PM – Programming Parallel Models

Version 0.1

Language Reference – Preliminary Draft (incomplete)

© Tim Bellerby, University of Hull, UK

Contents

Background

PM (Programming Parallel Models) is a new open source programming language designed for implementing environmental models, particularly in a research context where ease of coding and performance of the implementation are both essential but frequently conflicting requirements. PM is designed to allow a natural coding style, including familiar forms such as loops over array elements. Language elements have been included (or more importantly excluded) from the language in a careful combination that enables the language implementation to both vectorise and parallelise the code. The following goals guided the development of the PM language:

- Language structures facilitate the generation of vectorised and parallelised code
- Vectorisation and parallelisation should be explicit and configurable
- Programmers should be able to concentrate on coding the algorithm – not on how to get it to run efficiently using whole-array operations and/or in parallel
- Support for data structures required in modelling: matrices, vectors, grids (including grids of matrices, vectors and sub-grids) and meshes with generalised connectivity.
- PM code should be as readable as possible by those not familiar with the language.
- PM should be formally specified – not defined by an implementation.

Conventions used to define language syntax

PM syntax will be described using the following extended BNF notation:

name ::= *list*	Define a non-terminal element in terms of other elements
name	Non terminal element
dim	Keyword
'>='	Character combination
[*elements*]	Elements are optional – may appear zero or one times
{ *elements* }	Elements may appear zero, one or more times
el1 | *el2*	Exactly one of the listed element sequences must be present
(el1 | el2)	Brackets may be used to group selections that are not enclosed by { } or [ ]

Lexical Elements

Comments start with a '!' and continue to the end of the line

```
! Comments are ignored
```

PM uses upper and lower case letters, digits and a number of special symbols:

letter::= **'a'** |**'b'** |**'c'** |**'d'** |**'e'** |**'f'** |**'g'** |**'h'** |**'i'** |**'j'** |**'k'** |**'l'** |**'m'** |**'n'** |**'o'** |**'p'** |**'q'** |**'r'** |**'s'** |**'t'** |**'u'** |**'v'** |**'w'** |**'x'** |**'y'** | **'z'**
'A' |**'B'** |**'C'** |**'D'** |**'E'** |**'F'** |**'G'** |**'H'** |**'I'** |**'J'** |**'K'** |**'L'** |**'M'** |**'N'** |**'O'** |**'P'** |**'Q'** |**'R'** |**'S'** |**'T'** |**'U'** |**'V'** |**'W'** |**'X'** |**'Y'** | **'Z'**
digit::= **'0'** | **'1'** | **'2'** |**'3'** | **'4'** | **'5'** | **'6'** | **'7'** | **'8'** | **'9'**

The following single characters and character combinations are used as ***delimiters***:

'$'   '&'   '('   ')'   ''   '*'   ','   '//'   '/'   '=>'   ':'   '||'   ';'   '@'   '['**
']'   '_'   '{'   '|'   '}'   '-'   '"'   '+'   '<'   '<='   '='   '=='   '>'   '>='   '%'
'..'   '.'   '->'   '::'

To cater for environments with restricted character sets, the following substitutions are always permitted:
'**(/**' for '**[**'   '**/)**' for '**]**'   '**(:**' for '**{**'   '**:)**' for '**}**'   '**%%**' for '**@**'   '**:/**' for '**|**'
'**://**' for '**||**'

White space characters (space, newline, form feed and horizontal tab) may appear between lexical elements (delimiters, names, keywords, numeric constants) but not within them. Names, keywords and numbers must be separated from each other by white space. Otherwise white space is optional.

New lines are not generally significant, but in any context where a semicolon ';' is expected, the semicolon may be omitted if the following lexical (non-white-space) symbol starts on a new line.

Names have a maximum length of 100 characters. They are comprised of upper and lower case letters, decimal digits, and the underscore character '_'. They may not start with a digit. Letter case is significant.

stem ::= *letter*{'_'|*letter*|*digit*}
name ::= ['_']*name*

Names that start with '_' are unique to the module within which they are defined. A module entity defined using such a name may not be referred to from another module.

The following are ***keywords*** and may not be used as names:

also	**and**	**any**	**arg**	**argc**	**as**
by	**case**	**check**	**const**	**debug**	**default**
dim	**div**	**do**	**else**	**elseif**	**enddebug**
enddo	**endfind**	**endfor**	**endif**	**endlet**	**endproc**
endselect	**endtype**	**endwhile**	**excludes**	**false**	**find**
follow	**for**	**found**	**from**	**high**	**if**
in	**include**	**includes**	**is**	**key**	**let**
low	**mod**	**not**	**null**	**opt**	**or**
otherwise	**over**	**param**	**proc**	**prod**	**rec**
reduce	**repeat**	**result**	**select**	**seq**	**step**
struct	**then**	**true**	**type**	**until**	**use**
using	**when**	**where**	**while**		

Numeric constants take the following forms*:*

number ::= [*integer_constant* | *real_constant* | *imaginary_constant*] [*bits*]
integer_constant ::= [{ *digit* } '**r**'] { [*digit* | *letter*] } ['**l**']
real_constant ::= { *digit* } ['**.**'*digit* { *digit* }] ['**e**' ['**+**' | '**-**'] digit { *digit* }]['**d**']
imaginary_constant ::= [*integer_constant* | *real_constant*] '**i**'
bits ::= '_' { *digit* }

String constants are enclosed by double quotes'"' and may include any character other than '"'. They may not span more than one line.

```
"Hello World"
```

Modular Structure

A PM program consists of one or more source files, each corresponding to a ***module***. A module defines a collection of ***procedures***, ***types*** and ***parameters*** (module level constants). A **program module** also contains executable statements. **Interface modules** specify the interface (API) that must be implemented by an actual model. Module names are constructed from a sequence of standard PM names optionally joined by ‘**.**‘delimiters. They are linked to source file names in an implementation-dependant manner. Module names starting with **lib.** refer to standard libraries.

```
module::= program_module | library_module
program_module::=
        [ decls ] statements
library_module::=
        decls [ debug statements enddebug  ]
modname::=
        name { '.' name }
```

Names may optionally start with an underscore character. Such names are local to the module in which they are used and do not match names in any other module into which they are imported.

```
decls::=
        { import ';' }  ( import | decl )  { ';' decl }
import::=
        use modname  [ modifiers ]  { ',' modname  [ modifiers ]  }
```

Modules may include other modules using a **use** statement. In its simplest form, **use** *module_name* the statement imports all definitions in the named module into the current module. Type and parameter declarations should not conflict with other definitions in the module containing the import statement – whether directly defined or imported from other modules. Imported procedure definitions merge with definitions imported to or declared in the module containing the include statement. This merging joins definitions across a PM program – if two different modules define a procedure with the same name, the merging of their definitions affects both the module imported to and both imported modules.

```
modifiers::=
        '{' modifier { ',' modifier } '}'
modifier::=
        [ type | param | proc  ]  name ‘=>’ name
```

Name clashes may be avoided by using a qualifying clause in the **use** statement. This lists specific elements to import and optionally renames them:

```
use model4 {
      type model_params
      param theta => model_theta
      test => test_model
}
```

The optional => operator in this context renames the given element. If an element in the list is not qualified as **type**, **param**, or **proc** then it is assumed to be a procedure (**proc**).

Types, procedures and parameters occupy separate name spaces – they may have the same name as each other.

Type, procedure and parameter declarations follow any include statements in the module. The simplest of these is the **param** declaration which simply defines a named constant:

```
paramdec::=
        param  name '=' xexpr
```

For example:

```
param pi = 3.14159265
```

param statements may refer to each other, irrespective of the order in which they are defined. However, such references must not be recursive or mutually recursive.

Type and procedure declarations are described later in this document.

In a program module, the (optional) declarations are followed by executable statements. Such a module may be run as a program. It may not, however, be included in other modules. Library modules may be included in other modules. They may not contain executable statements after the declarations except for an optional single debug statement that should contain module testing code. The PM system may optionally check that the library module debug statement references all procedures and types in the module –directly or indirectly - and issue a warning if this is not the case. Interface modules are described later in this document.

Types, values and objects

PM programs operate on *values* stored in *objects*. A *type* defines a (possibly conceptually infinite) set of values and is constructed using a *type constraint expression*. A type may consist of a set containing only a single value. However, it remains distinct from that value. A *conformance* relationship is defined between types and between values and types:

- A value V conforms to an abstract type T if $V \in T$
- An abstract type T conforms to abstract type T if $T \subseteq U$

While the PM type system provides flexibility similar to some dynamically typed languages, it is designed for static analysis. Run time type inference and procedure selection should only be necessary in situations where specifically polymorphic values are generated (using **any** or **opt** types or reference objects).

The universal type, which includes all other types if denoted using an underscore ('_'). It is usually not required, the absence of a type constraint serves a similar purpose in most cases where a type expression is expected.

User defined types

User defined types associate a name with a given set of type expressions, creating a new type that is the union of the listed types. A type is defined in a type definition:

```
typedecl::=
        type name [ '{' typeparams  '}' ] [ in namelist ] ( is typelist | includes typelist  | also includes typelist )
typeparams::=
        name [ ':' type ] { ',' name [ ':' type ] }
```

The simplest type definition simply associates a name with a set of types:

```
type x is int, struct{ x:int}
```

A parameterised type declaration defines a template from which to create types, parameterised by one of more other types:

```
      type point{ t:num} is struct{ x:t, y:t}
      type integer_point is point{int}
```

An open type declaration using the **includes** form provides an incomplete definition for a type which may be added to later. The type definition may be augmented using an **also includes** type definition or by using by declaring another expression to be **in** that type:

```
      type a includes int, real
      type a also includes string
      type b in a is cpx             ! Type a is now int, real, string, cpx
```

Augmenting type declarations do not have to be in the same module as the original declaration.

Open type declarations may also be parameterised:

```
      type point{T:num} includes rec point{ x:T,y:T }
      type point{T:num} also includes rec point{ x:T,y:T,z:T}
      type point{T:int16} also includes rec point{ combined_index: int32 }
```

If an **also includes** definition is parameterised, then the type constraints associated with its parameters must either be absent or must conform, parameter by parameter, with the type constraints in the original **includes** definition to which it refers. If type constraints are present, they control the circumstances under which the **also includes** template extends the original type template.

A type definition may not refer to itself recursively unless that recursive use occurs within the definition of an optional type.

The PM system defines a number of *intrinsic types*. These act as user defined types in declared in a system module implicitly imported into every other module.

<u>Numeric types</u>

PM supports a range of integer types, defined in the following table. The definitions of these types are flexible to enable portability of code. Code requiring, e.g., a strict bit size should use inquiry functions to check that a given type meets the expected requirements.

int	*short integer*	*System defined standard integer*	
long	*long integer*	*Integer capable of counting elements in larges possible array*	
int8	*8-bit integer*	*OR smallest integer holding ≥2 decimal digits*	*OR same as* **long**
int16	*16-bit integer*	*OR smallest integer holding ≥4 decimal digits*	*OR same as* **int8**
int32	*32-bit integer*	*OR smallest integer holding ≥9 decimal digits*	*OR same as* **int16**
int64	*64-bit integer*	*OR smallest integer holding ≥18 decimal digits*	*OR same as* **int32**
int128	*128-bit integer*	*OR smallest integer holding ≥36 decimal digits*	*OR same as* **int64**

Integer constants may be defined with respect to any base between 2 and 62 by specifying the base followed by '**r**' and then the digits, with '**a**'..'**z**'and '**A**'..'**Z**' representing 10 to 36 respectfully (case insensitive).

```
          123        2r101011101   16rdeadbeef
```

By default integer constants yield values conforming to **int**. Long integer constants should append an **l** to the value. For other integer types, append an underscore and the corresponding (assumed) number of bits:

```
889874324897284746                  ! long
255_8                               ! int8
2r10101010101010101010101010_32    ! int32
```

PM also defines a set of real (floating point) types:

real	*Standard (system defined) floating point value*		
double	*Standard (system defined) double precision floating point value*		
real32	*32-bit floating point*	*OR real number with ≥1 decimal digits in mantissa*	*OR same as* **real**
real64	*64-bit floating point*	*OR real number with ≥15 decimal digits in mantissa*	*OR same as* **real32**
real128	*128- bit floating point*	*OR real number with ≥24 decimal digits in mantissa*	*OR same as* **real64**

Real constants must contain a decimal point and may contain an exponent preceded by '**e**'.

```
-5.2       2e3          4.2e-20
```

By default real constants yield values conforming to type **real**. To specify a **double** constant, append a **d**. For other real types append and underscore and the assumed number of bits:

```
-5.2d      ! double precision
3.2e-3     ! real
1.2e-2_32  ! real32
```

Complex types are defined as follows:

cpx	*Complex number formed from* ***real*** *components*	
double_cpx	*Complex number formed from* ***double*** *components*	
cpx64	*64-bit complex number*	*OR complex number based on* **real32** *components*
cpx128	*128-bit complex number*	*OR complex number based on* **real64** *components*
cpx256	*256-bit complex number*	*OR complex number based on* **real128** *components*

Imaginary constants terminate with a letter '**i**' or '**j**'.

```
3.0i        ! cpx
1.0i_64     ! cpx64
-2.2-30di   ! double_cpx
```

The following intrinsic types define groups of numeric types:

- any_int is int, long, int8, int16, int32, int128
- any_real is real, double, real32, real64, real128
- any_cpx is cpx, double_cpx, cpx64, cpx128, cpx256
- int_num includes any_int
- real_num includes any_int, any_real
- cpx_num includes any_int, any_real, any_cpx
- num includes cpx_num
- std_int is int, long
- std_real is real,double
- std_cpx is cpx, double_cpx
- std_num is std_int, std_real, std_cpx

Numeric type balancing

Values of different numeric types may appear together in numeric expressions. The rules more mixed types are:

> The most general type is selected for the results and the least general value converted to that type. Generality is defined as follows (least..most):
>
> > **int long int8 int16 int32 int64 int128 real double real32 real64 real128 cpx double_cpx cpx64 cpx128 cpx256**

The **div** (integer divide) operator applies numeric balancing, and then selects the result type from the balanced type as follows:

int long int8 int16 int32 int64 int128	*as balanced type*
real double	long
real32	int32
real64	int64
real128	int128

Non-numeric types

The **bool** type contains the logical values **true** and **false**

The **string** type contains values that are arbitrary length character strings. ***String constants*** are enclosed by double quotes '"' and may include any character other than '"'. They may not span more than one line.

```
"Hello World"
```

The **name** type contains values that are valid PM names. Name constants are formed by preceding a name by the dollar symbol:

```
$x
```

The **proc** type contains values that are valid PM procedure names (including operator symbol). Procedure constants are formed by preceding a name by the **proc** keyword:

```
proc cell_model
proc +
```

Structures and records

Structures and records provide a mechanism to create aggregate values. They are very similar, except that it is not possible to alter a single component of a record, while this is possible for a structure.

A structure or record values is created using a generating expression:

[**struct** | **rec**] [*name*] '**{**' *name* '**=**' *exp* { '**;**' *name* '**=**' *exp* } '**}**'

For example:

```
struct { name="John Smith", age=42 }
rec point { x=2, y=4 }
```

A structure or record value tags each component with a name. The whole value may optionally tagged with a further name. Component names are local to the structure or value. The name tagging the whole structure or record resides in a name space that is global to the program (unless preceded by an '_' in which case it is local to the given module)

A structure or record type is constructed using:

[**struct** | **rec**] [*name*] '**{**' *name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] } '**}**'

For example:

```
struct { name:string, age:int }
rec point { x:int, y:int }
```

A structure or record value is a member of a given structure or record type if all of the following apply:

1. The value is a record and the type is defined using **rec** or the value is a structure and the type **struct**
2. The whole-value name matches the corresponding name given in the **struct**/**rec** type, or both are absent
3. The same component names are present (not necessarily in the same order)
4. If a type constraint is associated with a component name in the type expression then the corresponding component of the structure or record value, associated with that name, conforms to that type constraint.

A given component of a structure or record may be accessed using the '.' operator. Structure components may be modified using the same operator.

```
person:=struct { name="John Smith", age=42 }
location:=rec point { x=2, y=4 }
person.age=person.age+1
print(person.age)
print(location.x)
```

Polymorphic types

A polymorphic value has an internal representation that is capable of representing any value in a given type. A value of a given polymorphic type is created using a generating expression:

any [*type*] '**{**' *expr* '**}**'

If a type is not given then it is set equal to the concrete type of the expression (see section on variables below). The value contained within a polymorphic value is obtained using the unary '*' operator.
For example:

```
type int_or_string is int,string
a:= any int_or_string{ 1 }
a= any { 'Hello' }
print(*a)
```

A polymorphic type expression has the following form:

any '**{**' [*type*] '**}**'

A polymorphic value conforms to type **any{***T***}** if the value it contains conforms to *T*.

Optional types

Optional types are extensions of polymorphic types. An optional value has an internal representation that is capable of representing any value in a given type or the special value **null** (**null** is the single member of type **null**)

opt [*type*] '**{**' *expr* '**}**'

As with a polymorphic type, the value contained within an optional value is obtained using the unary '*****' operator. This will raise a runtime error if the contained value is **null**. An optional type expression has the following form:

opt '**{**' [*type*] '**}**'

An optional value conforms to type **opt{***T***}** if the value it contains conforms to *T* or is **null**.

Ranges

Range types represent open or closed intervals. Range values are created using the '..', '<..','..<' or '<..<' operators. For numerical values mixed type promotion rules apply. E.g:

```
closed_range:= 0..10
part_open_range:= 0<..10          ! Contains 1..10
part_open_range2:= 0..<10         ! Contains 0..9
real_open_range:=0<..<10.0        ! Contains { x | x>0.0 and x<10.0 }
```

The two bounds of a range value R may be obtained using the procedures low(R) and high(R).

Range types are denoted by:

range '**{**' [*type*] '**}**'
open_start_range '**{**' [*type*] '**}**'
open_finish_range '**{**' [*type*] '**}**'
open_range '**{**' [*type*] '**}**'

Sequences

A sequence type represents a sequence of values. Sequence values are created by applying the **by** operator to a range value. The second argument to **by** gives the step between sequence elements. Numeric balancing will apply between the start/finish values of the range and the step argument.

```
integer_sequence := 1..6 by 2
real_sequence := 3.2 .. 5.4 by 0.7
```

The step of a sequence S may be found using the procedure step(S). The two bounds of the underlying range value may be obtained using the procedures low(S) and high(S).

Sequence types are denoted by:

seq '**{**' [*type*] '**}**'

Sets

Set types incorporate arbitrary sets of values. Set values are formed using a set generator expression:

```
        '{' exprlist '}'
exprlist ::=
        expr { ',' expr }
```

For example:

```
      colours := { $red, $green, $blue, 16r1A3F2A }
      numbers:= { 1 ,2, sqrt(2), a+b }
```

The **in** operator allows set membership to be checked:

```
      if colour in colours then
```

Set types are denoted by:

```
        set '{' [ type ] '}'
```

A set value conforms to type **set{***T***}** if every value in the set conforms to *T*.

Grids

A grid type defines an *N*-dimensional (2<*N*≤7) grid of points. Each dimension of the grid may be defined by a value conforming to **grid_base**. By default this is either (1) a numeric range (2) an integer value *M* denoting the range 1..*M* (3) a set – in which case the ordering of elements along this dimension is arbitrary. There are seven specific grid types of varying number of dimensions, and a generalised grid type encompassing all of these:

- grid_base includes int,long,range{std_int},seq{}
- grid1d is grid{grid_base}
- grid{T:grid_base} is T
- grid2d is grid{grid_base,grid_base}, matrix_dom
- grid3d is grid{ grid_base,grid_base,grid_base }
- grid4d is grid{ grid_base,grid_base,grid_base ,grid_base }
- grid5d is grid{ grid_base,grid_base,grid_base ,grid_base ,grid_base }
- grid6d is grid{ grid_base,grid_base,grid_base,grid_base,grid_base,grid_base}
- grid7d is grid{ grid_base,grid_base,grid_base ,grid_base ,grid_base ,grid_base ,grid_base }
- grid is grid1d, grid2d, grid3d, grid4d, grid5d, grid6d, grid7d

Grid values are created by calling the **grid** procedure:

```
model_grid := grid (0..num_cols, 0..num_rows,{$Pressure,$Temp,$Humid} )
```

The value corresponding to each element of a grid value G may be obtained using the procedures **d1**(G), **d2**(G), … , **d7**(G).

Domains

A domain defines a set of indices over which a (possibly parallel) iteration may be performed and over which arrays may be defined. The intrinsic domain type domain is defined as:

```
type dom includes grid_base,grid
```

All valid domain values D support the num_elem(D) procedure which returns the number of elements in the domain.

Base domains

All domains have a corresponding base domain, which may be determined using the **base** procedure. This is defined as follows:

- The base domain of a vector or matrix domain is equivalent to the original domain
- The base domain of a grid domain **grid{**T_1,T_2,... **}** is given by
  proc base(g: grid{ T_1,T_2,... **})= grid(num_elem(d1(g)),num_elem(d2(g), ...)**
- The base domain of any other domain *D* is given by default by **num_elem(***D***)**

Arrays

An array type is a generic type indicating the storage of a set of values over corresponding to elements of a given domain. Array values may be created using the **dim** operator:

expr **dim** *expr*

e.g.: `0.0 dim grid(0..3,0..3)`. The operator creates a new array by replicating the element value over every point in the given domain. The second argument to the **dim** operator must conform to domain.

Array types are defined using:

type '**[**' *typelist* '**]**' *typelist*::= *type* { '**,**' *type* }

An array value conforms to array type T[U] if all of its elements conform to T and its domain conforms to U.

One and two dimensional array values, with domains **int** and **grid{int,int}** respectively, may be defined using the following syntax:

[' *list2d* '**]**' *list2d*::= *exprlist* { '**;**' *exprlist* } ['**;**']

For example:

```
array_1d := [ 1, 2, 3 ]                        ! domain is 3
array_2d := [ 1, 2, 3 ; 4, 5, 6; 7, 8, 9]     ! domain is grid(3,3)
```

Array assignment requires that the two participating arrays have the same base domain. Arrays cannot change size – flexible size arrays must be implemented using **any** or **opt** types. An array of type int[int] cannot change size. A value of type any { int[int] } can refer to arrays of different sizes.

Vectors and Matrices

A vector is a numeric array defined over the **vect_dom** domain, which is a special one dimensional domain with a long integer dimension. A **vect_dom** value is created using the **vect dom** procedure:

```
        zero_vector = 0.0 dim vect_dom(3l)
```

A matrix is a numeric array defined over the **mat_dom** domain, which is a special two dimensional domain with two long integer dimensions. A **mat_dom** value is created using the **matrix dom** procedure:

```
        zero_matrix = 0.0 dim mat_dom(3l,3l)
```

Vector and matrix values may be created using a generating expression:

```
        '('list2d ')'
list2d::=
        exprlist  {  ';' exprlist } [';']
```

For example:

```
       v:=    (1, 3, 1)
       a : = ( 3, 0, 0
               0, 2, 1
               0, 0, 1 )
```

Note the use of the optional semicolon rule in this example.

Matrix and vector values follow the usual rules for matrix multiplication. Vectors act as row or column matrices as appropriate in the context of matrix multiplication.
The vector and matrix intrinsic types are defined as:

- vect is [vect_dom]
- mat is [mat_dom]

Partitions and Distributions

A *partition* exhaustively divides a domain into non-intersecting sub-domains each associated with a point in a second, partitioning, domain. Conceptually, a partition acts as an array of domain values although it is not defined as such.

A *distribution* extends this concept a stage further, conceptually consisting of an array of arrays of domains. There is no single way to create a partition or distribution, they must be generated using appropriate intrinsic procedures.

The corresponding intrinsic types are **part{t:dom,p:dom}** and **distr{t:dom,p:dom,d:dom}.**

For example:

```
        domain:= grid(16,16)
        part_domain:= grid(4,4)
        partition:= block(domain,part_domain)
        block:=partition[2,2]                    ! yields grid(5..8,5..8)
        distribution:= block_cyclic(domain,partition,block=grid(2,2))
        sub_block:=distribution[2,2][1]          ! yields grid(3..4,3..4)
```

Variables, constants and references

Objects are created and associated with names using a declaration statement:

```
x := 4
const message = "Hello World"
```

An object has a *concrete type* that defines the set of values it is capable of storing. The concrete type of an object is determined by as follows:

1. If the initial value conforms to any of: **int long int8 int16 int32 int64 int128 real double real32 real64 real128 cpx double_cpx cpx64 cpx128 cpx256 bool string name proc** then then that type is the concrete type of the object.
2. If the initial value is a structure or record, then the concrete type is a structure or record type with component type constraints given by the concrete types of the corresponding components, determined by applying these rules recursively.
3. If the initial value is a polymorphic value created using **poly** *T* **{** *expr* **}** then the concrete type is **poly {** *T* **}**
4. If the initial value is an array, then the concrete type is an array type with element and domain types equal to concrete types obtained recursively from corresponding components of the array value.

If the name is declared using **var** or '**:=**', values in the object may be changed later in the program. If it is declared using **const** then no changes to its value are permitted. The restriction for **const** names is implemented by forbidding them from appearing on the left hand side of an assignment statement or as a reference argument to a procedure. A name is defined until the end of the block of statements in which it is defined. Local (var, const and reference - defined later) names may not local shadow names defined in enclosing blocks or clash with procedure parameter names.

The values stored in variables may change over time. Values are changed in one of two ways: (i) using an assignment statement and (ii) by passing the variable as a reference argument to a procedure.

```
var x = 4     ! Value stored in x is 4
x = x + 1     ! Value changed to 5
float(&x)     ! Value changed to 10
              ! - assuming procedure float has been defined
```

Some procedures return more than one value. In this case, multiple left hand sides are permitted:

assignment ::=
    lhs '**=**' *expr*
    lhs '**,**' *lhs* { '**,**' *lhs* } '**=**' *call*
    name '**->**' *name qualifier*
lhs::=
    name qualifier | '**_**'

qualifier::=
    { '**.**' *name* | '**[**' *exprlist* '**]**' | '**{**' *exprlist* '**}**' }

For example:

```
x,y,_, z = returns_four_args(a)
```

An underscore may be used as a place filler, allowing a given returned value to be ignored. It is also possible for the left hand side of an assignment to refer to a component of an object.

```
a.f = b
a[3,2]=c
```

A reference declaration assigns a name directly to an existing object, without creating a new object first. A reference is both created and changed using the same statement form:

> *assignment* ::=
> *name* '**->**' *name qualifier*

For example:

```
r -> x
r -> y.f
r -> a[3..9]
```

A reference declaration has the same scope as a variable or constant declaration. References can change type and are intrinsically polymorphic.

Procedures

A procedure defines an operation on a set of objects, which may change some of their stored values. Procedures may also return one of more values. A procedure declaration defines the name of the procedure, parameters on which it operates and their associated type constrains and any values returned.

```
proc square(x) = x**2

proc calc_stats(data: float dim any) do
     ……
     result=mean,std_dev
endproc
```

Procedure names may be simple names (with or without a leading underscore) or may be one of the following operators:

+ - div mod * / ** == /= > >= and or // => dim =[] {} opt
.. <.. ..< <..< by

Note the absence of **<** or **<=** from this list – these operators are always defined relative to **>** or **>=** respectfully.

Each procedure is associated with a **signature** consisting of the following:

1. The name of the procedure
2. The number of values the procedure will return.
3. The number of arguments the procedure will accept
4. The type constrains on the procedure's parameters
5. Which parameters are declared to be reference parameters

Keyword parameters (defined below) are not part of the signature.

A procedure signature *P* is said to be **strictly more specific** than another procedure signature *Q* if:

1. The procedure names are the same.
2. The numbers of parameters are the same (allowing for variable argument definitions in either procedure).
3. All reference parameters occur in the same position.
4. The type constraint for each parameter of *P* conforms to the type constraint for the equivalent parameter of *Q*
5. Type constrains are not strictly equal to one another for at least one parameter position.

A procedure call invokes a procedure, supplying it with a list of values and/or objects to operate on. If the call is part of an assignment statement (discussed in detail below) then the call also provides a list of objects to receive any values generated or declarations to receive created objects.

```
u, var v = myproc(x,&y,z*2)
```

If a procedure call is nested inside another procedure call, then the nested call is assumed to return a single object to the definition of an anonymous intermediate constant (again see the discussion below for details on declarations). Thus the following are equivalent:

```
y = f(g(x))

const _anon = g(x)
y = f(_anon)
```

Reference parameters, denoted by **&**, indicate that the procedure may change the value stored in corresponding object. Any procedure call must precede the corresponding argument with an **&**.

When encountering a procedure call, PM finds all procedures with signatures that ***match*** the call. A call matches a procedure signature if :

1. The procedure name is the same.
2. The number of arguments equals the number of parameters defined by the signature, or is greater than or equal to the number of parameters for a variable arguments signature.
3. All reference parameters are associated with a reference argument, starting with an &, in the same position.
4. The object supplied for each argument conforms to the type constraint for the corresponding parameter.

If more than one procedure matches a given call then PM will try to find a procedure that is strictly more specific than all the other candidate procedures. If no such procedure exists, then the call is ***ambiguous*** and the procedure selection is arbitrary. This condition may be checked for by the PM system and warnings issued.

Keyword arguments, if present, are not considered when finding a matching procedure. However, if the matching procedure will not accept a supplied keyword parameter then an error condition will arise.

A procedure may accept a variable number of arguments:

```
proc set_numbers(&dest:int[],arg:int...) do  ....
```

From the body of the procedure, the excess arguments are accessed using **arg[]** and **argc**. **argc** returns the number of arguments in positions corresponding to and following that of **arg…** as an **int** value. **arg[]** must be supplied an **int** value in the range **1..argc**, otherwise it will raise and error condition. It returns the value of the corresponding argument**.**

```
for i in 1..argc do
    dest[i] = arg[i]
endfor
```

It is also possible to pass optional arguments *en masse* to a procedure call:

```
proc process(a) do
    print("Value is"//a)
enproc

proc process(a,arg...) do
    process(a)
    process(arg...)
endproc

process_values(1,2,3,4)
```

Arguments passed in this way <u>do</u> form part of the signature of the call – they are used in the matching process and do not necessarily have to correspond to the variable argument portion of the called procedures parameter list.

Keyword parameters provide optional values to the procedure call. They are specified by providing a default value. Keyword parameters follow all other parameters and **arg ...** (if present). Keyword parameters <u>do not</u> form a part of the signature of the procedure. An error results if the procedure selected by signature matching is not able to receive the keyword arguments provided in the call.

```
proc run_model(params : model_params, iterations = 1000,
    relaxation = 0.02) do
         .......
endproc

run_model(my_param_set, relaxation=0.042)
```

It is possible to define a procedure that takes additional unknown keyword parameters and passes these on to other procedures. This is achieved using the **key** keyword:

```
proc process_opts(&optarray,first_opt=.false,second_opt=33,key...) do
    ....
    process_other_opts(key...)
endproc
```

These additional keyword arguments may only be passed to another procedure – there is no facility to inspect them individually.

Expressions

An expression consists of a set of nested procedure calls. The usual infix notation is provided for common mathematical operations, but this is *syntactic sugar* for procedure calls.

Operation	Prior -ity	Equivalent call	Description
`x[a,b]`	1	`proc [] (x,a,b)`	Array subscript
`-x`	2	`proc - (x)`	Minus
`x**y`	3	`proc ** (x,y)`	Power
`x*y`	4	`proc * (x,y)`	Multiplication (including matrix multiplication)
`x/y`	4	`proc / (x,y)`	Division (float result)
`x div y`	5	`proc div (x,y)`	Division (integer result)
`x mod y`	5	`proc mod(x,y)`	Remainder
`x + y`	6	`proc + (x,y)`	Addition
`x - y`	6	`proc - (x,y)`	Subtraction
`x == y`	7	`proc == (x,y)`	Equals
`x /= y`	7	`proc /= (x,y)`	Not equal
`x > y`	7	`proc > (x,y)`	Greater than
`x >= y`	7	`proc >= (x,y)`	Greater than or equal
`x < y`	7	`proc > (y,x)`	Less than
`x <= y`	7	`proc >= (y,x)`	Less than or equal
`x and y`	8	`proc and (x,y)`	Logical and
`x or y`	9	`proc or (x,y)`	Logical or
`x // y`	10	`proc // (x,y)`	String concatenation
`x .. y`	11	`proc .. (x,y)`	Range creation
`x <.. y`	11	`proc <.. (x,y)`	Range creation
`x ..< y`	11	`proc ..< (x,y)`	Range creation
`x <..< y`	11	`proc <..< (x,y)`	Range creation
`x by y`	11	`proc by (x,y)`	Sequence creation
`x => y`	12	`proc => (x,y)`	If x is true then y else null optional value
`x || y`	13	`proc || (x,y)`	If x is not null (or null optional value) then x else y
`x => y || z`	13	`proc||(proc=>(x,y),z)`	If x is true then y else z
`x dim y`	14	`proc dim(x,y)`	Array creation – spread copies of x over domain y

Sub-expressions may be separated from the main expression using **where**.

```
        s = a * -exp (b/a) where a = c/sqrt(b)
```

There may be multiple values defined after a where keyword. These clauses may not refer to each other, but it is possible to follow with a second where statement and associated clauses:

```
        a = x **2 / y**3 where x = s/p, y=s/q where s = sqrt(p**2 + q**2)
```

Subscripts

Subscript expressions have the following syntax:

'**[**' *sexpr* { '**,** ' *sexpr* } '**]**' | '**{**' *sexpr* { '**,** ' *sexpr* } '**}**' }
sexpr::=
expr | [*expr*] ('..' | '<..' | '..<' | '<..<') [*expr*] [**by** [*expr*]]

Subscript expressions using square brackets [] return a single value from a domain or array or alternatively a sub-domain or sub-array. If the subscript expression falls out of the bounds of the domain then an error condition arises. Subscript expressions using braces {} return an optional value or array of optional values. This will be null for points which fall outside of the domain.

Subscript expressions may contain the keywords **high**, **low**, or **step**. If this occurs in a given index position of the subscript expression list, then then these are converted to calls to the equivalent procedures on the corresponding dimensions of the array's domain.

```
x[low..high/2,low..high by step*2 ]
```

is equivalent to:

```
x[low(d1(x))..high(d1(x))/2,low(d1(x))..high(d1(x))/2 by step(d2(x))*2]
```

If the subscript expression consists of a range generator or a direct combination of range and sequence generators (**x..y by z**) then the start, end or step expressions may be omitted, in which case they are set to **low**, **high** or **step** respectfully. Thus the above expression could be written:

```
x [ .. high/2 , .. by step*2 ]
```

Statements

PM provides a range of conventional control statements:

If *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
select *xexpr* { (**case** *xexprlist* | **default**) **do** *statements* } **endselect**
while *xexpr* **do** *statements* **endwhile**
repeat *statements* **until** *xexpr*

For example:

```
if x > y then
     print("X is greater")
endif

if x>y then
     print("X is greater")
else
     print("Y is greater or equal")
endif

select digit
     case '1','3','5','7','9' do print("Odd")
     case '2','4','6','8' do print("Even")
     otherwise do print("?")
endselect

repeat
     x = x/2
     y = y+1
until x == 0

while x>0 do
     x = x/2
     nbits = nbits + 1
endwhile
```

These statements have usual interpretations. Conditional expressions must return a **bool** value. A runtime error will result if they do not. The **select** statement builds sets of values (which do not have to be constants) and uses the set **in** operator to check the applicability of each case.

Two statements allow for debugging and may optionally not be compiled and executed. The **check** statement confirms that an expression yields a **true** value and raises an error if this is not the case. The **debug** statement executes a block of code only if debugging is enabled.

check *xexpr*
debug *statements* **enddebug**

For example:

```
check value /= null

debug print("Entering main loop") enddebug
```

<u>Parallel execution</u>

The **for** statement executes its body in parallel for every element of a domain or array:

for *iter subexp* [**seq** | [**using** *assignlist*]] *loopbody* **endfor**
iter::=
name { **‘,’** *name* } **in** *exp* { **‘,’** *exp* }
name **from** *exp* **follow** *qualifier*
loopbody::=
do *statements*
find *statements* [**otherwise** *statements*]

For example:

```
for pixel in image do
      pixel = min(pixel,threshold)
endfor
```

It is possible to iterate over more than one domain and/or array, providing they all share the same base domain:

```
for pixel in image1, pixel2 in image2 do
      if pixel+pixel2>threshold then
            pixel=pixel-threshold/2
            pixel2=pixel2-threshold/2
      endif
endfor
```

Statements in the body of a parallel loop are not normally allowed to modify variables defined outside of the scope of the **for** statement. It is also possible to require that the loop body is executed sequentially, in which case these restrictions do not apply.

```
      for s in array seq do
            sum = sum + f(s)
      endfor
```

Numerical models will usually require interaction between loop invocations. PM enables this using the '@' operator. Used as a unary operator (@x) this provides an array view of a given expression across all invocations. Used as a binary operator (x@ndb_desc) it returns values from a local neighbourhood of the calling invocation. For example:

```
    for cell in model_array do
        advection=advection_model(cell)
        advected_cell=cell@advection
        cell=model(advected_cell,parameters)
    endfor
```

In the latter case, the result will be an array with an optional type and off-edge values undefined.

```
      for pixel in image do
            pixel = median(pixel@grid(-1..1,-1..1))
      endfor
```

The neighbourhood is defined with respect to the common base domain of the domains/arrays being iterated over, shifted so that the zero point lies at the location of the point associated with the current invocation.

The '**::**' operator repeatedly applies a given binary operation applied between the value of a given expression across all invocations of the enclosing parallel loop:

```
      for cell in model_grid do
            cell=model(cell,parameters)
            cell=cell/sum::cell     ! Normalise
      enddo
```

A reduction operation must be defined using a specialised procedure definition:

```
      proc sum:: (x,y) = x + y
```

The binary operation is applied with arbitrary ordering and bracketing. This will only give a deterministic result if the operation is both commutative and associative.

It is also possible to define procedures incorporating '**@**' and '**::**' operators that can only be called inside parallel for statements:

```
            proc @mean_filter(x,n)=sum(nbd)/count(nbd)
                  where nbd=grid(-n..n,-n..n)

            for pixel in image do
                  pixel = @mean_filter(x,1)
            endfor
```

To ensure synchronisation, an '**@**' operator, '**@**' procedure call or '**::**' operator may not occur inside a conditional statement (if, select, while, repeat) or a sequential for statement, unless its iteration domain is entirely independent of the current invocation of the enclosing parallel for statement.

The version of the **for** statement described above always invokes the loop body for every element of the domain. A second version executes arbitrary invocations of the loop body until at least one such body encounters a **found** statement:

found *assignlist*

Out of the loop invocations that encounter the **found** statement, only one will execute that statement. If all loop invocations complete without encountering a **found** statement, then the statements in the **otherwise** clause (if present) are executed. This clause is only executed once – not as a parallel invocation – and is thus not restricted with respect to side effects.

```
for pixel in image find
     if pixel>threshold then
          found high_pixel=pixel
     endif
otherwise
     high_pixel=-1
endfor
```

The body of a **find** loop may not use the **@** or **::** operators, except within nested parallel **for** – **do** statements.

Task parallelism is supported by the **do**.. **enddo** construct.

[using *keyargs* **] do** *statements* **{ also do** *statements* **} enddo**

The semantics of a **do** statement are defined according to an equivalent **for** statement.

using *opts* **do** *statements*$_1$ **also do** *statements*$_2$ … **also do** *statements*$_N$ **enddo**

is semantically equivalent to:

```
for _index in 1..N using opts do
     select _index
          case 1 do statements_1
          case 2 do statements_2
           …
          case N do statements_N
     endselect
endfor
```

The exception to this direct equivalence is that the implied **select** statement does not prevent the presence of **@** or **::** forms within the its scope.

Parallel expressions

FORTHCOMING…

Tasks and processors

FORTHCOMING…

<u>Syntax</u>

module::= *program_module* | *library_module*

program_module::=
  [*decls*] *statements*

library_module::=
  decls [**debug** *statements* **enddebug**]

decls::=
  { *import* '**;**' } (*import* | *decl*) { '**;**' *decl* }

import::=
  use *modname* [*modifiers*] { '**,**' *modname* [*modifiers*] }

modname::=
  name { '**.**' *name* }

modifiers::=
  '**{**' *modifier* { '**,**' *modifier* } '**}**'

modifier::=
  [**type** | **param** | **proc**] *name* '**=>**' *name*

decl::=
  procdecl | *typedecl* | *paramdecl*

signature::=
  procsig | *typesig* | *paramsig*

procdecl::=
  proc *procname* *params* '**=**' *xexprlist* [**do** *statements* **endproc**]
  proc *procname* *params* [**check** *exprlist*] **do** *statements* [**result** '**=**' *xexprlist*] **endproc**

procname::=
  [*name* | *op* | *name* '**::**' | **low** | **high** | **step** | '**@**' *name*]

op::=
  '+' | **'-'** |**div** | **mod** |**'*'** | **'/'** | **'**'**| **'=='** | **'/='** | **'>'**| **'>='** | **and** | **or** | **'//'** | **'=>'** | **'||'** | **dim** | **'='**| **'[]'** | **'{}'** | **opt** |
  '..' | **'<..'** | **'..<'** | **'<..<'** | **by**

params::=
  '**(**' { *param* '**,**' } [*param* ['**,**' *keyparams*] | **arg** '**...**' ['**,**' *keyparams*] | *keyparams*] '**)**'

param::=
  ['**&**'] *name* ['**:**' *type*]

keyparams::=
  { *name* '**=**' *expr* '**,**'} (*name* '**=**' *expr* | **key** '**...**')

namelist::=
  name { '**,**' *name* }

typedecl::=
  type *name* ['**{**' *typeparams* '**}**'] [**in** *namelist*] (**is** *typelist* | **includes** *typelist* | **also includes** *typelist*)

typeparams::=
  name '**:**' *type* { '**,**' *name* '**:**' *type* }

paramdec::=
  param *name* '**=**' *xexpr*

type::=
  name ['**{**' *typelist* '**}**']
  type '**[**' *typelist* '**]**'
  [**any** | **opt**] '**{**' [*type*] '**}**'
  [**struct** | **rec**] [*name*] '**{**' *name* ['**:**' *type*] { '**,**' *name* ['**:**' *type*] } '**}**'

statements::=
  statement { '**;**' *statement* } ['**;**']

statement::=
  If *xexpr* **then** *statements* { **elseif** *xexpr* **then** *statements* } [**else** *statements*] **endif**
  select *xexpr* { (**case** *xexprlist* | **default**) **do** *statements* } **endselect**
  while *xexpr* **do** *statements* **endwhile**
  repeat *statements* **until** *xexpr*

for *iter subexp* [**seq** | [**using** *keyargs*]] *loopbody* **endfor**
[using *keyargs* **] do** *statements* **{ also do** *statements* **} enddo**
check *xexpr*
debug *statements* **enddebug**
found *assignlist*
const *assignlist*
let *assignlist*
assignment
call

iter::=
name { **‘,’** *name* } **in** *exp* { *‘,’ exp* }
name **from** *exp* **follow** *qualifier*

loopbody::=
do *statements*
find *statements* [**otherwise** *statements*]

definition ::=
(*name*| '_') { **','** (*name* | '_') } **':='** *xexpr*

assignlist::=
assignment { ',' *assignment* } *subexp*

assignment ::=
lhs **'='** *expr*
lhs **','** *lhs* { **','** *lhs* } **'='** *call*
name **'->'** *name qualifier*

lhs::=
name qualifier | **'_'**

expr::=
lowest to highest precedence
expr **dim** *expr*
expr **'||'** *expr*
expr **'=>'** *expr*
expr [**'..'** | **'<..'** | **'..<'** | **'<..<'** | **by**] *expr*
expr **'//'** *expr*
expr **or** *expr*
expr **and** *expr*
not *expr*
expr [**'=='** | **'/='** | **'>'**| **'<'** | **'>='** | **'<='** | **in**] *expr*
expr [**'+'** |**'-'**] *expr*
expr **mod** *expr*
expr **div** *expr*
expr [**'*'**|**'/'**] *expr*
expr **'**'** *expr*
'-' *term*
term qualifier

subexp::=
[**check** *exprlist*] { **where** *namelist* '='*expr* { **','** *namelist* '='*expr* } }

xexpr::=
expr subexp

xexprlist::=
exprlist subexp

exprlist ::=
expr { **','** *expr* }

call::=
[*name* | **low** | **high** | **step**] '**(**'*arglist* '**)**'
proc *procname* '**(**'*arglist*'**)**'

term::=
subterm
value
'**@**' *subterm*
'**@**' *name* '**(**' *arglist* '**)**'
subterm '**@**' *subterm*
subterm::=
[**any** | **opt**] *type* '**{**' *subterm* '**}**'
name
'**(**'*expr*'**)**'
call
array
[**struct** | **rec**] [*name*] '**{**' *name* '**=**' *exp* { '**;**' *name* '**=**' *exp* } '**}**'
value::=
constant
arg '**[**' *expr* '**]**' | **argc** | **low** | **high** | **true** | **false**
proc *procname* | **type** *type*
constant::=
number | *string* | '**$**' *name*
arglist ::=
{ *arg* '**,**' } (*arg* | **arg** '**...**') ['**,**' *keyargs*]
[*keyargs*]
arg::=
'**&**' *name qualifier* | *expr*
keyargs::=
{ *name* '**=**' *expr* '**,**' } (*name* '**=**' *expr* | **key** '**...**')
qualifier::=
{ '**.**' *name* | '**[**' *sexpr* { '**,**' *sexpr* } '**]**' | '**{**' *sexpr* { '**,**' *sexpr* } '**}**' }
sexpr::=
expr | [*expr*] ('..' | '<..' | '..<' | '<..<') [*expr*] [**by** [*expr*]]
array::=
['**(**'*list2d* '**)**' | '**[**' *list2d* '**]**' | '**{**' *exprlist* '**}**' | '**(**' *generator* '**)**' | '**[**' *generator* '**]**' | '**{**' *generator* '**}**']
list2d::=
exprlist { '**;**' *exprlist* } ['**;**']
generator::=
expr '**:**' *iter* [**using** *keyargs*] { '**;**' *iter* [**using** *keyargs*] } ['**|**' *expr*]