \rho t))) \\
\qquad \land\ \operatorname{ran} Ls = L \\
\qquad \land\ \theta State = \sigma \\
\qquad \land\ (\forall\, i : 2 \upto \# Ls \bullet Ls(i-1) < Ls\ i) \\
\qquad \land\ (\# V = 0 \Rightarrow \Sigma' = \Sigma) \\
\qquad \land\ (\# V = 1 \land \# L = 1 \Rightarrow \\
\qquad\quad \Sigma' = \Sigma \oplus \{\ l : L;\ v : \operatorname{ran} V \bullet l \mapsto initValue\ v\ B\ \}) \\
\qquad \land\ (\# V = 1 \land 1 < \# L \Rightarrow \\
\qquad\quad \Sigma' = \Sigma \oplus \{\ i : 1 \upto \# Ls;\ v : \operatorname{ran} V \bullet \\
\qquad\qquad\qquad Ls\ i \mapsto initValue\ v\ B\ \}) \\
\qquad \land\ (1 < \# V \Rightarrow \\
\qquad\quad \Sigma' = \Sigma \oplus \{\ i : 1 \upto \# Ls \bullet Ls\ i \mapsto initValue(V\ i) B\ \}) \bullet \\
\quad \mathcal{M}_{VI}\ \theta VarDecl\ B\ \rho t\ \sigma = (\!|\ \Sigma == \Sigma', i == i, o == o\ |\!)
\end{array}
$$

- Ls : $\operatorname{seq} LOCATION$ a sequence of the locations allocated to ξ, in ascending order
- $\# V = 0$: no initialisation values, so no change to the store
- $\# V = 1 \land \# L = 1$: one initialisation value, one location, so a simple variable initialisation
- $\# V = 1 \land 1 < \# L$: one initialisation value, several locations, so a block array initialisation
- $1 < \# V$: several initialisation values, so an array initialisation

9.5 Simple declarations

The meaning of a simple declaration is the meaning of the relevant free type branch.

Where the branch definition uses a block *ID*, this refers to the last block in the stack. Where the branch definition is absent, the identity mapping of the environment is applied.

$$\mathcal{D}_{SD} : SIMPLE\_DECL \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

$$\begin{aligned}
&\forall\, \delta : ConstDecl;\ B : STACK \bullet \mathcal{D}_{SD}(constDecl\ \delta)B = \mathcal{D}_{NC}\ \delta(last\ B)\\
&\forall\, \delta : TYPE_DEF;\ B : STACK \bullet \mathcal{D}_{SD}(typeDecl\ \delta)B = \mathcal{D}_{TD}\ \delta(last\ B)\\
&\forall\, \delta : VarDecl \bullet \mathcal{D}_{SD}(varDecl\ \delta) = \mathcal{D}_{V}\ \delta
\end{aligned}$$

$$DeclType \mathrel{\widehat=} [\ \rho\tau t : EnvTTrace;\ tc : Env[CHECK]\ ]$$

$$\mathcal{T}_{SD} : SIMPLE\_DECL \nrightarrow STACK \nrightarrow DeclType \nrightarrow DeclType$$

$$\begin{aligned}
&\forall\, \delta : ConstDecl;\ B : STACK;\ DeclType \bullet\\
&\quad \mathcal{T}_{SD}(constDecl\ \delta)B\ \theta DeclType =\\
&\qquad \langle\!|\ \rho\tau t == \mathcal{T}_{NC}\ \delta(last\ B)\rho\tau t, tc == tc\ |\!\rangle\\
&\forall\, \delta : TYPE_DEF;\ B : STACK;\ DeclType \bullet\\
&\quad \mathcal{T}_{SD}(typeDecl\ \delta)B\ \theta DeclType =\\
&\qquad \langle\!|\ \rho\tau t == \mathcal{T}_{TD}\ \delta\ B\ \rho\tau t, tc == tc\ |\!\rangle\\
&\forall\, \delta : VarDecl;\ B : STACK;\ DeclType \bullet\\
&\quad \mathcal{T}_{SD}(varDecl\ \delta)B\ \theta DeclType =\\
&\qquad \langle\!|\ \rho\tau t == \mathcal{T}_{V}\ \delta\ B\ \rho\tau t, tc == tc\ |\!\rangle
\end{aligned}$$

$$\mathcal{U}_{SD} : SIMPLE\_DECL \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace$$

$$\begin{aligned}
&\forall\, \delta : ConstDecl;\ B : STACK \bullet \mathcal{U}_{SD}(constDecl\ \delta)B = \mathrm{id}\ EnvUTrace\\
&\forall\, \delta : TYPE_DEF;\ B : STACK \bullet t1\mathcal{U}_{SD}(typeDecl\ \delta)B = \mathrm{id}\ EnvUTrace\\
&\forall\, \delta : VarDecl \bullet \mathcal{U}_{SD}(varDecl\ \delta) = \mathcal{U}_{V}\ \delta
\end{aligned}$$

The dynamic semantics for simple declarations is split into two parts: location allocation and initialisation.

The allocation change to the environment is

$$
\begin{array}{|l}
\mathcal{M}_{SD} : SIMPLE\_DECL \pfun STACK \pfun EnvMTrace \pfun EnvMTrace \\
\hline
\forall\, \delta : ConstDecl \bullet \mathcal{M}_{SD}(constDecl\ \delta) = \mathcal{M}_{NC}\ \delta \\
\forall\, \delta : TYPE\_DEF;\ B : STACK \bullet \mathcal{M}_{SD}(typeDecl\ \delta)B = \mathrm{id}\, EnvMTrace \\
\forall\, \delta : VarDecl \bullet \mathcal{M}_{SD}(varDecl\ \delta) = \mathcal{M}_{V}\ \delta
\end{array}
$$

The initialisation change to the state is

$$
\begin{array}{|l}
\mathcal{M}_{SDI} : SIMPLE\_DECL \pfun STACK \pfun EnvMTrace \pfun State \pfun State \\
\hline
\forall\, \delta : ConstDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
\quad \mathcal{M}_{SDI}(constDecl\ \delta)B\ \rho t = \mathrm{id}\, State \\
\forall\, \delta : TYPE\_DEF;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
\quad \mathcal{M}_{SDI}(typeDecl\ \delta)B\ \rho t = \mathrm{id}\, State \\
\forall\, \delta : VarDecl;\ B : STACK;\ \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \mathcal{M}_{SDI}(varDecl\ \delta)B\ \rho t\ \sigma = \mathcal{M}_{VI}\ \delta\ B\ \rho t\ \sigma
\end{array}
$$

Multiple declaration initialisation is the state change resulting from the composition of the individual state changes.

$$
\begin{array}{|l}
\mathcal{M}_{SDI^*} : \mathrm{seq}\, SIMPLE\_DECL \pfun \\
\qquad STACK \pfun EnvMTrace \pfun State \pfun State \\
\hline
\forall\, B : STACK;\ \rho t : EnvMTrace \bullet \mathcal{M}_{SDI^*}\langle\,\rangle\ B\ \rho t = \mathrm{id}\, State \\
\forall\, \delta : SIMPLE\_DECL;\ \Delta : \mathrm{seq}\, SIMPLE\_DECL;\ B : STACK; \\
\qquad \rho t : EnvMTrace;\ \sigma : State \bullet \\
\quad \exists\, \sigma', \sigma'' : State \mid \\
\qquad \sigma' = \mathcal{M}_{SDI}\ \delta\ B\ \rho t\ \sigma \\
\qquad \land\ \sigma'' = \mathcal{M}_{SDI^*}\ \Delta\ B\ \rho t\ \sigma' \bullet \\
\qquad \mathcal{M}_{SDI^*}(\langle\delta\rangle\ \frown \Delta)B\ \rho t\ \sigma = \sigma''
\end{array}
$$

9.6 Procedure declaration

9.6.1 Procedure declaration – Declaration before use semantics

Because of the complexity of procedure declarations, the checking is broken down into checks on the individual components.

First we define how to check the declaration of the formal parameters. The name of the parameter is added to the environment, and if the type is user defined, it is checked.

$$\mathcal{D}_{PMD} : ParamDecl \to STACK \to EnvDTrace \to EnvDTrace$$

$$\begin{aligned}
&\forall\, ParamDecl;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t' : EnvDTrace \mid \\
&\qquad \rho\delta t' = \mathbf{if}\ \tau \in \operatorname{ran} typeName \\
&\qquad\quad \mathbf{then}\ checkSymbol(typeName^{\sim}\ \tau) B\ \rho\delta t \\
&\qquad\quad \mathbf{else}\ \rho\delta t \bullet \\
&\quad \mathcal{D}_{PMD}\ \theta ParamDecl\ B\ \rho\delta t = addSymbol\ \xi(last\ B)\rho\delta t'
\end{aligned}$$

To check the procedure body, we check the local declarations, and then check the body statement, all in the given block.

$$\mathcal{D}_B : Body \to STACK \to EnvDTrace \to EnvDTrace$$

$$\begin{aligned}
&\forall\, Body;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t', \rho\delta t'' : EnvDTrace \mid \\
&\qquad \rho\delta t' = \mathcal{D}_{SD}\ {}_*\ \Delta SD\ B\ \rho\delta t \\
&\qquad \land\ \rho\delta t'' = \mathcal{D}_S\ \gamma\ B\ \rho\delta t' \bullet \\
&\quad \mathcal{D}_B\ \theta Body\ B\ \rho\delta t = \rho\delta t''
\end{aligned}$$

Putting these definitions together, we obtain the following semantics, where the parameters and body are checked in the nested block (whose name is the same as that of the procedure), and then the procedure name is added to the surrounding block.

$$\mathcal{D}_{PD} : ProcDecl \to STACK \to EnvDTrace \to EnvDTrace$$

$$\begin{aligned}
&\forall\, ProcDecl;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, B' : STACK;\ \rho\delta t', \rho\delta t'' : EnvDTrace \mid \\
&\qquad B' = B\ ^\frown \langle \xi \rangle \\
&\qquad \land\ \rho\delta t' = \mathcal{D}_{PMD}\ {}_*\ \Pi\ B'(\rho\delta t \oplus \{\xi \mapsto \varnothing\}) \\
&\qquad \land\ \rho\delta t'' = \mathcal{D}_B\ \theta Body\ B'\ \rho\delta t' \bullet \\
&\quad \mathcal{D}_{PD}\ \theta ProcDecl\ B\ \rho\delta t = addSymbol\ \xi(last\ B)\rho\delta t''
\end{aligned}$$

9.6.2 Procedure declaration – Type checking semantics

Each formal parameter is added to the type trace environment.

$$
\begin{array}{|l}
\mathcal{T}_{PMD} : ParamDecl \pfun STACK \pfun EnvTTrace \pfun EnvTTrace \\
\hline
\forall\, ParamDecl;\ B : STACK;\ \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, FormalType';\ idt : IDTYPE;\ \chi : CHECK \mid \\
\qquad \xi' = \xi \land c' = c \land A' = \operatorname{ran} A \\
\qquad \land\ (\chi, \theta SubrangeType') = \mathcal{T}_T\ \tau\ B\ \rho\tau t \\
\qquad \land\ \theta ArrayType' = \mathcal{T}_{SR^*}\ SR\ B\ \rho\tau t \\
\qquad \land\ idt = \textbf{if}\ \mathcal{T}_{A^*}(A, \tau) \bowtie \chi = checkOK \\
\qquad\qquad \land\ A' \subset \{readOnly, writeOnly\} \\
\qquad\qquad \land\ (c = ref \lor \#SR = 0 \land A' = \varnothing) \\
\qquad\quad \textbf{then}\ formalParam\ \theta FormalType' \\
\qquad\quad \textbf{else}\ formalParam \langlebar\, \xi == \xi, c == c, A == A', \\
\qquad\qquad\qquad SR == \langle\ \rangle, \tau a == \tau a', lb == 0, ub == 0, \\
\qquad\qquad\qquad \tau == typeWrong \,\ranglebar \bullet \\
\qquad \mathcal{T}_{PMD}\ \theta ParamDecl\ B\ \rho\tau t = update(\rho\tau t, B, \{\xi \mapsto idt\})
\end{array}
$$

No attribute other than read only or write only is allowed for a formal parameter. Also, the parameter must be called by reference *unless* it is a simple variable (that is, is not an array and has no attributes).

Type checking the procedure body carries out two functions. The environment is updated with the types of the local variables and constants, and the type usage of the body statement is checked. The block is labelled with this check status in the mapping *BlockTypeCheck*.

$$
\begin{array}{|l}
\mathcal{T}_B : Body \pfun STACK \pfun DeclType \pfun DeclType \\
\hline
\forall\, Body;\ B : STACK;\ DeclType \bullet \\
\quad \exists\, DeclType';\ StmtType'';\ tc'' : Env[CHECK] \mid \\
\qquad \theta DeclType' = \mathcal{T}_{SD\ *}\ \Delta SD\ B\ \theta DeclType \\
\qquad \land\ \theta StmtType'' = \mathcal{T}_S\ \gamma\ B\ \rho\tau t' \\
\qquad \land\ tc'' = tc' \oplus \{last\ B \mapsto c''\} \bullet \\
\qquad \mathcal{T}_B\ \theta Body\ B\ \theta DeclType = \langlebar\, \rho\tau t == \rho\tau t'', tc == tc'' \,\ranglebar
\end{array}
$$

We define a function that extracts the parameter names from a sequence of

parameter declarations.

$$
\begin{array}{|l}
extractParamIds : \operatorname{seq} ParamDecl \rightarrow \operatorname{seq} ID \\
\hline
extractParamIds\langle\,\rangle = \langle\,\rangle \\
\forall\, ParamDecl;\ \Pi : \operatorname{seq} ParamDecl \bullet \\
\quad extractParamIds(\langle \theta ParamDecl \rangle \frown \Pi) = \langle \xi \rangle \frown extractParamIds\ \Pi
\end{array}
$$

We now define the type checking semantics of procedure declarations.

$$
\begin{array}{|l}
\mathcal{T}_{PD} : ProcDecl \nrightarrow STACK \nrightarrow DeclType \nrightarrow DeclType \\
\hline
\forall\, ProcDecl;\ B : STACK;\ DeclType \bullet \\
\quad \exists\, ProcType';\ \rho\tau t', \rho\tau t''' : EnvTTrace;\ DeclType'' ; \\
\qquad\qquad \Xi : \operatorname{seq} ID;\ \rho\tau : EnvT \mid \\
\qquad\qquad B' = B\ \frown \langle \xi \rangle \\
\qquad\qquad \land\ \rho\tau t' = \mathcal{T}_{PMD}\ {}_{*}\ \Pi\ B'(\rho\tau t \oplus \{\xi \mapsto \varnothing\}) \\
\qquad\qquad \land\ \theta DeclType'' = \mathcal{T}_B\ \theta Body\ B' \llparenthesis\ \rho\tau t == \rho\tau t', tc == tc\ \rrparenthesis \\
\qquad\qquad \land\ \Xi = extractParamIds\ \Pi \\
\qquad\qquad \land\ I' = \rho\tau t''\ \xi \circ \Xi \\
\qquad\qquad \land\ \rho\tau = \{\xi \mapsto procedure\ \theta ProcType'\} \\
\qquad\qquad \land\ \rho\tau t''' = update(\rho\tau t'', B, \rho\tau) \bullet \\
\qquad \mathcal{T}_{PD}\ \theta ProcDecl\ B\ \theta DeclType = \llparenthesis\ \rho\tau t == \rho\tau t''', tc == tc''\ \rrparenthesis
\end{array}
$$

The procedure identifier itself is entered into the type environment with the block stack relevant to the procedure body and the types of the parameters. The parameters are declared inside the block labelled by the procedure name.

9.6.3 Procedure declaration – Use semantics

Most of the information required from the declarations of the formal parameters, local constants, types, and variables is extracted from the type trace environment. The usage of defined types in parameter and variable declarations is checked.

$$
\begin{array}{|l}
\mathcal{U}_{PMD} : ParamDecl \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace \\
\hline
\forall\, ParamDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t' : EnvUTrace;\ \rho\upsilon : EnvU \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_T\ \tau\ B\ \rho\upsilon t \\
\qquad \rho\upsilon = \{\xi\} \times \{unread, unwritten\} \bullet \\
\quad \mathcal{U}_{PMD}\ \theta ParamDecl\ B\ \rho\upsilon t = update(\rho\upsilon t', B, \rho\upsilon)
\end{array}
$$

The use semantics of the body is just the use semantics of the local variable declarations. In addition a use value is returned that contains the body statement and the procedure name marked as *Unused*.

$$
\begin{array}{|l}
\mathcal{U}_B : Body \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace \\
\hline
\forall\, Body;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, \rho\upsilon t', \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t' = \mathcal{U}_{SD}\ {}_*\ \Delta SD\ B\ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_S\ \gamma\ B\ \rho\upsilon t' \bullet \\
\quad \mathcal{U}_B\ \theta Body\ B\ \rho\upsilon t = \rho\upsilon t''
\end{array}
$$

All that is then required is to apply the semantics of the parameter declarations and the body.

$$
\begin{array}{|l}
\mathcal{U}_{PD} : ProcDecl \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace \\
\hline
\forall\, ProcDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, B' : STACK;\ \rho\upsilon t', \rho\upsilon t'' : EnvUTrace \mid \\
\qquad B' = B\ ^\frown \langle \xi \rangle \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_{PMD}\ {}_*\ \Pi\ B'(\rho\upsilon t \oplus \{\xi \mapsto \varnothing\}) \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_B\ \theta Body\ B'\ \rho\upsilon t' \bullet \\
\quad \mathcal{U}_{PD}\ \theta ProcDecl\ B\ \rho\upsilon t = update(\rho\upsilon t'', B, \{\xi \mapsto uncalled\})
\end{array}
$$

9.6.4 Procedure declaration – Dynamic semantics

The dynamic semantics of parameter declarations allocates a single location for the formal parameter. This is used to store the value of a call-by-value actual parameter (which would be a simple variable only, not an array) and to store the start location of a call-by-reference actual parameter.

$$\mathcal{M}_{PMD} : ParamDecl \pfun STACK \pfun EnvMTrace \pfun EnvMTrace$$

$$\begin{aligned}
&\forall \pi : ParamDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
&\quad \exists \xi : ID;\ l : LOCN;\ f : \mathrm{seq}\ VALUE \pinj LOCN \mid \\
&\qquad \langle \xi \rangle = extractParamIds\ \langle \pi \rangle \\
&\qquad \land\ l \notin usedLocations\ \rho t \bullet \\
&\quad \mathcal{M}_{PMD}\ \pi\ B\ \rho t = update(\rho t, B, \{\xi \mapsto loc\{\langle\ \rangle \mapsto l\}\})
\end{aligned}$$

The procedure body is mapped to the meaning of the declaration initialisations followed by the body statement. This is calculated with an environment updated by the declarations of the local variables. This meaning is a state transition.

$$\mathcal{M}_B : Body \pfun STACK \pfun EnvMTrace \pfun EnvMTrace \times (State \pfun State)$$

$$\begin{aligned}
&\forall Body;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
&\quad \exists \rho t' : EnvMTrace;\ fd, fb : State \pfun State \mid \\
&\qquad \rho t' = \mathcal{M}_{SD}\ {}_*\ \Delta SD\ B\ \rho t \\
&\qquad \land\ fd = \mathcal{M}_{SDI^*}\ \Delta SD\ B\ \rho t' \\
&\qquad \land\ fb = \mathcal{M}_S\ \gamma\ B\ \rho t' \bullet \\
&\quad \mathcal{M}_B\ \theta Body\ B\ \rho t = (\rho t', fd \fatsemi fb)
\end{aligned}$$

The meaning of a procedure declaration is the environment obtained from the dynamic semantics of the body applied to the meaning of the parameters. The procedure name is added to the environment, and is mapped to the state transformation given by the body statement.

$$\mathcal{M}_{PD} : ProcDecl \pfun STACK \pfun EnvMTrace \pfun EnvMTrace$$

$$\begin{aligned}
&\forall ProcDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
&\quad \exists B' : STACK;\ \rho t', \rho t'' : EnvMTrace;\ fs : State \pfun State \mid \\
&\qquad B' = B \frown \langle \xi \rangle \\
&\qquad \land\ \rho t' = \mathcal{M}_{PMD}\ {}_*\ \Pi\ B'(\rho t \oplus \{\xi \mapsto \varnothing\}) \\
&\qquad \land\ (\rho t'', fs) = \mathcal{M}_B\ \theta Body\ B'\ \rho t' \bullet \\
&\quad \mathcal{M}_{PD}\ \theta ProcDecl\ B\ \rho t = update(\rho t'', B, \{\xi \mapsto pval\ fs\})
\end{aligned}$$

9.7 Function declaration

9.7.1 Function declaration – Symbol declaration semantics

The declaration semantics of functions is very similar to that for procedures, except that a function declaration also effectively declares a local variable with the same name as the function. This is used to store the return value. So none of the formal parameters can have the same name as the function.

$$
\begin{array}{|l}
\mathcal{D}_{FD} : FunDecl \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace \\
\hline
\forall\, FunDecl;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, B' : STACK;\ \rho\delta t', \rho\delta t'', \rho\delta t''', \rho\delta t'''' : EnvDTrace \mid \\
\qquad B' = B \frown \langle \xi \rangle \\
\qquad \land\ \rho\delta t' = \textbf{if}\ \tau \in \operatorname{ran} typeName \\
\qquad\quad \textbf{then}\ checkSymbol\ \xi\ B\ \rho\delta t\ \textbf{else}\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = addSymbol\ \xi\ \xi(\rho\delta t' \oplus \{\xi \mapsto \varnothing\}) \\
\qquad \land\ \rho\delta t''' = \mathcal{D}_{PMD}\ {}_{*}\ \Pi\ B'\ \rho\delta t'' \\
\qquad \land\ \rho\delta t'''' = \mathcal{D}_{B}\ \theta Body\ B'\ \rho\delta t''' \bullet \\
\quad \mathcal{D}_{FD}\ \theta FunDecl\ B\ \rho\delta t = addSymbol\ \xi(last\ B)\rho\delta t'''
\end{array}
$$

Note that the function identifier is added to the trace environment of both the outer and the inner block. In the outer block it represent the function, while in the inner block it represents a local variable for the return value.

9.7.2 Function declaration – Type checking semantics

The type checking semantics for function declarations is similar to that for procedures.

$$
\begin{array}{|l}
\mathcal{T}_{FD} : FunDecl \pfun STACK \pfun DeclType \pfun DeclType \\
\hline
\forall\, FunDecl;\ B : STACK;\ DeclType \bullet \\
\quad \exists\, FunType';\ \rho\tau t', \rho\tau t'', \rho\tau t'''' : EnvTTrace;\ DeclType''';\\
\qquad\qquad \Xi : \mathrm{seq}\, ID;\ \rho\tau : EnvT \mid \\
\qquad\qquad B' = B \frown \langle \xi \rangle \\
\qquad\qquad \land\ \theta SubrangeType' = (\mathcal{T}_T\ \tau\ B\ \rho\tau t).2 \\
\qquad\qquad \land\ \rho\tau t' = \mathcal{T}_{PMD}\ {}_*\ \Pi\ B'\ (\rho\tau t \oplus \{\xi \mapsto \varnothing\}) \\
\qquad\qquad \land\ \rho\tau t'' = update(\rho\tau t', B, \\
\qquad\qquad\qquad\qquad \{\xi \mapsto variable \langle\!|\ A == \varnothing, SR == \langle\ \rangle, \\
\qquad\qquad\qquad\qquad\quad \tau a == unsigned, lb == lb', ub == ub', \\
\qquad\qquad\qquad\qquad\quad \tau == \tau'\ |\!\rangle\}) \\
\qquad\qquad \land\ \theta DeclType''' = \\
\qquad\qquad\quad \mathcal{T}_B\ \theta Body\ B'\ \langle\!|\ \rho\tau t == \rho\tau t'', tc == tc\ |\!\rangle \\
\qquad\qquad \land\ \Xi = extractParamIds\ \Pi \\
\qquad\qquad \land\ I' = \rho\tau t'''\ \xi \circ \Xi \\
\qquad\qquad \land\ \rho\tau = \{\xi \mapsto function\ \theta FunType'\} \\
\qquad\qquad \land\ \rho\tau t'''' = update(\rho\tau t''', B, \rho\tau) \bullet \\
\qquad\quad \mathcal{T}_{FD}\ \theta FunDecl\ B\ \theta DeclType = \langle\!|\ \rho\tau t == \rho\tau t'''', tc == tc'''\ |\!\rangle
\end{array}
$$

A local variable with the same name and type as the function is added to the type trace environment for the inner block. This variable is for the return value of the function.

9.7.3 Function declaration – Use semantics

The use semantics of a function declaration is defined a similar way to that of a procedure declaration, with the addition of a check on the return type.

$$
\begin{array}{|l}
\mathcal{U}_{FD} : FunDecl \pfun STACK \pfun EnvUTrace \pfun EnvUTrace \\
\hline
\forall\, FunDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet \\
\quad \exists\, B' : STACK;\ \rho\upsilon t', \rho\upsilon t'', \rho\upsilon t''' : EnvUTrace \mid \\
\qquad B' = B \frown \langle \xi \rangle \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_{PMD}\ {}_{*}\ \Pi\ B'(\rho\upsilon t \oplus \{\xi \mapsto \varnothing\}) \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_B\ \theta Body\ B'\ \rho\upsilon t' \\
\qquad \land\ \rho\upsilon t''' = \mathcal{U}_T\ \tau\ B'\ \rho\upsilon t'' \bullet \\
\quad \mathcal{U}_{FD}\ \theta FunDecl\ B\ \rho\upsilon t = update(\rho\upsilon t''', B, \{\xi \mapsto uncalled\})
\end{array}
$$

9.7.4 Function declaration – Dynamic semantics

The dynamic semantics of function declarations are similar to those of procedure declarations. The return value location l is also added to the environment.

$$
\begin{array}{|l}
\mathcal{M}_{FD} : FunDecl \pfun STACK \pfun EnvMTrace \pfun EnvMTrace \\
\hline
\forall\, FunDecl;\ B : STACK;\ \rho t : EnvMTrace \bullet \\
\quad \exists\, B' : STACK;\ \rho t', \rho t'', \rho t''' : EnvMTrace;\ fc : State \pfun State; \\
\qquad\qquad l' : LOCN;\ \sigma' : State;\ ExprValue' \mid \\
\qquad B' = B \frown \langle \xi \rangle \\
\qquad \land\ \rho t' = \mathcal{M}_{PMD}\ {}_{*}\ \Pi\ B'(\rho t \oplus \{\xi \mapsto \varnothing\}) \\
\qquad \land\ (\ \exists\, l : LOCN \mid l \notin usedLocations\ \rho t' \bullet l = l'\ ) \\
\qquad \land\ \rho t'' = update(\rho t', B, \{\xi \mapsto loc\{\langle\ \rangle \mapsto l'\}\}) \\
\qquad \land\ (\rho t''', fc) = \mathcal{M}_B\ \theta Body\ B'\ \rho t'' \\
\qquad \land\ \theta State' = fc\ \sigma \\
\qquad \land\ v' = \Sigma'\ l' \bullet \\
\quad \mathcal{M}_{FD}\ \theta FunDecl\ B\ \rho t = \\
\qquad update(\rho t''', B, \{\xi \mapsto fval\{\sigma \mapsto \theta ExprValue'\}\})
\end{array}
$$

9.8 Procedure or function declarations

The meaning of a procedure or function declaration is the meaning of the relevant free type branch.

$$\mathcal{D}_{PFD} : PROC\_FUN\_DECL \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

$$\forall \delta : ProcDecl \bullet \mathcal{D}_{PFD}(procDecl\ \delta) = \mathcal{D}_{PD}\ \delta$$
$$\forall \delta : FunDecl \bullet \mathcal{D}_{PFD}(funDecl\ \delta) = \mathcal{D}_{FD}\ \delta$$

$$\mathcal{T}_{PFD} : PROC\_FUN\_DECL \nrightarrow STACK \nrightarrow DeclType \nrightarrow DeclType$$

$$\forall \delta : ProcDecl \bullet \mathcal{T}_{PFD}(procDecl\ \delta) = \mathcal{T}_{PD}\ \delta$$
$$\forall \delta : FunDecl \bullet \mathcal{T}_{PFD}(funDecl\ \delta) = \mathcal{T}_{FD}\ \delta$$

$$\mathcal{U}_{PFD} : PROC\_FUN\_DECL \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace$$

$$\forall \delta : ProcDecl \bullet \mathcal{U}_{PFD}(procDecl\ \delta) = \mathcal{U}_{PD}\ \delta$$
$$\forall \delta : FunDecl \bullet \mathcal{U}_{PFD}(funDecl\ \delta) = \mathcal{U}_{FD}\ \delta$$

$$\mathcal{M}_{PFD} : PROC\_FUN\_DECL \nrightarrow STACK \nrightarrow EnvMTrace \nrightarrow EnvMTrace$$

$$\forall \delta : ProcDecl \bullet \mathcal{M}_{PFD}(procDecl\ \delta) = \mathcal{M}_{PD}\ \delta$$
$$\forall \delta : FunDecl \bullet \mathcal{M}_{PFD}(funDecl\ \delta) = \mathcal{M}_{FD}\ \delta$$

10 Import Declarations

10.1 Introduction

In this section, the semantics for import declarations are defined.

The procedure and function declarations are dealt with separately, although they have very similar semantics, because the function import declares a local variable with the same name as the function import declaration. None of the formal parameters can have the same name as the import.

10.2 Meaning function signatures

10.2.1 Symbol declaration semantics

$$\mathcal{D}_{IM} : IMPORT\_DECL \nrightarrow STACK \nrightarrow EnvDTrace \nrightarrow EnvDTrace$$

10.2.2 Type checking semantics

$$\mathcal{T}_{IM} : IMPORT\_DECL \nrightarrow STACK \nrightarrow DeclType \nrightarrow DeclType$$

10.2.3 Use semantics

$$\mathcal{U}_{IM} : IMPORT\_DECL \nrightarrow STACK \nrightarrow EnvUTrace \nrightarrow EnvUTrace$$

10.2.4 Dynamic semantics

There is no dynamic semantics for imports, as the modules themselves have no defined meaning: there is only a meaning for the whole $MPROG$.

10.3 Import Constant

An import constant declaration is semantically similar to an ordinary constant declaration.

10.3.1 Import Constant – Symbol decl semantics

The name of the constant is added as an imported symbol.

$$\forall \xi : ID;\ \tau : TYPE;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet$$
$$\mathcal{D}_{IM}(constHdr(\xi, \tau))B\ \rho\delta t =$$
$$addImportSymbol\ \xi(last\ B)\rho\delta t$$

10.3.2 Import Constant – Type checking semantics

The type of the constant must be one of *unsigned*, *byte* or *boolean*.

$$\forall \xi : ID;\ \tau : TYPE;\ B : STACK;\ DeclType \bullet$$
$$\exists\, DeclType';\ ValueType' \mid$$
$$\rho\tau t' = update(\rho\tau t, \langle last\ B\rangle, \{\xi \mapsto const\ \theta ValueType'\})$$
$$\land\ (tc', \tau') = \textbf{if}\ \tau \in \{unsigned, pbyte, boolean\}$$
$$\textbf{then}\ (tc \oplus \{\xi \mapsto checkOK\}, \tau)$$
$$\textbf{else}\ (tc \oplus \{\xi \mapsto checkTypeWrong\}, typeWrong) \bullet$$
$$\mathcal{T}_{IM}(constHdr(\xi, \tau))B\ \theta DeclType = \theta DeclType'$$

10.3.3 Import Constant – Use semantics

The constant is marked as initially unread.

$$\forall \xi : ID;\ \tau : TYPE;\ B : STACK;\ \rho\upsilon t : EnvUTrace \bullet$$
$$\mathcal{U}_{IM}(constHdr(\xi, \tau))B\ \rho\upsilon t = update(\rho\upsilon t, B, \{\xi \mapsto unread\})$$

10.4 Import Variable

An import variable declaration is semantically similar to an ordinary variable declaration. An imported variable is read only in the module to which it is imported.

10.4.1 Import Variable – Symbol decl semantics

Using the *VAR_DECL* semantics as a basis, we get the semantics for import variable declarations to be

$$
\begin{array}{|l}
\hline
\forall\, VarDecl;\ B : STACK;\ \rho\delta t : EnvDTrace \mid \\
\qquad A = \langle readOnly \rangle \land V = \langle\ \rangle \bullet \\
\quad \exists\, \rho\delta t' : EnvDTrace \mid \\
\qquad\quad \rho\delta t' = \textbf{if}\ \tau\ \in \operatorname{ran} typeName \\
\qquad\qquad\quad \textbf{then}\ checkSymbol(typeName^{\sim}\ \tau)B\ \rho\delta t\ \textbf{else}\ \rho\delta t \bullet \\
\qquad \mathcal{D}_{IM}(varHdr\ \theta VarDecl)B\ \rho\delta t = \\
\qquad\quad addImportSymbol\ \xi(last\ B)\rho\delta t' \\
\hline
\end{array}
$$

10.4.2 Import Variable – Type checking semantics

The type checking semantics of import variable declarations is the same as for variables.

$$
\begin{array}{|l}
\hline
\forall\, VarDecl;\ B : STACK;\ DeclType \mid \\
\qquad A = \langle readOnly \rangle \land V = \langle\ \rangle \bullet \\
\quad \exists\, DeclType' \mid \\
\qquad\quad \rho\tau t' = \mathcal{T}_V\ \theta VarDecl\ B\ \rho\tau t \\
\qquad\quad \land\ tc' = tc \oplus \{\xi \mapsto checkOK\} \bullet \\
\qquad \mathcal{T}_{IM}(varHdr\ \theta VarDecl)B\ \theta DeclType = \theta DeclType' \\
\hline
\end{array}
$$

10.4.3 Import Variable – Use semantics

The use semantics is the same as that for variables.

$$
\begin{array}{|l}
\hline
\forall\, VarDecl;\ B : STACK;\ \rho\upsilon t : EnvUTrace \mid \\
\qquad A = \langle readOnly \rangle \land V = \langle\ \rangle \bullet \\
\quad \exists\, \rho\upsilon t' : EnvUTrace \mid \rho\upsilon t' = \mathcal{U}_V\ \theta VarDecl\ B\ \rho\upsilon t \bullet \\
\qquad \mathcal{U}_{IM}(varHdr\ \theta VarDecl)B\ \rho\upsilon t = \rho\upsilon t' \\
\hline
\end{array}
$$

If a variable is imported, it must be used in the module, otherwise the use after declaration semantics gives a warning.

10.5 Import Procedure

An import procedure declaration is semantically similar to a *ProcDecl* with a *skip* body.

10.5.1 Import Procedure – Symbol decl semantics

Using the *ProcDecl* semantics as a basis, we get the semantics for import declarations to be the same, except that the variable is added as *ImportOk*, rather than *checkOk*.

$$
\begin{array}{|l}
\hline
\forall\, ProcHdr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t' : EnvDTrace \mid \\
\qquad \rho\delta t' = \mathcal{D}_{PMD}\ {}_*\ \Pi(B \frown \langle \xi \rangle)(\rho\delta t \oplus \{\xi \mapsto \varnothing\}) \bullet \\
\quad \mathcal{D}_{IM}(procHdr\ \theta ProcHdr)B\ \rho\delta t = \\
\qquad addImportSymbol\ \xi(last\ B)\rho\delta t' \\
\hline
\end{array}
$$

10.5.2 Import Procedure – Type checking semantics

The type checking semantics of an import procedure declaration is the same as for a procedure with a skip body.

$$
\begin{array}{|l}
\hline
\forall\, ProcHdr \bullet \\
\quad \mathcal{T}_{IM}(procHdr\ \theta ProcHdr) = \\
\qquad \mathcal{T}_{PD} \llparenthesis\ \xi == \xi, \Pi == \Pi, \Delta SD == \langle\ \rangle, \gamma == skip\ \rrparenthesis \\
\hline
\end{array}
$$

10.5.3 Import Procedure – Use semantics

The use semantics of imported procedure are the same as those for procedures.

$$
\begin{array}{|l}
\hline
\forall\, ProcHdr \bullet \\
\quad \mathcal{U}_{IM}(procHdr\ \theta ProcHdr) = \\
\qquad \mathcal{U}_{PD}(\!|\ \xi == \xi, \Pi == \Pi, \Delta SD == \langle\ \rangle, \gamma == skip\ |\!) \\
\hline
\end{array}
$$

For procedures imported into a module, they must be subsequently called, otherwise their use after declaration semantics will fail like an unused procedure.

10.6 Import Function

An import function declaration is semantically similar to a *FunDecl* with a *skip* body.

10.6.1 Import Function – Symbol decl semantics

Using the *FunDecl* semantics as a basis, we get the semantics for import declarations to be the same, except that the variable is added as *ImportOk*, rather than *checkOk*.

$$
\begin{array}{|l}
\hline
\forall\, FunHdr;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta t', \rho\delta t'', \rho\delta t''' : EnvDTrace \mid \\
\qquad \rho\delta t' = \textbf{if}\ \tau \in \operatorname{ran} typeName \\
\qquad\quad \textbf{then}\ checkSymbol(typeName^{\sim}\ \tau)B\ \rho\delta t\ \textbf{else}\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = addSymbol\ \xi\ \xi(\rho\delta t' \oplus \{\xi \mapsto \varnothing\}) \\
\qquad \land\ \rho\delta t''' = \mathcal{D}_{PMD}\ {}_{*}\ \Pi(B \frown \langle\xi\rangle)\rho\delta t'' \bullet \\
\quad \mathcal{D}_{IM}(funcHdr\ \theta FunHdr)B\ \rho\delta t = \\
\qquad addImportSymbol\ \xi(last\ B)\rho\delta t''' \\
\hline
\end{array}
$$

10.6.2 Import Function – Type checking semantics

The type checking semantics of an import function declaration is the same as for a function declaration with a skip body.

$$\forall\, FunHdr \bullet \\ \quad \mathcal{T}_{IM}(funcHdr\ \theta FunHdr) = \\ \quad\quad \mathcal{T}_{FD} \llparenthesis\, \xi == \xi, \Pi == \Pi, \tau == \tau, \Delta SD == \langle\,\rangle, \gamma == skip \,\rrparenthesis$$

10.6.3 Import Function – Use semantics

The use semantics for imported functions are those for function declarations.

$$\forall\, FunHdr \bullet \\ \quad \mathcal{U}_{IM}(funcHdr\ \theta FunHdr) = \\ \quad\quad \mathcal{U}_{FD} \llparenthesis\, \xi == \xi, \Pi == \Pi, \tau == \tau, \Delta SD == \langle\,\rangle, \gamma == skip \,\rrparenthesis$$

For functions imported into a module, they must be subsequently called, otherwise their use after declaration semantics will fail as for unused functions.

11 Export Declarations

In this section, the semantics for export declarations are defined. There is only one export declaration in each of the standard modules. This should be declaring the names of procedures and functions already declared completely in that module. No procedures or functions imported to a module, should be exported from the same module.

11.1 Export Declaration Semantics

The only semantics which need to be checked are the symbol declaration semantics, there we ensure that the export names are not also imported to the current module. If this is satisfactory, then the environment is restricted to those exported identifiers. This has the functionality that any declaration errors are still trapped within the semantics, and if this is correct then the resultant environment can be used to trap declaration errors across modules.

11.1.1 Symbol declaration semantics

The export identifiers are checked in the outermost module block.

$\mathcal{D}_{EXname}$ checks that an exported name has been declared, and has not been imported.

$$
\begin{array}{|l}
\mathcal{D}_{EXname} : ID \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace \\
\hline
\forall\, \xi : ID;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta : EnvD;\ \chi : CHECK \mid \\
\qquad \rho\delta = \oplus/\ (\rho\delta t \circ B) \\
\qquad \land\ (\xi \in \operatorname{dom} \rho\delta \Rightarrow \\
\qquad\quad \chi = (\ \textbf{let}\ \chi' == \rho\delta\ \xi \bullet \\
\qquad\qquad\quad \textbf{if}\ \chi' = ImportOK \\
\qquad\qquad\quad \textbf{then}\ ImportExport\ \textbf{else}\ \chi'\ )) \\
\qquad \land\ (\xi \notin \operatorname{dom} \rho\delta \Rightarrow \chi = Undecl) \bullet \\
\quad \mathcal{D}_{EXname}\ \xi\ B\ \rho\delta t = update(\rho\delta t, B, \{\xi \mapsto \chi\})
\end{array}
$$

Each name in the export list is checked to ensure that the symbol has been previously declared in the module.

$$\mathcal{D}_{EX} : EXPORT\_DECL \rightarrow STACK \rightarrow EnvDTrace \rightarrow EnvDTrace$$

$$\begin{aligned}
&\forall\, \Xi : \operatorname{seq} ID;\ B : STACK;\ \rho\delta t : EnvDTrace \bullet \\
&\quad \exists\, \rho\delta t' : EnvDTrace \mid \rho\delta t' = \mathcal{D}_{EXname\ *}\ \Xi\ B\ \rho\delta t \bullet \\
&\qquad \mathcal{D}_{EX}\ \Xi\ B\ \rho\delta t = \rho\delta t'
\end{aligned}$$

11.1.2 Type checking semantics

Each exported name is checked to have been declared as a constant name, a non-*writeOnly* variable name, a procedure name, or a function name.

$$\mathcal{T}_{EXname} : ID \nrightarrow STACK \nrightarrow DeclType \nrightarrow DeclType$$

$$\begin{aligned}
&\forall\, \xi : ID;\ B : STACK;\ DeclType \bullet \\
&\quad \exists\, \chi : CHECK;\ idt : IDTYPE;\ tc' : Env[CHECK] \mid \\
&\qquad idt = \rho\tau t\ (last\ B)\xi \\
&\qquad \land\ \chi = \\
&\qquad\quad \textbf{if } idt \in \operatorname{ran} const \\
&\qquad\qquad \cup \{\ v : \operatorname{ran} variable \mid writeOnly \notin attributeOf\ v\ \} \\
&\qquad\qquad \cup \operatorname{ran} procedure \cup \operatorname{ran} function \\
&\qquad\quad \textbf{then } tc\ \xi\ \bowtie\ checkOK \textbf{ else } checkTypeWrong \\
&\qquad \land\ tc' = tc \oplus \{\xi \mapsto \chi\} \bullet \\
&\quad \mathcal{T}_{EXname}\ \xi\ B\ \theta DeclType = \langle\!\langle\ \rho\tau t == \rho\tau t, tc == tc'\ \rangle\!\rangle
\end{aligned}$$

11.1.3 Use semantics

For the use semantics, no further information can determined, so no semantics is defined.

11.1.4 Dynamic semantics

Similarly to the import declarations, the export declaration has no meaning semantics as none are required. The meaning is defined only for a whole *MPROG*.

12 Modules

12.1 Introduction

In this section, the checking semantics for modules are defined. This includes the main and standard modules. The semantics are given in the usual order: symbol declaration, type checking, use semantics. No dynamic semantics are defined, only an *MPROG* has dynamic semantics defined.

12.2 Module header

12.2.1 Module header – Symbol declaration semantics

The declaration meaning of a module header maps it to an environment. A module header's check status is the result of checking the declarations; note that the declarations may include import declarations, as well as procedures and functions of which the bodies are also checked.

$$
\begin{array}{|l}
\mathcal{D}_{MH} : ModuleHdr \pfun EnvDTrace \\
\hline
\forall\, ModuleHdr \bullet \\
\quad \exists\, \rho\delta t, \rho\delta t', \rho\delta t'' : EnvDTrace \mid \\
\qquad \rho\delta t = \mathcal{D}_{IM} \ast \Delta I \ \langle\xi\rangle\{\xi \mapsto \rho\delta 0\} \\
\qquad \land\ \rho\delta t' = \mathcal{D}_{SD} \ast \Delta S\langle\xi\rangle\rho\delta t \\
\qquad \land\ \rho\delta t'' = \mathcal{D}_{PFD} \ast \Delta PF\langle\xi\rangle\rho\delta t' \bullet \\
\quad \mathcal{D}_{MH}\ \theta ModuleHdr = \rho\delta t''
\end{array}
$$

Firstly the import declaration list is checked in an initial trace environment. (containing the reserved constant identifiers), yielding an updated trace environment. Then the simple declarations are checked in this updated environment, yielding a trace environment complete with all the global constant, variable, and import declarations. The procedure/function declaration list is checked in this; the procedure and function names are checked to ensure that they have not already been defined, and each nested block is also checked in the relevant environment.

12.2.2 Module header – Type checking semantics

The type checking meaning of a module header maps it to a trace type environment (preserved for use in the use, dynamic, and operational semantics) and a map from blocks to check status.

$$
\begin{array}{|l}
\mathcal{T}_{MH} : ModuleHdr \pfun DeclType \\
\hline
\forall\, ModuleHdr \bullet \\
\quad \exists\, DeclType;\ DeclType';\ DeclType'' \mid \\
\qquad \theta DeclType = \\
\qquad\quad \mathcal{T}_{IM\ *}\ \Delta I \langle \xi \rangle \llparenthesis\ \rho\tau t == \{\xi \mapsto \rho\tau 0\}, tc == \varnothing\ \rrparenthesis \\
\qquad \land\ \theta DeclType' = \mathcal{T}_{SD\ *}\ \Delta S \langle \xi \rangle \theta DeclType \\
\qquad \land\ \theta DeclType'' = \mathcal{T}_{PFD\ *}\ \Delta PF \langle \xi \rangle \theta DeclType' \bullet \\
\quad\ \mathcal{T}_{MH}\ \theta ModuleHdr = \theta DeclType''
\end{array}
$$

12.2.3 Module header – Use semantics

We now define the use semantics of the module header. The environment status is the result of successively checking the various declarations starting from the initially empty use environment.

$$
\begin{array}{|l}
\mathcal{U}_{MH} : ModuleHdr \pfun EnvUTrace \\
\hline
\forall\, ModuleHdr \bullet \\
\quad \exists\, \rho\upsilon t, \rho\upsilon t', \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t = \mathcal{U}_{IM\ *}\ \Delta I \langle \xi \rangle \varnothing \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_{SD\ *}\ \Delta S \langle \xi \rangle \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_{PFD\ *}\ \Delta PF \langle \xi \rangle \rho\upsilon t' \bullet \\
\quad\ \mathcal{U}_{MH}\ \theta ModuleHdr = \rho\upsilon t''
\end{array}
$$

12.2.4 Module header – Dynamic semantics

There are no dynamic semantics for a module header.

12.3 Standard module

12.3.1 Standard module – Symbol declaration semantics

The declaration meaning of a module maps it to an environment. A module's check status is the result of checking the body statement in the environment of the declarations, to ensure that no exports are also imports to the module.

$$
\begin{array}{|l}
\mathcal{D}_M : Module \pfun EnvDTrace \\
\hline
\forall\, Module \bullet \\
\quad \exists\, \rho\delta t, \rho\delta t' : EnvDTrace \mid \\
\qquad \rho\delta t = \mathcal{D}_{MH}\ \theta ModuleHdr \\
\qquad \land\ \rho\delta t' = \mathcal{D}_{EX}\ \ e\langle\xi\rangle \rho\delta t \bullet \\
\quad \mathcal{D}_M\ \theta Module = \rho\delta t'
\end{array}
$$

The condition for the rest of the semantics to be defined is that every identifier in the final trace environment maps to *checkOK* or to *ImportOK*.

$$
\begin{array}{|l}
\textit{ModuleDeclOkay} \\
Module \\
\hline
(\mathrm{ran} \circ \bigcup \circ \mathrm{ran})(\mathcal{D}_M\ \theta Module) \subseteq \{checkOK, ImportOK\}
\end{array}
$$

Some of this symbol declaration information is needed to perform cross-module checks (either when flattening Pasp modules to a program, or when linking AspAL modules). $\mathcal{D}_{ML}$ returns an environment comprising the imported names (mapping to *ImportOK*), and the exported and module names (mapping to *checkOK*).

$$
\begin{array}{|l}
\mathcal{D}_{ML} : Module \pfun EnvD \\
\hline
\forall\, Module \mid ModuleDeclOkay \bullet \\
\quad \exists\, \rho\delta t : EnvDTrace;\ \Xi : \mathbb{P}\, ID;\ \rho\delta : EnvD \mid \\
\qquad \rho\delta t = \mathcal{D}_M\ \theta Module \\
\qquad \land\ \Xi = \mathrm{dom}(\bigcup(\mathrm{ran}\, \rho\delta t) \rhd \{ImportOK\}) \\
\qquad \land\ \rho\delta = (\Xi \times \{ImportOK\}) \\
\qquad\qquad \cup\, (\mathrm{ran}\, e\ \cup \{\xi\} \times \{checkOK\}) \bullet \\
\quad \mathcal{D}_{ML}\ \theta Module = \rho\delta
\end{array}
$$

12.3.2 Standard module – Type checking semantics

The type checking meaning of a standard module maps it to a trace type environment (preserved for use in the use, dynamic, and operational semantics) and a map from blocks to check status. The statement is checked in the environment of the declarations.

$$
\begin{array}{|l}
\mathcal{T}_M : Module \pfun DeclType \\
\hline
\forall\, Module \mid ModuleDeclOkay \bullet \\
\quad \exists\, DeclType;\ DeclType' \mid \\
\qquad \theta DeclType = \mathcal{T}_{MH}\ \theta ModuleHdr \\
\qquad \land\ \theta DeclType' = \mathcal{T}_{EXname\ *}\ e\langle\xi\rangle\theta DeclType \bullet \\
\quad \mathcal{T}_M\ \theta Module = \theta DeclType'
\end{array}
$$

So the type trace environment from type checking a module looks like (where ξ represents the module name, pf represents a procedure or function name, v represents a simple declaration name, and ap represents a parameter name):

$$
\begin{array}{l}
\{\xi \mapsto \{pf_1 \mapsto procedure(\langle \xi, pf_1\rangle, \\
\qquad\qquad \langle formalParam(ap_1, \ldots), \ldots, formalParam(ap_i, \ldots)\rangle), \\
\qquad \ldots, \\
\qquad pf_n \mapsto function(\langle \xi, pf_n\rangle, \\
\qquad\qquad \langle formalParam(ap_1, \ldots), \ldots, formalParam(ap_i, \ldots)\rangle, \ldots), \\
\qquad v_1 \mapsto variable(\ldots), \ldots v_m \mapsto const(\ldots)\}, \\
\quad pf_1 \mapsto \{ap_1 \mapsto formalParam(ap_1, \ldots), \ldots, ap_i \mapsto formalParam(ap_i, \ldots)\}, \\
\quad \ldots, \\
\quad pf_n \mapsto \{ap_1 \mapsto formalParam(ap_1, \ldots), \ldots, ap_j \mapsto formalParam(ap_j, \ldots)\}\}
\end{array}
$$

The corresponding *BlockTypeCheck* value looks like

$$\{pf_1 \mapsto checkOK, \ldots, pf_n \mapsto checkTypeWrong\}$$

The additional condition for the later semantics to be defined is that the type checking semantics yields *checkOK* for every block in the module:

ModuleTypeOkay

$$
\begin{array}{l}
ModuleDeclOkay \\
\hline
\exists\, tc : Env[CHECK] \mid \\
\qquad \mathcal{T}_M\ \theta Module = \langle\!|\ \rho\tau t == \rho\tau t0, tc == tc\ |\!\rangle \bullet \\
\quad \operatorname{ran} tc = \{checkOK\}
\end{array}
$$

Later semantics make use of the global type environment $\rho\tau t0$.

Some of this type information is needed to perform cross-module checks (either when flattening Pasp modules to a program, or when linking AspAL modules). $\mathcal{T}_{ML}$ returns a type environment comprising the types claimed for the imported names and of the types given to the exported names.

$$
\begin{array}{l}
\mathcal{T}_{ML} : Module \pfun EnvT \\
\hline
\forall\, Module \mid ModuleTypeOkay \bullet \\
\quad \exists\, \rho\delta t : EnvDTrace;\ \Xi : \mathbb{P}\, ID;\ \rho\tau : EnvT \mid \\
\qquad \rho\delta t = \mathcal{D}_M\ \theta Module \\
\qquad \land\ \Xi = \operatorname{dom}(\bigcup(\operatorname{ran} \rho\delta t) \rhd \{ImportOK\}) \\
\qquad \land\ \rho\tau = (\Xi\ \cup \operatorname{ran} e) \lhd \rho\tau t0\ \xi \bullet \\
\quad \mathcal{T}_{ML}\ \theta Module = \rho\tau
\end{array}
$$

So the type environment used for cross module checks looks like (where *pf* represents an imported or exported procedure or function name, *v* represents an exported variable name, and *ap* represents a parameter name):

$$
\begin{array}{l}
\{pf_2 \mapsto procedure(\langle \xi, pf_1 \rangle, \\
\qquad \langle formalParam(ap_1, \ldots), \ldots, formalParam(ap_i, \ldots) \rangle), \\
\ldots, \\
pf_n \mapsto function(\langle \xi, pf_n \rangle, \\
\qquad \langle formalParam(ap_1, \ldots), \ldots, formalParam(ap_i, \ldots) \rangle, \ldots), \\
v_3 \mapsto variable(\ldots)\}
\end{array}
$$

12.3.3 Standard module – Use semantics

We now define the use semantics of the module. The environment status is the result of checking the various header declarations; no further checking of the exports is done.

$$
\begin{array}{|l}
\mathcal{U}_M : Module \pfun EnvUTrace \\
\hline
\forall\, Module \mid ModuleTypeOkay \bullet \\
\quad \exists\, \rho\upsilon t : EnvUTrace \mid \rho\upsilon t = \mathcal{U}_{MH}\ \theta ModuleHdr \bullet \\
\quad\quad \mathcal{U}_M\ \theta Module = \rho\upsilon t
\end{array}
$$

All names (except reference parameters and exported variables) should be properly used. An implementation should deliver a *warning* if this is not the case.

ModuleUseOkay

$$
\begin{array}{l}
ModuleTypeOkay \\
\hline
\exists\, refs : ID \rel ID;\ \rho\upsilon t, \rho\upsilon t' : EnvUTrace \mid \\
\quad refs = \{\ b : \mathrm{dom}\, \rho\tau t0;\ \xi : ID \mid \\
\quad\quad\quad \xi \in \mathrm{dom}(\rho\tau t0\ b) \\
\quad\quad\quad \land\ \rho\tau t0\ b\ \xi \in \mathrm{ran}\, formalParam \\
\quad\quad\quad \land\ (formalParam^{\sim}(\rho\tau t0\ b\ \xi)).c = ref\ \} \\
\quad\quad \cup\, (\{\xi\} \times \mathrm{ran}\, e) \\
\quad \land\ \rho\upsilon t = \mathcal{U}_M\ \theta Module \\
\quad \land\ \rho\upsilon t' = (\mathrm{dom}\, refs \domres \rho\upsilon t) \\
\quad\quad \cup \{\ b : \mathrm{dom}\, \rho\upsilon t \cap \mathrm{dom}\, refs \bullet \\
\quad\quad\quad b \mapsto (refs(\!|\ \{b\}\ |\!) \ransub \rho\upsilon t\ b)\} \bullet \\
\quad \rho\upsilon t' = \varnothing
\end{array}
$$

12.3.4 Standard module – Dynamic semantics

There are no dynamic semantics for an individual module, only for a complete *MPROG*. This is because it is impossible to define the meaning of a module unless all the declarations are complete, and for imported declarations insufficient information is known at compile time. Therefore, the meaning of a module forms part of the meaning of a whole *MPROG* with all the other modules.

12.4 Main module

12.4.1 Main module – Symbol declaration semantics

$$
\begin{array}{|l}
\mathcal{D}_{MA} : MainModule \pfun EnvDTrace \\
\hline
\forall\, MainModule \bullet \\
\quad \exists\, \rho\delta t, \rho\delta t' : EnvDTrace \mid \\
\qquad\qquad \rho\delta t = \mathcal{D}_{MH}\ \theta ModuleHdr \\
\qquad\qquad \land\ \rho\delta t' = \mathcal{D}_S\ \gamma\ \langle \xi \rangle\ \rho\delta t \bullet \\
\qquad\quad \mathcal{D}_{MA}\ \theta MainModule = \rho\delta t'
\end{array}
$$

The body statement is checked in the environment of the checked header.

The condition for the rest of the semantics to be defined is that every identifier in the final trace environment maps to *checkOK* or to *ImportOK*.

$$
\begin{array}{|l}
\hline
MainModuleDeclOkay \\
MainModule \\
\hline
(\mathrm{ran} \circ \bigcup \circ \mathrm{ran})(\mathcal{D}_{MA}\ \theta MainModule) \subseteq \{checkOK, ImportOK\} \\
\hline
\end{array}
$$

Some of this symbol declaration information is needed to perform cross-module checks (either when flattening Pasp modules to a program, or when linking AspAL modules). $\mathcal{D}_{MAL}$ returns an environment comprising the imported names (mapping to *ImportOK*), and the main module name (mapping to *checkOK*).

$$
\begin{array}{|l}
\mathcal{D}_{MAL} : MainModule \pfun EnvD \\
\hline
\forall\, MainModule \mid MainModuleDeclOkay \bullet \\
\quad \exists\, \rho\delta t : EnvDTrace;\ \Xi : \mathbb{P}\, ID;\ \rho\delta : EnvD \mid \\
\qquad\qquad \rho\delta t = \mathcal{D}_{MA}\ \theta MainModule \\
\qquad\qquad \land\ \Xi = \mathrm{dom}(\bigcup(\mathrm{ran}\, \rho\delta t) \rhd \{ImportOK\}) \\
\qquad\qquad \land\ \rho\delta = (\Xi \times \{ImportOK\}) \cup (\{\xi\} \times \{checkOK\}) \bullet \\
\qquad\quad \mathcal{D}_{MAL}\ \theta MainModule = \rho\delta
\end{array}
$$

12.4.2 Main module – Type checking semantics

The type checking meaning of a main module maps it to a a trace type environment (preserved for use in the use, dynamic, and operational semantics) and a map from blocks to check status. The statement is checked in the environment of the declarations.

$$
\begin{array}{|l}
\mathcal{T}_{MA} : MainModule \pfun DeclType \\
\hline
\forall\, MainModule \mid MainModuleDeclOkay \bullet \\
\quad \exists\, DeclType;\ StmtType';\ tc' : Env[CHECK] \mid \\
\qquad \theta DeclType = \mathcal{T}_{MH}\ \theta ModuleHdr \\
\qquad \land\ \theta StmtType' = \mathcal{T}_S\ \gamma\langle\xi\rangle\rho\tau t \\
\qquad \land\ tc' = tc \oplus \{\xi\ \mapsto c'\} \bullet \\
\quad \mathcal{T}_{MA}\ \theta MainModule = \langle\!|\ \rho\tau t == \rho\tau t', tc == tc'\ |\!\rangle
\end{array}
$$

The condition for the rest of the semantics to be defined is that the type checking semantics yields *checkOK* for every block in the module:

MainModuleTypeOkay

$$
\begin{array}{|l}
MainModuleDeclOkay \\
\hline
\exists\, tc : Env[CHECK] \mid \\
\quad \mathcal{T}_{MA}\ \theta MainModule = \langle\!|\ \rho\tau t == \rho\tau t0, tc == tc\ |\!\rangle \bullet \\
\quad \operatorname{ran} tc = \{checkOK\}
\end{array}
$$

Later semantics make use of the global type environment $\rho\tau t0$.

Some of this type information is needed to perform cross-module checks (either when flattening Pasp modules to a program, or when linking AspAL modules). $\mathcal{T}_{MAL}$ returns a type environment comprising the types claimed for the imported names.

$$
\begin{array}{|l}
\mathcal{T}_{MAL} : MainModule \pfun EnvT \\
\hline
\forall\, MainModule \mid MainModuleTypeOkay \bullet \\
\quad \exists\, \rho\delta t : EnvDTrace;\ \Xi : \mathbb{P}\, ID;\ \rho\tau : EnvT \mid \\
\qquad \rho\delta t = \mathcal{D}_{MA}\ \theta MainModule \\
\qquad \land\ \Xi = \operatorname{dom}(\bigcup(\operatorname{ran}\rho\delta t) \rhd \{ImportOK\}) \\
\qquad \land\ \rho\tau = \Xi\ \lhd \rho\tau t0\ \xi \bullet \\
\quad \mathcal{T}_{MAL}\ \theta MainModule = \rho\tau
\end{array}
$$

12.4.3 Main module – Use semantics

We now define the use semantics of the module. The environment status is the result of checking the body statement in the environment of the checked header declarations.

$$
\begin{array}{|l}
\mathcal{U}_{MA} : MainModule \pfun EnvUTrace \\
\hline
\forall\, MainModule \mid MainModuleTypeOkay \bullet \\
\quad \exists\, \rho\upsilon t, \rho\upsilon t' : EnvUTrace \mid \\
\qquad \rho\upsilon t = \mathcal{U}_{MH}\ \theta ModuleHdr \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_S\ \gamma\ \langle \xi \rangle\ \rho\upsilon t \bullet \\
\quad \mathcal{U}_{MA}\ \theta MainModule = \rho\upsilon t'
\end{array}
$$

All names (except reference parameters) should be properly used. An implementation should deliver a *warning* if this is not the case.

MainModuleUseOkay

$$
\begin{array}{|l}
MainModuleTypeOkay \\
\hline
\exists\, refs : ID \rightarrow ID;\ \rho\upsilon t, \rho\upsilon t' : EnvUTrace \mid \\
\quad refs = \{\ b : \mathrm{dom}\, \rho\tau t0;\ \xi : ID \mid \\
\qquad \xi \in \mathrm{dom}(\rho\tau t0\ b) \\
\qquad \land\ \rho\tau t0\ b\ \xi \in \mathrm{ran}\, formalParam \\
\qquad \land\ (formalParam^{\sim}(\rho\tau t0\ b\ \xi)).c = ref\ \} \\
\quad \land\ \rho\upsilon t = \mathcal{U}_{MA}\ \theta MainModule \\
\quad \land\ \rho\upsilon t' = (\mathrm{dom}\, refs \ndres \rho\upsilon t) \\
\qquad \cup \{\ b : \mathrm{dom}\, \rho\upsilon t \cap \mathrm{dom}\, refs \bullet \\
\qquad\quad b \mapsto (refs(\!|\ \{b\}\ |\!) \rhd \rho\upsilon t\ b)\} \bullet \\
\quad \rho\upsilon t' = \varnothing
\end{array}
$$

12.4.4 Main module – Dynamic semantics

There are no dynamic semantics for an individual module, only for a complete *MPROG*. This is because it is impossible to define the meaning of a module unless all the declarations are complete, and for imported declarations insufficient information is known at compile time. Therefore, the meaning of

a module forms part of the meaning of a whole *MPROG* with all the other modules.

13 Modularised Program

13.1 Introduction

The Pasp interpreter takes a modularised program (collection of Pasp modules), and flattens it into an unmodularised program, which it can then execute.

We define symbol declaration, type declaration, and use semantics for the modularised program. The dynamic semantics define the meaning semantics of a modularised program.

The Pasp meaning of the modularised program (dynamic semantics) can be shown to correspond to the Asp meaning of the same program, with each module compiled, and then all of the modules linked together. (The linker takes the compiled Pasp modules (now in AspAL), and links these together into one AspAL program. The hexer then takes this AspAL program, and makes an Asp program.)

13.2 Modularised program – Symbol declaration

All the symbols within each module are checked by the individual module semantics. $\mathcal{D}_{ML}$ provides a module's symbol declaration environment of those *ID*s visible outside the module (which includes exports and module names). For a modularised program, we want to ensure that

- module names are not multiply declared
- each import to a module has been declared as an export in a previous module

Therefore the environment returned by $\mathcal{D}_{ML}$ in addition includes all imported *ID*s, so that these can be checked for previous export declarations.

It is known by pre-checks that the declarations within the modules are consistent. The checking takes the form of two parts depending on whether the value is *checkOK*, or *ImportOK* for that identifier.

If the check state is *checkOK*, then it is either an export from the module, or the name of the module. This identifier is checked to ensure that it is not already declared within any previous module.

If the check state is *ImportOK*, then it is an import to the module. This identifier is then checked to ensure that it is properly declared (by being exported in a previous module).

The function *checkLink* performs these checks. Each identifier in the environment is checked, and a corresponding entry is created in the updated *EnvDL* environment.

$$
\begin{array}{|l}
checkLink : ID \times EnvD \pfun EnvDTrace \pfun EnvDTrace \\
\hline
\forall\, m : ID;\ \rho\delta : EnvD;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, \rho\delta' : EnvD \mid \\
\qquad \rho\delta' = \{\ \xi : \dom \rho\delta;\ \chi : CHECK \mid \\
\qquad\quad \exists\, \rho\delta'' : ID \rel CHECK;\ \Xi : \power ID \mid \\
\qquad\qquad \rho\delta'' = \bigcup(\ran \rho\delta t) \land \Xi = \dom \rho\delta'' \bullet \\
\qquad\qquad \chi = \textbf{if}\ \rho\delta\ \xi = checkOK \\
\qquad\qquad\quad \textbf{then if}\ \xi \in \Xi\ \textbf{then}\ MultiDecl\ \textbf{else}\ checkOK \\
\qquad\qquad\quad \textbf{else if}\ \xi \in \Xi \land \{checkOK\} = \rho\delta''(\!|\ \{\xi\}\ |\!) \\
\qquad\qquad\quad\quad \textbf{then}\ ImportOK\ \textbf{else}\ ImportUndecl\ \} \bullet \\
\qquad checkLink(m, \rho\delta)\rho\delta t = update(\rho\delta t, \langle m \rangle, \rho\delta')
\end{array}
$$

The standard module semantics checks the declaration environment produced from the module, and updates the modularised program's semantics correspondingly. This maintains any incorrect declarations, so that the correct state can be determined in the top level modularised program semantics. If the module's declaration semantics are correctly defined, it applies *checkLink* to the *EnvDL* environment produced so far.

$$
\begin{array}{|l}
\mathcal{D}_{MD} : Module \pfun EnvDTrace \pfun EnvDTrace \\
\hline
\forall\, Module;\ \rho\delta t : EnvDTrace \mid ModuleDeclOkay \bullet \\
\quad \exists\, \rho\delta : EnvD \mid \rho\delta = \mathcal{D}_{ML}\ \theta Module \bullet \\
\qquad \mathcal{D}_{MD}\ \theta Module\ \rho\delta t = checkLink(\xi, \rho\delta)\rho\delta t
\end{array}
$$

The main module semantics is checked similarly.

$$\mathcal{D}_{MAD} : MainModule \pfun EnvDTrace \pfun EnvDTrace$$

$$\begin{aligned}
&\forall\, MainModule;\ \rho\delta t : EnvDTrace \mid MainModuleDeclOkay \bullet \\
&\quad \exists\, \rho\delta : EnvD \mid \rho\delta = \mathcal{D}_{MAL}\ \theta MainModule \bullet \\
&\qquad \mathcal{D}_{MAD}\ \theta MainModule\ \rho\delta t = checkLink(\xi, \rho\delta)\rho\delta t
\end{aligned}$$

The $\mathcal{D}_{MP}$ semantic checks the complete modularised program.

$$\mathcal{D}_{MP} : MPROG \pfun EnvDTrace$$

$$\begin{aligned}
&\forall\, M : \operatorname{seq} Module;\ m : MainModule \bullet \\
&\quad \exists\, \rho\delta t, \rho\delta t' : EnvDTrace \mid \\
&\qquad \rho\delta t = \mathcal{D}_{MD} * M\ \varnothing \land \rho\delta t' = \mathcal{D}_{MAD}\ m\ \rho\delta t \bullet \\
&\quad \mathcal{D}_{MP}(M, m) = \rho\delta t'
\end{aligned}$$

The modularised program environment *EnvDTrace* contains sufficient information to check whether a name is exported. The environment $\rho\delta t0$ is made global for use in later semantics.

$$\rho\delta t0 : EnvDTrace$$

The modularised program's declaration semantics are correct if the following condition is satisfied.

MProgDeclOkay

$$\begin{aligned}
&M : \operatorname{seq} Module \\
&m : MainModule \\
&\hline
&\mathcal{D}_{MP}\ (M, m) = \rho\delta t0 \\
&(\operatorname{ran} \circ \bigcup \circ \operatorname{ran})\rho\delta t0\ \subseteq \{checkOK, ImportOK\}
\end{aligned}$$

This check encapsulates all the modules' and the program's declaration checking. If it fails then the particular module and identifier can be found that is incorrectly declared.

13.3 Modularised program – Type checking

We know, from the symbol declaration semantics for the modularised program, that any import declaration is previously exported. The type checking

for the modularised program ensures that the types of imported declarations match those of the actual declaration in the exporting module. A type check environment is produced that defines the check state for each import to modules within the program.

The *verify* function is used to check imported against actual declarations. Each of these should be compatible in type definitions, and if they are not then a failure check status is returned.

$$
\begin{array}{|l}
verify : IDTYPE \times IDTYPE \pfun CHECK \\
\hline
\forall\, idt, idt' : IDTYPE \bullet \\
\quad verify\ (idt, idt') \\
\qquad = \mathbf{if} (\exists\, ValueType;\ ValueType' \mid \\
\qquad\qquad idt = const\ \theta ValueType \\
\qquad\qquad \land\ idt' = const\ \theta ValueType' \bullet \\
\qquad\qquad \tau = \tau') \\
\qquad\qquad \lor\ (\exists\, VarType;\ VarType' \mid \\
\qquad\qquad\qquad idt = variable\ \theta VarType \\
\qquad\qquad\qquad \land\ idt' = variable\ \theta VarType' \bullet \\
\qquad\qquad\qquad \theta ArrayType = \theta ArrayType' \\
\qquad\qquad\qquad \land\ \theta SubrangeType = \theta SubrangeType' \\
\qquad\qquad\qquad \land\ A' = \{readOnly\}) \\
\qquad\qquad \lor\ (\exists\, ProcType;\ ProcType' \mid \\
\qquad\qquad\qquad idt = procedure\ \theta ProcType \\
\qquad\qquad\qquad \land\ idt' = procedure\ \theta ProcType' \bullet \\
\qquad\qquad\qquad I = I') \\
\qquad\qquad \lor\ (\exists\, FunType;\ FunType' \mid \\
\qquad\qquad\qquad idt = function\ \theta FunType \\
\qquad\qquad\qquad \land\ idt' = function\ \theta FunType' \bullet \\
\qquad\qquad\qquad I = I' \land \theta SubrangeType = \theta SubrangeType') \\
\qquad\qquad \mathbf{then}\ checkOK \\
\qquad\qquad \mathbf{else}\ checkTypeWrong
\end{array}
$$

It is known by the symbol declaration checks, that the declarations within the program are correct (both within, and between modules).

The function *checkLinkType* performs the checks to ensure that the imports to a particular module are correctly typed. This is done by comparing the

type definition from the exporting module with the import declaration. Its parameters are

- m: the name of this module
- $\rho\tau$: this module's type environment
- $\rho\tau t$: the preceding modules' combined type trace environment of their exports
- $\rho\delta t$: the preceding modules' combined check trace environment of their incorrectly typed imports

The environments are updated with the appropriate information.

$$
\begin{array}{l}
checkLinkType : ID \times EnvT \pfun \\
\qquad EnvTTrace \times EnvDTrace \pfun EnvTTrace \times EnvDTrace \\
\hline
\forall\, m : ID;\ \rho\tau : EnvT;\ \rho\tau t : EnvTTrace;\ \rho\delta t : EnvDTrace \bullet \\
\quad \exists\, import : \mathbb{P}\, ID;\ \rho\delta : EnvD;\ \rho\tau' : EnvT \mid \\
\qquad import = \mathrm{dom}(\rho\delta t0\ m \rhd \{ImportOK\}) \\
\qquad \land\ \rho\delta = \{\ \xi : import;\ b : \mathrm{dom}\, \rho\delta t0;\ \chi : CHECK \mid \\
\qquad\qquad \xi \in \mathrm{dom}(\rho\delta t0\ b) \\
\qquad\qquad \land\ \chi = verify(\rho\tau\ \xi, \rho\tau t\ b\ \xi) \neq checkOK \bullet \\
\qquad\quad \xi \mapsto \chi\ \} \\
\qquad \land\ \rho\tau' = import \lhd \rho\tau \bullet \\
\quad checkLinkType(m, \rho\tau)(\rho\tau t, \rho\delta t) = \\
\qquad (\rho\tau t \cup \{m \mapsto \rho\tau'\}, \rho\delta t \cup \{m \mapsto \rho\delta\})
\end{array}
$$

checkLinkType works as follows:

- *import* : the set names imported by this module, determined from the global symbol declaration semantics.
- The block b where that function or procedure is exported is determined from the previous modules' type tract environment.
- If the type of the import and the type of the export are not the same, the fact is noted in the $\rho\delta t$ environment. ($\rho\delta$ is empty if all imports type check correctly.)

- This module's exports are added to the $\rho\tau t$ environment.

The single module semantics check the type environment produced from the module, and update the modularised program's semantics correspondingly. If the module's type semantics are correctly defined, it applies the *checkLinkType* function to the *EnvTTrace* environment produced so far.

$$
\begin{array}{|l}
\mathcal{T}_{MD} : Module \pfun EnvTTrace \times EnvDTrace \pfun \\
\qquad EnvTTrace \times EnvDTrace \\
\hline
\forall\, Module;\ \rho\tau t : EnvTTrace;\ \rho\delta t : EnvDTrace \mid ModuleTypeOkay \bullet \\
\quad \exists\, \rho\tau : EnvT \mid \rho\tau = \mathcal{T}_{ML}\ \theta Module \bullet \\
\qquad \mathcal{T}_{MD}\ \theta Module(\rho\tau t, \rho\delta t) = checkLinkType(\xi, \rho\tau)(\rho\tau t, \rho\delta t)
\end{array}
$$

The main module semantics check the type environment similarly.

$$
\begin{array}{|l}
\mathcal{T}_{MAD} : MainModule \pfun EnvTTrace \times EnvDTrace \pfun \\
\qquad EnvTTrace \times EnvDTrace \\
\hline
\forall\, MainModule;\ \rho\tau t : EnvTTrace;\ \rho\delta t : EnvDTrace \mid \\
\qquad MainModuleTypeOkay \bullet \\
\quad \exists\, \rho\tau : EnvT \mid \rho\tau = \mathcal{T}_{MAL}\ \theta MainModule \bullet \\
\qquad \mathcal{T}_{MAD}\ \theta MainModule(\rho\tau t, \rho\delta t) = \\
\qquad\quad checkLinkType(\xi, \rho\tau)(\rho\tau t, \rho\delta t)
\end{array}
$$

The $\mathcal{T}_{MP}$ semantic checks the complete modularised program.

$$
\begin{array}{|l}
\mathcal{T}_{MP} : MPROG \pfun EnvTTrace \times EnvDTrace \\
\hline
\forall\, M : \operatorname{seq} Module;\ m : MainModule \mid MProgDeclOkay \bullet \\
\quad \exists\, \rho\tau t, \rho\tau t' : EnvTTrace;\ \rho\delta t, \rho\delta t' : EnvDTrace \mid \\
\qquad (\rho\tau t, \rho\delta t) = \mathcal{T}_{MD}\ *\ M\ (\varnothing, \varnothing) \\
\qquad \land\ (\rho\tau t', \rho\delta t') = \mathcal{T}_{MAD}\ m(\rho\tau t, \rho\delta t) \bullet \\
\qquad \mathcal{T}_{MP}(M, m) = (\rho\tau t', \rho\delta t')
\end{array}
$$

The modularised program's type checking semantics are correct if all imported names type check okay.

MProgTypeOkay

$$MProgDeclOkay$$

$$(\mathrm{ran} \circ second)(\mathcal{T}_{MP}\ (M, m)) = \varnothing$$

13.4 Renaming, for dynamic semantics

It cannot be assumed that all constants, variables, and declarations are unique in name across modules. Therefore a renaming policy that generates such unique names is applied to all the modules in turn.

All the global constants, type definitions, variable declarations, procedure and function declarations in each module have to be modified, unless they are exported names. Exported names are not updated, which leaves the name global across all the modules. All the procedure and function calls, and value references, also have to be updated with the appropriate identifier.

It is not necessary to rename all the constants, variables, and type declarations within the procedures and functions. However, for simplicity of definition, this is done also.

13.4.1 Generating new names

All the required entities can be renamed by modifying the identifier to incorporate the module name (also an identifier). This is possible as in the concrete syntax, the identifier type is a string.

$$addMod : ID \pfun ID \pfun ID$$

addMod takes the name of the module and the current identifier, and generates a new, globally unique, identifier (by adding the module name to the identifier).

renID takes the name of the module and the current identifier, and generates the appropriate new identifier. Exported names, found in the global $\rho\delta t0$, are not renamed.

$$
\begin{array}{|l}
renID : ID \pfun ID \pfun ID \\
\hline
\forall\, m, \xi : ID \bullet \\
\quad renID\ m\ \xi = \textbf{if}\ \xi \in \mathrm{dom}(\rho\delta t0\ m)\ \textbf{then}\ \xi\ \textbf{else}\ addMod\ m\ \xi
\end{array}
$$

13.4.2 Renaming types

A name bound is renamed by adding the module name. A number literal bound needs no renaming.

$$
\begin{array}{|l}
renBnd : ID \pfun BOUND \pfun BOUND \\
\hline
\forall\, m, \xi : ID \bullet renBnd\ m(name\ \xi) = name(renID\ m\ \xi) \\
\forall\, m : ID;\ \kappa : VALUE \mid \kappa \notin \mathrm{ran}\, name \bullet renBnd\ m\ \kappa = \kappa
\end{array}
$$

A subrange is renamed by renaming both bounds.

$$
\begin{array}{|l}
renSR : ID \pfun Subrange \pfun Subrange \\
\hline
\forall\, m : ID;\ Subrange \bullet \\
\quad renSR\ m\ \theta Subrange = \\
\qquad \langle\!|\ lb == renBnd\ m\ lb, ub == renBnd\ m\ ub\ |\!\rangle
\end{array}
$$

A sequence of subranges is renamed by renaming each element of the sequence.

$$
\begin{array}{|l}
renSRs : ID \pfun \mathrm{seq}\, Subrange \pfun \mathrm{seq}\, Subrange \\
\hline
\forall\, m : ID \bullet renSRs\ m\langle\,\rangle = \langle\,\rangle \\
\forall\, m : ID;\ sr : Subrange;\ SR : \mathrm{seq}\, Subrange \bullet \\
\quad renSRs\ m(\langle sr \rangle \frown SR) = \langle renSR\ m\ sr \rangle \frown renSRs\ m\ SR
\end{array}
$$

Types are then renamed in the obvious way

$$
\begin{array}{|l}
renType : ID \pfun TYPE \pfun TYPE \\
\hline
\forall\, m : ID;\ \tau : \{unsigned, pbyte, boolean, typeWrong, indirectAddr, returnAddr\} \bullet \\
\quad renType\ m\ \tau = \tau \\
\forall\, m : ID;\ sr : Subrange \bullet \\
\quad renType\ m(subrange\ sr) = subrange(renSR\ m\ sr) \\
\forall\, m, \xi : ID \bullet \\
\quad renType\ m(typeName\ \xi) = typeName(renID\ m\ \xi) \\
\quad \land\ renType\ m(typeValue\ \xi) = typeValue(renID\ m\ \xi) \\
\forall\, m : ID;\ ScopedType \bullet \\
\quad renType\ m(scopedTypeName\ \theta ScopedType) = \\
\qquad scopedTypeName \langle\!|\ \xi == renID\ m\ \xi, b == renID\ m\ b\ |\!\rangle \\
\quad \land\ renType\ m(scopedTypeValue\ \theta ScopedType) = \\
\qquad scopedTypeValue \langle\!|\ \xi == renID\ m\ \xi, b == renID\ m\ b\ |\!\rangle
\end{array}
$$

13.4.3 Renaming expressions

Expressions are renamed by renaming their constituent parts. Each value reference and function call name are renamed, unless they refer to an exported name.

$$
\begin{array}{|l}
renExpr : ID \pfun EXPR \pfun EXPR \\
\hline
\forall\, m : ID;\ \kappa : VALUE \bullet renExpr\ m(constant\ \kappa) = constant\ \kappa \\
\forall\, m : ID;\ ValueRefExpr \bullet \\
\quad renExpr\ m(valueRef\ \theta ValueRefExpr) = \\
\qquad valueRef \langle\!|\ \xi == renID\ m\ \xi, E == renExpr\ m \circ E\ |\!\rangle \\
\forall\, m : ID;\ UnyExpr \bullet \\
\quad renExpr\ m(unyExpr\ \theta UnyExpr) = \\
\qquad unyExpr \langle\!|\ \Psi == \Psi, \epsilon == renExpr\ m\ \epsilon\ |\!\rangle \\
\forall\, m : ID;\ BinExpr \bullet \\
\quad renExpr\ m(binExpr\ \theta BinExpr) = \\
\qquad binExpr \langle\!|\ \Omega == \Omega, \epsilon 1 == renExpr\ m\ \epsilon 1, \epsilon 2 == renExpr\ m\ \epsilon 2\ |\!\rangle \\
\forall\, m : ID;\ FunCallExpr \bullet \\
\quad renExpr\ m(funCall\ \theta FunCallExpr) = \\
\qquad funCall \langle\!|\ \xi == renID\ m\ \xi, E == renExpr\ m \circ E\ |\!\rangle
\end{array}
$$

13.4.4 Renaming statements

Statements can be renamed similarly to expressions, with the only constructs that require any work being *assign* and *procCall*.

$$renStmt : ID \pfun STMT \pfun STMT$$

13.4.5 Simple Declaration

We rename a simple declaration by renaming the specific kind of declaration.

$$renSimpleDecl : ID \pfun SIMPLE\_DECL \pfun SIMPLE\_DECL$$

$$\begin{aligned}
&\forall\, m : ID;\ ConstDecl \bullet \\
&\quad renSimpleDecl\ m(constDecl\ \theta ConstDecl) = \\
&\qquad constDecl \lbag\, \xi == renID\ m\ \xi, \kappa == \kappa \,\rbag \\
&\forall\, m : ID;\ EnumDecl \bullet \\
&\quad renSimpleDecl\ m(typeDecl(enumDecl\ \theta EnumDecl)) = \\
&\qquad typeDecl(enumDecl \lbag\, \xi == renID\ m\ \xi, \\
&\qquad\quad \Xi == renID\ m \circ \Xi \,\rbag) \\
&\forall\, m : ID;\ VarDecl \bullet \\
&\quad renSimpleDecl\ m(varDecl\ \theta VarDecl) = \\
&\qquad varDecl \lbag\, \xi == renID\ m\ \xi, A == A, \\
&\qquad\quad SR == renSRs\ m\ SR, \tau == renType\ m\ \tau, V == V \,\rbag
\end{aligned}$$

The renamed simple declarations of a *Module* and a *MainModule* are

$$renSDofM : Module \pfun \mathrm{seq}\, SIMPLE\_DECL$$

$$\forall\, Module \bullet renSDofM\ \theta Module = renSimpleDecl\ \xi\ \circ \Delta S$$

$$renSDofMA : MainModule \pfun \mathrm{seq}\, SIMPLE\_DECL$$

$$\forall\, MainModule \bullet renSDofMA\ \theta MainModule = renSimpleDecl\ \xi \circ \Delta S$$

13.4.6 Renaming procedures and functions

Formal parameter declarations and routine body are renamed in the obvious way.

$$renParamD : ID \pfun ParamDecl \pfun ParamDecl$$

$$renBody : ID \pfun Body \pfun Body$$

The rename function, and procedure declarations are defined next. The routine names are renamed only if they are not exported.

$$renProcHdr : ID \pfun ProcHdr \pfun ProcHdr$$

$$\begin{aligned}
&\forall\, m : ID;\ ProcHdr \bullet \\
&\quad renProcHdr\ m\ \theta ProcHdr = \\
&\qquad \langle\!|\, \xi == renID\ m\ \xi, \Pi == renParamD\ m \circ \Pi \,|\!\rangle
\end{aligned}$$

$$renProcD : ID \pfun ProcDecl \pfun ProcDecl$$

$$\begin{aligned}
&\forall\, m : ID;\ ProcDecl \bullet \\
&\quad \exists\, ProcDecl' \mid \\
&\qquad \theta ProcHdr' = renProcHdr\ m\ \theta ProcHdr \\
&\qquad \land\ \theta Body' = renBody\ m\ \theta Body \bullet \\
&\quad renProcD\ m\ \theta ProcDecl = \theta ProcDecl'
\end{aligned}$$

$$renFunD : ID \pfun FunDecl \pfun FunDecl$$

$$\begin{aligned}
&\forall\, m : ID;\ FunDecl \bullet \\
&\quad \exists\, FunDecl' \mid \\
&\qquad \theta ProcDecl' = renProcD\ m\ \theta ProcDecl \\
&\qquad \land\ \tau' = renType\ m\ \tau \bullet \\
&\quad renFunD\ m\ \theta FunDecl = \theta FunDecl'
\end{aligned}$$

The procedure and function declaration is defined in terms of its constituent parts.

$$
\begin{array}{|l}
renPFD : ID \pfun PROC\_FUN\_DECL \pfun PROC\_FUN\_DECL \\
\hline
\forall\, m : ID;\ \delta : ProcDecl \bullet \\
\quad renPFD\ m(procDecl\ \delta) = procDecl(renProcD\ m\ \delta) \\
\forall\, m : ID;\ \delta : FunDecl \bullet \\
\quad renPFD\ m(funDecl\ \delta) = funDecl(renFunD\ m\ \delta)
\end{array}
$$

The renamed procedure and function declarations of a *Module* are

$$
\begin{array}{|l}
renPFDofM : Module \pfun \mathrm{seq}\, PROC\_FUN\_DECL \\
\hline
\forall\, Module \bullet renPFDofM\ \theta Module = renPFD\ \xi\ \circ \Delta PF
\end{array}
$$

The renamed procedure and function declarations of a *MainModule* are

$$
\begin{array}{|l}
renPFDofMA : MainModule \pfun \mathrm{seq}\, PROC\_FUN\_DECL \\
\hline
\forall\, MainModule \bullet renPFDofMA\ \theta MainModule = renPFD\ \xi\ \circ \Delta PF
\end{array}
$$

13.5 Modularised program – Dynamic Semantics

The dynamic semantics for the modularised program define the Pasp meaning of the program. Each module on its own does not have any meaning and it is only the combination of all the modules into a modularised program which has any meaning.

The meaning of the *MPROG* program is given by the meaning of the corresponding flattened *Prog* program.

Only various selected parts of each *MPROG* are required for use in the program. The imports and exports can be discarded. Each required sequence of declarations can be extracted from the *MPROG* in question by extracting the required part from each *Module*.

$$
\begin{array}{|l}
\mathit{flatten} : \mathit{MPROG} \rightarrow \mathit{ATTR} \rightarrow \mathit{Prog} \\
\hline
\forall\, M : \operatorname{seq} \mathit{Module};\ m : \mathit{MainModule};\ a : \mathit{ATTR} \bullet \\
\quad \exists\, \mathit{Prog} \mid \\
\qquad \xi = m.\xi \\
\qquad \land\ \Delta S = \frown/(\mathit{renSDofM} \circ M) \frown \mathit{renSDofMA}\ m \\
\qquad \land\ \Delta PF = \frown/(\mathit{renPFDofM} \circ M) \frown \mathit{renPFDofMA}\ m \\
\qquad \land\ \gamma = \mathit{renStmt}\ \xi\ m.\gamma \bullet \\
\quad \mathit{flatten}(M, m)a = \theta \mathit{Prog}
\end{array}
$$

The dynamic (meaning) semantics of the modularised program can now be defined. The meaning of the *MPROG* is written in terms of the meaning of a *Prog*.

The *MPROG* is checked to be correctly defined for the symbol declaration and type checking semantics. This ensures that when the *MPROG* is split apart, and re-assembled without the imports or exports, then the remaining function and procedure calls make sense. This is also achieved by renaming definitions within each module to avoid name clashes when amalgamated together.

The declared of $\mathcal{M}_P$ is required for the modularised program re-writing, but is defined in the next section.

$$
\begin{array}{|l}
\mathcal{M}_P : \mathit{Prog} \nrightarrow \mathit{Store} \nrightarrow \mathit{Store}
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{M}_{MP} : \mathit{MPROG} \nrightarrow \mathit{ATTR} \nrightarrow \mathit{Store} \nrightarrow \mathit{Store} \\
\hline
\forall\, M : \operatorname{seq} \mathit{Module};\ m : \mathit{MainModule};\ a : \mathit{ATTR} \mid \mathit{MProgTypeOkay} \bullet \\
\quad \exists\, \mathit{Prog} \mid \theta \mathit{Prog} = \mathit{flatten}(M, m)a \bullet \\
\qquad \mathcal{M}_{MP}(M, m)a = \mathcal{M}_P\ \theta \mathit{Prog}
\end{array}
$$

14 Program

14.1 Program – Symbol declaration semantics

The declaration meaning of a program maps it to an environment. A program's check status is the result of checking the body statement in the environment of the declarations; note that the declarations may include procedures and functions of which the bodies are also checked. The outermost, global, block is given the name of the program.

$$
\begin{array}{|l}
\mathcal{D}_P : Prog \pfun EnvDTrace \\
\hline
\forall\, Prog \bullet \\
\quad \exists\, \rho\delta t, \rho\delta t', \rho\delta t'' : EnvDTrace \mid \\
\qquad \rho\delta t = \mathcal{D}_{SD}\ {}_*\ \Delta S\ \langle\xi\rangle\ \{\xi \mapsto \rho\delta 0\} \\
\qquad \land\ \rho\delta t' = \mathcal{D}_{PFD}\ {}_*\ \Delta PF\ \langle\xi\rangle\ \rho\delta t \\
\qquad \land\ \rho\delta t'' = \mathcal{D}_S\ \gamma\ \langle\xi\rangle\ \rho\delta t' \bullet \\
\quad \mathcal{D}_P\ \theta Prog = \rho\delta t''
\end{array}
$$

The constant declaration list is checked in an initial trace environment containing the reserved constant identifiers, yielding an updated trace environment. Then the variable declaration list is checked in this, yielding a trace environment complete with all the global constant and variable declarations. Finally, the procedure/function declaration list is checked in this; the procedure and function names are checked to ensure that they have not already been defined, and each nested block is also checked in the relevant environment.

The condition for the rest of the semantics to be defined is that every identifier in the final trace environment maps to $checkOK$.

$$
\begin{array}{|l}
\hline
ProgDeclOkay \\
Prog \\
\hline
(\mathrm{ran} \circ \bigcup \circ \mathrm{ran})(\mathcal{D}_P\ \theta Prog) \subseteq \{checkOK\} \\
\hline
\end{array}
$$

It is a subset relation, not equality, because it is possible that the program declares no identifiers.

14.2 Program – Type checking semantics

The type checking meaning of a program maps it to a a trace type environment (preserved for use in the use, dynamic, and operational semantics) and a map from blocks to check status. The statement is checked in the environment of the declarations.

$$
\begin{array}{|l}
\mathcal{T}_P : Prog \pfun DeclType \\
\hline
\forall\, Prog \mid ProgDeclOkay \bullet \\
\quad \exists\, DeclType;\ DeclType';\ StmtType'';\ tc'' : Env[CHECK] \mid \\
\qquad \theta DeclType = \\
\qquad\quad \mathcal{T}_{SD}\ {}_*\ \Delta S \langle \xi \rangle \llparenthesis\ \rho\tau t == \{\xi \mapsto \rho\tau 0\}, tc == \varnothing\ \rrparenthesis \\
\qquad \land\ \theta DeclType' = \mathcal{T}_{PFD}\ {}_*\ \Delta PF \langle \xi \rangle \theta DeclType \\
\qquad \land\ \theta StmtType'' = \mathcal{T}_S\ \gamma \langle \xi \rangle \rho\tau t' \\
\qquad \land\ tc'' = tc' \oplus \{\xi \mapsto c''\} \bullet \\
\quad \mathcal{T}_P\ \theta Prog = \llparenthesis\ \rho\tau t == \rho\tau t'', tc == tc''\ \rrparenthesis
\end{array}
$$

The condition for the rest of the semantics to be defined is that the type checking semantics yields *checkOK* for every block in the program:

ProgTypeOkay

$$
\begin{array}{|l}
ProgDeclOkay \\
\hline
\mathrm{ran}(\mathcal{T}_P\ \theta Prog).tc = \{checkOK\}
\end{array}
$$

14.3 Program – Use semantics

The meaning function defined for the use after declaration semantics on declarations of variables just checks the use of defined types. The initial use environment is empty.

The initial value of use semantics for procedures and functions is formed when checking their declaration, since it contains the body statement.

We now define the use semantics of a program. The environment status is the result of checking the statement in the environment formed from the initially empty environment.

$$
\begin{array}{|l}
\mathcal{U}_P : Prog \pfun EnvUTrace \\
\hline
\forall\, Prog \mid ProgTypeOkay \bullet \\
\quad \exists\, \rho\upsilon t, \rho\upsilon t', \rho\upsilon t'' : EnvUTrace \mid \\
\qquad \rho\upsilon t = \mathcal{U}_{SD} \ast \Delta S \ \langle \xi \rangle \ \varnothing \\
\qquad \land\ \rho\upsilon t' = \mathcal{U}_{PFD} \ast \Delta PF \ \langle \xi \rangle \ \rho\upsilon t \\
\qquad \land\ \rho\upsilon t'' = \mathcal{U}_S \ \gamma \ \langle \xi \rangle \ \rho\upsilon t' \bullet \\
\quad \mathcal{U}_P \ \theta Prog = \rho\upsilon t''
\end{array}
$$

All names should be properly used. An implementation should deliver a *warning* if this is not the case.

ProgUseOkay

$$
\begin{array}{|l}
ProgTypeOkay \\
\hline
\varnothing = \mathrm{ran}(\mathcal{U}_P \ \theta Prog)
\end{array}
$$

14.4 Program – Dynamic semantics

Before defining the dynamic semantics of a Pasp program, it is necessary to define a function that allocates locations for temporary variables. These variables are associated with case statements and have no declaration; hence locations must be assigned by extracting information from the type trace environment.

$$
\begin{array}{|l}
\mathcal{M}_{Env} : EnvTTrace \pfun EnvMTrace \\
\hline
\forall\, \rho\tau t : EnvTTrace \bullet \\
\quad \exists\, \rho t : EnvMTrace \mid \\
\qquad (\ \forall\, b, \xi : ID \bullet \\
\qquad\quad (b\ \in \dom \rho t \Leftrightarrow b\ \in \dom \rho\tau t) \\
\qquad\quad \land\ (\xi\ \in \dom(\rho t\ b) \Leftrightarrow \rho\tau t\ b\ \xi\ \in \ran\, tempVar)\ ) \\
\qquad \land\ (\ \forall\, b : \dom \rho\tau t;\ \xi : ID \mid \xi\ \in \dom(\rho t\ b) \bullet \\
\qquad\quad \exists\, f : \seq \mathbb{N}\ \rightarrowtail LOCN \mid \dom f = \{\langle\ \rangle\} \bullet \\
\qquad\quad\quad \rho t\ b\ \xi = loc\ f\ ) \\
\qquad \land\ (\ \forall\, b, b' : \dom \rho\tau t;\ \xi, \xi' : ID \mid \\
\qquad\quad\quad \{\rho\tau t\ b\ \xi, \rho\tau t\ b'\ \xi'\} \subseteq \ran\, tempVar \bullet \\
\qquad\quad \rho t\ b\ \xi = \rho t\ b'\ \xi' \Leftrightarrow b' = b \land \xi = \xi'\ ) \bullet \\
\quad\ \mathcal{M}_{Env}\ \rho\tau t = \rho t
\end{array}
$$

This loose specification has three conditions on the resulting trace environment.

- The environment is defined for precisely the set of temporary variables.
- Each temporary variable is mapped to a single location.
- The environment is injective; no location is used more than once.

Given type and meaning environments, the locations allocated to variables with a certain attribute are

$$
\begin{array}{|l}
locnOfAttr : EnvTTrace \times EnvMTrace \pfun ATTR \pfun \mathbb{P}\, LOCN \\
\hline
\forall\, \rho\tau t : EnvTTrace;\ \rho t : EnvMTrace;\ a : ATTR \bullet \\
\quad \exists\, l : \mathbb{P}\, LOCN \mid \\
\qquad l = \bigcup \{\ b : \dom \rho\tau t;\ \xi : ID \mid \\
\qquad\qquad\qquad \xi \in \dom(\rho\tau t\ b) \\
\qquad\qquad\qquad \land\ a \in attributeOf(\rho\tau t\ b\ \xi) \bullet \\
\qquad\qquad \ran(loc^{\sim}(\rho t\ b\ \xi))\} \\
\quad\ locnOfAttr(\rho\tau t, \rho t)a = l
\end{array}
$$

The dynamic meaning of a program is its final output streams as a function of its input streams.

$$
\begin{array}{|l}
\mathcal{M}_P : Prog \pfun Store \pfun Store \\
\hline
\forall\, Prog;\ \Sigma : Store \mid ProgTypeOkay \bullet \\
\quad \exists\, DeclType;\ \rho t, \rho t', \rho t'' : EnvMTrace\sigma, \sigma', \sigma'' : State \mid \\
\qquad \theta DeclType = \mathcal{T}_P\ \theta Prog \\
\qquad \land\ \rho t = \mathcal{M}_{Env}\ \rho\tau t \\
\qquad \land\ \rho t' = \mathcal{M}_{SD}\ {}_*\ \Delta S\langle\xi\rangle\rho t \\
\qquad \land\ \rho t'' = \mathcal{M}_{PFD}\ {}_*\ \Delta PF\langle\xi\rangle\rho t' \\
\qquad \land\ \sigma.\Sigma = \Sigma \\
\qquad \land\ \sigma.i = locnOfAttr(\rho\tau t, \rho t) readOnly \\
\qquad \land\ \sigma.o = locnOfAttr(\rho\tau t, \rho t) writeOnly \\
\qquad \land\ \mathrm{dom}(\Sigma \rhd \mathrm{ran}\ stream) = \sigma.i \cup \sigma.o \\
\qquad \land\ \sigma' = \mathcal{M}_{SDI^*}\ \Delta S\langle\xi\rangle\rho t''\ \sigma \\
\qquad \land\ \sigma'' = \mathcal{M}_S\ \gamma\langle\xi\rangle\rho t''\ \sigma' \bullet \\
\quad \mathcal{M}_P\ \theta Prog\ \Sigma = outOf\ \sigma''
\end{array}
$$

The semantics are defined only if the static checks are satisfied.

The dynamic semantics of the declarations is used to define the environment (which allocates locations) and the input and output subsets. These must satisfy the following constraints:

- The locations allocated to all *readOnly* and *writeOnly* variables are the sets of input and output locations respectively.
- The locations mapped by the store to a stream value are precisely these input and output sets.

The program's meaning is given by the output streams of the meaning of the statement applied to the store with the variables initialised, and with the input and output locations determined as described above.

A General DeCCo Toolkit

This section contains definitions in common to several of the DeCCo specifications.

A.1 Raising to a power

Z has no 'raise to a power' operator, so we define one.

$$\textbf{function}\ 30\ \textbf{leftassoc}(\_ \uparrow \_)$$

$$\_ \uparrow \_ : \mathbb{N}_1 \times \mathbb{N} \to \mathbb{N}$$

$$\forall\, n : \mathbb{N}_1 \bullet n \uparrow 0 = 1$$

$$\forall\, n, p : \mathbb{N}_1 \bullet n \uparrow p = n * (n \uparrow (p - 1))$$

A.1.1 Byte

The byte type is defined so as to take advantage of the greater speed offered by using single register operations in the compiled code.

$$BYTE == 0 \,..\, 255$$

A.1.2 Boolean

We model the boolean type with a free type of two constants:

$$BOOLEAN ::= ptrue \mid pfalse$$

Note: Z has *true* and *false* as keywords, so those names cannot be used in the definition of $BOOLEAN$.

A.2 Bits

In order to define the logical operators such as *or*, *and*, and *xor*, it is necessary to be able to write numbers as sequences of bits. We first define a BIT.

$$BIT == \{0, 1\}$$

The first element of the sequence is the least significant bit.

$$\begin{array}{|l} NatToBits : \mathbb{N} \rightarrow \operatorname{seq} BIT \\ \hline NatToBits\, 0 = \langle\,\rangle \\ \forall\, n : \mathbb{N}_1 \bullet NatToBits\, n = \langle n \bmod 2 \rangle \frown NatToBits(n \operatorname{div} 2) \end{array}$$

This definition ensures that the most significant bit (if there is one) is always one (i.e. there is no leading zero).

The inverse of the above function is given by

$$\begin{array}{|l} BitsToNat : \operatorname{seq} BIT \nrightarrow \mathbb{N} \\ \hline BitsToNat\langle\,\rangle = 0 \\ \forall\, b : BIT;\ B : \operatorname{seq} BIT \bullet \\ \quad BitsToNat(\langle b \rangle \frown B) = b + 2 * BitsToNat\ B \end{array}$$

Logical operators are defined for single bits. The definitions are given in terms of arithmetic operations, for concision.

$$\begin{array}{|l} BitNot : BIT \rightarrowtail BIT \\ BitOr, BitXor, BitAnd : BIT \times BIT \rightarrow BIT \\ \hline BitNot = \{0 \mapsto 1, 1 \mapsto 0\} \\ \forall\, b, c : BIT \bullet \\ \quad BitOr(b, c) = 1 - (1 - b) * (1 - c) \\ \quad \land\ BitXor(b, c) = (b + c) \bmod 2 \\ \quad \land\ BitAnd(b, c) = b * c \end{array}$$

These definitions are extended to bit sequences. First we define OR. Recall that the empty sequence represents zero.

function 30 **leftassoc**($\_OR\_$)

$$\_\,OR\,\_ : \operatorname{seq} BIT \times \operatorname{seq} BIT \to \operatorname{seq} BIT$$

$$\begin{aligned}
&\forall\, s, t : \operatorname{seq} BIT;\ b, c : BIT \bullet \\
&\quad s\ OR\ \langle\rangle = s \land \langle\rangle\ OR\ t = t \\
&\quad \land\ (\langle b\rangle \frown s) OR (\langle c\rangle \frown t) = \langle BitOr(b, c)\rangle \frown (s\ OR\ t)
\end{aligned}$$

function 30 **leftassoc**($\_XOR\_$)

$$\_\,XOR\,\_ : \operatorname{seq} BIT \times \operatorname{seq} BIT \to \operatorname{seq} BIT$$

$$\begin{aligned}
&\forall\, s, t : \operatorname{seq} BIT;\ b, c : BIT \bullet \\
&\quad s\ XOR\ \langle\,\rangle = s \land \langle\,\rangle\ XOR\ t = t \\
&\quad \land\ (\langle b\rangle \frown s) XOR (\langle c\rangle \frown t) = \langle BitXor(b, c)\rangle \frown (s\ XOR\ t)
\end{aligned}$$

function 30 **leftassoc**($\_AND\_$)

$$\_\,AND\,\_ : \operatorname{seq} BIT \times \operatorname{seq} BIT \to \operatorname{seq} BIT$$

$$\begin{aligned}
&\forall\, s, t : \operatorname{seq} BIT;\ b, c : BIT \bullet \\
&\quad s\ AND\ \langle\,\rangle = s \land \langle\,\rangle\ AND\ t = t \\
&\quad \land\ (\langle b\rangle \frown s) AND (\langle c\rangle \frown t) = \langle BitAnd(b, c)\rangle \frown (s\ AND\ t)
\end{aligned}$$

B Pasp Toolkit

B.1 Distributed override

The generic function $\oplus/$ distributes function overriding over a sequence.

$$[L, X]$$

$$\oplus/ : \mathrm{seq}(L \pfun X) \to (L \pfun X)$$

$$\oplus/ \langle\rangle = \varnothing$$

$$\forall f : L \pfun X;\ s : \mathrm{seq}(L \pfun X) \bullet \oplus/(s \frown \langle f \rangle) = \oplus/\, s \oplus f$$

B.2 Extending meanings to sequences

Many of the meaning functions on sequences of items can be derived from the relevant meaning function on single items in a generic manner.

Some meaning functions $\mathcal{M}$ take an item ξ in the context of some block b and environment ρ, to yield a new environment ρ'. The meaning of a sequence of such items is often the meaning of tail of the sequence, evaluated in that environment given by the meaning of the first item. This meaning is $\mathcal{M}_*$, where

function$(\_*)$

$$[X, B, E]$$

$$\_* : (X \to B \to E \to E) \to \mathrm{seq}\, X \to B \to E \to E$$

$$\begin{aligned}&\forall\, m : X \to B \to E \to E;\ \xi : X;\ \Xi : \mathrm{seq}\, X;\ b : B \bullet \\ &\quad m\,*\,\langle\rangle\, b = \mathrm{id}\, E \\ &\quad \land\ m\,*(\langle\xi\rangle \frown \Xi)b = m\,*\,\Xi\, b \circ m\,\xi\, b\end{aligned}$$

Some meaning functions $\mathcal{M}$ take an item ξ in the context of some idType i, blockstack s, block b and environment ρ, to yield a new environment ρ'. The meaning of a sequence of such items is often the meaning of tail of the

sequence, evaluated in that environment given by the meaning of the first item. This meaning is $\mathcal{M}_*$, where

function$(\_*)$

$$
\begin{array}{|l}
[X, I, S, B, E] \\
\hline
\_* : (X \to I \to S \to B \to E \to E) \to \\
\qquad (\mathrm{seq}\ X \to \mathrm{seq}\, I \to S \to B \to E \to E) \\
\hline
\forall\, m : X \to I \to S \to B \to E \to E;\ \xi : X; \\
\qquad \Xi : \mathrm{seq}\, X;\ i : I;\ ii : \mathrm{seq}\, I;\ s : S;\ b : B \bullet \\
\qquad m\, *\, \langle\,\rangle\, \langle\,\rangle\, s\ b = \mathrm{id}\, E \\
\qquad \land\ m\, *(\langle \xi \rangle \frown \Xi)(\langle i \rangle \frown ii) s\ b = m\, *\, \Xi\ ii\ s\ b \circ m\, \xi\ i\ s\ b
\end{array}
$$

B.3 Vector operations

The following two functions perform vector summation and dot products respectively.

$$
\begin{array}{|l}
sum : \mathrm{seq}\, \mathbb{Z} \times \mathrm{seq}\, \mathbb{Z} \pfun \mathrm{seq}\, \mathbb{Z} \\
dotProduct : \mathrm{seq}\, \mathbb{Z} \times \mathrm{seq}\, \mathbb{Z} \pfun \mathbb{Z} \\
\hline
sum(\langle\,\rangle, \langle\,\rangle) = \langle\,\rangle \\
dotProduct(\langle\,\rangle, \langle\,\rangle) = 0 \\
\forall\, s1, s2 : \mathrm{seq}\, \mathbb{Z};\ n1, n2 : \mathbb{Z} \mid \#s1 = \#s2 \bullet \\
\qquad sum(\langle n1 \rangle \frown s1, \langle n2 \rangle \frown s2) = \langle n1 + n2 \rangle \frown sum(s1, s2) \\
\qquad \land\ dotProduct(\langle n1 \rangle \frown s1, \langle n2 \rangle \frown s2) = \\
\qquad\qquad n1 * n2 + dotProduct(s1, s2)
\end{array}
$$

References

[Formaliser] Susan Stepney. *The Formaliser Manual*. Logica.

[ISO-Z] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*. 2002.

[Spivey 1992] J. Michael Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Stepney *et al.* 1991] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *BCS Formal Aspects of Computing*, 3:58–101, 1991.

[Stepney 1993] Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[Stepney 1998] Susan Stepney. Incremental development of a high integrity compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington DC*, 1998.

[Stringer-Calvert *et al.* 1997] David W.J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, number 1313 in Lecture Notes in Computer Science, pages 573–588. Springer Verlag, September 1997.

Index