Towards a Semantics For Multidimensional Array Assignment in Fortran 95

N. Reid, P. Kilpatrick, M. Clint
School of Computer Science
The Queen’s University of Belfast
Northern Ireland

Abstract

Fortran is in its sixth decade of use, with little to show in the way of support for formal approaches to establishing program correctness. Fortran 95, the current dialect, exhibits parallelism through data-parallel assignment. This paper presents an approach to the formalisation of the data-parallel assignment aspects of the language, by providing a model of multidimensional arrays and data-parallel assignment over those arrays, utilising the VDM-SL map type.

1 Introduction

It is widely acknowledged that it is desirable to reason about the correctness of programs, most especially those of a safety critical nature. To this end the use of formal software specification techniques have been developed which facilitate the expansion of provably correct specifications.

Ultimately, though, formally specified software must be implemented. If the implementation is to be proved to meet its specification then the semantics of the implementation language must be formally defined.

Fortran is now in its sixth decade of use - its most recent dialect, Fortran 95, having been released in 1997. The fact that all but the concrete syntax of Fortran 95 is given in English has led to a situation in which both users and implementers can, and do, interpret the informal descriptions of features in different ways.

As one of the longest established and widely used languages it is ironic that Fortran 95 has been ignored by semantic theorists. In general the ‘new’ features of later dialects have been taken from other languages’ features: for example, the block structured nature of Pascal was described formally in 1986 [Bjørner & Jones 1986] and block structuring was adopted, in 1991, in Fortran 90.

Fortran 95 has broken this mould. New constructs, dealing with data-parallel assignment, have been introduced that have not been borrowed from elsewhere. These features, which include simple data parallel assignment have not been formally defined.

While much work has been carried out in recent years into the abstract idea of data-parallel assignment [Bougé et al 1996, Cachera and Utard 1996, Stewart 1995], in all of these cases the focus has been on data-parallel assignment over single dimension arrays, and has rarely addressed the multi-dimensional arrays supported by real scientific programming languages.

However, for a model to be considered of benefit to the users of a real language such as Fortran 95 it must encompass those concepts which are actually used. In such a language the
The majority of users will be utilising data-parallel assignment over multidimensional arrays. A model of data-parallel array assignment over single dimensional arrays is most readily understood, but it is not inherently scalable to multiple dimensions and is too specific to be of much benefit in the general case.

An overall goal of the work, some of which is reported upon here, is to provide a definition of key features of the Fortran 95 standard [ISO/IEC 1997, Adams et al 1997]. This semantics will provide aid for software engineers who wish to reason about their programs and offers a guide for those engaged in the definition of future dialects of the language.

The initial phase of the work has been completed, resulting in the provision of an abstract syntax of the language, expressed in VDM-SL [Jones 1990], together with a static semantics of the main program unit.

In this paper discussion is restricted to data-parallel assignment over multidimensional arrays: these multidimensional arrays are formally addressed by viewing them as VSM-SL maps. Attention is focused on the data-parallel assignment forms supported by Fortran 95.

A brief guide to the notation of VDM-SL is included in Appendix A.

2 Array Assignment in Fortran 95

The Fortran 95 standard includes three types of array reference:

1. a whole array, e.g. A
2. an array section: a sub-array of a full array is selected by means of a list of subscripts, e.g. A(1:10) selects the sub-array comprising elements one to ten of array A,
3. an array element: is selected by means of a subscript list, e.g. A(1) selects the contents of the array element at index one.

A reference to a whole array is achieved by referring to the array variable. An array variable followed by a list of array subscripts allows access either to an array element or an array section. The subscripts in the subscript list are applied in left to right order to the corresponding dimensions of the array: the length of the subscript list must therefore conform to the rank (number of dimensions) of the array.

This precludes the use of expressions such as \( A_{3d}(1) \) to select the top 'slice' of the array \( A_{3d} \), where \( A_{3d} \) is defined as being an array of 2 x 3 x 4 elements: the three dimensions must be given explicitly, in this example \( A_{3d}(1,1:3,1:4) \).

2.1 Array Subscripting

Each array subscript may assume one of three forms: simple subscript, vector subscript and triplet subscript. Mixing the three forms within a single subscript list is permitted.

These three subscripting forms are now described.
2.1.1 Simple Subscript

A *simple subscript* in a subscript list selects a single element of the appropriate dimension. For example, given an array:

```
INTEGER A(10)
```

the reference A(3) selects the third element of the array A.

2.1.2 Vector Subscript

A *vector-subscript* selects a number of non-contiguous elements by means of an array constant or value which itself specifies the elements:

1. **An array constructor constant**: for example
   
   \[ A(1,10,7/) \]
   
   selects the first, tenth and seventh elements of the array \( A \), in that order.

2. **An array**: the contents of a single dimension array (or section) may be used to select specific elements - e.g.

   \[ \text{let } k = (/1, 3, 5/) \]
   
   \[ A(k) \]
   
   where \( k \) is itself an array, then selects the first, third and fifth elements of the array \( A \), i.e. the *elements* of \( k \) are used to select the indices of \( A \).

The rank of a resulting array section is the number of *vector subscripts* and *triplet subscripts* in the subscript list. Thus, where the *vector subscript* is of length one, this means that the *vector subscript* selects an *array section*, not an *element*.

For example, using the array \( A_{3d}(2,3,4) \) from the previous section:

```
A_{3d} =
```

```
1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

def

**extent**: the *extent* of a dimension of an array or array section is the number of elements in that dimension, e.g. given a declaration \( A(1:5, 1:10) \) the extents of the dimensions of \( A \) are five and ten.

**shape**: the *shape* of an array or array section is taken as the list of extents of the dimension, in the order of declaration, e.g. given \( A(1:5, 1:10) \) the *shape* of \( A \) is \((5, 10)\).

**rank**: the rank of an array or section is the number of dimensions it possesses, e.g. \( A(10,10) \) is a rank two array.

It could be argued that the use of a single element vector subscript is akin to the use of an ordinary subscript. However, its use is not precluded and thus has an impact on how assignment is treated, as will be discussed later.

2.1.3 Triplet Subscript

In this form a lower-bound, upper-bound and (optional) stride are given, thereby selecting a number of contiguous elements, e.g. given the declaration INTEGER \( A(20) \), then \( A(1:10) \)
selects the sub-array containing the elements A(1) and A(10) and all the elements in between, in the same order. Similarly A(i:j:stride) selects elements A(i) and each stride\textsuperscript{th} element thereafter up to A(j). Thus, for example, A(1:10:3) selects the sub-array comprising A(1), A(4), A(7) and A(10). A zero stride is not permitted. Where the (optional) stride is omitted a default stride of one is assumed.

Where only a single element, e.g. A(1:1), is selected then a non-zero rank portion of the sub-array is denoted - not the value of A(1). A triplet subscript selecting no elements, e.g. A(1:0), denotes both a zero rank array portion and a zero rank array, neither of which is considered to be the same as a scalar.

To summarise:
- The length of the subscript list must equal the rank of the array to which it is applied,
- Each position of the subscript list may be occupied by any of the three forms of subscript,
- The rank of an array section is the number of vector subscripts and triplet subscripts in the subscript list, e.g. given a rank four array B: B(4,4,4,4) of shape (4,4,4,4), the array section B(1:2,2,2/1:2) has rank three and shape (2,1,2).

### 2.1.4 Array Declarations

Array declarations may take a number of forms:

1. \textit{extent}: for an extent style declaration only the extent of a dimension is given, which is then implicitly extrapolated to a default lower-bound of 1, upper-bound of extent and default stride of 1. Thus INTEGER A(5, 10) defines a rank two array of shape (5, 10), with default lower-bound of 1 and implicit upper-bounds of five and ten respective to the dimensions, having an integer range.

2. \textit{explicit bounds}: here a lower-bound and upper-bound are explicitly provided, e.g. A(1:5, 1:10), which declares precisely the same array as that of (1) above; a default stride of one is assumed.

3. \textit{stride}: this provides an explicit stride. A stride may only be supplied in conjunction with a declaration providing explicit bounds. Thus A(10:1:-2) declares an array with indices 10, 8, 6, 4 and 2.

It is possible for the domain declared for an array to be empty, e.g. A(1:0). While a stride is optional, when it is used in conjunction with explicit bounds, it may not be zero.

### 2.2 Multidimensional-Array Assignment

Before embarking on the formal modelling of multidimensional arrays and Fortran 95 array assignment both the general and specific forms that array assignment may take will be discussed informally.

The lhs of an array assignment may be either an array element, an array or an array sub-object (i.e. an array section). The rhs of an array assignment may be either a scalar (e.g. an integer), an array, an array element or an array section.
2.2.1 Assignment to a whole array

Where the lhs of the array assignment is simply composed of an array identifier, every element of the array referred to is assigned a value. This rhs value may be either a scalar, in which case the scalar value is assigned to each of the elements of the lhs array, or an array (of the same shape as the lhs array), in which case an element-wise assignment is carried out. Figure 2.1 provides an illustrative example of these.

![Figure 2.1: Whole Array Assignment Example](image)

2.2.2 Assignment to a sub-array

It is possible to carry out an assignment to a specified section of an array, where the section may be specified by a list of subscripts, with the subscripts being applied in left to right order to the dimensions of the array, as described earlier.

It is also possible to have an array section on the rhs of an assignment, provided its shape conforms with that of the lhs. Only one difference between the permissible forms for the lhs and rhs exists: in the case of an array subscript it is not permissible for the l.h.s. array subscript to refer to the same element more than once, i.e. a many-one section is permissible on the rhs but not on the left. For example:

A(7, 1, 7)

may appear on the rhs of an assignment, but not on the lhs.

The following are legal Fortran 95 assignments:

```fortran
INTEGER A(10), B(10,10), C(10,10,10)
B = 10
C = 20
A = B(1:10, 1):  A|1) = B|1|, A|2) = B|2|, ..., A|10) = B|10|
A(1:4) = B(1:4, 1):  A|1) = B|1|, A|2) = B|2|, A|3) = B|3|, A|4) = B|4|
A(1:4) = B(1, 1:4):  A|1) = B|1, A|2) = B|1, A|3) = B|1, A|4) = B|1
A(/10, 1, 4, 2/) = C(1, 3, /9, 4, 6, 7 /):  A|10) = C|1,3,9|, A|1) = C|1,3,4|,
                                           A|4) = C|1,3,6|, A|2) = C|1,3,7|
```
Now that the forms of the Fortran 95 array assignment have been described informally, a formal description will be proposed. We begin by introducing a formal model of multidimensional arrays.

3 Array Modelling using VDM Mappings

The approach taken here is to begin with a single dimensional array model and to move in stages to the multidimensional case.

In the first instance a single dimension array is modelled. This model is then extended to two-dimensional and three-dimensional arrays. These results are then extrapolated to the 'general' multi-dimensional case.

It is necessary here to introduce some of the complexity of the chosen programming language, Fortran 95, so that concrete examples of the arrays referred to by the model can be included. To build concrete arrays the following Fortran 95 features are required:

1. **The array constructor**: one dimensional instances of arrays may be generated using the array constructor, which accepts a sequence of values enclosed within '','/'. Instances of multidimensional arrays may not be directly generated: the RESHAPE function may be used to impose the appropriate shape on an array and according to the Fortran 95 storage discipline.

2. **RESHAPE**: The RESHAPE function provides a number of facilities, only one of which is of concern here, namely, the reshaping of a sequence of items into an array of the appropriate shape. For example, RESHAPE(/1, 2, 3, 4, 5, 6/, /2, 3/) will return a two dimensional array of extents two and three, with the following (Fortran) column-dominant element ordering:

   ![2x3 array]

   1 3 5  
   2 4 6

3.1 Modelling 1D Arrays

The modelling of arrays utilises the VDM-SL mapping, which inherently lends itself to both programmers' and theoreticians' views of an array.

Let $A_{1d} = (/11, 12/)$

$A_{1d} = \begin{bmatrix} 11 & 12 \end{bmatrix}$

In this single dimension case the mapping from indices to values is intuitive for programmers.

Abstractly the type of a one-dimensional array, $1D_{map}$ can thus be represented by:

**TYPE** $1D_{map} = N_{\rightarrow} N$
The abstract value of $A_{1d}$, $A_{1d-map}$, is then:

$$A_{1d-map} = \{ 1 \rightarrow 11, \ 2 \rightarrow 12 \}$$

### 3.2 Modelling 2D Arrays

Let $A_{3d} = \text{RESHAPE}(1, 2, \ldots, 6/2, 3/)$

$$A_{2d} = \begin{pmatrix}
1 & 3 & 5 \\
2 & 4 & 6
\end{pmatrix}$$

At this stage it is reasonably clear how to represent the array as a mapping: a two dimensional array can be represented as a mapping from natural numbers to a mapping of type $1^{D-map}$, where the range of the mapping is itself a mapping. Such an array is of type $2^{D-map}$, given by:

**TYPE** $2^{D-map} = N \rightarrow 1^{D-map}$

Thus the abstract value of $A_{2d}$, $A_{2d-map}$, is:

$$A_{2d-map} = \begin{cases}
1 \rightarrow \{1 \rightarrow 1, \ 2 \rightarrow 3, \ 3 \rightarrow 5\}, \\
2 \rightarrow \{1 \rightarrow 2, \ 2 \rightarrow 4, \ 3 \rightarrow 6\}
\end{cases}$$

### 3.3 Modelling 3D Arrays

Let $A_{3d} = \text{RESHAPE}(1, 2, \ldots, 24/2, 3/4)$

At this stage the recursive structure of multidimensional arrays becomes obvious, so that:

**TYPE** $3^{D-map} = N \rightarrow 2^{D-map}$
Thus $A_{3d\text{-map}}$ has the value:

\[
A_{3d\text{-map}} =
\begin{cases}
1 \mapsto \{1 \mapsto 1, 2 \mapsto 7, 3 \mapsto 13, 4 \mapsto 19\}, \\
2 \mapsto \{1 \mapsto 3, 2 \mapsto 9, 3 \mapsto 15, 4 \mapsto 21\}, \\
3 \mapsto \{1 \mapsto 5, 2 \mapsto 11, 3 \mapsto 17, 4 \mapsto 23\},
\end{cases}
\]

\[
\begin{cases}
1 \mapsto \{1 \mapsto 2, 2 \mapsto 8, 3 \mapsto 14, 4 \mapsto 20\}, \\
2 \mapsto \{1 \mapsto 4, 2 \mapsto 10, 3 \mapsto 16, 4 \mapsto 22\}, \\
3 \mapsto \{1 \mapsto 6, 2 \mapsto 12, 3 \mapsto 18, 4 \mapsto 24\},
\end{cases}
\]

### 3.4 General Multidimensional Arrays Model

The model clearly exhibits scaleable qualities, based on its recursive structure. Thus:

\[
\text{TYPE} \quad n^D = N \mapsto (n-1)^D
\]

\[
(n-1)^D = N \mapsto (n-2)^D
\]

\[
\cdot
\]

\[
\cdot
\]

\[
1^D = N \mapsto N
\]

**Definition 1: Array**

\[
\text{TYPE} \quad \text{Array} = N \mapsto \text{Array-Range}
\]

\[
\text{Array-Range} = N \mid \text{Array}
\]

A type definition for the multidimensional array is provided in **Definition 1**, which reflects this structure.

Having decided on a map representation for an array, it is possible to specify the RESHAPE function formally. This is given in **Definition 2**.
The natural development of the mapping model results in the model having the first dimension outermost, so that, for example, in an array:

\[
\text{INTEGER } A(X, Y, Z)
\]

the element \(A(x, y, z)\) in the programming model is accessed in the mapping model by:

\[
((A(x))(y))(z),
\]

where \(A\) is defined abstractly by:

\[
A : \text{Array}
\]

**Definition 2: the RESHAPE Function**

\[
\text{RESHAPE}: N^* \times N^* \rightarrow \text{Array}
\]

\[
\text{RESHAPE}(vals, shape) \triangleq \\
\quad \text{if } \text{len} \ shape = 1 \text{ then} \\
\qquad \{ i \mapsto \text{vals}(i) \mid i \in \text{inds} \ vals \} \\
\quad \text{else} \\
\qquad \{ i \mapsto \text{Reshape}( \{ \text{vals}(i + (k \times \text{hd} \ shape)) \mid k \in \{0, \ldots, (\text{len} \ vals \ / \text{hd} \ shape) - 1\} \}, \text{tl} \ shape) \mid i \in \{1, \ldots, \text{hd} \ shape\} \}
\]

**Example**

\[
\text{RESHAPE}([1,2,3,4,5,6],[2,3]) = \\
\quad \{ 1 \mapsto \text{RESHAPE}([1,3,5],[3]), \} \\
\quad \{ 2 \mapsto \text{RESHAPE}([2,4,6],[3]) \}
\]

\[
\quad = \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, \} \\
\quad \{ 2 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \}
\]

4. **Multidimensional Array Assignment**

The model of multidimensional array assignment is based on the mapping model for arrays defined above.

An array assignment in Fortran 95 may have one of the following five forms:

i. assignment of a scalar value to an array,
ii. assignment of a scalar value to an array section,
iii. assignment of a full array to a full array,
iv. assignment of an array section to an array section,
v. assignment of an array section to a full array, or assignment of a full array to an array section. These may be dealt with as a single case, since both involve a full array on one side of the assignment and an array section on the other.

In the first instance assignment forms (i) to (iv), above, are built as separate functions - form (v) is a combination of forms (iii) and (iv) and will not be dealt with separately. These functions are then combined into a general function, \(\text{Assign}_1\). The presentation reflects the development process, where the function \(\text{Assign}_1\) was created in an incremental fashion, reflecting the complexity of multidimensional array assignment. Another function is then constructed which, in addition, deals with some idiosyncrasies of Fortran 95 viz. inversely ordered indices and vector subcribing and which escaped notice during the earlier analysis. These will be discussed in more detail in section 4.2.
4.1 Assignment (1)

In this section the four functions corresponding to assignment forms (i) to (iv) above are presented, followed by the general function, $Assign_1$, which covers all four cases, the functions are as follows:

$AssignAs$: assign a scalar value to a full array
$AssignSs$: assign a scalar value to a (subscripted) array section
$AssignAA$: assign a full array to a full array
$AssignSS$: assign an array section to an array section
$Assign_1$: assignment function which encompasses all four of the preceding forms.

The following assumptions are made in all cases:
1. Where both the left and right hand sides of the assignment involve arrays, then both arrays are of the same shape. The rank (number of dimensions) of both of the arrays or sections are therefore implicitly the same. This assumption is based on the relevant Fortran 95 constraints of the standard: to have any other forms constitutes an illegal assignment.
2. Any subscripting of an array is provided, in the formal model, as a sequence of sets, where the set in each position of the sequence indicates the indices that are selected in the corresponding dimension. It is therefore required that the length of the subscript list be equal to the rank of the array to which it is being applied.

4.1.1 Assign Scalar to Full Array: $AssignAs$

In the function $AssignAs$ no restriction is placed on which indices in a dimension are selected, as all indices are required in all dimensions.

$$AssignAs : Array \times N \rightarrow Array$$

\[
\begin{align*}
\text{AssignAs}(A, val) & \triangleq \\
\text{cases } \text{rng } A \text{ of } \\
N & \rightarrow A \uparrow \{ i \mapsto val \mid i \in \text{dom } A \} \\
\text{else} & \rightarrow A \uparrow \{ i \mapsto \text{AssignAs}(A(i), val) \mid i \in \text{dom } A \}
\end{align*}
\]

Examples 4.1 and 4.2 trace applications of $AssignAs$. These emphasise that it is only when the final, innermost, dimension is reached that overwriting with the required value takes place, with the result then being propagated back through the dimensions.
Example 4.1

\[ \text{AssignAs( } \{1 \rightarrow 11, 2 \rightarrow 12\}, 5) \]

\[ = \{1 \rightarrow 11, 2 \rightarrow 12\} \uplus \{1 \rightarrow 5, 2 \rightarrow 5\} \]

\[ = \{1 \rightarrow 5, 2 \rightarrow 5\} \]

Example 4.2

\[ \text{AssignAs( } \{\begin{array}{l}
1 \mapsto \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6\}\end{array}, 5) \]

\[ = \{\begin{array}{l}
1 \mapsto \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6\}\end{array} \uplus \{\begin{array}{l}
1 \mapsto \text{AssignAs}(\{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}, 5) \\
2 \mapsto \text{AssignAs}(\{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6\}, 5)\end{array}\} \]

\[ = \{\begin{array}{l}
1 \mapsto \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6\}\end{array} \uplus \{\begin{array}{l}
1 \mapsto \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6\}\end{array} \uplus \{\begin{array}{l}
1 \mapsto \{1 \mapsto 5, 2 \mapsto 5, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 5, 2 \mapsto 5, 3 \mapsto 5\}\end{array}\} \]

\[ = \{\begin{array}{l}
1 \mapsto \{1 \mapsto 5, 2 \mapsto 5, 3 \mapsto 5\}, \\
2 \mapsto \{1 \mapsto 5, 2 \mapsto 5, 3 \mapsto 5\}\end{array}\} \]

4.1.2 Assign Scalar to Array Section

The function AssignSs, below, addresses specifically the assignment of a scalar value/expression to an array section - e.g. \(A(1:3) = 6\) - again using a recursive overwriting of the array. The indices in any dimension to which the assignment is to be restricted are specified in the index-set. For example, a call to AssignSs with parameters consisting of a two dimensional array \(A\), index-set-seq \([\{1,2\}, \{4,7\}]\) and a scalar value \(s\) results in the value \(s\) being assigned to all those elements of \(A\) which have index 1 or 2 in the first dimension and index 4 or 7 in the second dimension, i.e. \(A(1, 4), A(2, 4), A(1, 7)\) and \(A(2, 7)\).

\[ \text{AssignSs : Array} \times \text{N-SET}^+ \times \text{N} \rightarrow \text{Array} \]

\[ \text{AssignSs}(A, \text{index-set-seq}, \text{val}) \triangleq \]

\[ \begin{align*}
& \text{if } \text{len} \text{ index-set-seq} = 1 \text{ then } \\
& \quad A \uplus \{i \mapsto \text{val} \mid i \in \text{hd} \text{ index-set-seq} \} \\
& \text{else } \\
& \quad A \uplus \{i \mapsto \text{AssignSs}(A(i), \text{tl} \text{ index-set-seq}, \text{val}) \mid i \in \text{hd} \text{ index-set-seq} \} 
\end{align*} \]
The precondition/assumption relating the length of index-set-seq to the rank of the array - and in the mapping model the depth of the mapping - ensures that recursion can be terminated.

The functionality of AssignSs is thus essentially the same as that of AssignAs, except that, in each dimension, the selected elements are restricted to those determined by the index-set corresponding to that dimension. Thus a domain restriction on the recursive overwriting is introduced.

Example 4.3 traces an application of AssignSs.

**Example 4.3**

Assume $\{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, 2 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \} \mapsto \{ \{2\}, \{1,3\} \}, 5)$

\[
\begin{array}{c|c|c|c}
  & 1 & 3 & 5 \\
\hline
1 & 2 & 4 & 6 \\
\end{array}
\]

\[
= \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, 2 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \} \mapsto \{ 2 \mapsto AssignSs\{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \}, \{ \{1,3\}, 5 \} \}
\]

\[
= \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, 2 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \} \mapsto \{ 2 \mapsto \{ 1 \mapsto 5, 3 \mapsto 5 \} \}
\]

\[
= \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, 2 \mapsto \{ 1 \mapsto 5, 2 \mapsto 4, 3 \mapsto 5 \} \}
\]

The situation is more complex when arrays occur on both sides of the assignment.

### 4.1.3 AssignAA

The index of an element of an array may have only an indirect correspondence with its relative position within the array. For example, the following array declarations:

```plaintext
INTEGER A(1:10:2), B(2:15:3)
```

specify that array A is defined to have elements with indices 1, 3, 5, 7, 9 (i.e. a lower-bound, upper-bound and stride); similarly, array B has elements with indices 2, 5, 8, 11, 14.

In this case the index of the first element of array A is 1, while the index of the first element of array B is 2, so that for the assignment $A = B$: 

```plaintext
A = B
```
A(1) is overwritten with the value of B(2),
A(3) is overwritten with the value of B(5),
A(5) is overwritten with the value of B(8),
A(7) is overwritten with the value of B(11) and
A(9) is overwritten with the value of B(14).

Thus it is necessary to introduce a function which establishes a positional correspondence
between two arrays. To this end the function POS, as defined, which generates a mapping
from relative position to actual array index. The functions for assignment involving two
arrays are then based on this function.

It should, however, be noted that POS assumes that array indices are
ordered (in ascending order), while allowing for the fact that indices may be in an arithmetic
progression rather than being contiguous integers. This arbitrary restriction will be removed in section
4.2.

Min is assumed to return the smallest number in a set (of numbers).

**AssignAA: Array x Array → Array**

\[ AssignAA(A_1, A_2) \triangleq \]

**cases** rng \( A_1 \) of

- **N-set** \( \rightarrow \)  
  let \( \text{pos} A_1 = \text{POS}(\text{dom } A_1) \) in
  let \( \text{pos} A_2 = \text{POS}(\text{dom } A_2) \) in
  \( A_1 \uparrow \{ \text{pos} A_1(i) \mapsto A_2(\text{pos} A_2(i)) \mid i \in \text{dom } \text{pos} A_1 \} \)

- **else** \( \rightarrow \)  
  \( A_1 \uparrow \{ \text{pos} A_1(i) \mapsto AssignAA( A_1(\text{pos} A_1(i)), A_2(\text{pos} A_2(i))) \mid i \in \text{dom } \text{pos} A_1 \} \)

The functionality follows the now familiar form, with the exception that the index of an
element is now referred to indirectly, via the element’s relative position. Examples 4.4 and
4.5 trace applications of \( AssignAA \).

It should be noted that, given that both array sections are of the same shape:
\[
\text{dom } \text{POS}(\text{dom } A_i) = \text{dom } \text{POS}(\text{dom } A_j).
\]
Example 4.4

Let \( A_1 = \{3 \mapsto 11, 4 \mapsto 12\}, A_2 = \{1 \mapsto 21, 3 \mapsto 22\} \) in

\[ AssignAA(A_1, A_2) \]

\[ = A_1 \uplus \{ posA_1(1) \mapsto A_2(posA_2(1)), posA_1(2) \mapsto A_2(posA_2(2)) \} \]

\[ = A_1 \uplus \{ 3 \mapsto A_2(1), 4 \mapsto A_2(3) \} \]

\[ = A_1 \uplus \{ 3 \mapsto 21, 4 \mapsto 22 \} \]

\[ = \{ 3 \mapsto 21, 4 \mapsto 22 \} \]

Example 4.5

\[ AssignAA( \{ 1 \mapsto \{ 1 \mapsto 1, 3 \mapsto 3, 5 \mapsto 5 \}, \} \), \{ 1 \mapsto \{ 1 \mapsto 3, 2 \mapsto 6, 3 \mapsto 9 \}, \} \)

\[ = \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, \}

\[ \{ 3 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \} \uplus \{ 1 \mapsto AssignAA(\{1\mapsto1, 2\mapsto3, 3\mapsto5\}, \{1\mapsto3, 2\mapsto6, 3\mapsto9\}), \}

\[ \{ 3 \mapsto AssignAA(\{1\mapsto1, 2\mapsto4, 3\mapsto6\}, \{1\mapsto4, 2\mapsto8, 3\mapsto10\}) \}

\[ = \{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 5 \}, \}

\[ \{ 3 \mapsto \{ 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6 \} \} \uplus \{ 1 \mapsto \{ 1 \mapsto 3, 2 \mapsto 6, 3 \mapsto 9 \}, \}

\[ \{ 3 \mapsto \{ 1 \mapsto 6, 2 \mapsto 8, 3 \mapsto 10 \} \}

\[ = \{ 1 \mapsto \{ 1 \mapsto 3, 2 \mapsto 6, 3 \mapsto 9 \}, \}

\[ \{ 3 \mapsto \{ 1 \mapsto 6, 2 \mapsto 8, 3 \mapsto 10 \} \}

4.1.4 Assign Array Section to Array Section: AssignSS

In the function AssignSS the full effect of possible indirect correspondence between an index value and its relative position is addressed. Here it is necessary to establish a relative position for each index not within the array but rather within the array section. For example, given

integer A(1:8:2)

the domain of A is

\{1, 3, 5, 7\}

so that with \( A_{\text{map}} \) denoting the map representation of A:

\[ \text{POS(} \text{dom } A_{\text{map}} \text{)} = \{ 1\mapsto1, 2\mapsto3, 3\mapsto5, 4\mapsto7 \}. \]

However, when dealing with the section \( A(1:5:3) \), the domain of the section is:

\{1, 4\} \cap \{1, 3, 5, 7\}

corresponding to:

\[ \text{POS(} \text{dom } A_{\text{mapping}} \cap \text{section-indices} \text{)}. \]
**AssignSS**: $\text{Array x N-SET} \times \text{Array x N-SET} \rightarrow \text{Array}$

AssignSS($A_1, A_1\text{indices}, A_2, A_2\text{indices}$) $\triangleq$

let $posA_1 = (\text{POS}_m(\text{dom } A_1 \cap \text{hd } A_1\text{indices}))$ in
let $posA_2 = (\text{POS}_m(\text{dom } A_2 \cap \text{hd } A_2\text{indices}))$ in

if rng $A_1 \in \mathbb{N}$ then
  $A_1 \uparrow \{ \text{pos}A_1(i) \mapsto A_2(\text{pos}A_2(i)) \mid i \in \text{dom pos}A_1 \}$
else
  $A_1 \uparrow \{ \text{pos}A_1(i) \mapsto \text{AssignSS}(A_1(\text{pos}A_1(i)), A_2(\text{pos}A_2(i))) \mid i \in \text{dom pos}A_1 \}$

Example 4.6 traces an application of AssignSS.

**Example 4.6**

AssignSS

\[
\begin{align*}
\{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \}, 3 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \} \} & \uparrow \{ 1 \mapsto \{ 1 \mapsto 3, 2 \mapsto 6 \}, 3 \mapsto \{ 1 \mapsto 4, 2 \mapsto 8 \} \} \\
\{ 3 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \} \} & \uparrow \{ 3 \mapsto \text{AssignSS}(\{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \}, [\{3\}], [\{1\mapsto 3, 2 \mapsto 6\}, [\{2\}]) \} \\
\{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \}, 3 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \} \} & \uparrow \{ 3 \mapsto ( \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 5 \} \uparrow \{ 3 \mapsto 6 \} ) \} \\
\{ 1 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1 \}, 3 \mapsto \{ 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 6 \} \} & \uparrow \{ 3 \mapsto \} \\
\end{align*}
\]

### 4.1.5 $\text{Assign}_1$

The functionality of $\text{AssignAs}$, $\text{AssignSs}$, $\text{AssignAA}$ and $\text{AssignSS}$ can now be assembled into a single function, $\text{Assign}_1$, which handles forms (i) to (iv) of array assignment.

Two additional types are introduced:
1. $\text{Array-Ref}_i$: this type is used to bring together the concept of an array and optional subscripting on that array,
2. $\text{exp}$: is introduced to deal with the rhs of an assignment which may be either an array reference or an integer scalar.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Array-Ref$_i$ ::</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>array-part : Array</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subscript-part : N-set</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>$= N \mid \text{Array-Ref}_i$</td>
<td></td>
</tr>
</tbody>
</table>
Assign : Array-Ref x exp → Array
Assign₁(lhs, rhs) Δ
let sₐ = subscript-part(lhs) in
let A = array-part(lhs) in
if sₐ = [ ] then
  let posₐ = (POS(dom A)) in
  cases rng A of
  N → cases e of
  N → A ⊩ \{posₐ(i) ↦ e | i ∈ dom posₐ\}
else → let e = array-part(rhs) in
  let sₑ = subscript-part(rhs) in
  if sₑ = [ ] then
    let posₑ = (POS(dom e)) in
    A ⊩ \{posₑ(i) ↦ e(posₑ(i)) | i ∈ dom posₑ\}
  else
    let posₑ = (POS(dom e ∩ hd sₑ)) in
    A ⊩ \{posₑ(i) ↦ e(posₑ(i)) | i ∈ dom posₑ\}
else → cases e of
N → A ⊩ \{posᵢ(i) ↦ \assign₁(\textit{mk-Array-Ref}(A(posᵢ(i)), [ ]), e) | i ∈ dom posᵢ\}
else → let e = array-part(rhs) in
  let sₑ = subscript-part(rhs) in
  if sₑ = [ ] then
    let posₑ = (POS(dom e)) in
    A ⊩ \{posₑ(i) ↦ \assign₁(\textit{mk-Array-Ref}(A(posₑ(i)), [ ]), \textit{mk-Array-Ref}(e(posₑ(i)), [ ])) | i ∈ dom posₑ\}
  else
    let posₑ = (POS(dom e ∩ hd sₑ)) in
    A ⊩ \{posₑ(i) ↦ \assign₁(\textit{mk-Array-Ref}(A(posₑ(i)), [ ]), \textit{mk-Array-Ref}(e(posₑ(i)), tl sₑ)) | i ∈ dom posₑ\}
else → cases e of
N → A ⊩ \{posᵢ(i) ↦ \assign₁(\textit{mk-Array-Ref}(A(posᵢ(i)), tl sᵢ), e) | i ∈ dom posᵢ\}
else → let e = array-part(rhs) in
    let s_e = subscript-part(rhs) in
    if s_e = [] then
        let pos_e = (POS(dom e)) in
        A ⊩ \{ pos_e(i) → Assign1(mk-Array-Ref(A(pos_e(i)), tl s_A),
                                  mk-Array-Ref(e(pos_e(i)), []) ) |
                        i ∈ dom pos_A \}
    else
        let pos_e = (POS(dom e ∩ hd s_e)) in
        A ⊩ \{ pos_A(i) → Assign1(mk-Array-Ref(A(pos_A(i)), tl s_A),
                                  mk-Array-Ref(e(pos_A(i)), tl s_e)) |
                        i ∈ dom pos_A \}

When dealing with a full array the list of subscripts is empty. As the VDM-SL function hd, which returns the item at the head of a given sequence, is undefined for an empty sequence it is necessary to deal separately with a full array with no subscripting and a full array with subscripting. This has been achieved, in Assign1, by the use of if then else constructs to differentiate between an empty and non-empty subscript list.

No traces of application of Assign1 are provided, as its application replicates that of an appropriately chosen earlier function.

Assign1 provides a definition of multidimensional assignment for the straightforward cases. It is necessary, however, to address the problems of inversely ordered indices and vector subscripting. These are now discussed.

4.2 Assign (2)

In the development of the function Assign2 two problems are addressed:

1. descending ordered indices: earlier it was assumed that indices follow, by convention, an increasing order. This assumption does not hold for Fortran 95 arrays. While the indices are regularly spaced they may, in fact, be in descending order- e.g. A(10:1:-2) declares an array A with indices 10, 8, 6, 4, 2 in that order so that, for example, the second element of the array is A(8).

   The impact that this has on the mapping model of arrays is highlighted by the following example:

   INTEGER A(1:4), B(4:1:-1)
   B = 7, 5, 10, 3

   results in the assignments B(4) = 7, B(3) = 5, B(2) = 10 and B(1) = 3.

   Abstractly B = { 1 → 3, 2 → 10, 3 → 5, 4 → 7 }

   Suppose that the assignment A = B is required. The question arises as to how this may be defined abstractly when the domain of a map is unordered.
2. **vector subscripting**: in this form of Fortran 95 subscripting, instead of a number of contiguous array elements being selected, a sequence of elements (of the appropriate dimension) is determined by a sequence of indices, e.g.

```
INTEGER A(2), B(6)
B = /10, 5, 17, 23, 42, 19/
A = B(/6, 3/)
```

results in the assignments $A(1) = B(6) = 19$ and $A(2) = B(3) = 17$.

To avoid non-determinism, where a vector subscript appears in an array reference on the lhs of an assignment, all elements referred to in that vector subscript must be distinct. Thus:

$A(/1,3,1/) = B(1:3)$

is an illegal assignment in Fortran 95. This constraint is assumed to have been met, and will be formalised elsewhere.

Rather than dealing with the subscripts of an array as a sequence of *sets* it becomes necessary to treat the subscripts as a sequence of *sequences*, thereby:

- preserving the ordering of any vector subscript. For example, it is possible to indicate that the elements of an array are to be selected in inverse order, e.g.

```
INTEGER A(1:3), B(1:3)
B = /7, 11, 13/
A(/3, 2, 1/) = B
```

results in the assignments $A(3) = B(1)$, $A(2) = B(2)$ and $A(1) = B(3)$.

- implicitly providing the order of an array’s indices. An unfortunate drawback is that, where an array in its entirety is to be dealt with, the entire array must be explicitly subscripted, in order to provide the ordering information.

```
INTEGER A(3:1:-1), B(1:3)
B = /7, 11, 13/
A = B
  Comment: in the formal model the indices would have to be given in their entirety in the assignment.
```

results in the assignments:

$A(3) = B(1) = 7$,
$A(2) = B(2) = 11$ and
$A(1) = B(3) = 13$.

Taken in conjunction with the preservation of the ordering of a vector subscript this now permits more complex combinations of the two facilities, e.g.

```
INTEGER A(1:3), B(3:1:-1)
B = /7, 11, 13/
A(/2, 3, 1/) = B
```

results in the assignments:
A(2) = B(3) = 7,
A(3) = B(2) = 11 and
A(1) = B(1) = 13.

Since all forms of Fortran 95 subscripting such as vector subscripting must be addressed, additional VDM-SL types are introduced.

**TYPE**

\[
Array = N \rightarrow Array\text{-Range}
\]

\[
Array\text{-Range} = N | Array
\]

\[
Array\text{-Ref}_2 :: array\text{-part} : Array
\]

\[
\text{subscript\text{-part}:Subscript}^+
\]

\[
Subscript = Vector\text{-Subscript} | Triplet\text{-Subscript} | \text{simple\text{-subscript}}
\]

\[
simple\text{-subscript} = N
\]

\[
Vector\text{-Subscript} = N^k
\]

\[
Triplet\text{-Subscript} :: lb : N
ub : N
by : N
\]

\[
exp = N | Array\text{-Ref}_2
\]

Note, again, that the Fortran 95 standard defines the rank of an array reference as the number of *vector-subscripts* and *triplet-subscripts* in the subscript list. This leads to the situation in which a single element selected by a *simple-subscript* is dealt with in a subtle but significantly different way from the selection of the *same* element by a *vector-subscript* or *triplet-subscript*. The use of a *simple subscript* means that that dimension of the parent array may be ignored, while the other forms demand that that dimension be taken into account in determining the desired section. For example, given an array defined as:

```
INTEGER A(10,10,10,10)
```

1. A(1,1,1,1): selects a single element in each dimension, while indicating that that dimension plays no part in the final result. The result is a *value*.
2. A(1:1, 1:1, 1:1, 1:1): since the rank of an array reference is the number of vector and triplet subscripts in the reference, the rank of this reference is 5. The result is an *array*.

Thus care must be taken when using vector or triplet subscripts which select a single array component. These subtleties are highlighted by the following examples:

```
INTEGER A(10,10,10), x, B(1:2)
```

```
1. x = A(1,1,1): legal,
2. x = A(1:1,1:1,1:1): illegal assignment as it attempts to assign a rank three array to a scalar variable,
3. B = A(1,1,1:2): legal,
4. B = A(1:1, 1:1, 1:2): illegal assignment, as it attempts to assign an array of shape (1,1,2) to an array of shape (2).```
The array assignment function is now recast in the light of the subscript related problems discussed above under the assumption that all assignments are legal in Fortran 95. The auxiliary function \texttt{extract-indices}, which produces a sequence of indices from a given subscript, is provided in Appendix B.

\textbf{Assign}$_2$: \texttt{Array-Ref} $x$ \texttt{exp} $\rightarrow$ \texttt{Array}
\begin{align*}
\text{Assign}_2(\text{lhs}, \text{rhs}) & \triangleq \\
\text{let } A = \text{array-part}(\text{lhs}) \text{ in} \quad & \text{let } S_d = \text{subscript-part}(\text{lhs}) \text{ in} \\
\text{let } S_e = \text{extract-indices}(\text{hd} \ S_d) \text{ in} \quad & \text{let } S_c = \text{extract-indices}(\text{hd} \ S_c) \text{ in} \\
\text{cases } \text{rng} \ A \text{ of} \\
N & \rightarrow \quad \text{cases } e \text{ of} \\
& \rightarrow \quad N \rightarrow A \uparrow \{S_d(i) \mapsto e \mid i \in \text{inds} \ S_e\} \\
& \text{else} \rightarrow \text{let } e = \text{array-part}(\text{rhs}) \text{ in} \\
& \text{let } S_e = \text{subscript-part}(\text{rhs}) \text{ in} \\
& A \uparrow \{S_d(i) \mapsto e(S_e(i)) \mid i \in \text{inds} \ S_c\} \\
& \text{else} \rightarrow \text{cases } e \text{ of} \\
& \rightarrow \quad N \rightarrow A \uparrow \{S_d(i) \mapsto \text{Assign}_2(A(S_d(i)), \text{tl} \ S_d, e, \text{Sub}_c \mid i \in \text{inds} \ S_a\} \\
& \text{else} \rightarrow \text{let } e = \text{array-part}(\text{rhs}) \text{ in} \\
& \text{let } S_e = \text{subscript-part}(\text{rhs}) \text{ in} \\
& A \uparrow \{S_d(i) \mapsto \text{Assign}_2(A(S_d(i)), \text{tl} \ S_d, e, \text{tl} \ S_c \\
& \mid i \in \text{inds} \ S_a\}
\end{align*}

Example 4.7 traces an application of \texttt{Assign}$_2$.

\textbf{Example 4.7}

\begin{align*}
\text{Assign}_2(\{1 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, 2 \mapsto \{1 \mapsto 4, 2 \mapsto 8\}), [[3], [1, 2]], [1 \mapsto \{1 \mapsto 3, 2 \mapsto 6\}], [1, [1, 2]]) \\
= \{1 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, 3 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \} \uparrow \{3 \mapsto \text{Assign}_2(\{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, [1, 2]) \}
\end{align*}

\begin{align*}
= \{1 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, 3 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \} \uparrow \{3 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \uparrow \{1 \mapsto 3, 3 \mapsto 6\}\}
\end{align*}

\begin{align*}
= \{1 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, 3 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \} \uparrow \{3 \mapsto \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 6\}\}
\end{align*}

\begin{align*}
= \{1 \mapsto \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}, 3 \mapsto \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 6\} \}
\end{align*}
5 Evaluation

The recursive VDM-SL array model defined in this paper is both concise and elegant. That the functionality of data-parallel array assignment can then easily be described recursively using this model goes some way to justifying the selection of the model.

The definition of the function $Assign_1$ addresses array assignment in Fortran 95 where the indices are in increasing order. The indices are further assumed to constitute an arithmetic progression $x, x+1, x+2, \ldots$.

While the assumption that the order is always increasing allows for simplification in the case of full array assignment - in that the indices need not be made explicit - this prevents more complex forms of Fortran 95 array indexing being dealt with.

$Assign_2$ rectifies this deficiency by employing sequences rather than sets to represent the array indices - so that the order of the indices is made explicit within the sequence. The utilisation of sequences to represent the indices allows the full range of Fortran 95 indexing to be accommodated, with the additional benefit of permitting a more concise assignment function to be defined. The only disadvantage of this approach is that the index set must now be made explicit for all cases - including full array assignment.

The assignment functions presented in this paper embrace all forms of Fortran 95 array assignment in which the lhs variable and rhs expression have the same rank.

The descriptions furnished in this paper clarify for language semanticists and users the forms that Fortran 95 assignment may take, and define precisely what those forms may be used to accomplish.

6 Conclusions

In this paper a formal model of multidimensional array assignment in Fortran 95 has been proposed. The arrays are modelled as recursively defined map structures and the assignment operations are defined as recursive functions over these map types.

The wide range and complexity of array assignment in Fortran 95 (see section 2) suggested the strategy of incremental development of assign functions $Assign_As$, $Assign_AA$, $Assign_Ss$, and $Assign_SS$, each dealing with a clearly defined sub-case of generalised assignment. These were then combined into a single general function, $Assign_1$. This approach was found to be useful as it allowed the complexity to be tackled in a manageable way. However, when dealing with an operation as complex and ill-described as the Fortran 95 array assignment it is not easy to anticipate all of the special cases that may arise.

The use of sets to represent the index sets in $Assign_1$ was a natural counterpart to the use of maps for the underlying array data-type. However, this choice was later found to be inadequate for the representation of the more complex orderings of indices permitted by Fortran 95 array assignment.

The work reported upon in this paper is the first step in the creation of a formal semantics, in a number of styles, for Fortran 95 - in particular for WHERE and FORALL constructs, as
discussed briefly below. The array model and assignment functions described in this paper will provide a solid foundation on which to base the semantics of these constructs.

In addition, to enable programmers to conduct proofs about their products an axiomatic semantics will be developed. In the construction of program proofs and in establishing the validity of the proposed semantic functions the theories of the VDM-SL types employed will be drawn upon. These developments will be the subject of further papers.

6.1 The WHERE Construct

The WHERE construct provides a mechanism whereby the elements of an array to which an assignment is to be applied are selected by means of a mask: this mask is dependent on the values of the elements of this, or another, array. For example:

**VALUE OF A**

\[
A = \{2, 4, 6, 8, 10, 13/\}
\]

WHERE \(\text{mod}(A, 4) = 0\)
\[
A = A + 2 \quad A = \{2, 6, 6, 10, 10, 13/\} \quad (i)
\]

ELSEWHERE \(\text{mod}(A, 2) = 0\)
\[
A = A - 1 \quad A = \{1, 6, 5, 10, 9, 13/\} \quad (ii)
\]

ELSEWHERE \(\text{mod}(A, 5) = 0\)
\[
A = A + 10 \quad A = \{1, 6, 5, 10, 9, 23/\} \quad (iii)
\]

ELSEWHERE
\[
A = A - 10 \quad A = \{1, 6, 5, 10, 9, 23/\} \quad (iv)
\]

END WHERE

(i) Assignment is made to elements \(A(2)\) and \(A(4)\)
(ii) Assignment is made to elements \(A(1), A(3)\) and \(A(5)\). While \(A(2)\) and \(A(4)\) are divisible by 4, these elements are not permitted to have a second assignment made to them.
(iii) Assignment is made to no elements, since \(A(3)\) and \(A(4)\) have previously had an assignment made to them.
(iv) Assignment is made to element \(A(6)\), since it is the only element to which no assignment has so far been made.

6.2 The FORALL Construct

The FORALL construct provides a mechanism to carry out assignment to specific elements of an array, where the selection of the elements is based on their index values, e.g. given the statement:

\[
\text{FORALL } (i = 1:3:1)
A(i,i) = 0
\text{END FORALL}
\]
if the value of the array \( A \) prior to this statement has the value:

\[
\begin{array}{ccc}
17 & 23 & 13 \\
31 & 47 & 3 \\
12 & 22 & 63 \\
\end{array}
\]

then \( A \)'s value following the statement will be:

\[
\begin{array}{ccc}
0 & 23 & 13 \\
31 & 0 & 3 \\
12 & 22 & 0 \\
\end{array}
\]

i.e. the diagonal is set to zero.

### 6.3 Summary and Future Developments

The development of Fortran continues: release of the next ISO Fortran standard, Fortran 2000, is expected in 2004. The semantics for Fortran 95 assignment presented here provides a formalisation of the meaning of the operation which should be more useful to members of the Standards Committee than the informal natural language descriptions which they currently use.

Finally, the models and functions provided in this paper provide a solid foundation for the following future developments:
- the provision of a denotational semantics, in the style presented in this paper, for the WHERE and FORALL constructs
- ensuring the correctness of the assignment functions by reasoning about their adequacy within the VDM theories of maps and sequences
- the provision of an axiomatic semantics for array assignment and the WHERE and FORALL constructs
- demonstration of how the correctness proofs of Fortran 95 programs may be constructed, on the basis of the semantics provided.

---

**Acknowledgement**

N. Reid was supported by DFHETE, through a Research Studentship.
7 References


Appendix A

Glossary of Symbols

Functions

| $f : D_1 \times D_2 \to R$ | signature |
| $f(d)$ | application |
| if ... then ... else | conditional |
| let $x = ...$ in ... | local definition |

Logic

| $\neg E$ | negation (not) |
| $\exists x \in S . \ E$ | existential quantifier |
| $\forall x \in S . \ E$ | universal quantifier |
| $E_1 \lor E_2$ | disjunction (or) |
| $E_1 \land E_2$ | conjunction (and) |

Sets

| $F$-set | all finite subsets of $F$ |
| $\{e_1, e_2, ..., e_n\}$ | set enumeration |
| $\{ \ x \in S \mid p(x) \}$ | set comprehension |
| $\{n, ..., n+k\}$ | subset of integers (from $n$ to $n+k$ inclusive) |
| $x \in S$ | set membership |
| $x \notin S$ | $\neg (x \in S)$ |
| $S_1 \cap S_2$ | set intersection |
| $S_1 \cup S_2$ | set union |
**Composite Objects**

| ::
| nil
| mk-\(N\) \(\ldots\)
| \(c(o)\)
| \(\mu(o, c \rightarrow v)\) | compose
| omitted object
| generate object
| select component of composite object
| modify component \(c\) of composite object \(o\), giving it new value \(v\)

E.g.

**TYPE**

\[ Month = \{\text{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}\} \]

1. **Date::**
   - \(\text{day} : \{1, \ldots, 31\}\)
   - \(\text{month} : Month\)
   - \(\text{year} : \{2000, 2001, \ldots\}\)

   \(x : Date\)

2. \(x = \text{mk-Date}(1, \text{Jan, 2001})\): creates an instance of the object, \(Date\), with the given values.

3. \(\text{day}(x)\): selects the \(day\) component of the composite object \(x\) i.e. 1.

4. \(x = \mu(x, day \rightarrow 2)\): changes the \(component, day\), of the \(composite object, x\), to the \(value 2\), resulting in \(x\) having the value \((2, \text{Jan, 2001})\).
Maps

\[ D \xrightarrow{=} R^1 \]
\[ \{ d_1 \mapsto r_1, d_2 \mapsto r_2, \ldots, d_n \mapsto r_n \}^2 \]
\[ \text{dom } m^3 \]
\[ m(d)^4 \]
\[ \{ d \mapsto f(d) \mid p(d) \}^5 \]
\[ m_1 \uplus m_2^6 \]

finite map: many-to-one mapping
map enumeration
domain of the map \( m \)
map application
map comprehension
map overwriting

E.g.

TYPE

1. \( Days\_in\_Month = Month \xrightarrow{=} N \)
   \[ dim : Days\_in\_Month \]

2. \( dim = \{ \text{Jan } \mapsto 31, \text{Feb } \mapsto 28, \text{Mar } \mapsto 31, \text{Apr } \mapsto 30, \text{May } \mapsto 31, \text{Jun } \mapsto 30, \]
   \( \text{Jul } \mapsto 31, \text{Aug } \mapsto 31, \text{Sep } \mapsto 30, \text{Oct } \mapsto 31, \text{Nov } \mapsto 30, \text{Dec } \mapsto 31 \} \)

3. \( \text{dom } dim : \{ \text{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec} \} \)

4. \( dim(\text{Feb}) = 28 \)

5. \( \{ x \mapsto dim(x) \mid x \in \text{dom } dim \land dim(x) = 30 \} = \{ \text{Nov } \mapsto 30, \text{Jun } \mapsto 30, \]
   \( \text{Sep } \mapsto 30, \text{Apr } \mapsto 30 \} \)

6. If \( m_1 = \{ 1 \mapsto 'd', 2 \mapsto 'a', 3 \mapsto 'g' \} \) and \( m_2 = \{ 1 \mapsto 'w', 3 \mapsto 't' \} \), then \( m_1 \uplus m_2 = \{ 1 \mapsto 'w', 2 \mapsto 'a', 3 \mapsto 't' \} \), i.e. the resulting map contains all the maplets from the second map, and all the maplets from the first whose first component is not in the domain of the second.
**Sequences**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T^*$</td>
<td>finite sequences</td>
</tr>
<tr>
<td>$T^+$</td>
<td>non-empty finite sequences</td>
</tr>
<tr>
<td>$[t_1, t_2, \ldots, t_n]$</td>
<td>sequence enumeration</td>
</tr>
<tr>
<td>$\text{len } s$</td>
<td>length of $s$</td>
</tr>
<tr>
<td>$s_1 \circ s_2$</td>
<td>sequence concatenation</td>
</tr>
<tr>
<td>$\text{hd } s$</td>
<td>item at the head of the (non-empty) sequence $s$</td>
</tr>
<tr>
<td>$\text{tl } s$</td>
<td>tail: the sequence $s$ with $\text{hd } s$ removed</td>
</tr>
<tr>
<td>$\text{inds } s$</td>
<td>the set of indices of $s$</td>
</tr>
<tr>
<td>$\text{elems } s$</td>
<td>the set of elements of $s$</td>
</tr>
<tr>
<td>${ f(d) \mid p(d) }$</td>
<td>sequence comprehension</td>
</tr>
</tbody>
</table>

Let $t_1 = [1, 13, 1]$ and $t_2 = [7, 17]$ in the following examples:

1. $\text{len } t_1 = 3$, $\text{len } t_2 = 2$
2. $t_1 \circ t_2 = [1, 13, 1, 7, 17]$
3. $\text{hd } t_1 = 1$, $\text{hd } t_2 = 7$
4. $\text{tl } t_1 = [13, 1]$, $\text{tl } t_2 = [17]$
5. $\text{inds } t_1 = \{1, 2, 3\}$, $\text{inds } t_2 = \{1, 2\}$
6. $\text{elems } t_1 = \{1, 13\}$, $\text{elems } t_2 = \{7, 17\}$
7. $\{ t_1(i) \mid i \in \text{inds } t_1 \land t_1(i) > 10 \} = \{13\}$
Appendix B

Auxiliary Functions

\textbf{extract-indices}

extract-indices: Subscript \rightarrow \mathbb{N}^+

\text{extract-indices}(s) \triangleq
\begin{align*}
\text{cases } s \text{ of} \\
\text{simple-subscript} \rightarrow [s] \\
\text{Vector-Subscript} \rightarrow s \\
\text{else} \rightarrow \text{let } sl = \text{mk-Triplet-Subscript}(lb, ub, by) \text{ in} \notag \\
\quad \text{if } by = [] \notag \\
\quad \text{then} \notag \\
\quad \quad [lb] \leadsto \text{extract-indices}(	ext{mk-Subscript-Triplet}(lb+1, ub, by)) \\
\quad \text{else} \notag \\
\quad \quad \text{if } (by > 0 \land lb > ub) \text{ then } [] \\
\quad \quad \text{elseif } (by > 0 \land lb < ub) \notag \\
\quad \quad \quad [lb] \leadsto \notag \\
\quad \quad \quad \text{extract-indices}(	ext{mk-Subscript-Triplet}(lb+by, ub, by)) \\
\quad \quad \text{elseif } (by < 0 \land lb < ub) \text{ then } [] \\
\quad \quad \text{elseif } (by < 0 \land lb > ub) \notag \\
\quad \quad \quad [lb] \leadsto \notag \\
\quad \quad \quad \text{extract-indices}(	ext{mk-Subscript-Triplet}(lb+by, ub, by))
\end{align*}

\textbf{Example A.1}

extract-indices(<1,3,2>)
\begin{align*}
\notag &= [1] \leadsto \text{extract-indices}(<3, 3, 2>) \\
\notag &= [1] \leadsto [3] \leadsto \text{extract-indices}(<5,3,2>) \\
\notag &= [1, 3] \leadsto [] \\
\notag &= [1,3]
\end{align*}