The DeCCo project papers III

Z Specification of Compiler Templates

Susan Stepney and Ian T. Nabney

University of York Technical Report YCS-2002-360

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 gives an algorithmic specification that defines the compiler; it defines what AspAL statements are produced for each Pasp fragment. Throughout the definition, wherever there has been a choice between clarity and efficiency, the latter has been sacrificed to the former, so that the correctness proofs will be made easier. The most obvious example of this is in the extensive use of indirect instructions (see the AspAL definition document) rather than immediate instructions in the definition of operators.

The specification has developed over time. This is due to adding additional functionality as well as correcting errors shown up by proof work, and further analysis. The additional functionality had been a combination of new instructions and registers being provided on the chip by AWE, as well as development, and optimisation of the operation of the compiler.

1.1 Critical assumption

Any module compiled under this specification has passed declaration and type checking as specified in the formal Pasp language semantics.

That is, for each ordinary module

$$\mathit{ModuleTypeOkay}$$

and for the main module

$$\mathit{MainModuleTypeOkay}$$

Parent Section: Pasp Specification

Parent Section: AspAL Specification

2 Translation domains

This section gives the definition of the translation environment. It also discusses the allocation of Asp data addresses to Pasp values. A semantics to determine allocation of space on the stack is discussed. Finally, a number of general functions and template fragments are defined.

2.1 Translation environment

During translation, Pasp variables are allocated Asp data addresses (memory locations); the translation environment models this relationship. In defining this environment, we also bear in mind the requirements for proving the correctness of the compiler. Specifically, we need to be able to retrieve a Pasp state from an Asp state and the translation environment, in order to show that variables in the retrieved state have the same values as those in the corresponding state produced by the Pasp dynamic semantics.

To help with this, the translation environment is divided into two parts, the operational environment *EnvO*, and the memory allocation environment, *MO*. This division of the environment allows for scoped declarations. (It also makes the proofs easier, because after translating the declarations, the location environment should be equal to the Pasp dynamic semantics environment. Thus in the remainder of the proof only the retrieved state and the dynamic state need be compared.)

2.1.1 Operational environment

The operational environment *EnvO* maps Pasp identifiers to their operational denotable values: the values that an identifier can take in the operational semantics. (This is analogous to the environment *Env* defined for the Pasp dynamic semantics.) The operational denotable values are:

$$\begin{aligned} OPDENVALUE ::=\ & opLoc\langle\langle \mathrm{seq}\,\mathbb{N} \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow LOCN\rangle\rangle \\ & \mid opPFval\langle\langle LABEL \times LOCN\rangle\rangle \\ & \mid opImport\langle\langle ID\rangle\rangle \mid opExport\langle\langle VALUE\rangle\rangle \end{aligned}$$

- *opLoc* is used for variables and formal parameters. It has the same form as the *DENVALUE* of *loc* in the Pasp denotable values: an injection from a sequence of array indexes to the start location of that array.
- *opPFval* is used for procedures and functions. When a procedure or function is called, the program stores the return address in the abstract *LOCN* and then jumps to the *LABEL*.
- *opImport* is used for imported constants and (read only) variables. When such a constant or variable is accessed, the actual value or memory location cannot be determined until link time.
- *opExport* is used for the value of all constants (some of which may be exported).

The operational environment is a mapping from identifiers to denotable values.

$$EnvO == Env[OPDENVALUE]$$

The strategy used for location allocation for variables and formal parameters is the same as that used in Pasp.

The trace environment associates an environment with each block in a program. (Recall that the current implementation there are only two block levels: the outermost 'main' block labelled by the module name, and the block inside any procedure or function body, labelled by that procedure's or function's name.)

$$EnvOTrace == EnvTr[OPDENVALUE]$$

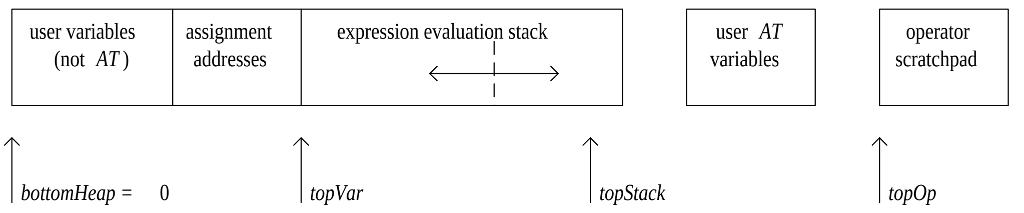

Figure 1: The memory map for a compiled Pasp module.

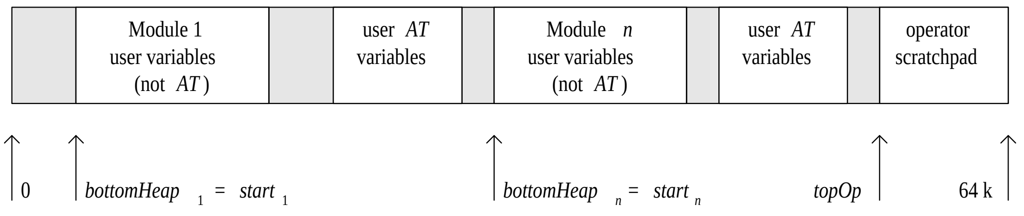

Figure 2: The memory map for a compiled and linked Pasp program.

2.1.2 Memory allocation map

Locations are abstract. Each value is stored in a single location, but Pasp values have different sizes (one byte or two bytes), so abstract locations can take up a variable number of physical Asp addresses.

Memory allocation maps an abstract location to the sequence of (one or two) physical data addresses it corresponds to.

$$MO == LOCN \pfun \operatorname{seq} ADDR$$

2.2 Allocation of data addresses

The memory allocation scheme for a single module is shown in figure 1. Each module is allocated an area of memory, which has

- a segment of memory for the module's variables (the *heap*), which starts at *bottomHeap*, and extends upwards to *topVar*.

- 'assignment addresses': two bytes for each block, used to implement the *assign* statement template.
- an expression evaluation stack (for storing temporary intermediate values when evaluating nested and binary expressions), which starts at *topStack* and expands *downwards*; the first value on the stack is stored at the addresses $(topStack, topStack + 1)$.

Here we are concerned with compiling a single module, so we need only one (relative) value for each of *topVar* and *topStack*. These are converted to absolute addresses (a different *bottomHeap* for each module) by the linker (figure 2), which also arranges the module memory blocks around any fixed location variables (those allocated by the programmer to particular physical addresses).

$$\begin{array}{|l} topVar : ADDR \\ topStack : ADDR \\ assignAddr : ID \rightarrowtail \operatorname{seq} ADDR \end{array}$$

Finally, there is a global 'operator scratchpad'. The 24 locations required for the operator scratchpad are reserved at the very top of the memory map. (For historical reasons, the bottom of this region has the name *topOp*.)

$$\begin{array}{|l} topOp : ADDR \end{array}$$

In performing arithmetic on several bytes it is usual to work from the least significant byte to the most significant. We store multi-byte Pasp unsigned values with the lo byte at address δ, and the hi byte at address $\delta + 1$, and use the address increment opcode to move along the value.

The function *sizeof* returns the number of bytes taken up by a value of some Pasp types[1]. The size of a defined type[2] is taken to be 1.

[1]Note that *subrange* is not in the domain of *sizeof*. *sizeof* should be called with a type calculated from the type environment, which knows the base type of subranges.

[2]Currently the only defined type is an enumerated type. If records are added in the future, it will be necessary to pass extra information to *sizeof*.

$$\begin{array}{|l} sizeof : TYPE \pfun \mathbb{N} \\ \hline sizeof = \\ \quad \{boolean \mapsto 1, pbyte \mapsto 1, unsigned \mapsto 2, indirectAddr \mapsto 2\} \\ \quad \cup \{ \ ScopedType \bullet scopedTypeValue \ \theta ScopedType \mapsto 1 \ \} \end{array}$$

2.3 Stack set-up semantics

Statements are used inside function call expressions, so we cannot assume that the expression evaluation stack is empty when we translate a statement. The offset from the top of the stack required for a statement is constant for all statements in any given block. (Note that the main program block always has an offset of zero.)

The size of this offset from the stack top for each block is given by the following function:

$$OFFSET == \mathbb{N}$$
$$blockOffset == ID \pfun OFFSET$$

In order to calculate this function, it is necessary to traverse the program blocks in calling order, from last to first. This constitutes a 'pass' through the program, and we define a 'static semantics' to carry it out.

The stack set-up semantics $\mathcal{S}$ defines the block offset function for a program. It is viewed as part of the compiler definition, as it is specific to the requirements of the target device and target language.

The final block offset for a complete module is made global, for use in the operational semantics. Its value is defined by the operational semantics of a module.

$$\begin{array}{|l} \rho st0 : blockOffset \end{array}$$

2.4 Utility template fragments

There are some AspAL fragments, sequences of X_INSTRs, that are used in a large number of templates[3]. Here, functions that yield these fragments are defined. This modularisation is helpful for two reasons: the specification should be made clearer, and the proof requirements should be made simpler.

2.4.1 Store byte or word

The fragment $\mathcal{O}_{s\tau}$ stores a byte or a word, depending on the size of the supplied type τ.

$$
\begin{array}{|l}
\mathcal{O}_{s\tau} : TYPE \rightarrowtail\!\!\!\!\rightarrow \mathrm{seq}\, X\_INSTR \\
\hline
\forall \tau : TYPE \bullet \\
\quad \mathcal{O}_{s\tau}\ \tau = \mathbf{if}\ sizeof\ \tau = 1 \\
\qquad\qquad \mathbf{then}\ \langle indirect\ \texttt{bstd} \rangle\ \mathbf{else}\ \langle indirect\ \texttt{abstd} \rangle
\end{array}
$$

2.4.2 Load byte or word

The fragment $\mathcal{O}_{l\tau}$ loads a byte or a word, depending on the size of the supplied type τ.

$$
\begin{array}{|l}
\mathcal{O}_{l\tau} : TYPE \rightarrowtail\!\!\!\!\rightarrow \mathrm{seq}\, X\_INSTR \\
\hline
\forall \tau : TYPE \bullet \\
\quad \mathcal{O}_{l\tau}\ \tau = \mathbf{if}\ sizeof\ \tau = 1 \\
\qquad\qquad \mathbf{then}\ \langle indirect\ \texttt{bldd} \rangle\ \mathbf{else}\ \langle indirect\ \texttt{abldd} \rangle
\end{array}
$$

2.4.3 Register manipulations

The fragment $\mathcal{O}_{lwi}$ loads a word-length constant into the a and b registers.

[3]There used to be more, but some have been optimised into single new Asp instructions.

$$\begin{array}{|l}
\mathcal{O}_{lwi} : WORD \rightarrow \operatorname{seq} X\_INSTR \\
\hline
\forall w : WORD \bullet \\
\quad \mathcal{O}_{lwi}\ w = \langle immediate\ (\texttt{aldi}, theHiByte\ w)\rangle \\
\qquad \frown \langle immediate\ (\texttt{bldi}, theLoByte\ w)\rangle
\end{array}$$

The fragment $\mathcal{O}_{l\tau i}$ stores an immediate byte or a word, depending on the size of the supplied type τ.

$$\begin{array}{|l}
\mathcal{O}_{l\tau i} : TYPE \times \mathbb{N} \nrightarrow \operatorname{seq} X\_INSTR \\
\hline
\forall \tau : TYPE;\ n : \mathbb{N} \mid \\
\qquad sizeof\ \tau = 1 \land n\ \in BYTE \\
\qquad \lor\ sizeof\ \tau = 2 \land n\ \in WORD \bullet \\
\quad \mathcal{O}_{l\tau i}\ (\tau, n) = \mathbf{if}\ sizeof\ \tau = 1 \\
\qquad\quad \mathbf{then}\ \langle immediate\ (\texttt{bldi}, n)\rangle\ \mathbf{else}\ \mathcal{O}_{lwi}\ n
\end{array}$$

3 Stack set-up semantics

This semantics calculates the offset from the top of the expression evaluation stack needed when compiling a statement (needed because the statement may be in a function, and function calls may occur in nested expressions taking up stack space). This offset is the same for all statements in any given block.

The $\mathcal{S}$ semantics take an offset and a block offset, and return a new block offset. (Recall that a *blockOffset* maps a block name to the offset for that block.) The 'interesting' part of the definition is in procedure and function calls, where the supplied offset is max'ed with the block offset at the procedure/function name.

3.1 Utility functions

We need to be able to combine *blockOffset* functions pessimistically, so we define $\Uparrow$ to do this. This returns the maximum of the two offsets for each block.

For example, $\{x \mapsto 2, y \mapsto 4\} \Uparrow \{x \mapsto 3, y \mapsto 1\} = \{x \mapsto 3, y \mapsto 4\}$

$$\textbf{function}\ 30\ \textbf{leftassoc}(\_ \Uparrow \_)$$

$$\begin{array}{|l}
\_ \Uparrow \_ : \mathit{blockOffset} \times \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\forall f, g : \mathit{blockOffset} \mid \mathrm{dom}\, f = \mathrm{dom}\, g \bullet \\
\qquad f \Uparrow g = (\lambda\, b : \mathrm{dom}\, f \bullet \mathit{max}\{f\ \ b, g\ \ b\})
\end{array}$$

3.2 Operators

No stack set-up semantics need be defined for operators.

3.3 Expressions

The stack set-up semantics updates the block offset.

The offset passed inductively down to component expressions (and which represents the current stack offset) is incremented if the operational semantics will use space on the expression stack.

3.3.1 Actual parameters

These are checked as expressions.

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

To check a list of actual parameters, the stack semantics are applied to each element in turn, and the results combined pessimistically.

$$\mathcal{S}_{AP^*} : \operatorname{seq} \mathit{EXPR} \rightarrowtail\!\!\!\!\!\rightarrow \mathit{OFFSET} \nrightarrow \mathit{blockOffset} \nrightarrow \mathit{blockOffset}$$

$$\forall \delta : \mathit{OFFSET} \bullet \mathcal{S}_{AP^*}\ \langle\ \rangle\ \delta = \operatorname{id} \mathit{blockOffset}$$
$$\forall E : \operatorname{seq} \mathit{EXPR};\ \epsilon : \mathit{EXPR};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_{AP^*}(\langle \epsilon \rangle \frown E)\delta\ \rho st = \mathcal{S}_{AP^*}\ E\ \delta\ \rho st \Uparrow \mathcal{S}_{AP}\ \epsilon\ \delta\ \rho st$$

3.3.2 Constant

A constant has no effect on the block offset function.

$$\mathcal{S}_E : \mathit{EXPR} \nrightarrow \mathit{OFFSET} \nrightarrow \mathit{blockOffset} \nrightarrow \mathit{blockOffset}$$

$$\forall \kappa : \mathit{VALUE};\ \delta : \mathit{OFFSET} \bullet \mathcal{S}_E(\mathit{constant}\ \kappa)\delta = \operatorname{id} \mathit{blockOffset}$$

3.3.3 Value reference

A value reference uses up no more stack than its expression list.

$$\forall \mathit{ValueRef} \bullet \mathcal{S}_E(\mathit{valueRef}\ \theta \mathit{ValueRef}) = \mathcal{S}_{E^*}\ E$$

3.3.4 Unary expression

A unary expression uses up no more stack than its sub-expression.

$$\forall\, UnyExpr \bullet \mathcal{S}_E(unyExpr\ \theta UnyExpr) = \mathcal{S}_E\ \epsilon$$

3.3.5 Binary expression

A binary expression uses up no more stack than the worse of its two sub-expressions.

However, the offset for the second sub-expression is calculated with 2 added to the current offset, because when it is translated (with the operational semantics) a word of stack space is used to store the result of the first sub-expression. Also, the *umod* binary operator requires an extra 2 bytes of stack space to store an intermediate result, which is also covered by the 2 added to the second sub-expression.

$$\begin{aligned}&\forall\, BinExpr;\ \delta : OFFSET;\ \rho st : blockOffset \bullet\\&\quad \mathcal{S}_E(binExpr\ \theta BinExpr)\delta\ \rho st =\\&\qquad \mathcal{S}_E\ \epsilon\ \delta\ \rho st \Uparrow \mathcal{S}_E\ \epsilon'(\delta + 2)\rho st\end{aligned}$$

3.3.6 Function call

If the value given by the block offset function is less than the current offset in the expression stack (given by the argument δ), then its value is updated. (This updated value will be used when calculating the offset for the function body.) The offset for the function arguments is calculated using the modified block offset function.

$$\begin{aligned}&\forall\, FunCallExpr;\ \delta : OFFSET;\ \rho st : blockOffset \bullet\\&\quad \mathcal{S}_E(funCall\ \theta FunCallExpr)\delta\ \rho st =\\&\qquad \mathcal{S}_{AP^*}\ E\ \delta(\rho st \oplus \{\xi \mapsto max\{\delta, \rho st\ \xi\}\})\end{aligned}$$

3.3.7 Multiple expressions

The operational semantics of expression lists (given by $\mathcal{O}_{E^*}$) use an additional 2 locations on the stack if and only if the expression list is non-empty. (The operational semantics of variable locations (given by $\mathcal{O}_{VL}$) use no additional locations.) So each individual expression is calculated as if it has this larger offset, then the resulting block offset functions from all the expressions are combined pessimistically.

Hence the stack set-up semantics of multiple expressions can be defined as follows.

$$
\begin{array}{|l}
\mathcal{S}_{E^*} : \operatorname{seq} \mathit{EXPR} \leftrightarrow \mathit{OFFSET} \leftrightarrow \mathit{blockOffset} \leftrightarrow \mathit{blockOffset} \\
\hline
\forall \delta : \mathit{OFFSET} \bullet \mathcal{S}_{E^*} \langle\ \rangle \delta = \operatorname{id} \mathit{blockOffset} \\
\forall \epsilon : \mathit{EXPR};\ \delta : \mathit{OFFSET} \bullet \mathcal{S}_{E^*} \langle \epsilon \rangle \delta = \mathcal{S}_E\ \epsilon(\delta + 2) \\
\forall E, E' : \operatorname{seq} \mathit{EXPR};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet \\
\qquad \mathcal{S}_{E^*}(E \frown E')\delta\ \rho st = \mathcal{S}_{E^*}\ E\ \delta\ \rho st \Uparrow \mathcal{S}_{E^*}\ E'\ \delta\ \rho st
\end{array}
$$

3.4 Statements

3.4.1 Multiple statements

The semantics is extended inductively to multiple statements.

$$
\begin{array}{|l}
\mathcal{S}_{S^*} : \operatorname{seq} \mathit{STMT} \leftrightarrow \mathit{OFFSET} \leftrightarrow \mathit{blockOffset} \leftrightarrow \mathit{blockOffset} \\
\hline
\forall \delta : \mathit{OFFSET} \bullet \mathcal{S}_{S^*} \langle\ \rangle \delta = \operatorname{id} \mathit{blockOffset} \\
\forall \gamma : \mathit{STMT};\ \Gamma : \operatorname{seq} \mathit{STMT};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet \\
\qquad \mathcal{S}_{S^*}(\langle \gamma \rangle \frown \Gamma)\delta\ \rho st = \mathcal{S}_{S^*}\ \Gamma\ \delta\ \rho st \Uparrow \mathcal{S}_S\ \gamma\ \delta\ \rho st
\end{array}
$$

3.4.2 Block

Calculating the offset for a block statement is the same as calculating it for the sequence of body statements.

$$\begin{array}{|l}
\mathcal{S}_S : \mathit{STMT} \pfun \mathit{OFFSET} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\forall \Gamma : \operatorname{seq} \mathit{STMT} \bullet \mathcal{S}_S(\mathit{block}\ \Gamma) = \mathcal{S}_{S^*}\ \Gamma
\end{array}$$

3.4.3 Skip

A skip statement has no effect on the stack usage.

$$\begin{array}{|l}
\hline
\forall \delta : \mathit{OFFSET} \bullet \mathcal{S}_S\ \mathit{skip}\ \delta = \operatorname{id} \mathit{blockOffset}
\end{array}$$

3.4.4 Assignment

An assignment statement makes no extra use of the stack beyond that required for calculating the location and expression value.

$$\begin{array}{|l}
\hline
\forall \mathit{AssignStmt};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet \\
\qquad \mathcal{S}_S(\mathit{assign}\ \theta \mathit{AssignStmt})\delta\ \rho st = \mathcal{S}_{E^*}\ E\ \delta\ \rho st \Uparrow \mathcal{S}_E\ \epsilon\ \delta\ \rho st
\end{array}$$

3.4.5 If statement

The offsets required for conditional expression and each branch statement are combined pessimistically.

$$\begin{array}{|l}
\hline
\forall \mathit{IfStmt};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet \\
\qquad \mathcal{S}_S(\mathit{ifStmt}\ \theta \mathit{IfStmt})\delta\ \rho st = \\
\qquad\qquad \mathcal{S}_E\ \epsilon\ \delta\ \rho st \Uparrow \mathcal{S}_S\ \gamma\ \delta\ \rho st \Uparrow \mathcal{S}_S\ \gamma'\ \delta\ \rho st
\end{array}$$

3.4.6 Case statement

The stack semantics for a branch is that of the executed statement.

$$\begin{array}{|l}
\mathcal{S}_{CL} : \mathit{Branch} \pfun \mathit{OFFSET} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\forall \mathit{Branch} \bullet \mathcal{S}_{CL}\ \theta \mathit{Branch} = \mathcal{S}_S\ \gamma
\end{array}$$

This is extended in the usual manner to sequences of branches.

$$\mathcal{S}_{CL^*} : \operatorname{seq} \mathit{Branch} \pfun \mathit{OFFSET} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset}$$

$$\forall \delta : \mathit{OFFSET} \bullet \mathcal{S}_{CL^*} \langle\ \rangle \delta = \operatorname{id} \mathit{blockOffset}$$
$$\forall k : \mathit{Branch};\ K : \operatorname{seq} \mathit{Branch};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_{CL^*}(K \frown \langle k \rangle)\delta\ \rho st = \mathcal{S}_{CL^*}\ K\ \delta\ \rho st \Uparrow \mathcal{S}_{CL}\ k\ \delta\ \rho st$$

The results of calculating the offsets of the expression and the branches are combined pessimistically.

$$\forall \mathit{CaseStmt};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_S(\mathit{caseStmt}\ \theta \mathit{CaseStmt})\delta\ \rho st = \mathcal{S}_E\ \epsilon\ \delta\ \rho st \Uparrow \mathcal{S}_{CL^*}\ C\ \delta\ \rho st$$

3.4.7 Loop

The results of calculating the offsets of the expression and loop body are combined pessimistically.

$$\forall \mathit{WhileStmt};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_S(\mathit{whileStmt}\ \theta \mathit{WhileStmt})\delta\ \rho st = \mathcal{S}_E\ \epsilon\ \delta\ \rho st \Uparrow \mathcal{S}_S\ \gamma\ \delta\ \rho st$$

3.4.8 Procedure call

The calculation for a procedure call is similar to that of a function call.

$$\forall \mathit{ProcCallStmt};\ \delta : \mathit{OFFSET};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_S(\mathit{procCall}\ \theta \mathit{ProcCallStmt})\delta\ \rho st =$$
$$\qquad \mathcal{S}_{AP^*}\ E\ \delta(\rho st \oplus \{\xi \mapsto \mathit{max}\{\delta, \rho st\ \xi\}\})$$

3.5 Declarations

The operational semantics of named constant declarations, type definitions and variable declarations have no effect on the stack usage; hence there is no stack semantics to define for them.

3.5.1 Procedures

The procedure body is processed as a statement.

$$\mathcal{S}_B : \mathit{Body} \rightarrow\!\!\!\!\!\!+ \; \mathit{OFFSET} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset}$$
$$\forall \mathit{Body} \bullet \mathcal{S}_B \; \theta \mathit{Body} = \mathcal{S}_S \; \gamma$$

Then calculating the offset for the procedure declaration itself just calls the body semantics with the current offset of that procedure as its offset. This is used as the start offset for those procedures and functions called in the body.

$$\mathcal{S}_{PD} : \mathit{ProcDecl} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset}$$
$$\forall \mathit{ProcDecl};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_{PD} \; \theta \mathit{ProcDecl} \; \rho st = \mathcal{S}_B \; \beta(\rho st \; \xi) \rho st$$

3.5.2 Functions

A function is checked by applying the body semantics in the same way as for procedures.

$$\mathcal{S}_{FD} : \mathit{FunDecl} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset}$$
$$\forall \mathit{FunDecl};\ \rho st : \mathit{blockOffset} \bullet$$
$$\quad \mathcal{S}_{FD} \; \theta \mathit{FunDecl} \; \rho st = \mathcal{S}_B \; \beta(\rho st \; \xi) \rho st$$

3.5.3 Multiple declarations

The semantics for procedure and function declarations are defined in the usual way.

$$\mathcal{S}_{PFD} : \mathit{PROC\_FUN\_DECL} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrow\!\!\!\!\!\!+ \; \mathit{blockOffset}$$
$$\forall \delta p : \mathit{ProcDecl} \bullet \mathcal{S}_{PFD}(\mathit{procDecl} \; \delta p) = \mathcal{S}_{PD} \; \delta p$$
$$\forall \delta f : \mathit{FunDecl} \bullet \mathcal{S}_{PFD}(\mathit{funDecl} \; \delta f) = \mathcal{S}_{FD} \; \delta f$$

For sequences of procedure and function declarations, each block is checked in *reverse order of declaration.*

$$
\begin{array}{|l}
\mathcal{S}_{PFD^*} : \operatorname{seq} PROC\_FUN\_DECL \rightarrowtail blockOffset \rightarrowtail blockOffset \\
\hline
\mathcal{S}_{PFD^*} \langle\ \rangle = \operatorname{id} blockOffset \\
\forall\, \Delta : \operatorname{seq} PROC\_FUN\_DECL;\ \delta : PROC\_FUN\_DECL \bullet \\
\qquad \mathcal{S}_{PFD^*}(\Delta \frown \langle \delta \rangle) = \mathcal{S}_{PFD^*}\ \Delta \circ \mathcal{S}_{PFD}\ \delta
\end{array}
$$

3.6 Import and Export Declarations

Variable import declarations are semantically similar to variable declarations, which have no effect on the *blockOffset*.

Procedure and function import declarations are semantically similar to procedures and functions with *Unknown* bodies. Calculating a stack set-up semantics, this shows there is no change to the *blockOffset*.

For the exports, we are only declaring the names and so these can be ignored also.

3.7 Standard Module

The stack set-up semantics of a standard module are simple: there is no main block, so the procedure and function declarations are checked in the initial environment where all names in scope have zero offset.

$$
\begin{array}{|l}
\mathcal{S}_M : Module \rightarrowtail blockOffset \\
\hline
\forall\, Module \bullet \\
\qquad \exists\, B : \mathbb{P}\, ID \mid B = \operatorname{dom}(\mathcal{T}_M\ \theta Module).tc \bullet \\
\qquad\qquad \mathcal{S}_M\ \theta Module = \mathcal{S}_{PFD^*}\ \Delta PF(B \times \{0\})
\end{array}
$$

3.8 Main Module

The stack set-up semantics of a main module are simple: the offset for the main block is calculated with a zero current offset, in the initial environment where all names in scope have zero offset. Then the resulting block offset is used in the calculation of the offset for the procedure and function declarations.

$$\mathcal{S}_{MA} : \mathit{MainModule} \pfun \mathit{blockOffset}$$

$$\begin{array}{l}
\forall\, \mathit{MainModule} \bullet \\
\quad \exists\, B : \mathbb{P}\, \mathit{ID} \mid B = \mathrm{dom}(\mathcal{T}_{MA}\ \theta \mathit{MainModule}).\mathit{tc} \bullet \\
\quad\quad \mathcal{S}_{MA}\ \theta \mathit{MainModule} = (\mathcal{S}_{PFD^*}\ \Delta PF \circ \mathcal{S}_S\ \gamma\ 0)(B \times \{0\})
\end{array}$$

4 Stack amount set-up semantics

This semantics calculates how much expression evaluation stack is needed (which depends on how deeply nested expression evaluation is).

4.1 Operators

The binary modulus operators use two extra bytes of expression evaluation stack space to perform their processing. The other binary operators require no additional stack space.

$$\mathcal{A}_{OP} : \mathit{BIN\_OP} \rightarrow\!\!\!\!+ \mathit{OFFSET}$$
$$\begin{array}{l} \forall\, \Omega : \{\mathit{bmod}, \mathit{umod}\} \bullet \mathcal{A}_{OP}\ \Omega = 2 \\ \forall\, \Omega : \mathit{BIN\_OP} \setminus \{\mathit{bmod}, \mathit{umod}\} \bullet \mathcal{A}_{OP}\ \Omega = 0 \end{array}$$

4.2 Expressions

The stack set-up semantics updates the maximum amount of stack space used. The value is incremented if the operational semantics that translate the expression uses space on the expression stack.

4.2.1 Actual parameters

These are checked as expressions.

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

To check a list of actual parameters, the stack semantics are applied to each element in turn, and the results combined pessimistically.

$$\mathcal{A}_{AP^*} : \operatorname{seq} \mathit{EXPR} \rightarrow\!\!\!\!+ \mathit{blockOffset} \rightarrow\!\!\!\!+ \mathit{OFFSET}$$
$$\begin{array}{l} \forall\, \rho sa : \mathit{blockOffset} \bullet \mathcal{A}_{AP^*} \langle\ \rangle \rho sa = 0 \\ \forall\, E : \operatorname{seq} \mathit{EXPR};\ \epsilon : \mathit{EXPR};\ \rho sa : \mathit{blockOffset} \bullet \\ \qquad \mathcal{A}_{AP^*}(\langle \epsilon \rangle \frown E)\rho sa = \mathit{max}\{\mathcal{A}_{AP^*}\ E\ \rho sa, \mathcal{A}_{AP}\ \epsilon\ \rho sa\} \end{array}$$

4.2.2 Constant

A constant has no effect on the stack offset amount.

$$\begin{array}{|l}\mathcal{A}_E : \mathit{EXPR} \nrightarrow \mathit{blockOffset} \nrightarrow \mathit{OFFSET} \\ \hline \forall \kappa : \mathit{VALUE};\ \rho sa : \mathit{blockOffset} \bullet \mathcal{A}_E(\mathit{constant}\ \kappa)\rho sa = 0\end{array}$$

4.2.3 Value reference

A value reference use up no more stack than the expression list.

$$\begin{array}{|l}\hline \forall \mathit{ValueRef} \bullet \mathcal{A}_E(\mathit{valueRef}\ \theta \mathit{ValueRef}) = \mathcal{A}_{E^*}\ E\end{array}$$

4.2.4 Unary expression

A unary expression uses up no more stack than its sub-expression.

$$\begin{array}{|l}\hline \forall \mathit{UnyExpr} \bullet \mathcal{A}_E(\mathit{unyExpr}\ \theta \mathit{UnyExpr}) = \mathcal{A}_E\ \epsilon\end{array}$$

4.2.5 Binary expression

Binary expression evaluation first evaluates the left argument, then stores it on the stack and evaluates the right argument, then the operator combines the arguments. So the stack space needed is the maximum of: that required for the left argument; that for the right argument plus two bytes (as the left argument is stored on the stack at this time); and that for the operator plus two bytes (as the left argument is still stored on the stack at this time).

$$\begin{array}{|l}\hline \forall \mathit{BinExpr};\ \rho sa : \mathit{blockOffset} \bullet \\ \qquad \mathcal{A}_E(\mathit{binExpr}\ \theta \mathit{BinExpr})\rho sa = \\ \qquad\qquad \mathit{max}\{\mathcal{A}_E\ \epsilon\ \rho sa, 2 + \mathcal{A}_E\ \epsilon'\ \rho sa, 2 + \mathcal{A}_{OP}\ \Omega\}\end{array}$$

4.2.6 Function call

The stack size required is obtained by extracting the value from the *blockOffset* for this block, and comparing it with the stack space required for the parameters.

$$\begin{array}{l} \forall\, \mathit{FunCallExpr};\ \rho sa : \mathit{blockOffset} \bullet \\ \quad \mathcal{A}_E(\mathit{funCall}\ \theta \mathit{FunCallExpr})\rho sa = \mathit{max}\{\rho sa\ \xi, \mathcal{A}_{AP^*}\ E\ \rho sa\} \end{array}$$

4.2.7 Multiple expressions

The operational semantics of expression lists (given by $\mathcal{O}_{E^*}$) use an additional 2 locations on the stack if and only if the expression list is non-empty. The operational semantics of variable locations (given by $\mathcal{O}_{VL}$) use no additional locations. Hence the stack amount semantics of multiple expressions can be defined as follows.

$$\begin{array}{l} \mathcal{A}_{E^*} : \mathrm{seq}\, \mathit{EXPR} \rightarrow\!\!\!\!\!\!+\;\; \mathit{blockOffset} \rightarrow\!\!\!\!\!\!+\;\; \mathit{OFFSET} \\ \hline \forall\, \rho sa : \mathit{blockOffset} \bullet \mathcal{A}_{E^*}\langle\ \rangle \rho sa = 0 \\ \forall\, \epsilon : \mathit{EXPR};\ \rho sa : \mathit{blockOffset} \bullet \mathcal{A}_{E^*}\langle \epsilon \rangle \rho sa = 2 + \mathcal{A}_E\ \epsilon\ \rho sa \\ \forall\, E, E' : \mathrm{seq}\, \mathit{EXPR};\ \rho sa : \mathit{blockOffset} \bullet \\ \quad \mathcal{A}_{E^*}(E \frown E')\rho sa = \mathit{max}\{\mathcal{A}_{E^*}\ E\ \rho sa, \mathcal{A}_{E^*}\ E'\ \rho sa\} \end{array}$$

4.3 Statements

The stack set-up semantics of statements update the block stack amount.

4.3.1 Multiple statements

The semantics is extended inductively to multiple statements.

$$\mathcal{A}_{S^*} : \operatorname{seq} STMT \rightarrow blockOffset \rightarrow OFFSET$$

$$\forall \rho sa : blockOffset \bullet \mathcal{A}_{S^*}\langle\ \rangle \rho sa = 0$$

$$\forall \gamma : STMT;\ \Gamma : \operatorname{seq} STMT;\ \rho sa : blockOffset \bullet$$
$$\quad \mathcal{A}_{S^*}(\langle \gamma \rangle \frown \Gamma)\rho sa = max\{\mathcal{A}_{S^*}\ \Gamma\ \rho sa, \mathcal{A}_S\ \gamma\ \rho sa\}$$

4.3.2 Block

Checking a block statement is the same as checking the sequence of body statements.

$$\mathcal{A}_S : STMT \rightarrow blockOffset \rightarrow OFFSET$$

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

4.3.3 Skip

A skip statement has no effect on the stack usage.

$$\forall \rho sa : blockOffset \bullet \mathcal{A}_S\ skip\ \rho sa = 0$$

4.3.4 Assignment

An assignment statement makes no extra use of the stack beyond that required for calculating the location and expression value.

$$\forall AssignStmt;\ \rho sa : blockOffset \bullet$$
$$\quad \mathcal{A}_S(assign\ \theta AssignStmt))\rho sa = max\{\mathcal{A}_{E^*}\ E\ \rho sa, \mathcal{A}_E\ \epsilon\ \rho sa\}$$

4.3.5 If statement

The offsets required for conditional expression and each branch statement are combined.

$$\forall IfStmt;\ \rho sa : blockOffset \bullet$$
$$\quad \mathcal{A}_S(ifStmt\ \theta IfStmt)\rho sa = max\{\mathcal{A}_E\ \epsilon\ \rho sa, \mathcal{A}_S\ \gamma\ \rho sa, \mathcal{A}_S\ \gamma'\ \rho sa\}$$

4.3.6 Case statement

We define the stack semantics for branches; it is the semantics of the executed statement.

$$\mathcal{A}_{CL} : Branch \pfun blockOffset \pfun OFFSET$$

$$\forall BranchSTMT \bullet \mathcal{A}_{CL}\ \theta Branch = \mathcal{A}_S\ \gamma$$

This is extended in the usual manner to sequences of branches.

$$\mathcal{A}_{CL^*} : \mathrm{seq}\, Branch \pfun blockOffset \pfun OFFSET$$

$$\begin{array}{l}\forall \rho sa : blockOffset \bullet \mathcal{A}_{CL^*}\langle\ \rangle \rho sa = 0 \\ \forall k : Branch;\ K : \mathrm{seq}\, Branch;\ \rho sa : blockOffset \bullet \\ \qquad \mathcal{A}_{CL^*}(K \frown \langle k \rangle)\rho sa = max\{\mathcal{A}_{CL^*}\ K\ \rho sa, \mathcal{A}_{CL}\ k\ \rho sa\}\end{array}$$

The results of checking the expression and the branches are combined pessimistically.

$$\begin{array}{l}\forall CaseStmt;\ \rho sa : blockOffset \bullet \\ \qquad \mathcal{A}_S(caseStmt\ \theta CaseStmt)\rho sa = max\{\mathcal{A}_E\ \epsilon\ \rho sa, \mathcal{A}_{CL^*}\ K\ \rho sa\}\end{array}$$

4.3.7 Loop

The results of checking the expression and loop body are combined pessimistically.

$$\begin{array}{l}\forall WhileStmt;\ \rho sa : blockOffset \bullet \\ \qquad \mathcal{A}_S(whileStmt\ \theta WhileStmt)\rho sa = max\{\mathcal{A}_E\ \epsilon\ \rho sa, \mathcal{A}_S\ \gamma\ \rho sa\}\end{array}$$

4.3.8 Procedure call

The checking of a procedure call is similar to that of a function call.

$$\begin{array}{l}\forall ProcCallStmt;\ \rho sa : blockOffset \bullet \\ \qquad \mathcal{A}_S(procCall\ \theta ProcCallStmt)\rho sa = max\{\rho sa\ \xi, \mathcal{A}_{AP^*}\ E\ \rho sa\}\end{array}$$

4.4 Declarations

The operational semantics of named constant declarations, type definitions and variable declarations have no effect on the stack usage; hence there is no stack semantics to define for them.

4.4.1 Procedures

The procedure body is processed as a statement.

$$\begin{array}{|l} \mathcal{A}_B : \mathit{Body} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{OFFSET} \\ \hline \forall\, \mathit{Body} \bullet \mathcal{A}_B\ \theta \mathit{Body} = \mathcal{A}_S\ \gamma \end{array}$$

Then checking the procedure declaration itself just calls the body semantics with the current offset of the procedure as its argument. The value returned is the amount of stack space used by the procedure body.

$$\begin{array}{|l} \mathcal{A}_{PD} : \mathit{ProcDecl} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{blockOffset} \\ \hline \forall\, \mathit{ProcDecl};\ \rho sa : \mathit{blockOffset} \bullet \\ \qquad \mathcal{A}_{PD}\ \theta \mathit{ProcDecl}\ \rho sa = \rho sa \cup \{\xi \mapsto \mathcal{A}_B\ \beta\ \rho sa\} \end{array}$$

4.4.2 Functions

A function is checked by applying the body semantics in the same way as for procedures.

$$\begin{array}{|l} \mathcal{A}_{FD} : \mathit{FunDecl} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{blockOffset} \rightarrowtail\!\!\!\!\!\!+ \; \mathit{blockOffset} \\ \hline \forall\, \mathit{FunDecl};\ \rho sa : \mathit{blockOffset} \bullet \\ \qquad \mathcal{A}_{FD}\ \theta \mathit{FunDecl}\ \rho sa = \rho sa \cup \{\xi \mapsto \mathcal{A}_B\ \beta\ \rho sa\} \end{array}$$

4.4.3 Multiple declarations

There is clearly nothing to define for simple declarations. The semantics for procedure and function declarations are defined in the usual way.

$$
\begin{array}{|l}
\mathcal{A}_{PFD} : \mathit{PROC\_FUN\_DECL} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\forall \delta p : \mathit{ProcDecl} \bullet \mathcal{A}_{PFD}(\mathit{procDecl}\ \delta p) = \mathcal{A}_{PD}\ \delta p \\
\forall \delta f : \mathit{FunDecl} \bullet \mathcal{A}_{PFD}(\mathit{funDecl}\ \delta f) = \mathcal{A}_{FD}\ \delta f
\end{array}
$$

Each block is checked in *order of declaration*. (Compare the stack semantics $\mathcal{S}$, which is calculated in *reverse order*.)

$$
\begin{array}{|l}
\mathcal{A}_{PFD^*} : \operatorname{seq} \mathit{PROC\_FUN\_DECL} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\mathcal{A}_{PFD^*}\langle\ \rangle = \operatorname{id} \mathit{blockOffset} \\
\forall \Delta : \operatorname{seq} \mathit{PROC\_FUN\_DECL};\ \delta : \mathit{PROC\_FUN\_DECL} \bullet \\
\qquad \mathcal{A}_{PFD^*}(\langle \delta \rangle \frown \Delta) = \mathcal{A}_{PFD^*}\ \Delta \circ \mathcal{A}_{PFD}\ \delta
\end{array}
$$

4.5 Import and Export Declarations

The import declarations require definition in the *blockOffset* so that any calls to imported functions and procedures can be properly updated; nothing is required for imported variables. Each imported procedure or function declaration identifier is mapped to 0 stack amount.

$$
\begin{array}{|l}
\mathcal{A}_{IM} : \mathit{IMPORT\_DECL} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\forall \mathit{ProcHdr};\ \rho sa : \mathit{blockOffset} \bullet \\
\qquad \mathcal{A}_{IM}(\mathit{procHdr}\ \theta \mathit{ProcHdr})\rho sa = \rho sa \cup \{\xi \mapsto 0\} \\
\forall \mathit{FunHdr};\ \rho sa : \mathit{blockOffset} \bullet \\
\qquad \mathcal{A}_{IM}(\mathit{funcHdr}\ \theta \mathit{FunHdr})\rho sa = \rho sa \cup \{\xi \mapsto 0\}
\end{array}
$$

And by usual extension to sequences.

$$
\begin{array}{|l}
\mathcal{A}_{IM^*} : \operatorname{seq} \mathit{IMPORT\_DECL} \pfun \mathit{blockOffset} \pfun \mathit{blockOffset} \\
\hline
\mathcal{A}_{IM^*}\langle\ \rangle = \operatorname{id} \mathit{blockOffset} \\
\forall \Delta I : \operatorname{seq} \mathit{IMPORT\_DECL};\ \delta i : \mathit{IMPORT\_DECL} \bullet \\
\qquad \mathcal{A}_{IM^*}(\langle \delta i \rangle \frown \Delta I) = \mathcal{A}_{IM^*}\ \Delta I \circ \mathcal{A}_{IM}\ \delta i
\end{array}
$$

For the exports, we are only declaring the names to be exported and therefore this requires no stack usage, and can be ignored.

4.6 Standard Module

The stack set-up semantics of a standard module are simple: there is no main block, so the procedure and function declarations are checked in the environment generated from applying the import declarations to the initial environment. The export declarations cause no change to the *blockOffset*, and so are ignored.

$$
\begin{array}{|l}
\mathcal{A}_M : \mathit{Module} \mathrel{+\!\!\!\!\to} \mathit{blockOffset} \\
\hline
\forall\, \mathit{Module} \bullet \\
\quad \exists\, B : \mathbb{P}\, \mathit{ID} \mid B = \mathrm{dom}(\mathcal{T}_M \theta \mathit{Module}).\mathit{tc} \bullet \\
\qquad \mathcal{A}_M\ \theta \mathit{Module} = (\mathcal{A}_{PFD^*}\ \Delta PF \circ \mathcal{A}_{IM^*}\ \Delta I)(B \times \{0\})
\end{array}
$$

4.7 Main Module

The stack amount semantics of a main module are simple: the procedure and function declarations are checked in order, then the main statement is checked. As discussed before, export declarations cause no change to the *blockOffset*, and so are ignored. However the import declarations have to be set up in the semantics so they cause no extra use in the amount of stack used.

$$
\begin{array}{|l}
\mathcal{A}_{MA} : \mathit{MainModule} \mathrel{+\!\!\!\!\to} \mathit{blockOffset} \\
\hline
\forall\, \mathit{MainModule} \bullet \\
\quad \exists\, B : \mathbb{P}\, \mathit{ID};\ \rho sa : \mathit{blockOffset} \mid \\
\qquad\quad B = \mathrm{dom}(\mathcal{T}_{MA} \theta \mathit{MainModule}).\mathit{tc} \\
\qquad\quad \wedge\ \rho sa = (\mathcal{A}_{PFD^*}\ \Delta PF \circ \mathcal{A}_{IM^*}\ \Delta I)(B \times \{0\}) \bullet \\
\qquad \mathcal{A}_{MA}\ \theta \mathit{MainModule} = \rho sa \cup \{\xi \mapsto \mathcal{A}_S\ \gamma\ \rho sa\}
\end{array}
$$

5 Operators

The meaning function for operators yields a sequence of instructions that translate the operator on the assumption that one argument is contained in the a and b registers (with the most significant byte in the a register), and that the second argument, if any, is contained on the top of the expression evaluation stack, which is referenced by the data register. We also assume that boolean and byte values are contained in the b register as this is where they will be loaded when a value reference is executed.

The values of the carry bits are undefined before and after the fragment.

5.1 Unary Operators, single byte arguments

The single byte argument is in the b register. The single byte result is left in the b register.

$$\begin{array}{|l}
\mathcal{O}_{UO} : \mathit{UNY\_OP} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\qquad \mathcal{O}_{UO}\ \mathit{not}\ l = (l, \langle \mathit{immediate}(\mathtt{bxoi}, 1)\rangle) \\
\qquad \wedge\ \mathcal{O}_{UO}\ \mathit{bnot}\ l = (l, \langle \mathit{immediate}(\mathtt{bxoi}, \mathit{MaxByte})\rangle)
\end{array}$$

- *not*: boolean not. $b := b\ xor\ 1$
- *bnot*: byte not. $b := b\ xor\ FF$

$$\begin{array}{|l}
\mathcal{O}_{UO} : \mathit{UNY\_OP} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\qquad \mathcal{O}_{UO}\ \mathit{bleft}\ l = (l, \langle \mathit{indirect}\ \mathtt{ccb}\rangle \frown \langle \mathit{indirect}\ \mathtt{lrb}\rangle) \\
\qquad \wedge\ \mathcal{O}_{UO}\ \mathit{bright}\ l = (l, \langle \mathit{indirect}\ \mathtt{ccb}\rangle \frown \langle \mathit{indirect}\ \mathtt{rrb}\rangle)
\end{array}$$

- *bleft*: byte shift left, multiply by 2.
- *bright*: byte shift right, divide by 2.

$$\mathcal{O}_{UO} : UNY\_OP \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{aligned}
&\forall\, l : LABEL \bullet \\
&\quad \mathcal{O}_{UO}\ \mathit{byteToBool}\ l = (l, \\
&\qquad \langle \mathit{immediate}(\mathtt{beqi}, 0)\rangle \frown \langle \mathit{indirect}\ \mathtt{ncb}\rangle \\
&\qquad \frown \langle \mathit{immediate}(\mathtt{bldi}, 0)\rangle \frown \langle \mathit{indirect}\ \mathtt{lrb}\rangle) \\
&\quad \land \mathcal{O}_{UO}\ \mathit{boolToByte}\ l = (l, \langle\ \rangle) \\
&\quad \land \mathcal{O}_{UO}\ \mathit{unsgnToByte}\ l = (l, \langle\ \rangle)
\end{aligned}$$

- *byteToBool*: cast byte to boolean. Test if the byte is zero, leaving the result in the b-carry; negate it, then rotate it into cleared b register.
- *boolToByte*: cast boolean to byte. Do nothing; the representation of a boolean is a byte.
- *unsgnToByte*: cast unsigned to byte. Do nothing; this has the effect of truncating the hi byte, because the result is assumed to be a byte, and hence in the b register only.

$$\mathcal{O}_{UO} : UNY\_OP \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{aligned}
&\forall\, l : LABEL \bullet \\
&\quad \mathcal{O}_{UO}\ \mathit{ord}\ l = (l, \langle\ \rangle) \\
&\quad \land \mathcal{O}_{UO}\ \mathit{succ}\ l = (l, \langle \mathit{indirect}\ \mathtt{ccb}\rangle \frown \langle \mathit{immediate}(\mathtt{buai}, 1)\rangle) \\
&\quad \land \mathcal{O}_{UO}\ \mathit{pred}\ l = (l, \langle \mathit{indirect}\ \mathtt{ccb}\rangle \frown \langle \mathit{immediate}(\mathtt{busi}, 1)\rangle)
\end{aligned}$$

- *ord*: cast enumerate type to byte. Do nothing; the representation of an enum is a byte.
- *succ*: increment enum. $b_c := \mathit{clear};\ b := b + 1(+b_c)$
  The Pasp semantics does not define the result of incrementing *venum bmax*; the result here is $\mathit{bmax} + 1$, or zero if $\mathit{bmax} = 255$.
- *pred*: decrement enum. $b_c := \mathit{clear};\ b := b - 1(-b_c)$
  The Pasp semantics does not define the result of decrementing *venum* 0; the result here is 255.

5.2 Unary Operators, two byte arguments

The two byte argument is in the a and b register, with the hi byte in a and the lo byte in b. The two byte result is left in the a and b register.

$$
\begin{array}{|l}
\mathcal{O}_{UO} : \mathit{UNY\_OP} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{UO}\ \mathit{unot}\ l = (l, \\
\qquad \langle \mathit{immediate}(\mathtt{axoi}, \mathit{MaxByte}) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{bxoi}, \mathit{MaxByte}) \rangle)
\end{array}
$$

- *unot*: unsigned not. $a := a\ xor\ FF;\ b := b\ xor\ FF$

$$
\begin{array}{|l}
\mathcal{O}_{UO} : \mathit{UNY\_OP} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{UO}\ \mathit{uleft}\ l = (l + 1, \\
\qquad \langle \mathit{indirect}\ \mathtt{ccb} \rangle \frown \langle \mathit{indirect}\ \mathtt{lrb} \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{sca} \rangle \frown \langle \mathit{xLabel}(\mathit{xrjb}, l) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{cca} \rangle \frown \langle \mathit{xLabel}(\mathit{xli}, l) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{lra} \rangle) \\
\quad \wedge\ \mathcal{O}_{UO}\ \mathit{uright}\ l = (l + 1, \\
\qquad \langle \mathit{indirect}\ \mathtt{cca} \rangle \frown \langle \mathit{indirect}\ \mathtt{rra} \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{scb} \rangle \frown \langle \mathit{xLabel}(\mathit{xrja}, l) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{ccb} \rangle \frown \langle \mathit{xLabel}(\mathit{xli}, l) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{rrb} \rangle)
\end{array}
$$

- *uleft*: unsigned shift left.
- *uright*: unsigned shift right.

$$
\begin{array}{|l}
\mathcal{O}_{UO} : \mathit{UNY\_OP} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{UO}\ \mathit{byteToUnsgn}\ l = (l, \langle \mathit{immediate}(\mathtt{aldi}, 0) \rangle) \\
\quad \wedge\ \mathcal{O}_{UO}\ \mathit{hiByte}\ l = (l, \langle \mathit{indirect}\ \mathtt{blda} \rangle) \\
\quad \wedge\ \mathcal{O}_{UO}\ \mathit{loByte}\ l = (l, \langle\ \rangle)
\end{array}
$$

- *byteToUnsgn*: cast byte to unsigned. $a := 0$
  This clears any garbage out of the a register, effectively setting the hi byte to zero.
- *hiByte*: cast unsigned to value of its high byte
  divide by 256, or, move a into b
- *loByte*: cast unsigned to value of its low byte
  do nothing

5.3 Binary Operators – overview

The meaning function for binary operators is a map from an operator, a data address and a label to label and a sequence of instructions.

The primitive binary operator instructions provided by the Asp chip are designed to work with the left hand argument in the a, b registers, and the right hand argument either an immediate value, or in the memory pointed to by the data address register. But the Asp compiler translates binary expressions from left to right, and so first evaluates the left hand argument, storing the result in memory on the expression evaluation stack, then evaluates the right hand argument, leaving the result in the a, b registers, then applies the binary operator. This order difference causes a complication to some of the template algorithms, and should be noted.

The data address is the address of the first operand, on the expression evaluation stack. For two byte values, d points to the lo byte at address δ, and the hi byte is at address $\delta + 1$ (also remember the expression stack grows 'downwards in memory, so the 'top of the stack is at the lowest memory address.) If the contents of the data register are altered to store partial results, they must be restored to complete the calculation.

The labels are required as some of the larger arithmetic templates require relative jumps.

5.4 Binary Logical Operators

$$\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{array}{l}
\forall \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ and\ \delta\ l = (l, \langle indirect\ \texttt{band} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ or\ \delta\ l = (l, \langle indirect\ \texttt{bord} \rangle) \\
\forall \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ band\ \delta\ l = (l, \langle indirect\ \texttt{band} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ bor\ \delta\ l = (l, \langle indirect\ \texttt{bord} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ bxor\ \delta\ l = (l, \langle indirect\ \texttt{bxod} \rangle) \\
\forall \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ uand\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{band} \rangle \frown \langle immediate(\texttt{dpi}, 1) \rangle \frown \langle indirect\ \texttt{aand} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ uor\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{bord} \rangle \frown \langle immediate(\texttt{dpi}, 1) \rangle \frown \langle indirect\ \texttt{aord} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ uxor\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{bxod} \rangle \frown \langle immediate(\texttt{dpi}, 1) \rangle \frown \langle indirect\ \texttt{axod} \rangle)
\end{array}$$

- *and*: boolean and. $b := b \wedge mem\ D$
- *or*: boolean or. $b := b \vee mem\ D$
- *band*: byte and. $b := b \wedge mem\ D$
- *bor*: byte or. $b := b \vee mem\ D$
- *boxr*: byte xor. $b := b\ xor\ mem\ D$
- *uand*: unsigned and. $b := b \wedge mem\ D;\ a := a \wedge mem(D+1)$
- *uor*: unsigned or: $b := b \vee mem\ D;\ a := a \vee mem(D+1)$
- *uoxr*: unsigned xor: $b := b\ xor\ mem\ D;\ a := a\ xor\ mem(D+1)$

5.5 Binary Unsigned Comparison Operators

In the case of two byte comparison operators such as *ueq*, we want to combine the results of two tests, one on the lo byte, one on the hi byte, and leave the boolean comparison result in the b register. There are two ways to do this:

1. Rotate the test result into the corresponding register, store this and then use the corresponding logical instruction,
2. Build an 'if then else' construction.

The latter technique, which is the one we use, requires fewer instructions in general, and also makes the function definition simpler, as there is no need to use a temporary address as an argument; on the other hand, it is likely to be harder to reason about.

Note that although we are using an 'if then else' construction, we cannot include the test of the lower byte in the then clause (where it logically belongs) because we cannot decrement the data address register.

5.5.1 Unsigned equality

This template computes the result of the test $(\epsilon\ hi, \epsilon\ lo) = (a, b)$

```
$\mathbf{if}(b = \epsilon\ lo)\ \mathbf{then}\ res := false;\ jump\ end$
$res := (a = \epsilon\ hi)$
$label\ end$
```

$$\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{aligned}
&\forall \delta : OFFSET;\ l : LABEL \bullet \\
&\quad \mathcal{O}_{BO}\ ueq\ \delta\ l = (l + 1, \\
&\qquad \langle indirect\ \texttt{beqd} \rangle \\
&\qquad ^\frown \langle immediate(\texttt{bldi}, 0) \rangle ^\frown \langle indirect\ \texttt{ncb} \rangle \\
&\qquad ^\frown \langle xLabel(xrjb, l) \rangle \\
&\qquad ^\frown \langle immediate(\texttt{dpi}, 1) \rangle ^\frown \langle indirect\ \texttt{aeqd} \rangle \\
&\qquad ^\frown \langle indirect\ \texttt{cbldca} \rangle ^\frown \langle indirect\ \texttt{lrb} \rangle \\
&\qquad ^\frown \langle xLabel(xli, l) \rangle)
\end{aligned}$$

5.5.2 Unsigned greater than

This template computes the result of the test $(\epsilon\ hi, \epsilon\ lo) > (a, b)$

The algorithm is a little obscure, because we compare the lo bytes before incrementing the address register to compare the hi bytes.

```
**if**$(b < \epsilon\ lo)$ **then** $jump\ l$
**if**$(a = \epsilon\ hi)$ **then** $res := false;\ jump\ end$
$label\ l$
**if**$(a > \epsilon\ hi)$ **then** $res := false;\ jump\ end$
$res := true$
$label\ end$
```

$$\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \mathrm{seq}\, X\_INSTR$$

$$\begin{aligned}
&\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
&\quad \mathcal{O}_{BO}\ ugt\ \delta\ l = (l + 2, \\
&\qquad \langle indirect\ \texttt{scb}\rangle \frown \langle indirect\ \texttt{bucd}\rangle \\
&\qquad \frown \langle immediate(\texttt{bldi}, 0)\rangle \\
&\qquad \frown \langle immediate(\texttt{dpi}, 1)\rangle \frown \langle indirect\ \texttt{ncb}\rangle \\
&\qquad \frown \langle xLabel(xrjb, l)\rangle \\
&\qquad \frown \langle indirect\ \texttt{aeqd}\rangle \\
&\qquad \frown \langle xLabel(xrja, l + 1)\rangle \\
&\qquad \frown \langle xLabel(xli, l)\rangle \\
&\qquad \frown \langle indirect\ \texttt{cca}\rangle \frown \langle indirect\ \texttt{aucd}\rangle \\
&\qquad \frown \langle xLabel(xrja, l + 1)\rangle \\
&\qquad \frown \langle immediate(\texttt{bldi}, 1)\rangle \\
&\qquad \frown \langle xLabel(xli, l + 1)\rangle)
\end{aligned}$$

5.5.3 Unsigned greater than or equal to

This template computes the result of the test $(\epsilon\ hi, \epsilon\ lo) \geq (a, b)$

$$\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{array}{l}
\forall \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ uge\ \delta\ l = (l + 2, \\
\qquad \langle indirect\ \texttt{ccb}\rangle \frown \langle indirect\ \texttt{bucd}\rangle \\
\qquad \frown \langle immediate(\texttt{bldi}, 0)\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 1)\rangle \frown \langle indirect\ \texttt{ncb}\rangle \\
\qquad \frown \langle xLabel(xrjb, l)\rangle \\
\qquad \frown \langle indirect\ \texttt{aeqd}\rangle \\
\qquad \frown \langle xLabel(xrja, l + 1)\rangle \\
\qquad \frown \langle xLabel(xli, l)\rangle \\
\qquad \frown \langle indirect\ \texttt{cca}\rangle \frown \langle indirect\ \texttt{aucd}\rangle \\
\qquad \frown \langle xLabel(xrja, l + 1)\rangle \\
\qquad \frown \langle immediate(\texttt{bldi}, 1)\rangle \\
\qquad \frown \langle xLabel(xli, l + 1)\rangle)
\end{array}$$

5.5.4 Other Binary Unsigned Comparison Operators

It is worth reiterating a point made in the introduction to this document; wherever we can make a proof more easy or clear by sacrificing a bit of efficiency, we are likely to do so. It is clear that more efficient implementations of the *une* operator are possible. However, the one we have chosen will be quicker to prove correct than repeating a similar proof to that required for *ueq*.

$$
\begin{array}{|l}
\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR \\
\hline
\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
\quad \exists\, l', l'' : LABEL;\ I, I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ ueq\ \delta\ l \\
\qquad\qquad \land\ (l'', I') = \mathcal{O}_{UO}\ not\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ une\ \delta\ l = (l'', I \mathbin{^\frown} I') \\
\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
\quad \exists\, l', l'' : LABEL;\ I, I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ uge\ \delta\ l \\
\qquad\qquad \land\ (l'', I') = \mathcal{O}_{UO}\ not\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ ult\ \delta\ l = (l'', I \mathbin{^\frown} I') \\
\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
\quad \exists\, l', l'' : LABEL;\ I, I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ ugt\ \delta\ l \\
\qquad\qquad \land\ (l'', I') = \mathcal{O}_{UO}\ not\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ ule\ \delta\ l = (l'', I \mathbin{^\frown} I')
\end{array}
$$

5.6 Binary Byte and Enumeration Comparison Operators

Because they manipulate a single register, the comparison operators for byte values are simpler.

$$
\begin{array}{|l}
\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR \\
\hline
\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ beq\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{beqd}\rangle \mathbin{^\frown} \langle immediate(\texttt{bldi}, 0)\rangle \mathbin{^\frown} \langle indirect\ \texttt{lrb}\rangle) \\
\quad \land\ \mathcal{O}_{BO}\ blt\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{ccb}\rangle \mathbin{^\frown} \langle indirect\ \texttt{bucd}\rangle \\
\qquad\ \mathbin{^\frown} \langle immediate(\texttt{bldi}, 0)\rangle \mathbin{^\frown} \langle indirect\ \texttt{lrb}\rangle) \\
\quad \land\ \mathcal{O}_{BO}\ ble\ \delta\ l = (l, \\
\qquad \langle indirect\ \texttt{scb}\rangle \mathbin{^\frown} \langle indirect\ \texttt{bucd}\rangle \\
\qquad\ \mathbin{^\frown} \langle immediate(\texttt{bldi}, 0)\rangle \mathbin{^\frown} \langle indirect\ \texttt{lrb}\rangle)
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists l', l'' : \mathit{LABEL};\ I, I' : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ \mathit{beq}\ \delta\ l \\
\qquad\qquad \wedge\ (l'', I') = \mathcal{O}_{UO}\ \mathit{not}\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ \mathit{bne}\ \delta\ l = (l'', I \frown I') \\
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists l', l'' : \mathit{LABEL};\ I, I' : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ \mathit{ble}\ \delta\ l \\
\qquad\qquad \wedge\ (l'', I') = \mathcal{O}_{UO}\ \mathit{not}\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ \mathit{bgt}\ \delta\ l = (l'', I \frown I') \\
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists l', l'' : \mathit{LABEL};\ I, I' : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad (l', I) = \mathcal{O}_{BO}\ \mathit{blt}\ \delta\ l \\
\qquad\qquad \wedge\ (l'', I') = \mathcal{O}_{UO}\ \mathit{not}\ l' \bullet \\
\qquad \mathcal{O}_{BO}\ \mathit{bge}\ \delta\ l = (l'', I \frown I')
\end{array}
$$

The two comparison operators for enumerated values are the same as their byte equivalents.

$$
\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\mathcal{O}_{BO}\ \mathit{eeq} = \mathcal{O}_{BO}\ \mathit{beq} \\
\mathcal{O}_{BO}\ \mathit{ene} = \mathcal{O}_{BO}\ \mathit{bne}
\end{array}
$$

5.7 Binary Byte Arithmetic Operators

Some of the templates for arithmetic operators require data addresses in order to store partial results. These addresses are at and above *topOp*. We start with the byte arithmetic operators, which require comparatively simple templates.

$$\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{BO}\ \mathit{bplus}\ \delta\ l = (l, \\
\qquad \langle \mathit{indirect}\ \mathtt{ccb} \rangle \frown \langle \mathit{indirect}\ \mathtt{buad} \rangle \frown \langle \mathit{indirect}\ \mathtt{hltcb} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ \mathit{bminus}\ \delta\ l = (l, \\
\qquad \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \frown \langle \mathit{indirect}\ \mathtt{bstd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \langle \mathit{indirect}\ \mathtt{bldd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{ccb} \rangle \frown \langle \mathit{indirect}\ \mathtt{busd} \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{hltcb} \rangle)
\end{array}$$

$$\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{BO}\ \mathit{bmul}\ \delta\ l = (l, \\
\qquad \langle \mathit{immediate}(\mathtt{aldi}, 0) \rangle \frown \langle \mathit{indirect}\ \mathtt{aumd} \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{aeqi}, 0) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{nca} \rangle \frown \langle \mathit{indirect}\ \mathtt{hltca} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ \mathit{bdiv}\ \delta\ l = (l, \\
\qquad \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \frown \langle \mathit{indirect}\ \mathtt{bstd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \langle \mathit{indirect}\ \mathtt{bldd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{aldi}, 0) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{audd} \rangle)
\end{array}$$

Asp byte division yields the modulus in the a register; this is moved to the b register.

$$\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists\, \mathit{ldiv} : \mathit{LABEL};\ \mathit{Idiv} : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\quad (\mathit{ldiv}, \mathit{Idiv}) = \mathcal{O}_{BO}\ \mathit{bdiv}\ \delta\ l \bullet \\
\qquad \mathcal{O}_{BO}\ \mathit{bmod}\ \delta\ l = (\mathit{ldiv}, \mathit{Idiv} \frown \langle \mathit{indirect}\ \mathtt{blda} \rangle)
\end{array}$$

5.8 Binary Unsigned Addition and Subtraction Operators

The operands are the opposite way round to the way the Asp instructions work. Since *uplus* is commutative, this does not matter, however for *uminus* they have to be swapped.

$$\mathcal{O}_{BO} : \mathit{BIN\_OP} \to \mathit{OFFSET} \to \mathit{LABEL} \to \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR}$$

$$\begin{array}{l}
\forall\, \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{BO}\ \mathit{uplus}\ \delta\ l = (l, \\
\qquad \langle \mathit{indirect}\ \texttt{ccb} \rangle \frown \langle \mathit{indirect}\ \texttt{buad} \rangle \\
\qquad \frown \langle \mathit{indirect}\ \texttt{caldcb} \rangle \\
\qquad \frown \langle \mathit{immediate}(\texttt{dpi}, 1) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \texttt{auad} \rangle \frown \langle \mathit{indirect}\ \texttt{hltca} \rangle) \\
\quad \wedge\ \mathcal{O}_{BO}\ \mathit{uminus}\ \delta\ l = (l, \\
\qquad \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \frown \langle \mathit{indirect}\ \texttt{abstd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \langle \mathit{indirect}\ \texttt{abldd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \frown \langle \mathit{indirect}\ \texttt{ccb} \rangle \\
\qquad \frown \langle \mathit{indirect}\ \texttt{busd} \rangle \frown \langle \mathit{indirect}\ \texttt{caldcb} \rangle \\
\qquad \frown \langle \mathit{immediate}(\texttt{dpi}, 1) \rangle \\
\qquad \frown \langle \mathit{indirect}\ \texttt{ausd} \rangle \frown \langle \mathit{indirect}\ \texttt{hltca} \rangle)
\end{array}$$

These operators work from the low byte to the high byte.

The template for *uminus* can be optimised further when the instruction for copying C register to data addresses is provided. The C register can then be used as a temporary store during the swapping.

5.9 Binary Unsigned Multiplication Operator

The algorithm underlying the template for unsigned multiplication is based on conventional long multiplication. In the following algorithm only the lower two bytes of the answer are calculated, i.e. any overflow in the answer is quietly ignored.

A global optimisation has been incorporated into the system. This allows the *umul* and *udiv* operators to be moved to a common region, and called (similar to functions) when required. This allows a space saving for these operators, which increases with each Pasp call to the operators.

The templates for these operators are kept (but renamed eg $\mathcal{O}_{UMUL}$) for inclusion by the linker on a conditional linkage basis. The new templates for each of these operators consists of a sequence of XAspAL instructions. These store the arguments, return address (as a label), and handle the call to, and return from, the new function- style operators.

The linker will only include the template for one of these operators if a call actually exists in the compiled XAspAL instructions. To achieve this an XAspAL instruction *xCallOp* is used to identify calls to one of these operators. This instruction is trapped and handled at the link stage. A new function *binopfn* is used to map the operators to their unique label ids. This is valid as the operator templates will appear as the first module in the linked system, and so any subsequent modules will be labelled with higher labels.

$$\begin{array}{|l}
binopfn : BIN\_OP \rightarrowtail\!\!\!\!\rightarrow LABEL \\
\hline
binopfn\ umul = 0 \\
binopfn\ udiv = 1
\end{array}$$

The next label to be used is 10, as *udiv* operator requires 8 labels also.

$$firstAvailableLabel == 10$$

$$
\begin{array}{|l}
\mathcal{O}_{UMUL} : \operatorname{seq} X\_INSTR \\
\hline
\mathcal{O}_{UMUL} = \\
\quad \langle xLabel(xli, binopfn\ umul)\rangle \\
\quad {}^\frown \langle xData(xsdro, 16)\rangle {}^\frown \langle indirect\ \texttt{bldd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{aldi}, 0)\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 2)\rangle {}^\frown \langle indirect\ \texttt{aumd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, -16)\rangle {}^\frown \langle indirect\ \texttt{bstd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 15)\rangle {}^\frown \langle indirect\ \texttt{bldd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 1)\rangle {}^\frown \langle indirect\ \texttt{aumd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, -15)\rangle {}^\frown \langle indirect\ \texttt{bstd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 13)\rangle {}^\frown \langle indirect\ \texttt{bldd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{aldi}, 0)\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 3)\rangle {}^\frown \langle indirect\ \texttt{aumd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, -16)\rangle {}^\frown \langle indirect\ \texttt{ccb}\rangle \\
\quad {}^\frown \langle indirect\ \texttt{buad}\rangle {}^\frown \langle indirect\ \texttt{bstd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, -1)\rangle {}^\frown \langle indirect\ \texttt{cldd}\rangle \\
\quad {}^\frown \langle immediate(\texttt{dpi}, 18)\rangle \\
\quad {}^\frown \langle indirect\ \texttt{abldd}\rangle \\
\quad {}^\frown \langle xIndirect\ xag\rangle
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR \\
\hline
\forall\, \delta : OFFSET;\ l : LABEL \bullet \\
\quad \mathcal{O}_{BO}\ umul\ \delta\ l = (l+1, \\
\qquad \langle indirect\ \texttt{cldab}\rangle \\
\qquad {}^\frown \langle xData(xsdrs, \delta)\rangle {}^\frown \langle indirect\ \texttt{abldd}\rangle \\
\qquad {}^\frown \langle xData(xsdro, 18)\rangle {}^\frown \langle indirect\ \texttt{abstd}\rangle \\
\qquad {}^\frown \langle immediate(\texttt{dpi}, -2)\rangle {}^\frown \langle indirect\ \texttt{cstd}\rangle \\
\qquad {}^\frown \langle xLabel(xll, l)\rangle \\
\qquad {}^\frown \langle immediate(\texttt{dpi}, 4)\rangle {}^\frown \langle indirect\ \texttt{abstd}\rangle \\
\qquad {}^\frown \langle xCallOp(binopfn\ umul)\rangle \\
\qquad {}^\frown \langle xLabel(xli, l)\rangle \\
\qquad {}^\frown \langle indirect\ \texttt{abldc}\rangle)
\end{array}
$$

The template $\mathcal{O}_{UMUL}$ is now standalone. Its arguments and return labels are stored in the operator stack, from $topOp + 16$ to $topOp + 21$. The left hand argument u, from the memory, is stored in 16,17; the right hand operator v, from the a, b registers, is stored in 18,19; the return label is stored in 20,21. The dependence upon a stack address now no longer exists. This, and the introduction of the new Asp5 instructions allow the templates to be optimised quite substantially.

$topOp + 2$ and $topOp + 3$ are used to store the result of the multiplication. The Asp multiplication instruction calculates the value of $x * b + a$, where x is the data field. The 16-bit answer is stored with the most significant byte in the a register.

This template contains 33 instructions, however a number of these are due to the mechanism used to return from the operator.

First the value of $v_2 * u_2$ is calculated, and the low byte stored. The high byte is retained in the a register, and is added to $v_2 * u_1$. The low byte of the result is stored at $topOp + 3$. Finally, $v_1 * u_2$ is calculated and added to the contents of $topOp + 3$. There is no need to calculate $u_1 * v_1$.

5.10 Binary Unsigned Division Operator

The algorithm for unsigned division that is represented by the following template is based on ideas in volume 2 of ‘The Art of Computer Programming’ by D. E. Knuth.

The arguments and return labels are stored in the operator stack. The left hand argument u, from the memory, is stored in 0,1; the right hand operator v, from the a, b registers, is stored in 3,4; the return label is stored in 7,8.

$$
\begin{array}{|l}
opudivstart : LABEL \rightarrow \operatorname{seq} X\_INSTR \\
\hline
\forall l : LABEL \bullet \\
\quad opudivstart\ l = \\
\quad\quad \langle xLabel(xli, l) \rangle \\
\quad\quad {}^\frown \langle xData(xsdro, 4) \rangle {}^\frown \langle indirect\ \texttt{aldd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{aeqi}, 0) \rangle {}^\frown \langle indirect\ \texttt{nca} \rangle \\
\quad\quad {}^\frown \langle xLabel(xrja, l + 1) \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, -3) \rangle {}^\frown \langle indirect\ \texttt{bldd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{aldi}, 0) \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, 2) \rangle {}^\frown \langle indirect\ \texttt{audd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, 3) \rangle {}^\frown \langle indirect\ \texttt{bstd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, -6) \rangle {}^\frown \langle indirect\ \texttt{bldd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, 3) \rangle {}^\frown \langle indirect\ \texttt{audd} \rangle \\
\quad\quad {}^\frown \langle immediate(\texttt{dpi}, 3) \rangle {}^\frown \langle indirect\ \texttt{aldd} \rangle \\
\quad\quad {}^\frown \langle xLabel(xrgs, l + 8) \rangle
\end{array}
$$

$$
\begin{array}{|l}
opudivmain : LABEL \rightarrow \operatorname{seq} X\_INSTR \\
\hline
\forall\, l : LABEL \bullet \\
\quad opudivmain\ l = \\
\qquad opudivstart\ l\ \frown \langle xLabel(xli, l+1)\rangle \\
\qquad \frown \langle immediate(\texttt{auci}, 128)\rangle \\
\qquad \frown \langle xLabel(xrja, l+2)\rangle \\
\qquad \frown \langle immediate(\texttt{auai}, 1)\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 2)\rangle \frown \langle indirect\ \texttt{astd}\rangle \\
\qquad \frown \langle immediate(\texttt{aldi}, 1)\rangle \\
\qquad \frown \langle immediate(\texttt{bldi}, 0)\rangle \\
\qquad \frown \langle indirect\ \texttt{audd}\rangle \frown \langle indirect\ \texttt{bstd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, -6)\rangle \frown \langle indirect\ \texttt{bldd}\rangle \\
\qquad \frown \langle immediate(\texttt{aldi}, 0)\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 6)\rangle \frown \langle indirect\ \texttt{aumd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, -6)\rangle \frown \langle indirect\ \texttt{bstd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 1)\rangle \frown \langle indirect\ \texttt{bldd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 5)\rangle \frown \langle indirect\ \texttt{aumd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, -5)\rangle \frown \langle indirect\ \texttt{abstd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 2)\rangle \frown \langle indirect\ \texttt{bldd}\rangle \\
\qquad \frown \langle immediate(\texttt{aldi}, 0)\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 3)\rangle \frown \langle indirect\ \texttt{aumd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, -3)\rangle \frown \langle indirect\ \texttt{bstd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 1)\rangle \frown \langle indirect\ \texttt{bldd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, 2)\rangle \frown \langle indirect\ \texttt{aumd}\rangle \\
\qquad \frown \langle immediate(\texttt{dpi}, -2)\rangle \frown \langle indirect\ \texttt{bstd}\rangle \\
\qquad \frown \langle xLabel(xrgs, l+3)\rangle
\end{array}
$$

$$
\begin{array}{|l}
\mathit{opudivzero} : \mathit{LABEL} \rightarrow \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathit{opudivzero}\ l = \\
\qquad \mathit{opudivmain}\ l\ \frown \langle \mathit{xLabel}(\mathit{xli}, l + 2) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -2) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{aldi}, 0) \rangle \frown \langle \mathit{indirect}\ \mathtt{astd} \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l + 3) \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 2) \rangle \frown \langle \mathit{indirect}\ \mathtt{aldd} \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 2) \rangle \frown \langle \mathit{indirect}\ \mathtt{aeqd} \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrja}, l + 4) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -3) \rangle \frown \langle \mathit{indirect}\ \mathtt{bldd} \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 3) \rangle \frown \langle \mathit{indirect}\ \mathtt{audd} \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrgs}, l + 5) \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l + 4) \rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{bldi}, 255) \rangle
\end{array}
$$

$$
\begin{array}{|l}
\mathit{opudivrem} : \mathit{LABEL} \rightarrow \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathit{opudivrem}\ l = \\
\qquad \mathit{opudivzero}\ l\ \frown \langle \mathit{xLabel}(\mathit{xli}, l + 5)\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 1)\rangle \frown \langle \mathit{indirect}\ \mathtt{bstd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -2)\rangle \frown \langle \mathit{immediate}(\mathtt{aldi}, 0)\rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{aumd}\rangle \frown \langle \mathit{indirect}\ \mathtt{bstd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 1)\rangle \frown \langle \mathit{indirect}\ \mathtt{bldd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 1)\rangle \frown \langle \mathit{indirect}\ \mathtt{aumd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -3)\rangle \frown \langle \mathit{indirect}\ \mathtt{cca}\rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{aucd}\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrja}, l + 7)\rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{aeqd}\rangle \frown \langle \mathit{indirect}\ \mathtt{nca}\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrja}, l + 6)\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -1)\rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{ccb}\rangle \frown \langle \mathit{indirect}\ \mathtt{bucd}\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrjb}, l + 7)\rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{beqd}\rangle \frown \langle \mathit{indirect}\ \mathtt{ncb}\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrjb}, l + 6)\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, 2)\rangle \frown \langle \mathit{indirect}\ \mathtt{aldd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\mathtt{dpi}, -3)\rangle \frown \langle \mathit{indirect}\ \mathtt{aucd}\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrja}, l + 7)\rangle
\end{array}
$$

$$\begin{array}{l}
\mathit{opudivq} : \mathit{LABEL} \rightarrow \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \bullet \\
\quad \mathit{opudivq}\ l = \\
\qquad \mathit{opudivrem}\ l\ \frown \langle \mathit{xLabel}(\mathit{xli}, l + 6)\rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 5)\rangle \frown \langle \mathit{indirect}\ \texttt{bldd}\rangle \\
\qquad \frown \langle \mathit{immediate}(\texttt{aldi}, 0)\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrgs}, l + 8)\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l + 7)\rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 5)\rangle \\
\qquad \frown \langle \mathit{indirect}\ \texttt{bldd}\rangle \frown \langle \mathit{indirect}\ \texttt{ccb}\rangle \\
\qquad \frown \langle \mathit{immediate}(\texttt{aldi}, 0)\rangle \frown \langle \mathit{immediate}(\texttt{busi}, 1)\rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l + 8)\rangle
\end{array}$$

The stand-alone UDIV operation implements the division algorithm, then returns to the caller, saving/restoring the contents of the a, b registers. There are approximately 200 instructions in the complete template.

$$\begin{array}{l}
\mathcal{O}_{\mathit{UDIV}} : \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, l : \mathit{LABEL} \mid l = \mathit{binopfn}\ \mathit{udiv} \bullet \\
\quad \mathcal{O}_{\mathit{UDIV}} = \\
\qquad \mathit{opudivq}\ l\ \frown \langle \mathit{indirect}\ \texttt{cldab}\rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 7)\rangle \frown \langle \mathit{indirect}\ \texttt{abldd}\rangle \\
\qquad \frown \langle \mathit{xIndirect}\ \mathit{xag}\rangle
\end{array}$$

Unsigned division is called by placing the arguments and return address in the correct places in scratchpad memory, then jumping to *udiv*.

$$
\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \rightarrow \mathit{OFFSET} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \mathcal{O}_{BO}\ \mathit{udiv}\ \delta\ l = (l + 1, \\
\qquad \langle \mathit{xData}(\mathit{xsdro}, 3) \rangle \frown \langle \mathit{indirect}\ \texttt{abstd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \langle \mathit{indirect}\ \texttt{abldd} \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 0) \rangle \frown \langle \mathit{indirect}\ \texttt{abstd} \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xll}, l) \rangle \\
\qquad \frown \langle \mathit{xData}(\mathit{xsdro}, 7) \rangle \frown \langle \mathit{indirect}\ \texttt{abstd} \rangle \\
\qquad \frown \langle \mathit{xCallOp}(\mathit{binopfn}\ \mathit{udiv}) \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l) \rangle \frown \langle \mathit{indirect}\ \texttt{abldc} \rangle)
\end{array}
$$

5.11 Binary Unsigned Modulus Operator

The unsigned modulus operator is defined in terms of unsigned multiplication and division, using the equation

$$x \bmod y = x - y * (x \operatorname{div} y)$$

Consequently, the mod operator takes up a further two addresses on the expression evaluation stack.

$$
\begin{array}{|l}
\mathcal{O}_{BO} : \mathit{BIN\_OP} \rightarrow \mathit{OFFSET} \rightarrow \mathit{LABEL} \rightarrow \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists\, \mathit{ldiv}, \mathit{lmul}, \mathit{lminus} : \mathit{LABEL};\ \mathit{Idiv}, \mathit{Imul}, \mathit{Iminus} : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\quad (\mathit{ldiv}, \mathit{Idiv}) = \mathcal{O}_{BO}\ \mathit{udiv}\ \delta\ l \\
\qquad\quad \wedge (\mathit{lmul}, \mathit{Imul}) = \mathcal{O}_{BO}\ \mathit{umul}(\delta + 2)\mathit{ldiv} \\
\qquad\quad \wedge (\mathit{lminus}, \mathit{Iminus}) = \mathcal{O}_{BO}\ \mathit{uminus}\ \delta\ \mathit{lmul} \bullet \\
\qquad \mathcal{O}_{BO}\ \mathit{umod}\ \delta\ l = (\mathit{lminus}, \\
\qquad\quad \langle \mathit{xData}(\mathit{xsdrs}, \delta + 2) \rangle \\
\qquad\quad \frown \langle \mathit{indirect}\ \texttt{abstd} \rangle \frown \mathit{Idiv} \frown \mathit{Imul} \frown \mathit{Iminus})
\end{array}
$$

Note that this operator template uses the data addresses from *topOp* to $\mathit{topOp} + 10$ inclusive.

5.12 Binary Casting Operators

byteToEnum(*enumType*, *byte*) puts the *enumType* on the stack, and the *byte* in the *b* register. The result is the *byte* in the *b* register, so nothing needs to be done.

join(*hi*, *lo*) has the *hi* byte on the stack, and the *lo* byte in the *b* register. The *hi* byte needs to be loaded into the *a* register.

$$\mathcal{O}_{BO} : BIN\_OP \rightarrow OFFSET \rightarrow LABEL \rightarrow LABEL \times \operatorname{seq} X\_INSTR$$

$$\begin{array}{l}\forall \delta : OFFSET;\ l : LABEL \bullet \\ \quad \mathcal{O}_{BO}\ byteToEnum\ \delta\ l = (l, \langle\ \rangle) \\ \quad \land\ \mathcal{O}_{BO}\ join\ \delta\ l = (l, \langle indirect\ \texttt{aldd} \rangle)\end{array}$$

6 Expressions

The expression templates calculate the value denoted by the expression. The value is left in the a and b registers, or just the b register if the value is boolean or byte.

The meaning functions for expressions are quite complicated, as a large amount of information is required to calculate the templates. In particular

- the argument data address offset points to the top of the expression evaluation stack, 'above' which the locations may be used to store intermediate results
- the argument label is the next label name to be used
- the result label is the new next label name to be used, taking into account any labels used to translate the expression

6.1 Utility function — location of identifier

$\mathcal{O}_{Eaddr}$ calculates the code fragment that loads into the $[a, b]$ registers the data address corresponding to the relevant identifier (variable or parameter). It is used by $\mathcal{O}_{VL}$, which in turn is used to calculate the relevant address in both value reference expressions, and the lhs of assignment statements.

The parameters to $\mathcal{O}_{Eaddr}$ are:

- $IDTYPE$ of the identifier
- MO: the memory allocation map
- $OPDENVALUE$: of the identifier (from the $EnvOTrace$ environment)
- $\mathrm{seq}\, SubrangeN$: the sequence of lower and upper array bounds of the identifier

There are two cases: formal parameter and variable.

6.2 Utility function – location of identifier – formal parameter

formalParameter is call by reference, so indirect addressing:

1. $(xsdrh, \delta)$: set the data register D to δ offset from the heap, $D := \delta + bottomHeap$
2. `abldd`: load [a,b] with the address stored there

$$\begin{array}{l}
\mathcal{O}_{Eaddr} : IDTYPE \times MO \times OPDENVALUE \times \operatorname{seq} SubrangeN \rightarrow\!\!\!\!\!\!\nrightarrow \\
\qquad\qquad \operatorname{seq} X\_INSTR \\
\hline
\forall\, i : IDTYPE;\ \mu o : MO;\ odv : OPDENVALUE;\ NN : \operatorname{seq} SubrangeN \mid \\
\qquad i \in \operatorname{ran} formalParam \land (formalParam^{\sim}\ i).2 = ref \bullet \\
\quad \mathcal{O}_{Eaddr}(i, \mu o, odv, NN) = \\
\qquad (\mathbf{let}\ \delta == (startAddr \circ \mu o \circ opLoc^{\sim}\ odv)\langle\ \rangle \bullet \\
\qquad\quad \langle xData(xsdrh, \delta)\rangle \frown \langle indirect\ \texttt{abldd}\rangle)
\end{array}$$

6.3 Utility function – location of identifier – variable

There are three cases:

- declared 'AT' *variable*, placed at a fixed location:
  $(xlada, \delta)$: load $[a, b]$ with the immediate absolute address δ, $[a, b] := \delta$
- ordinary declared variable (no 'AT' attributes):
  $(xlrda, \delta)$: load $[a, b]$ with the immediate relative address δ, $[a, b] := \delta + bottomHeap$
- declared imported variable:
  xBvar: special command for the linker to fixup

$$
\begin{array}{l}
\forall i : IDTYPE;\ \mu o : MO;\ odv : OPDENVALUE;\ NN : \mathrm{seq}\, SubrangeN \mid \\
\quad\quad i \in \mathrm{ran}\, variable \bullet \\
\quad \exists A : \mathbb{P}\, ATTR \mid \\
\quad\quad\quad A = attributeOf\ i \\
\quad\quad\quad \land\ lbs = \{\ n : domNN \bullet n \mapsto (NNn).lb\ \} \bullet \\
\quad\quad (A \cap \mathrm{ran}\, dataAt \neq \varnothing \Rightarrow \\
\quad\quad\quad \mathcal{O}_{Eaddr}(i, \mu o, odv, NN) = \\
\quad\quad\quad\quad (\mathbf{let}\ \delta == (startAddr \circ \mu o \circ opLoc^{\sim}\ odv) lbs \bullet \\
\quad\quad\quad\quad\quad \langle xData(xlada, \delta)\rangle)) \\
\quad\quad \land\ (A \cap \mathrm{ran}\, dataAt = \varnothing \land odv \in \mathrm{ran}\, op\_loc \Rightarrow \\
\quad\quad\quad \mathcal{O}_{Eaddr}(i, \mu o, odv, NN) = \\
\quad\quad\quad\quad (\mathbf{let}\ \delta == (startAddr \circ \mu o \circ opLoc^{\sim}\ odv) lbs \bullet \\
\quad\quad\quad\quad\quad \langle xData(xlrda, \delta)\rangle)) \\
\quad\quad \land\ (odv \in \mathrm{ran}\, opImport \Rightarrow \\
\quad\quad\quad \mathcal{O}_{Eaddr}(i, \mu o, odv, NN) = \\
\quad\quad\quad\quad \langle xBvar(opImport^{\sim}\ odv, lbs)\rangle)
\end{array}
$$

6.4 Actual parameter

The translation of an actual parameter stores the required value at the correct addresses in the store. For call by value, the stored value is the value of the actual parameter, while for call by reference it is the start address of the variable.

$$\begin{array}{|l}
\mathcal{O}_{AP} : \mathit{EXPR} \mathrel{+\!\!\!\rightarrow} \\
\qquad\qquad \mathit{IDTYPE} \mathrel{+\!\!\!\rightarrow} \mathit{STACK} \mathrel{+\!\!\!\rightarrow} \mathit{ID} \mathrel{+\!\!\!\rightarrow} \mathit{EnvOTrace} \times \mathit{MO} \mathrel{+\!\!\!\rightarrow} \\
\qquad\qquad\qquad\qquad \mathit{OFFSET} \mathrel{+\!\!\!\rightarrow} \mathit{LABEL} \mathrel{+\!\!\!\rightarrow} \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall\, \epsilon : \mathit{EXPR};\ i : \mathit{IDTYPE};\ B : \mathit{STACK};\ b : \mathit{ID};\ \rho ot : \mathit{EnvOTrace}; \\
\qquad\qquad \mu o : \mathit{MO};\ \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \mid \\
\qquad\quad i \in \operatorname{ran} \mathit{formalParam} \bullet \\
\qquad \exists\, \mathit{FormalType};\ l' : \mathit{LABEL};\ I : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad\quad i = \mathit{formalParam}\ \theta \mathit{FormalType} \\
\qquad\qquad\quad \wedge (c = \mathit{val} \Rightarrow (l', I) = \mathcal{O}_E\ \epsilon\ B(\rho ot, \mu o)\delta\ l) \\
\qquad\qquad\quad \wedge (c = \mathit{ref} \Rightarrow \\
\qquad\qquad\qquad (\ \exists\, \xi' : \mathit{ID};\ i' : \mathit{IDTYPE};\ \mathit{odv} : \mathit{OPDENVALUE} \mid \\
\qquad\qquad\qquad\qquad \llparenthesis\ \xi == \xi', E == \langle\ \rangle\ \rrparenthesis \mathit{valueRef}^{\sim}\ \epsilon \\
\qquad\qquad\qquad\qquad \wedge\ i' = \mathit{lookup}(\xi', B, \rho\tau t0) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{odv} = \mathit{lookup}(\xi', B, \rho ot) \bullet \\
\qquad\qquad\qquad\quad I = \mathcal{O}_{Eaddr}(i', \mu o, \mathit{odv}, \mathit{NN}) \wedge l' = l)\ ) \bullet \\
\qquad\quad \mathcal{O}_{AP}\ \epsilon\ i\ B\ \ b(\rho ot, \mu o)\delta\ l = (l', I \frown \langle \mathit{xPa}(b, \xi) \rangle \frown \mathcal{O}_{s\tau}\ \tau)
\end{array}$$

In this fragment, the store address for the actual parameter is the formal parameter address in the declaring procedure or function. This is unknown at module compile time, and is fixed up by the linker.

1. I: load the parameter value/reference into $[a, b]$. Calculating a value might consume labels; calculating a reference location does not.
2. $\mathit{xPa}(b, \xi)$: the linker determines the correct formal location
3. $\mathcal{O}_{s\tau}$: store the parameter at that location

To translate a list of actual parameters, each parameter is translated in turn, and the results concatenated.

$$
\begin{array}{|l}
\mathcal{O}_{AP^*} : \operatorname{seq} EXPR \rightarrowtail\\
\qquad\qquad \operatorname{seq} IDTYPE \rightarrowtail STACK \rightarrowtail ID \rightarrowtail EnvOTrace \times MO \rightarrowtail\\
\qquad\qquad\qquad\qquad OFFSET \rightarrowtail LABEL \rightarrowtail LABEL \times \operatorname{seq} X\_INSTR\\
\hline
\forall B : STACK;\ b : ID;\ \rho\mu ot : EnvOTrace \times MO;\\
\qquad\qquad \delta : OFFSET;\ l : LABEL \bullet\\
\qquad \mathcal{O}_{AP^*}\langle\ \rangle\langle\ \rangle B\ \ b\ \ \rho\mu ot\ \ \delta\ \ l = (l, \langle\ \rangle)\\
\forall E : \operatorname{seq} EXPR;\ \epsilon : EXPR;\ I : \operatorname{seq} IDTYPE;\ i : IDTYPE;\\
\qquad\qquad B : STACK;\ b : ID;\ \rho\mu ot : EnvOTrace \times MO;\\
\qquad\qquad \delta : OFFSET;\ l : LABEL \mid\\
\qquad\quad \#E = \#I \bullet\\
\qquad \exists\, l', l'' : LABEL;\ P, P' : \operatorname{seq} X\_INSTR \mid\\
\qquad\qquad (l', P) = \mathcal{O}_{AP}\ \ \epsilon\ \ i\ \ B\ \ b\ \ \rho\mu ot\ \ \delta\ \ l\\
\qquad\qquad \wedge\ (l'', P') = \mathcal{O}_{AP^*}\ \ E\ \ I\ \ B\ \ b\ \ \rho\mu ot\ \ \delta\ \ l' \bullet\\
\qquad\quad \mathcal{O}_{AP^*}(\langle\epsilon\rangle \frown E)(\langle i\rangle \frown I)B\ \ b\ \ \rho\mu ot\ \ \delta\ \ l = (l'', P \frown P')
\end{array}
$$

6.5 Constant

The value of a boolean or byte constant is immediately loaded into the b register. The value of an unsigned constant is immediately loaded into the a and b registers.

This translation uses up no new labels.

$$
\begin{array}{|l}
\mathcal{O}_E : EXPR \rightarrow\\
\qquad\qquad STACK \rightarrowtail EnvOTrace \times MO \rightarrowtail OFFSET \rightarrowtail LABEL \rightarrowtail\\
\qquad\qquad\qquad\qquad LABEL \times \operatorname{seq} X\_INSTR\\
\hline
\forall v : VALUE;\ B : STACK;\ \rho\mu ot : EnvOTrace \times MO;\\
\qquad\qquad \delta : OFFSET;\ l : LABEL \bullet\\
\qquad \mathcal{O}_E(constant\ \ v)B\ \ \rho\mu ot\ \ \delta\ \ l = (l, \mathcal{O}_{l\tau i}(constType\ \ v, number\ \ v))
\end{array}
$$

6.6 Value Reference

6.6.1 Value Reference – named constant

$$\begin{array}{l}
\forall \mathit{ValueRefExpr};\ B : \mathit{STACK};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO}; \\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \mid \\
\qquad\quad E = \langle\,\rangle \bullet \\
\quad \forall i : \mathit{IDTYPE} \mid i = \mathit{lookup}(\xi, B, \rho\tau t0) \land i \in \operatorname{ran} \mathit{const} \bullet \\
\qquad \exists \mathit{odv} : \mathit{OPDENVALUE} \mid \mathit{odv} = \mathit{lookup}(\xi, B, \rho ot) \bullet \\
\qquad\quad (\mathit{odv} \in \operatorname{ran} \mathit{opImport} \Rightarrow \\
\qquad\qquad \mathcal{O}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr})B(\rho ot, \mu o)\delta\ l = \\
\qquad\qquad\quad (l, \langle \mathit{xConst}\ \xi \rangle)) \\
\qquad\quad \land (\mathit{odv} \notin \operatorname{ran} \mathit{opImport} \Rightarrow \\
\qquad\qquad (\exists \kappa : \mathit{VALUE} \mid \kappa = (\mathit{const}^{\sim}\ i).v \bullet \\
\qquad\qquad\quad \mathcal{O}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr})B(\rho ot, \mu o)\delta\ l = \\
\qquad\qquad\qquad \mathcal{O}_E(\mathit{constant}\ \kappa)B(\rho ot, \mu o)\delta\ l))
\end{array}$$

If the named constant is an imported constant, mark it using *xConst*, otherwise look up the value in the type environment, and load its value directly into $[a, b]$.

6.6.2 Value Reference – enumerated value

$$\begin{array}{l}
\forall \mathit{ValueRefExpr};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO}; \\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \mid \\
\qquad\quad E = \langle\,\rangle \bullet \\
\quad \forall i : \mathit{IDTYPE} \mid i = \mathit{lookup}(\xi, B, \rho\tau t0) \land i \in \operatorname{ran} \mathit{enumValue} \bullet \\
\qquad \exists \mathit{bmax}, b : \mathit{BYTE} \mid \mathit{venum}(\mathit{bmax}, b) = (\mathit{enumValue}^{\sim}\ i).v \bullet \\
\qquad\quad \mathcal{O}_E(\mathit{valueRef}\ \theta \mathit{ValueRefExpr})B\ \rho\mu ot\ \delta\ l = \\
\qquad\qquad (l, \langle \mathit{immediate}(\texttt{bldi}, b) \rangle)
\end{array}$$

The enumerated value is looked up in the type environment, and its value loaded directly into b.

6.6.3 Value Reference – enumerated type

$$
\begin{array}{|l}
\forall \, ValueRefExpr; \; B : STACK; \; \rho\mu ot : EnvOTrace \times MO; \\
\qquad\qquad \delta : OFFSET; \; l : LABEL \mid \\
\qquad E = \langle\,\rangle \bullet \\
\quad \forall \, i : IDTYPE \mid i = lookup(\xi, B, \rho\tau t0) \land i \in \operatorname{ran} enumType \bullet \\
\qquad \exists \, b : BYTE \mid b = \#((enumType^{\sim} \; i).\Xi) \bullet \\
\qquad\quad \mathcal{O}_E(valueRef \; \theta ValueRefExpr) B \; \rho\mu ot \; \delta \; l = \\
\qquad\qquad (l, \langle immediate(\texttt{bldi}, b)\rangle)
\end{array}
$$

The enumerated type's size is looked up in the type environment, and its value loaded directly into b.

6.6.4 Value Reference – variable

$$
\begin{array}{|l}
\forall \, ValueRefExpr; \; B : STACK; \; \rho\mu ot : EnvOTrace \times MO; \\
\qquad\qquad \delta : OFFSET; \; l : LABEL \bullet \\
\quad \forall \, i : IDTYPE \mid \\
\qquad\qquad i = lookup(\xi, B, \rho\tau t0) \\
\qquad\qquad \land \; i \notin \operatorname{ran} const \cup \operatorname{ran} enumValue \cup \operatorname{ran} enumType \bullet \\
\qquad \exists \, \tau : TYPE; \; l' : LABEL; \; I : \operatorname{seq} X\_INSTR \mid \\
\qquad\qquad \tau = typeOf \;\; i \land (l', I) = \mathcal{O}_{VL} \; E \; B \; \rho\mu ot \; \delta \; l \; \xi \bullet \\
\qquad\quad \mathcal{O}_E(valueRef \; \theta ValueRefExpr) B \; \rho\mu ot \; \delta \; l = \\
\qquad\qquad (l', I \frown \langle indirect \; \texttt{daldab}\rangle \frown \mathcal{O}_{l\tau} \; \tau)
\end{array}
$$

For a variable:

1. I: load the data address of the relevant array element into $[a, b]$. ($\mathcal{O}_{VL}$ evaluates the expression list to calculate the offset, then multiplies the offset by the size of the type and adds the position of the first element of the array.)
2. `daldab`: load $[a, b]$ into D
3. $\mathcal{O}_{l\tau}$: load the value from $mem \; D$ to $[a, b]$

6.7 Unary expression

In this case the body expression is translated (which results in its value being stored in the a and b registers) and then the translation of the unary operator is concatenated to the fragment.

$$
\begin{array}{l}
\forall\, \mathit{UnyExpr};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet\\
\quad \exists\, l', l'' : \mathit{LABEL};\ I, I' : \mathrm{seq}\, \mathit{X\_INSTR} \mid\\
\qquad\qquad (l', I) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ l\\
\qquad\qquad \land\ (l'', I') = \mathcal{O}_{UO}\ \Psi\ l' \bullet\\
\qquad\quad \mathcal{O}_E(\mathit{unyExpr}\ \theta\, \mathit{UnyExpr}) B\ \rho\mu ot\ \delta\ l = (l'', I \frown I')
\end{array}
$$

6.8 Binary expression

$$
\begin{array}{l}
\forall\, \mathit{BinExpr};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet\\
\quad \exists\, l1, l2, l3 : \mathit{LABEL};\ I1, I2, I3 : \mathrm{seq}\, \mathit{X\_INSTR} \mid\\
\qquad\qquad (l1, I1) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ l\\
\qquad\qquad \land\ (l2, I2) = \mathcal{O}_E\ \epsilon'\ B\ \rho\mu ot\ (\delta + 2) l1\\
\qquad\qquad \land\ (l3, I3) = \mathcal{O}_{BO}\ \Omega\ \delta\ l2 \bullet\\
\qquad\quad \mathcal{O}_E(\mathit{binExpr}\ \theta \mathit{BinExpr}) B\ \rho\mu ot\ \delta\ l = (l3,\\
\qquad\qquad I1\ \frown \langle \mathit{xData}\ (\mathit{xsdrs}, \delta)\rangle\\
\qquad\qquad \frown \langle \mathit{indirect}\ \texttt{abstd}\rangle \frown I2 \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta)\rangle \frown I3)
\end{array}
$$

Algorithm:

1. $I1$: translate expression ϵ, leaving result in $[a, b]$
2. $\mathit{xData}(\mathit{xsdrs}, \delta)$: set $D := \mathit{topStack} - \delta$, to next available space on expression evaluation stack.
3. `abstd`: store value of ϵ on expression stack

4. $I2$: translate expression ϵ, leaving result in $[a, b]$. This translation uses an expression stack offset of $\delta + 2$, so that the value of ϵ is not overwritten. So two addresses are used to store the value of the subexpression, even if it has boolean or byte type, because the information to calculate the type of the subexpression is not readily available at this point. (The binary expression can re-use the two addresses if it has to calculate a modtype expression.)

5. $xData(xsdrs, \delta)$: set $D := topStack - \delta$, so pointing to ϵ

6. $I3$: translate the binary operator, which leaves the result $\epsilon \; \Omega \; \epsilon$ in $[a, b]$

6.9 Function call

$$
\begin{array}{l}
\forall \mathit{FunCallExpr};\ B : \mathit{STACK};\ \rho\mu o t : \mathit{EnvOTrace} \times \mathit{MO}; \\
\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL} \bullet \\
\quad \exists \mathit{Bf} : \mathit{STACK};\ I : \mathrm{seq}\, \mathit{IDTYPE};\ \mathit{sr} : \mathit{Subrange};\ l' : \mathit{LABEL}; \\
\qquad\quad P : \mathrm{seq}\, \mathit{X\_INSTR} \mid \\
\qquad \mathit{function}(\mathit{Bf}, I, \mathit{sr}) = \mathit{lookup}(\xi, B, \rho\tau t0) \\
\qquad \wedge\ (l', P) = \mathcal{O}_{AP*}\ E\ I\ B\ \xi\ \rho\mu o t\ l\ \delta \bullet \\
\quad \mathcal{O}_E(\mathit{funCall}\ \theta \mathit{FunCallExpr}) B\ \rho\mu o t\ \delta\ l = (l' + 1, \\
\qquad P \frown \langle \mathit{xLabel}(\mathit{xll}, l') \rangle \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{cldab} \rangle \frown \langle \mathit{xCall}\ \xi \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, l') \rangle \frown \langle \mathit{indirect}\ \mathtt{abldc} \rangle)
\end{array}
$$

Algorithm:

1. P: calculate the values of the actual parameters, and store them at the correct formal parameter locations.

2. (xll, l): store the return label l in the C register. (This label is stored on the heap by the function itself, and used to return to the correct place.)

3. $xCall\ \xi$: call the function ξ.

4. (xli, l): place the return label l here.

5. `abldc`: load the result from C into $[a, b]$. (The function leaves its result in the C register.)

6.10 Multiple expressions

Multiple expressions occur as array indices. The templates that translate a sequence of expressions calculate the offset represented by the sequence of values from the position of the first element of the array.

The code calculates the array index offsets to give the same result as the Pasp *locationOffset* function. That is $\Sigma_i(ss_i * (si_i - (ssr_i).lb))$. The naming used here is instead $\Sigma_i(ne_i * (\epsilon_i - lb_i))$.

6.10.1 Single expression index

This template performs the calculation $ne_i * (\epsilon_i - lb_i)$. There is a (compile time) optimisation for the case $ne = 1$

$$
\begin{array}{|l}
\mathcal{O}_{E1} : \mathit{EXPR} \rightarrow \\
\qquad\qquad \mathit{STACK} \pfun \mathit{EnvOTrace} \times \mathit{MO} \pfun \mathit{OFFSET} \pfun \mathit{LABEL} \pfun \\
\qquad\qquad\quad \mathit{TYPE} \pfun \mathbb{N} \times \mathbb{N} \pfun \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \epsilon : \mathit{EXPR};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\ \delta : \mathit{OFFSET}; \\
\qquad\qquad l : \mathit{LABEL};\ \tau : \mathit{TYPE};\ lb : \mathbb{N} \bullet \\
\quad \exists\, le, lminus : \mathit{LABEL};\ Ie, Iminus : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad (le, Ie) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ l \\
\qquad\qquad \land (lminus, Iminus) = \\
\qquad\qquad\quad \mathbf{if}\ \tau = unsigned \\
\qquad\qquad\quad \mathbf{then}\ \mathcal{O}_{BO}\ uminus\ \delta\ le\ \mathbf{else}\ \mathcal{O}_{BO}\ bminus\ \delta\ le \bullet \\
\qquad \mathcal{O}_{E1}\ \epsilon\ B\ \rho\mu ot\ \delta\ l\ \tau (lb, 1) = (lminus, \\
\qquad\qquad Ie \frown \langle xData(xsdrs, \delta) \rangle \frown \mathcal{O}_{s\tau}\ \tau \frown \mathcal{O}_{l\tau i}(\tau, lb) \frown Iminus) \\
\forall \epsilon : \mathit{EXPR};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\ \delta : \mathit{OFFSET}; \\
\qquad\qquad l : \mathit{LABEL};\ \tau : \mathit{TYPE};\ lb, ne : \mathbb{N} \mid \\
\qquad ne > 1 \bullet \\
\quad \exists\, le, lmul : \mathit{LABEL};\ Ie, Imul : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad (le, Ie) = \mathcal{O}_{E1}\ \epsilon\ B\ \rho\mu ot\ \delta\ l\ \tau (lb, 1) \\
\qquad\qquad \land (lmul, Imul) = \\
\qquad\qquad\quad \mathbf{if}\ \tau = unsigned \\
\qquad\qquad\quad \mathbf{then}\ \mathcal{O}_{BO}\ umul\ \delta\ le\ \mathbf{else}\ \mathcal{O}_{BO}\ bmul\ \delta\ le \bullet \\
\qquad \mathcal{O}_{E1}\ \epsilon\ B\ \rho\mu ot\ \delta\ l\ \tau (lb, ne) = (lmul, \\
\qquad\qquad Ie \frown \langle xData(xsdrs, \delta) \rangle \frown \mathcal{O}_{s\tau}\ \tau \frown \mathcal{O}_{l\tau i}(\tau, ne) \frown Imul)
\end{array}
$$

6.10.2 Multiple expression template

This template performs the calculation $\Sigma_i \mathcal{O}_{E1}\ \epsilon_i$. The first expression in the sum is calculated at the top of the stack, δ. The remaining expressions are calculated at $\delta + 2$, and the running total is kept at δ.

$$
\begin{array}{|l}
\mathcal{O}_{E^*} : \mathrm{seq}_1\, \mathit{EXPR} \rightarrow \\
\qquad\qquad \mathit{STACK} \pfun \mathit{EnvOTrace} \times \mathit{MO} \pfun \mathit{OFFSET} \pfun \mathit{LABEL} \pfun \\
\qquad\qquad\quad \mathit{TYPE} \pfun \mathrm{seq}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N} \pfun \mathit{LABEL} \times \mathrm{seq}\, \mathit{X\_INSTR} \\
\hline
\forall\, \epsilon : \mathit{EXPR};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\ \delta : \mathit{OFFSET}; \\
\qquad\qquad l : \mathit{LABEL};\ \tau : \mathit{TYPE};\ \mathit{lb}, \mathit{ne} : \mathbb{N} \bullet \\
\quad \mathcal{O}_{E^*} \langle \epsilon \rangle B\ \rho\mu ot\ \delta\ l\ \tau(\langle \mathit{lb} \rangle, \langle \mathit{ne} \rangle) = \mathcal{O}_{E1}\ \epsilon\ B\ \rho\mu ot\ \delta\ l\ \tau(\mathit{lb}, \mathit{ne}) \\
\forall\, E : \mathrm{seq}_1\, \mathit{EXPR};\ \epsilon : \mathit{EXPR};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO}; \\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL};\ \tau : \mathit{TYPE}; \\
\qquad\qquad \mathit{LB}, \mathit{NE} : \mathrm{seq}\,\mathbb{N};\ \mathit{lb}, \mathit{ne} : \mathbb{N} \mid \\
\qquad \#\mathit{LB} = \#\mathit{NE} \bullet \\
\quad \exists\, \mathit{lE}, \mathit{le}, \mathit{lplus} : \mathit{LABEL};\ \mathit{IE}, \mathit{Ie}, \mathit{Iplus} : \mathrm{seq}\, \mathit{X\_INSTR} \mid \\
\qquad\qquad (\mathit{lE}, \mathit{IE}) = \mathcal{O}_{E^*}\ E\ B\ \rho\mu ot\ \delta\ l\ \tau(\mathit{LB}, \mathit{NE}) \\
\qquad\qquad \wedge\ (\mathit{le}, \mathit{Ie}) = \mathcal{O}_{E1}\ \epsilon\ B\ \rho\mu ot(\delta + 2)\mathit{lE}\ \tau(\mathit{lb}, \mathit{ne}) \\
\qquad\qquad \wedge\ (\mathit{lplus}, \mathit{Iplus}) = \\
\qquad\qquad\quad \mathbf{if}\ \tau = \mathit{unsigned} \\
\qquad\qquad\quad \mathbf{then}\ \mathcal{O}_{BO}\ \mathit{uplus}\ \delta\ \mathit{le}\ \mathbf{else}\ \mathcal{O}_{BO}\ \mathit{bplus}\ \delta\ \mathit{le} \bullet \\
\qquad \mathcal{O}_{E^*}(E \frown \langle \epsilon \rangle) B\ \rho\mu ot\ \delta\ l\ \tau(\mathit{LB} \frown \langle \mathit{lb} \rangle, \mathit{NE} \frown \langle \mathit{ne} \rangle) = \\
\qquad\qquad (\mathit{lplus}, \\
\qquad\qquad\quad \mathit{IE} \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \mathcal{O}_{s\tau}\ \tau \\
\qquad\qquad\quad \frown \mathit{Ie} \\
\qquad\qquad\quad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \mathit{Iplus})
\end{array}
$$

6.11 Variable location

$\mathcal{O}_{VL}$ calculates the data address of an array element reference, by adding a data address offset (the location offset multiplied by the size of the type) to the address of the first element. This address is known at compilation time (either absolute for ‘AT variables, or relative to *bottomHeap*) unless the variable is a call by reference formal parameter, in which case it is retrieved from a data address known at compilation time.

$$
\begin{array}{|l}
\mathcal{O}_{VL} : \operatorname{seq} \mathit{EXPR} \pfun \\
\qquad\qquad \mathit{STACK} \pfun \mathit{EnvOTrace} \times \mathit{MO} \pfun \mathit{OFFSET} \pfun \\
\qquad\qquad\quad \mathit{LABEL} \pfun \mathit{ID} \pfun \mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall E : \operatorname{seq}_1 \mathit{EXPR};\ B : \mathit{STACK};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO}; \\
\qquad\qquad \delta : \mathit{OFFSET};\ l : \mathit{LABEL};\ \xi : \mathit{ID} \bullet \\
\quad \exists i : \mathit{IDTYPE};\ odv : \mathit{OPDENVALUE};\ \mathit{NN} : \operatorname{seq} \mathit{SubrangeN}; \\
\qquad\qquad\qquad \mathit{Iaddr} : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad i = \mathit{lookup}(\xi, B, \rho\tau t0) \\
\qquad\qquad \land\ odv = \mathit{lookup}(\xi, B, \rho ot) \\
\qquad\qquad \land\ \mathit{NN} = \mathit{arrayOf}\ i \\
\qquad\qquad \land\ \mathit{Iaddr} = \mathcal{O}_{\mathit{Eaddr}}(i, \mu o, odv, \mathit{NN}) \bullet \\
\qquad\quad \mathcal{O}_{VL} \langle\ \rangle B\ (\rho ot, \mu o) \delta\ l\ \xi = (l, \mathit{Iaddr}) \\
\qquad\quad \land\ (\ \exists \tau a, \tau i : \mathit{TYPE};\ lE, \mathit{loff}1, \mathit{loff}2, \mathit{lplus} : \mathit{LABEL}; \\
\qquad\qquad\qquad\qquad \mathit{IE}, \mathit{Ioff}1, \mathit{Ioff}2, \mathit{Iplus} : \operatorname{seq} \mathit{X\_INSTR}; \\
\qquad\qquad\qquad\qquad \mathit{LB}, \mathit{NE} : \operatorname{seq} \mathbb{N} \mid \\
\qquad\qquad\qquad \tau a = \mathit{typeOf}\ i \\
\qquad\qquad\qquad \land\ \tau i = \mathit{arrayIndexType}\ i \\
\qquad\qquad\qquad \land\ (\ \forall n : \operatorname{dom} \mathit{NN} \bullet \\
\qquad\qquad\qquad\qquad \mathit{LB}\ n = (\mathit{NN}\ n).\mathit{lb} \\
\qquad\qquad\qquad\qquad \land\ \mathit{NE}\ n = \mathit{numElts}(((n+1) \mathrel{..} \#\mathit{NN}) \upharpoonleft \mathit{NN})\ ) \\
\qquad\qquad\qquad \land\ (lE, \mathit{IE}) = \mathcal{O}_{E^*}\ E\ B(\rho ot, \mu o)\delta\ l\ \tau i(\mathit{LB}, \mathit{NE}) \\
\qquad\qquad\qquad \land\ (\mathit{loff}1, \mathit{Ioff}1) = \mathbf{if}\ \tau a = \mathit{pbyte} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathcal{O}_{UO}\ \mathit{byteToUnsgn}\ lE\ \mathbf{else}\ (lE, \langle\ \rangle) \\
\qquad\qquad\qquad \land\ (\mathit{loff}2, \mathit{Ioff}2) = \mathbf{if}\ \tau i = \mathit{unsigned} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathcal{O}_{UO}\ \mathit{uleft}\ lE\ \mathbf{else}\ (\mathit{loff}1, \langle\ \rangle) \\
\qquad\qquad\qquad \land\ (\mathit{lplus}, \mathit{Iplus}) = \mathcal{O}_{BO}\ \mathit{uplus}\ \delta\ \mathit{loff}2 \bullet \\
\qquad\qquad \mathcal{O}_{VL}\ E\ B(\rho ot, \mu o)\delta\ l\ \xi = (\mathit{lplus}, \\
\qquad\qquad\qquad \mathit{IE} \frown \mathit{Ioff}1 \frown \mathit{Ioff}2 \\
\qquad\qquad\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \langle \mathit{indirect}\ \mathtt{abstd} \rangle \\
\qquad\qquad\qquad \frown \mathit{Iaddr} \\
\qquad\qquad\qquad \frown \langle \mathit{xData}(\mathit{xsdrs}, \delta) \rangle \frown \mathit{Iplus})\ )
\end{array}
$$

This template is very simple for variables of zero dimension, where the expression sequence is of zero length. In this case, the template I loads the variable's address into the $[a, b]$ registers

If the variable is an array, the array's start address must have the correct offset for that index added to it. The algorithm is:

1. *IE*: convert the sequence of array indexes into the relevant index offset, leaving the index offset in $[a, b]$.
2. *Ioff* 1: convert the index offset to an unsigned value (if the index type is a byte, convert it to an unsigned value)
3. *Ioff* 2: convert the index offset into an address offset in $[a, b]$ (if the array element type is unsigned, multiply the index offset by two)
4. $(xsdrs, \delta)$: $D := topStack - \delta$
5. `abstd`: store the array address offset on the stack
6. *Iaddr*: load the value start address into $[a, b]$
7. $(xsdrs, \delta)$: $D := topStack - \delta$
8. *Iplus*: add the start address to the address offset, leaving the required address in $[a, b]$

7 Statements

Translating statements does not alter the environment, but does produce a sequence of labelled instructions.

7.1 Multiple statements

The translation of a single statement is extended in a straightforward way to one of a list of statements.

$$
\begin{array}{|l}
\mathcal{O}_{S^*} : \mathrm{seq}\, STMT \rightarrowtail \\
\qquad STACK \rightarrowtail EnvOTrace \times MO \rightarrowtail LABEL \rightarrowtail OFFSET \rightarrowtail \\
\qquad\qquad LABEL \times \mathrm{seq}\, X\_INSTR \\
\hline
\forall B : STACK;\ \rho\mu ot : EnvOTrace \times MO;\ l : LABEL;\ \delta : OFFSET \bullet \\
\quad \mathcal{O}_{S^*} \langle\ \rangle B\ \rho\mu ot\ l\ \delta = (l, \langle\ \rangle) \\
\forall \gamma : STMT;\ \Gamma : \mathrm{seq}\, STMT;\ B : STACK;\ \rho\mu ot : EnvOTrace \times MO; \\
\qquad l : LABEL;\ \delta : OFFSET \bullet \\
\quad \exists\, l1, l2 : LABEL;\ I, I' : \mathrm{seq}\, X\_INSTR \mid \\
\qquad (l1, I) = \mathcal{O}_S\ \gamma\ B\ \rho\mu ot\ l\ \delta \\
\qquad \land\ (l2, I') = \mathcal{O}_{S^*}\ \Gamma\ B\ \rho\mu ot\ l1\ \delta \bullet \\
\qquad\quad \mathcal{O}_{S^*}(\langle\gamma\rangle \frown \Gamma) B\ \rho\mu ot\ l\ \delta = (l2, I \frown I')
\end{array}
$$

7.2 Block

Translating a block statement is the same as translating the sequence of body statements.

$$
\begin{array}{|l}
\mathcal{O}_{S} : STMT \rightarrowtail \\
\qquad STACK \rightarrowtail EnvOTrace \times MO \rightarrowtail LABEL \rightarrowtail OFFSET \rightarrowtail \\
\qquad\qquad LABEL \times \mathrm{seq}\, X\_INSTR \\
\hline
\forall \Gamma : \mathrm{seq}\, STMT \bullet \mathcal{O}_S(block\ \Gamma) = \mathcal{O}_{S^*}\ \Gamma
\end{array}
$$

7.3 Skip

A skip statement is compiled to a `nop` rather than to an empty sequence. This allows a Pasp program to use multiple skips to implement a delay.

$$\begin{array}{l}
\forall\, B : STACK;\ \rho\mu ot : EnvOTrace \times MO;\ l : LABEL;\ \delta : OFFSET \bullet \\
\qquad \mathcal{O}_S\ skip\ B\ \rho\mu ot\ l\ \delta = (l, \langle indirect\ \texttt{nop}\rangle)
\end{array}$$

7.4 Assignment

The assignment template has the function of updating the store with the value of the expression. To do this, the data address of the lhs is calculated and temporarily stored in the block's 'assignment address';; then the value of the expression is calculated and stored in the C register; then the value is stored at the lhs address.

$$\begin{array}{l}
\forall\, AssignStmt;\ \epsilon : EXPR;\ B : STACK;\ \rho\mu ot : EnvOTrace \times MO; \\
\qquad\qquad l : LABEL;\ \delta : OFFSET \bullet \\
\quad \exists\, l1, l2 : LABEL;\ Iloc, I : \mathrm{seq}\, X\_INSTR;\ i : IDTYPE; \\
\qquad\qquad\quad \tau : TYPE;\ \delta' : OFFSET \mid \\
\qquad\qquad (l1, Iloc) = \mathcal{O}_{VL}\ E\ B\ \rho\mu ot\ \delta\ l\ \xi \\
\qquad\qquad \wedge\ i = lookup(\xi, B, \rho\tau t0) \\
\qquad\qquad \wedge\ (l2, I) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ l1 \\
\qquad\qquad \wedge\ \tau = typeOf\ \ i \\
\qquad\qquad \wedge\ \delta' = head(assignAddr(last\ B)) \bullet \\
\qquad \mathcal{O}_S(assign\ \theta AssignStmt)B\ \rho\mu ot\ l\ \delta = (l2, \\
\qquad\qquad Iloc \frown \langle xData(xsdrh, \delta')\rangle \frown \langle indirect\ \texttt{abstd}\rangle \frown I \\
\qquad\qquad \frown \langle indirect\ \texttt{cldab}\rangle \\
\qquad\qquad \frown \langle xData(xsdrh, \delta')\rangle \frown \langle indirect\ \texttt{abldd}\rangle \\
\qquad\qquad \frown \langle indirect\ \texttt{daldab}\rangle \frown \langle indirect\ \texttt{abldc}\rangle \frown \mathcal{O}_{s\tau}\ \tau)
\end{array}$$

Algorithm:

1. *Iloc*: calculate data address of $\xi[E]$, result in $[a, b]$

2. $(xsdrh, \delta')$: $D := \delta' + bottomHeap$
3. `abstd`: store address of $\xi[E]$ at D, on heap
4. I: calculate expression ϵ, result in $[a, b]$
5. `cldab`: $C := [a, b]$; $C := \epsilon$
6. $(xsdrh, \delta')$: $D := \delta' + bottomHeap$, contains address of $\xi[E]$
7. `abldd`: load address of $\xi[E]$ into $[a, b]$
8. `daldab`: $D := [a, b]$; $D :=$ address of $\xi[E]$
9. `abldc`: $[a, b] := C$; $[a, b] := \epsilon$
10. $\mathcal{O}_{s\tau}$: store ϵ at address of $\xi[E]$

7.5 If statement

The expression is evaluated and the result put in the a carry bit. The a and b registers are then loaded with the address of the next instruction after the translation of the else statement and its associated *goto*. Then a jump is made on the contents of the a carry bit. The translation of the else statement comes next, followed by a *goto* to the statement following the translation of the entire ‘if then else’. This is followed by the translation for the then statement.

$$
\begin{array}{l}
\forall\, \mathit{IfStmt};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\ l : \mathit{LABEL};\\
\qquad\qquad \delta : \mathit{OFFSET} \bullet\\
\quad \exists\, l1, l2, l3 : \mathit{LABEL};\ I, I', \mathit{IE} : \mathrm{seq}\, X\_\mathit{INSTR} \mid\\
\qquad\qquad (l1, \mathit{IE}) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ l\\
\qquad\qquad \wedge\ (l2, I') = \mathcal{O}_S\ \gamma'\ B\ \rho\mu ot\ l1\ \delta\\
\qquad\qquad \wedge\ (l3, I) = \mathcal{O}_S\ \gamma\ B\ \rho\mu ot\ (l2 + 1)\delta \bullet\\
\qquad \mathcal{O}_S(\mathit{ifStmt}\ \theta \mathit{IfStmt}) B\ \rho\mu ot\ l\ \delta = (l3 + 1,\\
\qquad\qquad \mathit{IE}\\
\qquad\qquad ^\frown \langle \mathit{indirect}\ \mathtt{rrb} \rangle\\
\qquad\qquad ^\frown \langle \mathit{xLabel}(\mathit{xrjbl}, l2) \rangle\ ^\frown I'\\
\qquad\qquad ^\frown \langle \mathit{xLabel}(\mathit{xrg}, l3) \rangle\\
\qquad\qquad ^\frown \langle \mathit{xLabel}(\mathit{xli}, l2) \rangle\ ^\frown I\\
\qquad\qquad ^\frown \langle \mathit{xLabel}(\mathit{xli}, l3) \rangle)
\end{array}
$$

7.6 Case statement

The dynamic semantics of case statements are defined by calculating the dynamic semantics of an equivalent fragment of Pasp (written out in abstract syntax) which contains no case statements. The operational semantics are defined as the (operational) meaning of the same fragment, thus making the proof of the correctness immediate by structural induction. In order to evaluate the expression at the head of the case statement just once, a temporary variable is used to store the expression value.

The functions defining the equivalent Pasp fragment can be found in the Pasp specification.

$$
\begin{array}{l}
\forall\, \mathit{CaseStmt};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO};\ l : \mathit{LABEL};\\
\qquad\qquad \delta : \mathit{OFFSET} \bullet\\
\quad \mathcal{O}_S(\mathit{caseStmt}\ \theta \mathit{CaseStmt}) B\ \rho\mu ot\ l\ \delta =\\
\qquad \mathcal{O}_S(\mathit{block}(\langle \mathit{assign} \langle\!| \ \xi == \xi, E == \langle\ \rangle, \epsilon == \epsilon\ |\!\rangle \rangle\\
\qquad\qquad\qquad ^\frown \langle \mathit{limbSwitch}\ K\ \xi \rangle)) B\ \rho\mu ot\ l\ \delta
\end{array}
$$

7.7 Loop

The loop statement defined is a while-do, rather than a repeat-until loop. This means that the loop body could not be called at all if the condition is false.

$$
\begin{array}{l}
\forall\ \mathit{WhileStmt};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO}; \\
\qquad l : \mathit{LABEL};\ \delta : \mathit{OFFSET} \bullet \\
\quad \exists\ \mathit{lE}, \mathit{lS} : \mathit{LABEL};\ \mathit{IE}, \mathit{IS} : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad (\mathit{lE}, \mathit{IE}) = \mathcal{O}_E\ \epsilon\ B\ \rho\mu ot\ \delta\ (l + 1) \\
\qquad \wedge\ (\mathit{lS}, \mathit{IS}) = \mathcal{O}_S\ \gamma\ B\ \rho\mu ot\ \mathit{lE}\ \delta \bullet \\
\quad \mathcal{O}_S(\mathit{whileStmt}\ \theta\,\mathit{WhileStmt})B\ \rho\mu ot\ l\ \delta = (\mathit{lS} + 1, \\
\qquad \langle \mathit{xLabel}(\mathit{xli}, l) \rangle \frown \mathit{IE} \\
\qquad \frown \langle \mathit{indirect}\ \mathtt{rrb} \rangle \frown \langle \mathit{indirect}\ \mathtt{ncb} \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrjbl}, \mathit{lS}) \rangle \frown \mathit{IS} \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xrg}, l) \rangle \\
\qquad \frown \langle \mathit{xLabel}(\mathit{xli}, \mathit{lS}) \rangle)
\end{array}
$$

There is one jump and one goto in the translation of a loop statement. Firstly the expression is translated, the result is moved to the b carry bit, and the jump address is loaded into the a and b registers. Then the carry bit is inverted and a jump to the instruction after the end of the translation of the loop statement (i.e. the jump is performed if the result of the expression was false). The next sequence of instructions carries out the body of the loop, and jumps back to the beginning of the translation of the loop statement in order to re-evaluate the expression. The jumps are performed relative to labelled instructions.

7.8 Procedure call

The operational semantics of a procedure call is a simplified version of the operational semantics of a function call. The final part of the function call fragment that moves the return value from the C register to the a, b registers is omitted.

$$
\begin{array}{l}
\forall\ ProcCallStmt;\ B : STACK;\ \rho\mu ot : EnvOTrace \times MO; \\
\qquad\qquad l : LABEL;\ \delta : OFFSET \bullet \\
\quad \exists\ ProcType';\ l' : LABEL;\ I : \operatorname{seq} X\_INSTR \mid \\
\qquad\qquad procedure\ \theta ProcType' = lookup(\xi, B, \rho\tau t0) \\
\qquad\qquad \wedge\ (l', I) = \mathcal{O}_{AP*}\ E\ I'\ B\ \xi\ \rho\mu ot\ \delta\ l \bullet \\
\qquad \mathcal{O}_S(procCall\ \theta ProcCallStmt) B\ \rho\mu ot\ l\ \delta = (l' + 1, \\
\qquad\qquad I \frown \langle xLabel(xll, l') \rangle \\
\qquad\qquad \frown \langle indirect\ \texttt{cldab} \rangle \frown \langle xCall\ \xi \rangle \\
\qquad\qquad \frown \langle xLabel(xli, l') \rangle)
\end{array}
$$

8 Declarations

Declarations do not translate to instructions. Instead they cause modifications to the components of the translation environment. The definitions given here should be compared with the definition of the dynamic semantics of declarations.

We assume that the checks embodied in the static semantics are carried out before compilation, and that the type environment and constant table are available. There is no need, therefore, to define any semantics for the declaration of named constants or type definitions.

8.1 Utility functions

opUsedLocations returns the set of all locations used by an operational environment, those used by variables, and those used to store call return labels.

$$
\begin{array}{l}
opUsedLocations : EnvOTrace \rightarrow\!\!\!\!\!\!+\ \mathbb{P}\, LOCN \\
\hline
\forall \rho ot : EnvOTrace \bullet \\
\quad opUsedLocations\ \rho ot = \\
\qquad \bigcup \{\ b : \operatorname{dom} \rho ot;\ \xi : ID;\ f : \operatorname{seq} \mathbb{N} \rightarrowtail\!\!\!\!\!\!+\ LOCN \mid \\
\qquad\qquad \xi \in \operatorname{dom}(\rho ot\ \ b) \land \rho ot\ \ b\ \xi = opLoc\ f \bullet \\
\qquad\quad \operatorname{ran} f\ \} \\
\qquad \cup \{\ b : \operatorname{dom} \rho ot;\ \xi : ID;\ l : LABEL;\ locn : LOCN \mid \\
\qquad\qquad \xi \in \operatorname{dom}(\rho ot\ \ b) \land \rho ot\ \ b\ \xi = opPFval\ (l, locn) \bullet \\
\qquad\quad locn\ \}
\end{array}
$$

uss returns the set of all data addresses used by a memory allocation environment.

$$
\begin{array}{l}
usedAddrs : MO \rightarrow\!\!\!\!\!\!+\ \mathbb{P}\, ADDR \\
\hline
\forall \mu o : MO \bullet usedAddrs\ \mu o = (\operatorname{ran} \circ \bigcup \circ \operatorname{ran})\mu o
\end{array}
$$

newLabelAddr returns a data address suitable for storing a label, along with suitably updated environments.

$$
\begin{array}{l}
\mathit{newLabelAddr} : (\mathit{EnvOTrace} \times \mathit{MO}) \times (\mathit{ID} \times \mathit{ID}) \times \mathit{LABEL} \pfun \\
\qquad (\mathit{EnvOTrace} \times \mathit{MO}) \times \mathit{ADDR} \\
\hline
\forall \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO};\ \xi, \xi' : \mathit{ID};\ l : \mathit{LABEL} \bullet \\
\quad \exists \rho ot' : \mathit{EnvOTrace};\ \mu o' : \mathit{MO};\ locn : \mathit{LOCN};\ \delta : \mathit{ADDR} \mid \\
\qquad\qquad \mathsf{disjoint}\langle \{locn\}, \mathit{opUsedLocations}\ \rho ot\rangle \\
\qquad\qquad \land\ \mathsf{disjoint}\langle \{\delta, \delta + 1\}, \mathit{usedAddrs}\ \mu o\rangle \\
\qquad\qquad \land\ \rho ot' = \mathit{update}(\rho ot, \langle \xi' \rangle, \{\xi \mapsto \mathit{opPFval}(l, locn)\}) \\
\qquad\qquad \land\ \mu o' = \mu o \oplus \{locn \mapsto \langle \delta, \delta + 1\rangle\} \bullet \\
\qquad \mathit{newLabelAddr}((\rho ot, \mu ot), (\xi, \xi'), l) = ((\rho ot', \mu ot'), \delta)
\end{array}
$$

newVarAddr returns an allocation of new data addresses suitable for storing a value, along with the update to the memory map environment.

$$
\begin{array}{l}
\mathit{newVarAddr} : (\mathit{EnvOTrace} \times \mathit{MO}) \times \mathit{ID} \times \mathit{IDTYPE} \pfun \\
\qquad \mathit{OPDENVALUE} \times \mathit{MO} \\
\hline
\forall \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO};\ \xi : \mathit{ID};\ i : \mathit{IDTYPE} \bullet \\
\quad \exists \mu o' : \mathit{MO};\ f : \mathrm{seq}\,\mathbb{N} \pinj \mathit{LOCN};\ \tau : \mathit{TYPE};\ A : \mathbb{P}\,\mathit{ATTR}; \\
\qquad\qquad\quad \delta : \mathit{ADDR} \mid \\
\qquad\qquad f = \mathit{allocate}(i, \mathit{opUsedLocations}\ \rho ot) \\
\qquad\qquad \land\ \mathrm{dom}\,\mu o' = \mathrm{ran}\,f \\
\qquad\qquad \land\ \tau = \mathit{typeOf}\ i \\
\qquad\qquad \land\ A = \mathit{attributeOf}\ i \\
\qquad\qquad \land\ (\mathrm{ran}\,\mathit{dataAt} \cap A \neq \varnothing \Rightarrow \mathit{dataAt}\ \delta \in A) \\
\qquad\qquad \land\ (\ \forall l : \mathrm{dom}\,\mu o';\ n : \mathbb{N} \mid n = l - \mathit{min}(\mathrm{dom}\,\mu o') \bullet \\
\qquad\qquad\quad (\mathit{sizeof}\ \tau = 1 \Rightarrow \mu o'\ l = \langle \delta + n\rangle) \\
\qquad\qquad\quad \land\ (\mathit{sizeof}\ \tau = 2 \Rightarrow \mu o'\ l = \langle \delta + 2 * n, \delta + 2 * n + 1\rangle)\ ) \\
\qquad\qquad \land\ \mathsf{disjoint}\langle \mathit{usedAddrs}\ \mu o', \mathit{usedAddrs}\ \mu o\rangle \bullet \\
\qquad \mathit{newVarAddr}((\rho ot, \mu o), \xi, i) = (\mathit{opLoc}\ f, \mu o')
\end{array}
$$

The conditions on f here are the same as those on the f defined in $\mathcal{M}_V$ in the specification of Pasp. The extra conditions ensure the following:

- $\mathrm{dom}\,\mu o' = \mathrm{ran}\,f$
  The function $\mu o'$ defines the mapping from abstract locations to concrete data addresses. The condition ensures all and only those abstract locations used have data addresses associated with them.

- $\operatorname{ran} dataAt \cap A \neq \varnothing \Rightarrow \ \delta \in A$
  If the variable has an *at* attribute, that required address is used for δ.

- $sizeof\ \tau = 1 \Rightarrow \ldots$
  If the type requires one data address, then μo comprised a contiguous block of data addresses, each corresponding to a location.

- $sizeof\ \tau = 2 \Rightarrow \ldots$
  If the type requires two data addresses, then μo comprised a contiguous block of data addresses, with two contiguous addresses corresponding to each location.

- disjoint ...
  The new data addresses are previously unused (this puts a constraint on the allowed values of δ.

8.2 Variable declaration

8.2.1 Variable declaration – allocating locations

The location trace environment is updated to assign conceptual locations to all the array entries, and the memory allocation trace environment is updated to assign data addresses.

$$
\begin{array}{|l}
\mathcal{O}_V : VarDecl \nrightarrow STACK \nrightarrow EnvOTrace \times MO \nrightarrow EnvOTrace \times MO \\
\hline
\forall\, VarDecl;\ B : STACK;\ \rho ot : EnvOTrace;\ \mu o : MO \bullet \\
\quad \exists\, i : IDTYPE;\ odv : OPDENVALUE;\ \mu o' : MO \mid \\
\qquad\qquad i = lookup(\xi, B, \rho\tau t0) \\
\qquad\qquad \wedge\ (odv, \mu o') = newVarAddr((\rho ot, \mu o), \xi, i) \bullet \\
\qquad\quad \mathcal{O}_V\ \theta VarDecl\ B(\rho ot, \mu o) = \\
\qquad\qquad (update(\rho ot, B, \{\xi \mapsto odv\}), \mu o \cup \mu o')
\end{array}
$$

8.2.2 Variable declaration – initialisation

A sequence of instructions that initialises the variable is produced

$$
\begin{array}{|l}
\mathcal{O}_{VI} : \mathit{VarDecl} \rightarrow\!\!\!\!\!\!+ \mathit{STACK} \rightarrow\!\!\!\!\!\!+ \mathit{EnvOTrace} \times \mathit{MO} \rightarrow\!\!\!\!\!\!+ \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \mathit{VarDecl};\ B : \mathit{STACK};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO} \mid V = \langle\ \rangle \bullet \\
\quad \mathcal{O}_{VI}\ \theta \mathit{VarDecl}\ B\ \rho\mu ot = \langle\ \rangle \\
\forall \mathit{VarDecl};\ B : \mathit{STACK};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MOV} \neq \langle\ \rangle \bullet \\
\quad \exists\, i : \mathit{IDTYPE};\ odv : \mathit{OPDENVALUE};\ NN : \operatorname{seq} \mathit{SubrangeN}; \\
\qquad\qquad \mathit{Iloc}, \mathit{Irep} : \operatorname{seq} \mathit{X\_INSTR};\ sL : \mathbb{N};\ \tau' : \mathit{TYPE} \mid \\
\qquad\qquad i = \mathit{lookup}(\xi, B, \rho\tau t0) \\
\qquad\qquad \wedge\ odv = \mathit{lookup}(\xi, B, \rho ot) \\
\qquad\qquad \wedge\ NN = \mathit{arrayOf}\ i \\
\qquad\qquad \wedge\ \mathit{Iloc} = \mathcal{O}_{Eaddr}(i, \mu o, odv, NN) \\
\qquad\qquad \wedge\ sL = \#(\operatorname{dom}(\mathit{opLoc}^{\sim}\ odv)) \\
\qquad\qquad \wedge\ \tau' = \mathit{typeOf}\ i \\
\qquad\qquad \wedge\ (\# V = 1 \Rightarrow \\
\qquad\qquad\qquad \mathit{Irep} = \mathcal{O}_{l\tau i}(\tau', \mathit{number}(\mathit{initValue}(V\ 1)B)) \\
\qquad\qquad\qquad\qquad \frown \frown/\{\ n : 1 \mathinner{.\,.} (sL - 1) \bullet \\
\qquad\qquad\qquad\qquad\qquad n \mapsto (\mathcal{O}_{s\tau}\ \tau' \frown \langle \mathit{immediate}(\mathtt{dpi}, \mathit{sizeof}\ \tau')\rangle)\ \}) \\
\qquad\qquad \wedge\ (1 < \# V \Rightarrow \\
\qquad\qquad\qquad \mathit{Irep} = \frown/\{\ n : 1 \mathinner{.\,.} (sL - 1) \bullet \\
\qquad\qquad\qquad\qquad\qquad n \mapsto (\mathcal{O}_{l\tau i}(\tau', \mathit{number}(\mathit{initValue}(V\ n)B)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \frown \mathcal{O}_{s\tau}\ \tau' \frown \langle \mathit{immediate}(\mathtt{dpi}, \mathit{sizeof}\ \tau')\rangle)\ \} \\
\qquad\qquad\qquad\qquad \frown \mathcal{O}_{l\tau i}(\tau', \mathit{number}(\mathit{initValue}(V\ sL)B))) \bullet \\
\qquad \mathcal{O}_{VI}\ \theta \mathit{VarDecl}\ B(\rho ot, \mu o) \\
\qquad\qquad = \mathit{Iloc} \frown \langle \mathit{indirect}\ \mathtt{daldab}\rangle \frown \mathit{Irep} \frown \mathcal{O}_{s\tau}\ \tau'
\end{array}
$$

Algorithm:

- case $\# V = 0$: no initialisation, so no code produced.
- *Iloc*: the memory location of the (start of) the variable
- τ': the type of the variable from the type environment (any subranges replaced by base type)
- case $\# V = 1, \# L = 1$: simple variable.
  - `daldab`: store address in D

  - $\mathcal{O}_{l\tau i}(V\ 1)$: immediate load initialisation value into $[a, b]$
  - $\mathcal{O}_{s\tau}$: store initialisation value

- case $\# V = 1, \#L > 1$: block array initialisation.
  - `daldab`: store address in D
  - $\mathcal{O}_{l\tau i}(V\ 1)$: immediate load initialisation value into $[a, b]$
  - repeat $\#L - 1$ times
    - $\mathcal{O}_{s\tau}$: store initialisation value in nth array location
    - `dpi`: increment D
  - $\mathcal{O}_{s\tau}$: store initialisation value in last location

- case $\# V > 1, \#L > 1$: full array initialisation.
  - `daldab`: store address in D
  - repeat $\#L - 1$ times
    - $\mathcal{O}_{l\tau i}(V\ n)$: immediate load nth initialisation value into $[a, b]$
    - $\mathcal{O}_{s\tau}$: store initialisation value in nth array location
    - `dpi`: increment D
  - $\mathcal{O}_{l\tau i}(V\ \#L)$: immediate load last initialisation value into $[a, b]$
  - $\mathcal{O}_{s\tau}$: store initialisation value in last location

8.3 Simple Declarations

8.3.1 Simple Declarations – allocating locations

Simple declarations can update the translation environment; only constant and variable declarations result in any change.

$$
\begin{array}{|l}
\mathcal{O}_{SD} : \mathit{SIMPLE\_DECL} \rightarrow\!\!\!\!+ \mathit{STACK} \rightarrow\!\!\!\!+ \\
\qquad\qquad \mathit{EnvOTrace} \times \mathit{MO} \rightarrow\!\!\!\!+ \mathit{EnvOTrace} \times \mathit{MO} \\
\hline
\forall \mathit{ConstDecl};\ B : \mathit{STACK};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO} \bullet \\
\quad \exists \rho ot' : \mathit{EnvOTrace} \mid \rho ot' = \mathit{update}(\rho ot, B, \{\xi \mapsto \mathit{opExport}\ \kappa\}) \bullet \\
\qquad \mathcal{O}_{SD}(\mathit{constDecl}\ \theta \mathit{ConstDecl}) B(\rho ot, \mu o) = (\rho ot', \mu o) \\
\forall \delta : \mathit{TYPE\_DEF};\ B : \mathit{STACK} \bullet \\
\quad \mathcal{O}_{SD}(\mathit{typeDecl}\ \delta) B = \mathrm{id}(\mathit{EnvOTrace} \times \mathit{MO}) \\
\forall \delta : \mathit{VarDecl} \bullet \mathcal{O}_{SD}(\mathit{varDecl}\ \delta) = \mathcal{O}_{V}\ \delta
\end{array}
$$

8.3.2 Simple Declarations – initialisation

Simple declarations can produce initialisation instructions; only variable declarations result in any code.

$$
\begin{array}{|l}
\mathcal{O}_{SDI} : \mathit{SIMPLE\_DECL} \rightarrow\!\!\!\!+ \mathit{STACK} \rightarrow\!\!\!\!+ \\
\qquad\qquad \mathit{EnvOTrace} \times \mathit{MO} \rightarrow\!\!\!\!+ \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall \delta : \mathit{ConstDecl};\ B : \mathit{STACK};\ \rho\tau ot : \mathit{EnvOTrace} \times \mathit{MO} \bullet \\
\quad \mathcal{O}_{SDI}(\mathit{constDecl}\ \delta) B\ \rho\tau ot = \langle\ \rangle \\
\forall \delta : \mathit{TYPE\_DEF};\ B : \mathit{STACK};\ \rho\tau ot : \mathit{EnvOTrace} \times \mathit{MO} \bullet \\
\quad \mathcal{O}_{SDI}(\mathit{typeDecl}\ \delta) B\ \rho\tau ot = \langle\ \rangle \\
\forall \delta : \mathit{VarDecl} \bullet \mathcal{O}_{SDI}(\mathit{varDecl}\ \delta) = \mathcal{O}_{VI}\ \delta
\end{array}
$$

Multiple declarations produce instructions that are the concatenation of the individual declarations' instructions.

$$
\begin{array}{|l}
\mathcal{O}_{SDI^*} : \operatorname{seq} \mathit{SIMPLE\_DECL} \rightarrow\!\!\!\!+ \\
\qquad \mathit{STACK} \rightarrow\!\!\!\!+ \mathit{EnvOTrace} \times \mathit{MO} \rightarrow\!\!\!\!+ \operatorname{seq} \mathit{X\_INSTR} \\
\hline
\forall B : \mathit{STACK};\ \rho\tau ot : \mathit{EnvOTrace} \times \mathit{MO} \bullet \mathcal{O}_{SDI^*} \langle\ \rangle B\ \rho\tau ot = \langle\ \rangle \\
\forall \delta : \mathit{SIMPLE\_DECL};\ \Delta : \operatorname{seq} \mathit{SIMPLE\_DECL};\ B : \mathit{STACK}; \\
\qquad\qquad \rho\tau ot : \mathit{EnvOTrace} \times \mathit{MO} \bullet \\
\quad \mathcal{O}_{SDI^*}(\langle \delta \rangle \frown \Delta) B\ \rho\tau ot = \mathcal{O}_{SDI}\ \delta\ B\ \rho\tau ot \frown \mathcal{O}_{SDI^*}\ \Delta\ B\ \rho\tau ot
\end{array}
$$

8.4 Routines

8.4.1 Parameters

The operational semantics of parameter declarations allocates a single location for the formal parameter (as in the dynamic semantics) together with an appropriate number of data addresses. For parameters that are called by value, the appropriate number is given by the size of the value's type. For parameters that are called by reference, the appropriate number is given by the size of a data address, which is 2.

$$
\begin{array}{|l}
\mathcal{O}_{PMD} : \mathit{ParamDecl} \nrightarrow \mathit{STACK} \nrightarrow \\
\qquad\qquad \mathit{EnvOTrace} \times \mathit{MO} \nrightarrow \mathit{EnvOTrace} \times \mathit{MO} \\
\hline
\forall \mathit{ParamDecl};\ B : \mathit{STACK};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO} \bullet \\
\quad \exists\, \mathit{VarType}';\ \mathit{odv} : \mathit{OPDENVALUE};\ \mu o' : \mathit{MO} \mid \\
\qquad\qquad A' = \varnothing \land SR' = \langle\ \rangle \land \tau a' = \mathit{unsigned} \\
\qquad\qquad \land (c = \mathit{ref} \Rightarrow \\
\qquad\qquad\quad \theta \mathit{SubrangeType}' = \\
\qquad\qquad\qquad \langle\!| \mathit{lb} == 0, \mathit{ub} == 0, \tau == \mathit{indirectAddr} |\!\rangle) \\
\qquad\qquad \land (c \neq \mathit{ref} \Rightarrow \\
\qquad\qquad\quad (\exists\, \mathit{FormalType}'' \mid \\
\qquad\qquad\qquad\qquad \theta \mathit{FormalType}'' = \\
\qquad\qquad\qquad\qquad\quad \mathit{formalParam}^{\sim}(\rho\tau t0(\mathit{last} B)\xi) \bullet \\
\qquad\qquad\qquad \theta \mathit{SubrangeType}' = \theta \mathit{SubrangeType}'')) \\
\qquad\qquad \land (\mathit{odv}, \mu o') = \\
\qquad\qquad\quad \mathit{newVarAddr}((\rho ot, \mu o), \xi, \mathit{variable}\ \theta \mathit{VarType}') \bullet \\
\qquad \mathcal{O}_{PMD}\ \theta \mathit{ParamDecl}\ B(\rho ot, \mu o) \\
\qquad\qquad = (\mathit{update}((\rho ot, \mu o), B, \{\xi \mapsto \mathit{odv}\})\mu o \cup \mu o')
\end{array}
$$

This definition is extended in the usual inductive fashion to sequences of parameter declarations.

$$
\begin{array}{|l}
\mathcal{O}_{PMD^*} : \mathrm{seq}\, \mathit{ParamDecl} \nrightarrow \mathit{STACK} \nrightarrow \\
\qquad\qquad \mathit{EnvOTrace} \times \mathit{MO} \nrightarrow \mathit{EnvOTrace} \times \mathit{MO} \\
\hline
\forall B : \mathit{STACK} \bullet \mathcal{O}_{PMD^*} \langle\ \rangle B = \mathrm{id}(\mathit{EnvOTrace} \times \mathit{MO}) \\
\forall \pi : \mathit{ParamDecl};\ \Pi : \mathrm{seq}\, \mathit{ParamDecl};\ B : \mathit{STACK} \bullet \\
\quad \mathcal{O}_{PMD^*}(\langle \pi \rangle \frown \Pi) B = \mathcal{O}_{PMD^*}\ \Pi\ B \circ \mathcal{O}_{PMD}\ \pi\ B
\end{array}
$$

8.4.2 Body

The procedure body is mapped to the translation of the variable initialisations and the body statement. These is calculated with an environment updated by the declarations of the local variables. The correct offset from the top of the expression stack is passed as an argument.

$$
\begin{array}{|l}
\mathcal{O}_B : \mathit{Body} \nrightarrow \\
\qquad\qquad \mathit{STACK} \nrightarrow \mathit{LABEL} \nrightarrow \mathit{EnvOTrace} \times \mathit{MO} \nrightarrow \mathit{OFFSET} \nrightarrow \\
\qquad\qquad\qquad (\mathit{EnvOTrace} \times \mathit{MO}) \times (\mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR}) \\
\hline
\forall \mathit{Body};\ B : \mathit{STACK};\ l : \mathit{LABEL};\ \rho\mu ot : \mathit{EnvOTrace} \times \mathit{MO}; \\
\qquad\qquad \delta : \mathit{OFFSET} \bullet \\
\qquad \exists\, \rho\mu ot' : \mathit{EnvOTrace} \times \mathit{MO};\ l' : \mathit{LABEL};\ \mathit{Ii}, \mathit{Ib} : \operatorname{seq} \mathit{X\_INSTR} \mid \\
\qquad\qquad \rho\mu ot' = \mathcal{O}_{SD^*}\ \Delta\ B\ \rho\mu ot \\
\qquad\qquad \wedge\ \mathit{Ii} = \mathcal{O}_{SDI^*}\ \Delta\ B\ \rho\mu ot' \\
\qquad\qquad \wedge\ (l', \mathit{Ib}) = \mathcal{O}_S\ \gamma\ B\ \rho\mu ot'\ l\ \delta \bullet \\
\qquad\quad \mathcal{O}_B\ \theta \mathit{Body}\ B\ l\ \rho\mu ot\ \delta = (\rho\mu ot', (l', \mathit{Ii} \frown \mathit{Ib}))
\end{array}
$$

8.4.3 Procedures

We are now ready to define the operational semantics of procedure declarations. The label for the program address of the first instruction is passed as an argument to this semantics so that it can be stored in the environment and used to label the instructions used to translate the procedure. When the procedure is called, a pair of data addresses are allocated to store contents of the C register – the return address. The environment is updated to reserve these addresses, and location used. The body template is followed by a fragment to jump to the return address. We assume here that the return address is modelled by an unsigned integer (ie a label).

$$
\begin{array}{l}
\mathcal{O}_{PD} : \mathit{ProcDecl} \pfun \\
\qquad \mathit{STACK} \pfun \mathit{LABEL} \pfun \mathit{EnvOTrace} \times \mathit{MO} \pfun \\
\qquad\qquad (\mathit{EnvOTrace} \times \mathit{MO}) \times (\mathit{LABEL} \times \operatorname{seq} \mathit{X\_INSTR}) \\
\hline
\forall \mathit{ProcDecl};\ B : \mathit{STACK};\ l : \mathit{LABEL};\ \rho ot : \mathit{EnvOTrace};\ \mu o : \mathit{MO} \bullet \\
\quad \exists \rho\mu ot', \rho\mu ot'', \rho\mu ot''' : \mathit{EnvOTrace} \times \mathit{MO};\ l' : \mathit{LABEL}; \\
\qquad\quad I : \operatorname{seq} \mathit{X\_INSTR};\ \delta : \mathit{OFFSET} \mid \\
\qquad\qquad \rho\mu ot' = \mathcal{O}_{PMD^*}\ \Pi(B \frown \langle \xi \rangle)(\rho ot \oplus \{\xi \mapsto \varnothing\}, \mu o) \\
\qquad\qquad \wedge (\rho\mu ot'', (l', I)) = \mathcal{O}_B\ \beta(B \frown \langle \xi \rangle)(l+1)\rho\mu ot'(\rho st0\ \xi) \\
\qquad\qquad \wedge (\rho\mu ot''', \delta) = \mathit{newLabelAddr}(\rho\mu ot'', (\xi, \mathit{last}\ B), l) \bullet \\
\qquad \mathcal{O}_{PD}\ \theta \mathit{ProcDecl}\ B\ l\ (\rho ot, \mu o) = \\
\qquad\qquad (\rho\mu ot''', (l', \\
\qquad\qquad\qquad \langle \mathit{xProc}(\xi, l) \rangle \\
\qquad\qquad\qquad \frown \langle \mathit{xData}(\mathit{xsdrh}, \delta) \rangle \frown \langle \mathit{indirect}\ \texttt{cstd} \rangle \frown I \\
\qquad\qquad\qquad \frown \langle \mathit{xData}(\mathit{xsdrh}, \delta) \rangle \frown \langle \mathit{indirect}\ \texttt{abldd} \rangle \\
\qquad\qquad\qquad \frown \langle \mathit{xIndirect}\ \mathit{xag} \rangle))
\end{array}
$$

Algorithm:

- $\mathit{xProc}(\xi, l)$: the label l corresponding to the start of procedure ξ, used by the linker
- (xsdrh, δ): set D to $\delta + \mathit{bottomHeap}$. (The return label in MO is stored relative to $\mathit{bottomHeap}$, for δ the actual position on the heap is $\mathit{bottomHeap} + \delta$. New heap addresses are to store the label.)
- `cstd`: store return label from C to $\delta + \mathit{bottomHeap}$. (The label to return to is stored in the C register just before the procedure is called. This label is not necessarily known at compilation time, as the caller could be in a different module. The linker resolves labels across modules.)
- I: procedure body template
- (xsdrh, δ): set D to $\delta + \mathit{bottomHeap}$
- `abldd`: load return label from $\delta + \mathit{bottomHeap}$ into $[a, b]$
- xag: jump to label stored in $[a, b]$

8.4.4 Functions

The operational semantics of function declarations is identical to that of procedure declarations, apart for the allocation of memory for the return value. This value is stored in the C register before the function completes, and returns. The caller then retrieves the value from the C register, and puts it into the a and b registers.

The function has to retrieve the value to store in the C register. This is achieved similarly to how the value was previously retrieved by the caller. This is done by using the $EnvOTrace$ and MO variables ($\rho ot''$, and $\mu o''$) which have the correct information.

$$
\begin{array}{|l}
\mathcal{O}_{FD} : FunDecl \mathrel{+\!\!\!\to} \\
\qquad\qquad STACK \mathrel{+\!\!\!\to} LABEL \mathrel{+\!\!\!\to} EnvOTrace \times MO \mathrel{+\!\!\!\to} \\
\qquad\qquad\qquad (EnvOTrace \times MO) \times (LABEL \times \operatorname{seq} X\_INSTR) \\
\hline
\forall\, FunDeclB : STACK;\ l : LABEL;\ \rho ot : EnvOTrace;\ \mu o : MO;\ \bullet \\
\quad \exists\, FunType';\ VarType'; \\
\qquad\qquad\qquad \rho\mu ot', \rho\mu ot''', \rho\mu ot'''' : EnvOTrace \times MO; \\
\qquad\qquad\qquad \rho ot'' : EnvOTrace;\ \mu o', \mu o'' : MO; \\
\qquad\qquad\qquad odv : OPDENVALUE; \\
\qquad\qquad\qquad l' : LABEL;\ I : \operatorname{seq} X\_INSTR;\ \delta, \delta' : OFFSET \mid \\
\qquad\qquad \rho\mu ot' = \mathcal{O}_{PMD^*}\ \Pi(B \frown \langle \xi \rangle)(\rho ot \oplus \{\xi \mapsto \varnothing\}, \mu o) \\
\qquad\qquad \wedge\ \theta FunType' = function^{\sim}(\rho\tau t0(last\ B)\xi) \\
\qquad\qquad \wedge\ A' = \varnothing \wedge SR' = \varnothing \wedge \tau a' = unsigned \\
\qquad\qquad \wedge\ (odv, \mu o') = newVarAddr(\rho\mu ot', \xi, variable\ \theta VarType) \\
\qquad\qquad \wedge\ (\rho ot'', \mu o'') = (update(\rho\mu ot', \langle \xi \rangle, \{\xi \mapsto odv\}), \mu o \cup \mu o') \\
\qquad\qquad \wedge\ (\rho\mu ot''', (l', I)) = \\
\qquad\qquad\qquad \mathcal{O}_B\ \theta Body(B \frown \langle \xi \rangle)(l+1)(\rho ot'', \mu o'')(\rho st0\ \xi) \\
\qquad\qquad \wedge\ (\rho\mu ot'''', \delta') = newLabelAddr(\rho\mu ot''', (\xi, last\ B), l+1) \\
\qquad\qquad \wedge\ \delta = (head \circ \mu o''\ \xi \circ opLoc^{\sim}(\rho ot''\ \xi\ \xi))\langle\ \rangle\ \bullet \\
\qquad \mathcal{O}_{FD}\ \theta FunDecl\ B\ l(\rho ot, \mu o) = \\
\qquad\qquad (\rho\mu ot'''', (l', \\
\qquad\qquad\qquad\quad \langle xProc(\xi, l) \rangle \\
\qquad\qquad\qquad\quad \frown \langle xData(xsdrh, \delta') \rangle \frown \langle indirect\ \texttt{cstd} \rangle \frown I \\
\qquad\qquad\qquad\quad \frown \langle xData(xsdrh, \delta) \rangle \frown \langle indirect\ \texttt{cldd} \rangle \\
\qquad\qquad\qquad\quad \frown \langle xData(xsdrh, \delta') \rangle \frown \langle indirect\ \texttt{abldd} \rangle \\
\qquad\qquad\qquad\quad \frown \langle xIndirect\ xag \rangle))
\end{array}
$$

8.4.5 Routines

In order to use the generic extensions of semantic functions to sequences, we define an operational semantics for procedure/function declarations.

$$\begin{array}{l}
\mathcal{O}_{PFD} : PROC\_FUN\_DECL \rightarrowtail \\
\qquad STACK \rightarrowtail LABEL \rightarrowtail EnvOTrace \times MO \rightarrowtail \\
\qquad\qquad (EnvOTrace \times MO) \times (LABEL \times \mathrm{seq}\, X\_INSTR) \\
\hline
\forall \delta : ProcDecl \bullet \mathcal{O}_{PFD}(procDecl\ \delta) = \mathcal{O}_{PD}\ \delta \\
\forall \delta : FunDecl \bullet \mathcal{O}_{PFD}(funDecl\ \delta) = \mathcal{O}_{FD}\ \delta
\end{array}$$

8.5 Import declarations

Import declarations generate no code. The linker fixes up the addresses when the imported declarations are referenced in the code. Constant and variable import declarations change the environment.

$$\begin{array}{l}
\mathcal{O}_{ID} : IMPORT\_DECL \rightarrowtail EnvO \times MO \rightarrowtail EnvO \times MO \\
\hline
\forall \xi : ID;\ \tau : TYPE;\ \rho o : EnvO;\ \mu o : MO \bullet \\
\qquad \mathcal{O}_{ID}(constHdr(\xi, \tau))(\rho o, \mu o) = (\rho o \oplus \{\xi \mapsto opImport\ \xi\}, \mu o) \\
\forall\ VarDecl;\ \rho o : EnvO;\ \mu o : MO \bullet \\
\qquad \mathcal{O}_{ID}(varHdr\ \theta VarDecl)(\rho o, \mu o) = (\rho o \oplus \{\xi \mapsto opImport\ \xi\}, \mu o) \\
\forall \delta : ProcHde \bullet \mathcal{O}_{ID}(procHdr\ \delta) = \mathrm{id}(EnvO \times MO) \\
\forall \delta : FunHdr \bullet \mathcal{O}_{ID}(funcHdr\ \delta) = \mathrm{id}(EnvO \times MO)
\end{array}$$

8.6 Multiple declarations

Translating an empty declaration list has no effect on the environment. Translating a list of declarations is done by translating the head of the list (thereby changing the environment), and then translating the tail of the list in the new environment.

$$\begin{array}{l}
\mathcal{O}_{SD^*} : \mathrm{seq}\, SIMPLE\_DECL \rightarrowtail \\
\qquad STACK \rightarrowtail EnvOTrace \times MO \rightarrowtail EnvOTrace \times MO \\
\hline
\forall B : STACK \bullet \mathcal{O}_{SD^*}\langle\ \rangle B = \mathrm{id}(EnvOTrace \times MO) \\
\forall \delta : SIMPLE\_DECL;\ \Delta : \mathrm{seq}\, SIMPLE\_DECL;\ B : STACK \bullet \\
\qquad \mathcal{O}_{SD^*}(\langle\delta\rangle \frown \Delta)B = \mathcal{O}_{SD^*}\ \Delta\ B \circ \mathcal{O}_{SD}\ \delta\ B
\end{array}$$

$$
\begin{array}{|l}
\mathcal{O}_{PFD*} : \operatorname{seq} PROC\_FUN\_DECL \mathrel{+\!\!\!\rightarrow} \\
\qquad STACK \mathrel{+\!\!\!\rightarrow} LABEL \mathrel{+\!\!\!\rightarrow} EnvOTrace \times MO \mathrel{+\!\!\!\rightarrow} \\
\qquad\qquad (EnvOTrace \times MO) \times (LABEL \times \operatorname{seq} X\_INSTR) \\
\hline
\forall B : STACK;\ l : LABEL;\ \rho\mu ot : EnvOTrace \times MO \bullet \\
\quad \mathcal{O}_{PFD*}\langle\ \rangle B\ l\ \rho\mu ot = (\rho\mu ot, (l, \langle\ \rangle)) \\
\forall \Delta : \operatorname{seq} PROC\_FUN\_DECL;\ \delta : PROC\_FUN\_DECL;\ B : STACK; \\
\qquad l : LABEL;\ \rho\mu ot : EnvOTrace \times MO \bullet \\
\quad \exists \rho\mu ot', \rho\mu ot'' : EnvOTrace \times MO;\ l', l'' : LABEL; \\
\qquad\qquad I, I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\quad (\rho\mu ot', (l', I)) = \mathcal{O}_{PFD}\ \delta\ B\ l\ \rho\mu ot \\
\qquad\quad \wedge (\rho\mu ot'', (l'', I')) = \mathcal{O}_{PFD*}\ \Delta\ B\ l'\ \rho\mu ot' \bullet \\
\qquad \mathcal{O}_{PFD*}(\langle\delta\rangle \frown \Delta) B\ l\ \rho\mu ot = (\rho\mu ot'', (l'', I \frown I'))
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{O}_{ID*} : \operatorname{seq} IMPORT\_DECL \mathrel{+\!\!\!\rightarrow} EnvO \times MO \mathrel{+\!\!\!\rightarrow} EnvO \times MO \\
\hline
\mathcal{O}_{ID*}\langle\ \rangle = \operatorname{id}(EnvO \times MO) \\
\forall \delta : IMPORT\_DECL;\ \Delta : \operatorname{seq} IMPORT\_DECL \bullet \\
\quad \mathcal{O}_{ID*}(\langle\delta\rangle \frown \Delta) = \mathcal{O}_{ID*}\ \Delta \circ \mathcal{O}_{ID}\ \delta
\end{array}
$$

9 Modules and Programs

9.1 Utility Functions

The addition of *blockOffset* functions element-wise is required, so we define $\boxplus$ to do this.

$$\textbf{function}\,30\,\textbf{leftassoc}(\_ \boxplus \_)$$

$$\begin{array}{|l}
\_ \boxplus \_ : \mathit{blockOffset} \times \mathit{blockOffset} \nrightarrow \mathit{blockOffset} \\
\hline
\forall f, g : \mathit{blockOffset} \mid \mathrm{dom}\, f = \mathrm{dom}\, g \bullet \\
\qquad f \boxplus g = (\lambda\, b : \mathrm{dom}\, f \bullet f\ b + g\ b)
\end{array}$$

9.2 Address allocation

9.2.1 Case statement addresses

$\mathcal{O}_{\mathit{Env}}$ allocates locations and memory addresses for the temporary variables associated with case statements that have no explicit declaration; locations and addresses are assigned by extracting information from the type trace environment.

$$
\begin{array}{|l}
\mathcal{O}_{Env} : EnvTTrace \pfun EnvOTrace \times MO \\
\hline
\forall \rho\tau t : EnvTTrace \bullet \\
\quad \exists \rho ot : EnvOTrace;\ \mu o : MO \mid \\
\qquad\qquad \operatorname{dom} \rho ot = \operatorname{dom} \rho\tau t \\
\qquad\qquad \land (\ \forall b, b' : \operatorname{dom} \rho ot \bullet \\
\qquad\qquad\quad \operatorname{dom}(\rho ot\ b) = \operatorname{dom}(\rho\tau t\ b \rhd \operatorname{ran} tempVar) \\
\qquad\qquad\quad \land (\ \forall \xi : \operatorname{dom}(\rho ot\ b);\ \xi' : \operatorname{dom}(\rho ot\ b') \bullet \\
\qquad\qquad\qquad (\ \exists l : LOCN \bullet \rho ot\ b\ \xi = opLoc\{\langle\ \rangle \mapsto l\}\ ) \\
\qquad\qquad\qquad \land (\rho ot\ b\ \xi = \rho ot\ b'\ \xi' \Leftrightarrow b' = b \land \xi = \xi')\ ) \\
\qquad\qquad\quad \land \operatorname{dom} \mu o = \operatorname{ran}(\bigcup(opLoc^{\sim}(\!|\ (\operatorname{ran} \circ \bigcup \circ \operatorname{ran})\rho ot\ |\!))) \\
\qquad\qquad\quad \land (\ \forall \xi : \operatorname{dom}(\rho ot\ b) \bullet \\
\qquad\qquad\qquad \exists l : \operatorname{dom} \mu o;\ \delta : ADDR \mid \\
\qquad\qquad\qquad\qquad l = opLoc^{\sim}\ (\rho ot\ b\ \xi)\langle\ \rangle \bullet \\
\qquad\qquad\qquad\quad \mu o\ l = \langle \delta \rangle\ ) \\
\qquad\qquad\quad \land (\ \forall l, l' : \operatorname{dom} \mu o \bullet \mu o\ l = \mu o\ l' \Leftrightarrow l = l'\ )\ ) \bullet \\
\qquad\quad \mathcal{O}_{Env}\ \rho\tau t = (\rho ot, \mu o)
\end{array}
$$

This loose specification constructs from a type trace environment, an *EnvOTrace* environment and *MO* map with the following properties:

- The environment is defined for precisely the same blocks.
- The *EnvOTrace* environment is defined for precisely the temporary case variables.
- Each temporary variable is mapped to a single location.
- These locations are disjoint.
- The *MO* map for each block is defined for precisely the locations given in the *EnvOTrace*.
- Each location is mapped to a single data address.
- These addresses are disjoint.

9.3 Module address allocation

The function *addrs* takes a mapping from array indexes to abstract locations (that is, information about all the abstract locations used to store the array), and a memory allocation map (from locations to data addresses), and returns the set of all data addresses mapped to by the supplied locations (that is, all the data addresses used to store the array).

$$addrs : (\mathrm{seq}\,\mathbb{N} \rightarrowtail\!\!\!\!\!\rightarrow LOCN) \times MO \rightarrow \mathbb{P}\, ADDR$$

$$\forall f : \mathrm{seq}\,\mathbb{N} \rightarrowtail\!\!\!\!\!\rightarrow LOCN;\ \mu o : MO \bullet$$
$$\quad addrs(f, \mu o) = \mathrm{ran}(\bigcup(\mathrm{ran}(\mu o \circ f)))$$

moduleAddr takes an operational trace environment and memory allocation map, and returns an 'AT' variable data address map. It also puts some constraints on some global variables to do with memory allocation.

$$
\begin{array}{|l}
moduleAddr : EnvOTrace \times MO \pfun \mathbb{P}\, ATTR \pfun \mathbb{P}\, ADDR \\
\hline
\forall \rho ot : EnvOTrace;\ \mu o : MO \bullet \\
\quad \exists\, \alpha 1, \alpha 2 : \mathbb{P}\, ADDR;\ atMap : ID \times ID \times \mathbb{P}\, ATTR \pfun \mathbb{P}\, ADDR; \\
\qquad\qquad atMemMap : \mathbb{P}\, ATTR \pfun \mathbb{P}\, ADDR \mid \\
\qquad\quad atMap = \\
\qquad\qquad \{\ b : \operatorname{dom} \rho\tau t0;\ \xi : ID;\ \delta : ADDR;\ A, A' : \mathbb{P}\, ATTR \mid \\
\qquad\qquad\qquad \xi \in \operatorname{dom}(\rho\tau t0\ b) \\
\qquad\qquad\qquad \land A = attributeOf(\rho\tau t0\ b\ \xi) \\
\qquad\qquad\qquad \land dataAt\ \delta \in A \\
\qquad\qquad\qquad \land A' = A \cap \{readOnly, writeOnly, nvram\} \bullet \\
\qquad\qquad\quad (b, \xi, A') \mapsto addrs(opLoc^{\sim}(\rho ot\ b\ \xi), \mu o)\ \} \\
\qquad\quad \land (\ \forall\, b, \xi : ID;\ A, A' : \mathbb{P}\, ATTR \mid \\
\qquad\qquad\quad (b, \xi, A') \in \operatorname{dom} atMap \\
\qquad\qquad\quad \land A = attributeOf(\rho\tau t0\ b\ \xi) \bullet \\
\qquad\qquad dataAt(min(atMap(b, \xi, A'))) \in A\ ) \\
\qquad\quad \land \mathsf{disjoint}\ atMap \\
\qquad\quad \land atMemMap = \\
\qquad\qquad \{\ A' : \{\varnothing, \{readOnly\}, \{writeOnly\}, \{nvram\}\} \bullet \\
\qquad\qquad\quad a' \mapsto \bigcup \{\ b, \xi : ID \mid (b, \xi, A') \in \operatorname{dom} atMap \bullet \\
\qquad\qquad\qquad\qquad atMap(b, \xi, A')\ \}\ \} \\
\qquad\quad \land \alpha 1 = usedAddresses\ \mu o \setminus \bigcup(\operatorname{ran} atMemMap) \\
\qquad\quad \land \alpha 2 = \operatorname{ran}(\bigcup(\operatorname{ran} assignAddr)) \\
\qquad\quad \land \operatorname{dom} assignAddr = \operatorname{dom} \rho ot \\
\qquad\quad \land \mathsf{disjoint}\langle \alpha 1, \alpha 2 \rangle \\
\qquad\quad \land (\ \forall\, b : \operatorname{dom} assignAddr \bullet \\
\qquad\qquad \exists\, \delta : ADDR \bullet assignAddr\ b = \langle \delta, \delta + 1 \rangle\ ) \\
\qquad\quad \land topVar > max\ (\alpha 1\ \cup \alpha 2) \bullet \\
\qquad moduleAddr(\rho ot, \mu o) = atMemMap
\end{array}
$$

- $atMap = \ldots$: a mapping of all the 'AT' variables, from (block, variable name, attributes) to the set of data addresses used to store that variable. (This is one or two addresses for a simple variable, several for an array variable. The addresses are contiguous.)
- $dataAt \ldots \in A$: each 'AT' variable starts at the requested absolute address (the linker does not change these addresses).

- $\mathsf{disjoint}\ atMap$: the 'AT' data addresses do not overlap. (This check includes all the data addresses involved, not just the start ones explicitly mentioned in the 'AT' declaration.)
- $atMemMap = \ldots$: a mapping from each kind of 'AT' attribute (plain 'AT', read only 'AT', write only 'AT', and nvram 'AT') to all the data addresses requested for that attribute.
- $\alpha 1$: the relative addresses used by all the user variables (ordinary, case statement, but not 'AT')
- $\alpha 2$: the 'assignment addresses' (used in the *assign* template)
- $\mathrm{dom}\ assignAddr$: there are assignment addresses for precisely the declared blocks
- $\mathsf{disjoint}$: The assignment addresses are distinct from the user addresses.
- There are two contiguous assignment addresses per block
- $topVar$: the user addresses and assignment addresses are below $topVar$ (are all in the heap).

9.4 Compiled module

A *CompModule* is the output from the compilation of a module. It is used as the input to the linker, which requires not just the code, but also enough information to do cross-module checking, and to resolve cross-module jumps.

The *CompModule* consists of the name of the module (an identifier), the compiled sequence of instructions, the size of the memory required (heap and stack combined), the addresses used as fixed 'AT' addresses in the module, the declaration and type environments, the operation trace environment, and the memory allocation map.

$$
\begin{array}{|l}
\hline
\textit{CompModule} \\
\xi : \textit{ID} \\
I : \operatorname{seq} \textit{X\_INSTR} \\
\textit{size} : \mathbb{N} \\
\alpha : \mathbb{P}\,\textit{ATTR} \mapsto \mathbb{P}\,\textit{ADDR} \\
\rho\delta : \textit{EnvD} \\
\rho\tau : \textit{EnvT} \\
\rho ot : \textit{EnvOTrace} \\
\mu o : \textit{MO} \\
\hline
\end{array}
$$

9.5 Module

To compile a module, the various stack amounts are calculated, the declarations are compiled to I, the simple declarations' initialisations are compiled to Ii, and various environments are computed.

$$
\begin{array}{|l}
\mathcal{O}_M : \textit{Module} \mapsto \textit{CompModule} \\
\hline
\forall\, \textit{Module} \bullet \\
\quad \exists\, \rho sa : \textit{blockOffset};\ \textit{CompModule}';\ \rho ot : \textit{EnvOTrace}; \\
\qquad\qquad I, Ii : \operatorname{seq} \textit{X\_INSTR};\ l : \textit{LABEL} \mid \\
\qquad\quad \rho st0 = \mathcal{S}_M\ \theta \textit{Module} \land \rho sa = \mathcal{A}_M\ \theta \textit{Module} \\
\qquad\quad \land\ ((\rho ot, \mu o'), (l, I)) = \\
\qquad\qquad \mathcal{O}_{PFD^*}\ \Delta PF \langle \xi \rangle 1 (\mathcal{O}_{SD^*}\ \Delta SD \langle \xi \rangle (\mathcal{O}_{Env}\ \rho\tau t0)) \\
\qquad\quad \land\ \rho ot' = \textit{update}(\rho ot, \langle \xi \rangle, \textit{first}(\mathcal{O}_{ID^*}\ \Delta I(\varnothing, \varnothing))) \\
\qquad\quad \land\ Ii = \mathcal{O}_{SDI^*}\ \Delta SD \langle \xi \rangle (\rho ot', \mu o') \\
\qquad\quad \land\ I' = \langle \textit{xLabel}(\textit{xrg}, l) \rangle \frown I \frown \langle \textit{xLabel}(\textit{xli}, l) \rangle \frown Ii \\
\qquad\quad \land\ \alpha' = \textit{moduleAddr}(\rho ot', \mu o') \\
\qquad\quad \land\ \textit{size}' = \textit{topVar} + \textit{max}(\operatorname{ran}(\rho sa \boxplus \rho st0)) \\
\qquad\quad \land\ \rho\delta' = \mathcal{D}_{ML}\ \theta \textit{Module} \\
\qquad\quad \land\ \rho\tau' = \mathcal{T}_{ML}\ \theta \textit{Module} \bullet \\
\qquad \mathcal{O}_M\ \theta \textit{Module} = \theta \textit{CompModule}'
\end{array}
$$

- ρst : the stack set-up semantics, globally accessible to the module
- ρsa : the stack amount set-up semantics

- $((\rho ot, \mu o'), (l, I))$: the environments, next label, and instructions that result from compiling a module's declarations in the initial 'case statement' environments. (Recall that the environment stores *relative* addresses of non-'AT' variables, relative to the bottom of the heap, to be fixed up by the linker.) Each module has a contiguous sequence of label numbers starting from 1 (these are renumbered when the linking occurs to ensure all the labels are unique).
- $\rho ot'$: include any imported variables in the environment.
- Ii : simple declaration initialisation instructions
- I : jump around declarations, and perform initialisation
- α' : the 'AT' variable data addresses

The structure of the generated code is

- jump to label l
- code for procedures and functions I
- label l
- code to initialise simple declarations Ii

9.6 Main Module

The main module template is specified similarly to the standard module. It has a further sequence of compiled instructions I', for the body statement.

$$
\begin{array}{|l}
\mathcal{O}_{MA} : \mathit{MainModule} \rightarrowtail\!\!\!\!\rightarrow \mathit{CompModule} \\
\hline
\forall\, \mathit{MainModule} \bullet \\
\quad \exists\, \rho sa : \mathit{blockOffset};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mathit{CompModule}'; \\
\qquad\qquad I, \mathit{Ii}, \mathit{Io} : \mathrm{seq}\, X\_\mathit{INSTR};\ l, l' : \mathit{LABEL};\ | \\
\qquad\qquad \rho st0 = \mathcal{S}_{MA}\ \theta \mathit{MainModule} \\
\qquad\qquad \wedge\ \rho sa = \mathcal{A}_{MA}\ \theta \mathit{MainModule} \\
\qquad\qquad \wedge\ \xi' = \xi \\
\qquad\qquad \wedge\ ((\rho ot, \mu o'), (l, I)) = \\
\qquad\qquad\qquad \mathcal{O}_{PFD^*}\ \Delta PF \langle \xi \rangle 1 (\mathcal{O}_{SD^*}\ \Delta SD\ \langle \xi \rangle (\mathcal{O}_{Env}\ \rho\tau t0)) \\
\qquad\qquad \wedge\ \rho ot' = \mathit{update}(\rho ot, \langle \xi \rangle, \mathit{first}(\mathcal{O}_{ID^*}\ \Delta I(\varnothing, \varnothing))) \\
\qquad\qquad \wedge\ \mathit{Ii} = \mathcal{O}_{SDI^*}\ \Delta SD \langle \xi \rangle (\rho ot', \mu o') \\
\qquad\qquad \wedge\ (l', \mathit{Io}) = \mathcal{O}_S\ \gamma \langle \xi \rangle (\rho ot', \mu o') l\ \mathit{topStack} \\
\qquad\qquad \wedge\ I' = \langle \mathit{xLabel}(\mathit{xrg}, l') \rangle \frown I \frown \langle \mathit{xLabel}(\mathit{xli}, l') \rangle \frown \mathit{Ii} \frown \mathit{Io} \\
\qquad\qquad \wedge\ \alpha' = \mathit{moduleAddr}(\rho ot', \mu o') \\
\qquad\qquad \wedge\ \mathit{size}' = \mathit{topVar} + \mathit{max}(\mathrm{ran}(\rho sa \boxplus \rho st0)) \\
\qquad\qquad \wedge\ \rho\delta' = \mathcal{D}_{MAL}\ \theta \mathit{MainModule} \\
\qquad\qquad \wedge\ \rho\tau' = \mathcal{T}_{MAL}\ \theta \mathit{MainModule} \bullet \\
\qquad \mathcal{O}_{MA}\ \theta \mathit{MainModule} = \theta \mathit{CompModule}'
\end{array}
$$

The structure of the generated code is

- jump to label l'
- code for procedures and functions I
- label l'
- code to initialise simple declarations Ii
- code for body statement I'

9.7 Program

The operational semantics of a complete Pasp program is not defined in this document; it is given by the linker.

A Operator Optimisation

A.1 Introduction

This appendix discusses the validity of changes to the operator templates, and the global optimisation of *umul* and *udiv*.

The approach behind the optimisation is to remove the inline substitution of the templates for each call to one of the operators. The templates are replaced by specially optimised templates which are positioned at a fixed place in the memory. When a call to one of the operators is made, a function style call is made to the appropriate special template.

The function style call puts the two operands, and the label to return to in fixed locations in the operator scratchpad and then calls the operator template. This call is made by using unique labels to identify the beginning of each of the operators. A special XAspAL instruction is used by the compiler, because at the linking stage, conditional linking of these operators is performed so that only the templates for those operators called are included.

The return value from the operation is passed back from the optimised templates in the C register. This value is then recovered to the a and b registers and the operation is complete, as this emulates the operation of the original template.

A.2 Template algorithm

The algorithm used by the templates to calculate their operation is identical to previously. The optimisations possible are provided by the new Asp5 instructions to step the data address register back, as well as perform multiple operations in a single instruction (eg storing a word to memory locations pointed to by the data address, as well as multiplication and division). Each of these Asp instructions has been specified, and their operation can be shown to be equivalent to the sequence of previously used Asp instructions.

A.3 Function-Call algorithm

The function calling algorithm used for the operators is the same as for general function calls. The operands are put onto the scratchpad rather than into calculated positions in the heap, and the return label is put onto the scratchpad rather than passed into the function body in the C register. The actual XAspAL instructions used for the call are different, however this is purely used as a flag for the linker, and the instructions passed to the hexer are identical.

The calling template puts the RIGHT hand operator in the scratchpad addresses 16,17 (usually from the a and b registers) in the standard lo-byte, hi-byte format. The LEFT hand operator (usually from $addr$, $addr + 1$) is in 18,19, and the label within the calling template, for the optimised template return to, is put in addresses 20,21. (The case for $udiv$ is different.)

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