

PM –Parallel Models Programming Language

Version 0.4

Revision 02

Language Reference (incomplete)

© Tim Bellerby, University of Hull, UK

This work is licensed under a [Creative Commons Attribution 4.0 International License](http://creativecommons.org/licenses/by/4.0/).

November 2023

Contents

[1. Introduction 8](#_Toc150278210)

[i. Background 8](#_Toc150278211)

[ii. Inspiration 8](#_Toc150278212)

[iii. Executing a PM program. 8](#_Toc150278213)

[iv. Conventions used to define language syntax 8](#_Toc150278214)

[2. Lexical Structure 10](#_Toc150278215)

[3. Modular Structure 12](#_Toc150278216)

[i. Importing declarations from other modules 12](#_Toc150278217)

[ii. Type declarations 13](#_Toc150278218)

[iii. Procedure declarations 13](#_Toc150278219)

[iv. Parameter declarations 13](#_Toc150278220)

[4. Simple types 14](#_Toc150278221)

[i. Types and values 14](#_Toc150278222)

[ii. Type conformance 14](#_Toc150278223)

[iii. Implicit type conversions 14](#_Toc150278224)

[iv. Concrete types 15](#_Toc150278225)

[a) Integer types 15](#_Toc150278226)

[b) Real types 15](#_Toc150278227)

[c) Complex types 16](#_Toc150278228)

[d) Boolean type 16](#_Toc150278229)

[e) String type 16](#_Toc150278230)

[v. Compile-time types 16](#_Toc150278231)

[a) Null type 16](#_Toc150278232)

[b) Proc types 16](#_Toc150278233)

[c) Procedure signature types 17](#_Toc150278234)

[d) Value types 17](#_Toc150278235)

[e) Types as values 17](#_Toc150278236)

[f) Unique values 18](#_Toc150278237)

[5. Structured types 19](#_Toc150278238)

[i. Structures and records 19](#_Toc150278239)

[ii. Embedded elements in structures and records 20](#_Toc150278240)

[iii. Abstract embedded types and embedded type conversion 20](#_Toc150278241)

[iv. Tuples 21](#_Toc150278242)

[6. Abstract types and Type declarations 22](#_Toc150278243)

[i. Type constraint expressions 22](#_Toc150278244)

[ii. Universal type 22](#_Toc150278245)

[iii. Logical type expressions 22](#_Toc150278246)

[iv. Type declarations 23](#_Toc150278247)

[v. Parameterised type declarations 23](#_Toc150278248)

[vi. Open type declarations 24](#_Toc150278249)

[vii. Recursive type declarations 24](#_Toc150278250)

[7. Procedures 26](#_Toc150278251)

[i. Procedure definitions 26](#_Toc150278252)

[ii. Procedure calls 26](#_Toc150278253)

[iii. Reference parameters and arguments 27](#_Toc150278254)

[iv. Reference argument aliasing 27](#_Toc150278255)

[v. Variable-length argument lists 28](#_Toc150278256)

[vi. Keyword parameters and arguments 28](#_Toc150278257)

[vii. Procedure calls using procedure values 28](#_Toc150278258)

[viii. Procedure signatures 29](#_Toc150278259)

[ix. Matching procedure calls to procedure definitions 30](#_Toc150278260)

[x. Procedure specialisation 30](#_Toc150278261)

[xi. Type constraints between different arguments to a procedure 30](#_Toc150278262)

[xii. Procedure and procedure call attributes 31](#_Toc150278263)

[8. Objects 32](#_Toc150278264)

[i. Variables and constants 32](#_Toc150278265)

[ii. Assigning new values to variables 33](#_Toc150278266)

[iii. Assignment operators 33](#_Toc150278267)

[iv. Mixed initialisation/assignment 34](#_Toc150278268)

[v. Object resource management 34](#_Toc150278269)

[9. Expressions 35](#_Toc150278270)

[i. Operator expressions 35](#_Toc150278271)

[ii. Sub-expressions 36](#_Toc150278272)

[10. Sequential control statements 37](#_Toc150278273)

[i. Blocks 37](#_Toc150278274)

[ii. Proceed statement 37](#_Toc150278275)

[iii. Do statement 37](#_Toc150278276)

[iv. If statement 37](#_Toc150278277)

[v. Switch statement 38](#_Toc150278278)

[vi. Until statement 38](#_Toc150278279)

[vii. While statement 38](#_Toc150278280)

[viii. Foreach statement 38](#_Toc150278281)

[ix. Test statement 39](#_Toc150278282)

[11. Polymorphism and dynamic dispatch 41](#_Toc150278283)

[i. Polymorphic types 41](#_Toc150278284)

[ii. Dynamic dispatch using the any statement 41](#_Toc150278285)

[12. Ranges, Extents, Sequences and Grids 43](#_Toc150278286)

[i. Ranges 43](#_Toc150278287)

[ii. Extents 43](#_Toc150278288)

[iii. Strided ranges 43](#_Toc150278289)

[iv. Generic sequence type 44](#_Toc150278290)

[v. Open ranges and sequences 44](#_Toc150278291)

[vi. Blocked sequences 44](#_Toc150278292)

[vii. Mapped sequences 45](#_Toc150278293)

[viii. Grids 45](#_Toc150278294)

[ix. Sequence and Grid Operations 45](#_Toc150278295)

[13. Parallel and distributed execution model 47](#_Toc150278296)

[i. Strands, nodes and node ownership 47](#_Toc150278297)

[ii. Partitioned Global Address Space 47](#_Toc150278298)

[iii. Topologies and Work-sharing 48](#_Toc150278299)

[14. Distributions 49](#_Toc150278300)

[i. Overview 49](#_Toc150278301)

[ii. No distribution 49](#_Toc150278302)

[iii. Direct distribution 49](#_Toc150278303)

[iv. Fixed block distribution 49](#_Toc150278304)

[v. Variable block distribution 49](#_Toc150278305)

[vi. Cyclic distribution 50](#_Toc150278306)

[vii. Block cyclic distribution 50](#_Toc150278307)

[viii. User defined distributions 50](#_Toc150278308)

[15. Shapes 52](#_Toc150278309)

[i. Shapes 52](#_Toc150278310)

[ii. Mirrored Shapes 52](#_Toc150278311)

[iii. Distributed Shapes 52](#_Toc150278312)

[iv. Distributed Values 52](#_Toc150278313)

[v. Values with shapes 52](#_Toc150278314)

[vi. Conformance of shapes 52](#_Toc150278315)

[vii. Subscripting 52](#_Toc150278316)

[viii. Slicing 53](#_Toc150278317)

[16. Arrays, Vectors and Matrices 54](#_Toc150278318)

[i. Array templates 54](#_Toc150278319)

[ii. Arrays 54](#_Toc150278320)

[iii. Vectors and matrices 55](#_Toc150278321)

[iv. Accessing and updating array elements 55](#_Toc150278322)

[17. Parallel execution 56](#_Toc150278323)

[i. The for statement 56](#_Toc150278324)

[ii. The par statement 56](#_Toc150278325)

[iii. Parallel statement attributes 57](#_Toc150278326)

[iv. Implicitly defined constants in parallel statement 57](#_Toc150278327)

[v. The over statement 57](#_Toc150278328)

[vi. The forall statement 58](#_Toc150278329)

[18. Parallel contexts and value modes 59](#_Toc150278330)

[i. Parallel contexts 59](#_Toc150278331)

[ii. Object modes 60](#_Toc150278332)

[iii. Importing values into a parallel context 60](#_Toc150278333)

[iv. Determining the mode of an expression 61](#_Toc150278334)

[v. Assignment to variables created outside of the current parallel context 61](#_Toc150278335)

[vi. Invariant control statements 61](#_Toc150278336)

[19. Communicating operations 62](#_Toc150278337)

[i. Synchronised assignments: the sync statement 62](#_Toc150278338)

[ii. Channel objects 62](#_Toc150278339)

[iii. Shared and complete procedure calls 63](#_Toc150278340)

[iv. Indexed values 63](#_Toc150278341)

[20. Structured parallel communication 65](#_Toc150278342)

[i. Synchronisation points 65](#_Toc150278343)

[ii. Synchronisation in conditional contexts – label matching 66](#_Toc150278344)

[iii. Merging synchronised loops 67](#_Toc150278345)

[21. Communicating procedures 68](#_Toc150278346)

[i. Overview 68](#_Toc150278347)

[ii. Matching parameter and argument modes. 68](#_Toc150278348)

[iii. Communicating procedure definition attributes. 69](#_Toc150278349)

[iv. Communicating procedure call attributes 69](#_Toc150278350)

[v. Implicit arguments to communicating procedures 69](#_Toc150278351)

[22. Stencils: the nhd statement 70](#_Toc150278352)

[i. Overview 70](#_Toc150278353)

[ii. Orthogonal neighbourhoods 70](#_Toc150278354)

[iii. Boundaries 71](#_Toc150278355)

[23. Input/output 73](#_Toc150278356)

[i. Overview 73](#_Toc150278357)

[ii. Input/output in a parallel context 74](#_Toc150278358)

[iii. Input/output errors 74](#_Toc150278359)

[24. Syntax summary 75](#_Toc150278360)

[25. Intrinsic procedures and types 80](#_Toc150278361)

[i. Abstract numerical types 80](#_Toc150278362)

[ii. Numeric type balancing 80](#_Toc150278363)

[iii. Arithmeric operations 80](#_Toc150278364)

[iv. Numerical comparisons 81](#_Toc150278365)

[v. Numerical conversions 81](#_Toc150278366)

[vi. General comparisons 82](#_Toc150278367)

[vii. Logical operations 83](#_Toc150278368)

[viii. String operations 83](#_Toc150278369)

[ix. Array operations 83](#_Toc150278370)

[x. Type comparison 83](#_Toc150278371)

[xi. Ranges and sequences 84](#_Toc150278372)

[xii. Distributions 84](#_Toc150278373)

[xiii. Processor grouping 84](#_Toc150278374)

[xiv. Communicating procedures 85](#_Toc150278375)

# 

# Introduction

## Background

PM (Parallel Models) is a new open-source programming language designed for the implementation numerical models, particularly in a research context where ease of coding and performance of the implementation are both essential requirements that are frequently found in conflict with each other.

PM introduces a new paradigm – communicating operations – which combines the ability of Partitioned Global Address Space languages to use a straightforward ‘global view’ to access remote data (for example, by subscripting a global distributes array) with a structured approach to synchronisation.

The following goals guided the development of the PM language:

* Language structures facilitate the generation parallel and distributed code
* The language should concentrate on parallel structures required for numerical computation
* Distributed and parallel execution should be explicit and configurable
* There should be a clear cost model for all abstractions (in terms of both CPU time and communication costs)
* Programmers should be able to concentrate on coding the model
* 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.

## Inspiration

PM draws inspiration from many sources. In many ways the language is designed to imbed concepts behind *MPI* and *OpenMP* programming into a language. PM syntax and semantics draw inspiration from multiple sources, including: *C*, *C*++, *Fortran*, *Go* (golang.org) and *Rust* ([www.rust-lang.org](http://www.rust-lang.org)). Notable influences include: *ZPL* (<http://research.cs.washington.edu/zpl/overview/overview.html>), *Parasail* (parasail-lang.org), *NESL* (http://www.cs.cmu.edu/~scandal/nesl.html) , *llc* (Reyes *et al*.,2009, 16th European PVM/MPI Users Group Meeting) and *C*\* (<http://people.csail.mit.edu/bradley/cm5docs/CStarProgrammingGuide.pdf> ). Inspiration and ideas were also drawn from recent languages tackling similar problem domains: *Chapel* (chapel.cray.com) and *Julia* (julialang.org) together with language extensions *XcalableMP* (<https://xcalablemp.org/>) and ISPC (<https://ispc.github.io/>) and language projects looking to replace C for high-performance gaming applications, including *JAI* (<https://www.youtube.com/user/jblow888/playlists>)

## Executing a PM program.

A system for executing PM programs could be implemented using a variety of means (interpreters, direct compilation, just-in-time (JIT) compilation, source-to-source compilation, etc.) In terms of the language specification the following requirements are placed on any development system (which may itself consist of a suite of development tools).

* PM semantics require a global program analysis to be conducted prior to program execution
* At least one program execution option must gracefully halt program execution with an appropriate error message if an error condition occurs.
* At least one program execution option must be optimised for performance (in terms of an appropriate balance of execution speed, resources use and communications latency.) In this option the behaviour of the program if it enters and error state may be undefined.

## 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

**for** 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 ) Parentheses may be used to group selections that are not enclosed by {} or []

;OPT A special notation for a semicolon that may be omitted at a line break

# Lexical Structure

A PM program is defined using a set of ***modules***. Each module is defined using a single text file (or equivalent). PM module names are associated with file names (or equivalent) in an implementation specific manner. A valid PM module file will have lines of no more than 1000 characters.

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

// Comments are ignored

Comments may alternatively be delimited by '**/\***' and '**\*/**' which enclose the comment text, which may optionally span multiple lines and contain nested comment delimiters:

/\* Comments are ignored

/\* Even when nested \*/

// .. in different ways

\*/

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 ***operators*** or ***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 either by delimiters or by white space. Otherwise white space is optional.

***New lines*** are not generally significant, but in some of the contexts where a semicolon '**;**' is allowed or expected by the syntax (when used as a statement separator), 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.

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

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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **and** | **any** | **arg** | **as** | **bounds** | **by** |
| **case** | **chan** | **Check** | **complete** | **const** | **default** |
| **do** | **else** | **Elseif** | **except** | **false** | **fix** |
| **fmt** | **for** | **Foreach** | **if** | **in** | **inc** |
| **invar** | **is** | **Key** | **local** | **mod** | **new** |
| **nhd** | **not** | **Null** | **of** | **or** | **ortho** |
| **over** | **par** | **Param** | **partial** | **present** | **private** |
| **proc** | **proceed** | **Rec** | **return** | **shared** | **shift** |
| **struct** | **switch** | **Task** | **test** | **true** | **type** |
| **uniform** | **unique** | **Universal** | **until** | **use** | **var** |
| **where** | **while** | **With** | **xor** |  |  |

The following keywords are reserved for future use: **break continue** **exit** **extern interface** **render**

***Numeric constants*** take the following forms***:***

*number* ::= ( *integer\_constant* | *real\_constant* | *imaginary\_constant* ) [ *bits* ]

*integer\_constant* ::= { *digit* } [ ( '**r**' | '**R**' ) { ( *digit* | *letter* ) } ]

*real\_constant* ::= { *digit* } [ '**.**' *digit* { *digit* } ] [ ( '**e**' |'**d**' | '**E**' | '**D**' ) [ ['**+**' | '**-**' ] *digit* { *digit* } ] ] ]

*imaginary\_constant* ::= [ *integer*\_*constant* | *real\_constant*] ( '**i**' | '**j**' | '**I**' | '**J**')

*bits* ::= ''' { '**s**'| '**l**' | *digit* [ *digit* ] }

Details for interpreting numeric constants are given in the section on numeric types.

***String constants*** are enclosed by double quotes '**"**' and may include any character other than '**"**'. A '**"**' character in a string is represented using '**""**' without any white space between the '**"**' characters. String constants may not span more than one line.

"Hello World" // Hello World

"Hello ""World""" // Hello "World"

# Modular Structure

A PM program consists of one or more ***modules***. A module defines a collection of ***procedures***, ***types*** and ***parameters*** (module level constants). A ***program module*** additionally contains executable statements. A ***library module*** may optionally contain **test** statements containing module testing code but may not contain any other executable statements. Module names are constructed from a sequence of standard PM names (***module name components***) joined by '**.**' delimiters. They are linked to source file names in an implementation-dependant manner. Module names whose first component is '**lib**' refer to standard libraries. If a module name starts with '**.**' then this is taken as shorthand for name components of the current module, barring the last such component. Thus, when used in module **mylib.maths.trig** the module name **.private.support** will refer to module **mylib.maths.private.support**.

|  |
| --- |
| *module***::=** *program\_module* | *library\_module*  *program\_module*::=  [ *decls* ] *statements*  *library\_module*::=  *decls* [ *teststatement* ]  *modname*::=  ['**.**'] *name* { '**.**' *name* } |

All names used by the PM language may optionally start with an underscore character. Such names are local to the module in which they are used and do not match or conflict with names in any other module.

Type, procedure and parameter declarations follow any **use** statements in the module. Types, procedures and parameters occupy separate name spaces – they may have the same name as each other.

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 used by other modules. Library modules may be used by other modules. They may not contain executable statements after the declarations except for an optional **test** statements containing module testing code.

## Importing declarations from other modules

|  |
| --- |
| *decls*::=  { *import* '**;**' } ( *import* | *decl* ) { '**;**' *decl* }  *import*::=  **use** *modname* [ '**=>**' *name* ] [ *imports* ] { '**,**' *modname* [ *imports* ] } |

Modules may import declarations from library modules using a **use** declaration. The default form of this declaration imports all parameters, types, and procedures defined in the given module into the current module. However, the importing process does not import definitions indirectly accessed through **use** statements in the imported module. A **use** statement must not result in conflicting definitions. Name clashes may be avoided by using an import clause in the **use** statement. This lists specific elements to import.

|  |
| --- |
| *imports*::=  '**{**' *import* { '**,**' *import* } '**}**'  *import*::=  ( **type** | **param** | **proc** ) *namelist* |

use model4 {

type model\_params,

param theta,

proc test,

}

A item defined in a module listed in a **use** clause, but not in the import list, can be accessed using the notation *modulename* **'** *itemname* where *modulename* is the last component of the name of given module*.* To cater for circumstances where more than one module has a name with the same last component, the **use** clause allows the declaration of an alternative identifier that may be used to access a given module.

use mylib.maths.test => maths\_test {}

use mylib.graphics.test => graphics\_test {}

…

maths\_test'run\_tests()

graphics\_test'run\_tests()

It should be noted that there is no restriction on the dependency graph between modules that is created by **use** clauses – module *A* can use module *B* while module *B* also uses module *A*. Combined with the hierarchical naming convention, this allows larger packages to be created using clusters of interrelated modules.

## Type declarations

A ***type declaration*** introduces a new type or provides a name for an abstract type constraint expression. Types have their own namespace, separate from procedures and parameters. Types that have been declared as extensible may be extended by modules that import them and this extension will be visible in all modules, including the exporting module.

## Procedure declarations

A ***procedure declaration*** creates a procedure (procedure is a broad definition that encompasses subroutines, functions and templates from other languages). Procedures have their own namespace, separate from types and parameters. Procedures that have been declared as extensible may be extended by modules that import them and this extension will be visible by all modules, including the exporting module.

## Parameter declarations

A **param** declaration 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 directly or indirectly recursive. There is no explicit restriction on the procedures employed within a **param** declaration expression – they may be interpreted as zero-parameter procedures with their own separate name space. In expressions, parameter names may be shadowed by variable and constant names.

param sqrt\_pi = sqrt(pi)

# Simple types

## Types and values

The PM type system provides flexibility similar to some dynamically typed languages while being designed for static analysis. Run time type inference only occurs within a specific control structure: the **any** statement, all other type inference, including the selection of overloaded generic procedures, occurs at compile time. It should be noted that implementing the PM type system may require static type analysis across and between modules.

PM programs operate on ***values*** which are ***grouped*** into types.

* A ***concrete value*** is a particular run-time representation of a piece of information, such as a 16-bit binary encoding of the number 2 expressed in PM as **2'16**
* A ***compile-time value*** is a representation of a value that is known at compile time. Compile-time values are concrete values whose representation does not require any run-time storage.
* A ***concrete type*** defines a particular run-time representation of values (e.g. 16-bit integers). All concrete values (including compile-time values) have a concrete type. A concrete type may be viewed at the set of concrete values which share the same run-time representation.
* A ***polymorphic value*** is a particular kind of concrete value that has a run-time representation that is capable of representing concrete values of different concrete types. For example the number 2 stored using a run-time representation that is equally capable of storing 32-bit numbers or strings (this could be a tagged pointer to the data, although this is an implementation-specific detail).
* A ***polymorphic type*** is the concrete type associated with a polymorphic value.
* An ***abstract type*** is a (possibly unbounded) set of concrete and/or other abstract types defined using a ***type constraint expression*.**
* A ***parameterised type*** is an abstract type that can represent different types with a similar structure according to one or more supplied parameters, which are themselves types.

## Type conformance

C*onformance* relationships are defined between types and between values and types:

* A value V conforms to an type T if V∈T or for some type U, V∈U and U conforms to T
* Type T conforms to type U if T∈U or for some type V, T∈V and V conforms to U

Both of these definitions are applied recursively if required.

## Implicit type conversions

PM defines a number of implicit conversions between concrete values of different types. These are:

1. Conversions between numeric types
2. Conversion of a value to a polymorphic value capable of holding it
3. Conversion of a structured type value to a value that it embeds
4. Conversion of a procedure name value to a procedure signature value

These conversions are described later, together with the types to which they pertain.

Conversions are performed whenever a value is matched against a type constraint expression.

1. Declaring a variable or constant with a type constraint
2. Initialising a field of a **struct** or **rec** value
3. Returning a value from a procedure with a constrained return type
4. Passing a value as an argument in a procedure call where the corresponding parameter has a type constraint
5. In an explicit type conversion expression *value* **as** *type*

Each of these contexts is described in detail in subsequent chapters.

In all cases, only a single type conversion is performed (there are no chained implicit conversions) Conversions are considered in the following order:

1. procedure signature conversion
2. embedded value conversion
3. polymorphic value creation
4. numeric value conversion

Numeric value conversion is not performed when passing parameters to procedures.

## Concrete types

A concrete type represents a particular run-time encoding of a set of related values, such as the set of 16-bit signed integers. Concrete types are important in PM because variables do not change their concrete types over their lifetimes. Moreover, specialised copies of procedure definitions are created to cater for each combination of concrete types with which that procedure is called.

### Integer types

PM supports a range of concrete integer types, defined in the following table. The definitions of these types are flexible to enable portability of code. Code requiring a strict bit size should use inquiry functions to check that a given type meets the expected requirements. Each integer type defines a set of distinct values that conform to it.

|  |  |  |  |
| --- | --- | --- | --- |
| **sint** | *small integer* | *System defined standard integer* |  |
| **int** | *standard integer* | *Integer capable of counting elements into largest possible array* | |
| **lint** | *large integer* | *Integer capable of holding offsets into the largest possible file* | |
| **int8** | *8-bit integer* | *smallest integer holding -127..+127* |  |
| **int16** | *16-bit integer* | *smallest integer holding -32767..+32767* |  |
| **int32** | *32-bit integer* | *smallest integer holding -2147483647..+* *2147483647* | *OR same as* **int** |
| **int64** | *64-bit integer* | *smallest integer holding*  *-9,223,372,036,854,775,807.. +9,223,372,036,854,775,807* | *OR same as* **int32** |

***Integer constants*** are usually designated using whole decimal number, but may also 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**' representing 10 to 36 respectfully.

123 2r101011101 16rdeadbeef

By default integer constants yield values conforming to **int**. Integers of type **sint** are designated using a terminating **'s** and integers of type lint by a terminating **'l**. For other integer types, append a **'** and the corresponding number of bits:

12345's // sint

12345678904567889874324897284746'l // lint

2r1010101010101010101010101010100'l // lint

255'8 // int8

101010101010101010101010101010'32 // int32

### Real types

PM defines a set of concrete real types. Each is associated with a distinct set of floating point values that conform to it.

|  |  |  |
| --- | --- | --- |
| **sreal** | *Standard (implementation defined) single precision floating point value* |  |
| **real** | *Standard (implementation defined) double precision floating point value* |  |

**Real constants** must contain a decimal point and may contain an exponent preceded by '**e**', '**E**', '**d**' or '**D**'.

-5.2 2e3 4.2D-20

By default real constants yield values conforming to type **real** (regardless of the letter used to signify the exponent). Values of type **sreal** are designated using a terminating **'s**.

5.2d's ! sreal

3.2e-3 ! real

### Complex types

Concrete complex types are defined as follows.

|  |  |
| --- | --- |
| **scpx** | *Complex number formed from* **sreal** *components* |
| **cpx** | *Complex number formed from* **real** *components* |

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

3.0j ! cpx

-2.2e-3J's ! scpx

### Boolean type

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

### String type

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"

A string constant can contain a '**"**' character by repeating it twice:

"The last string was ""Hello World"""

## Compile-time types

Compile-time types contain a single value and thus need no storage at runtime (they are essentially concrete types whose representation requires zero bits). Combined with procedure genericity, compile-time types provide some of the benefits of languages with macro systems and compile-time execution as well as providing useful facilities such as user-controlled global constant propagation.

### Null type

The concrete **null** type contains the single value **null**. No other type conforms to **null**.

### Proc types

A compile-time ***proc type*** contains a single value that is a valid PM procedure name (or operator symbol). Both procedure types and procedure constants are formed by preceding a name by **$**:

'**$**' ( *mname* | *procname* )

$cell\_model

$+

$my\_module'init\_model

All procedure types conform to the abstract type **proc**.

### Procedure signature types

A procedure signature type describes a procedure with given argument constraints and result types. Procedure name types do not conform to procedure signature types. They may, however, be implicitly converted to a value of a procedure signature type. A procedure signature type has the form:

**proc** [ '**(**' *procarg* { '**,**' *procarg* } [ '**…**' ]'**)**' [ '-**>**' '**(**' *typelist* '**)**' ] ]

*procarg*::=

[ '**&**' ] [ *mode* ] *type*

For example:

proc (real,int)->real

A procedure name can be converted to a procedure signature type if a call to the procedure name with the given argument types and modes would return values conforming to the supplied return type constraints. If the corresponding procedure definition (or definitions) supplies return type constraints then these must conform to the corresponding constraints in the procedure signature type. If the corresponding procedure definitions do not supply return type constraints, then when a call made to the procedure signature value (as opposed to the procedure name value from which it has been converted), this call must not return a value that does not conform to the return type constraints specified in the procedure signature type.

proc f(x:real,y:int)->real { return x\*\*y }

proc g(x:real,y:int)=x\*\*y

func:proc(real,int)->real=f

func:proc(real,int)->real=g

a=f.(2.0,3)

b=g.(3.0,2) // Here an additional check of the return value of g is made

### Value types

A compile-time ***value type*** contains a single numeric or Boolean value. Value types do not require any run-time storage. Both the value type and its corresponding value is designated by:

'**'**' *number*

'**'**'' **true**

'**'**' **false**

A value type conforms to **integer** or **bool** accordingly

**'true** conforms to **bool**

**'123** conforms to **int**

Note that trailing '**'**' designations are not allowed in compile time numbers.

### Types as values

A ***type-value*** is a compile-time value representing a given type. Both the type value and the concrete type of that type value have the same representation:

'**<**' *type*  '**>**'

Type values are primarily used for type testing and type conversion:

1 as <real> // convert value to real

if x is <real> { } // test if type of x is real ('true or 'false)

required\_type=<string>

if x is required\_type { } // where required\_type is a type value

y = x as <real> // convert to real

### Unique values

Unique types have a single compile time value. They must be created using a type declaration of the following form:

**type** *name* [ **in** *namelist* ] **is unique** ['**{**' *name* '**}**' ]

This declaration both creates the concrete type of the given unique value and creates a parameter which is set to that value. By default, the parameter has the same name as the type. However, it is possible to give the parameter a different name, supplied between braces.

type CYLINDER is unique // parameter is named CYLINDER

type FLAG is unique {\_flag} // parameter is named \_flag

if CYLINDER is <CYLINDER> { … } // this test will succeed

if \_flag is <FLAG> { … } // this test will also succeed

if FLAG is <FLAG> { … } // this test will fail

// -- FLAG is a type but not a parameter

# Structured types

## Structures and records

***Structures*** and ***records*** provide a mechanism to create aggregate values. They are very similar, except that it is not possible to separately assign to a single component of a record, while this is possible for a structure. A structure or record value associates each component with a name. Component names are local to the structure or record.

A concrete structure or record type is defined using the following type definition:

**type** *name* [ *tparams* ] **is** [ **struct** | **rec** ] [ *name* ]

'**{**' [ **use** ] *name* [ '**:**' *type* ] ['**=**' expr] [ { '**,**' [ **use** ] *name* [ '**:**' *type*] ['**=**' expr] } '**}**'

For example:

type person is struct { name:string, age:int }

type \_point(t:num) is rec { x:t, y:t }

A structure or record value is then created using a generating expression:

**new** *name* '**{**' *name* '**=**'*exp* { '**,**' *name* '**=**'*exp* } '**}**'

For example:

person= new person{ name=”John Smith”, age=42 }

location= new \_point{ x=2, y=4 }

f\_location=new \_point{ x=2.0, y=3.0 }

The name following the **new** keyword must be the name given to the structure or record type directly in a structure or record type declaration - type synonyms are not acceptable.

Every field in a structure or record generating expression must be initialised, with the exception of (i) features constrained to be compile-time types and (ii) features with a default initial value supplied in the type definition:

type foo is rec { x:int =1, y: 'true, z: real }

a= new foo { x=2, y='true, z=1.2 }

b= new foo { y= 'true, z=1.2 } // b.x has default value of 1

c= new foo { x=2, z= 1.2 } // c.y can only have value fix true

d= new foo { x=2, y='true } // ERROR – d.z must be initialised

Note that default initial values cannot access variables and constants local to the procedure. They may, however, contain arbitrarily complex procedure calls (which may be executed each time the generating expression is executed.)

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

person.age=person.age+1

print(person.age)

print(location.x)

In order to maintain the no-modification rule, a '**.**' operation by not be applied to a record on the left hand side of an assignment or in the expression defining a reference parameter in a procedure call ('**&**')

## Embedded elements in structures and records

If an element in a structure value was defined with the **use** keyword present and if the value of that element is a structure or record, then the elements of that element may be referenced as if they were elements of the parent value.

type foo is rec{ use a,b }

type bar is rec { p, q }

x=new foo {a=new bar{p=1, q=2},b=3}

print(x.a.p) // Outputs 1

print(x.p) // Also outputs 1 – accesses the same element

The process works recursively:

type foo is rec{ use a,b }

type bar is rec { p, q }

x=new foo {new bar{p=new bar{p=4,q=5}},q=3}

print(x.a.p.p) // Outputs 5

print(x.q) // Also outputs 5 – accesses the same element

This form of access to ***embedded elements*** is only applies if the name of the embedded element is not the same as the name of an element in the parent value – parent elements always shadow embedded elements. In addition, if embedded element access could ambiguously refer to more than one element then that access is illegal.

type foo is rec{ use a,b }

type shadow is rec { use p, q}

type bar is rec { p, q }

type deep is rec { r }

x=new foo {

a= new shadow{

p= new bar{q=4,r=5},

q=4

},

b=new deep{

r=4

}

}

print(x.q) // This validly refers to x.a.q which shadows x.a.p.q

print(x.r) // Error – could refer to x.a.p.r or x.b.r

## Abstract embedded types and embedded type conversion

The use of embedded element access is primary designed to facilitate code reuse. Two facilities are provided to further this goal. A type expression:

'**.**' *type*

will match any type that embeds the given type. Thus **rec {use** **s=***S***,** … **}** will conform to **.***S*.

In addition, in contexts where type conversions are allowed, it is possible to convert to a given type from a value that embeds a value of that type.

type foo is {use x:int}

a=new foo{x=12}

b=a as <int>

## Tuples

A tuple is similar to a record. Tuples have one to seven elements named **1** .. **7** that may be accessed using the same dot notation as for structures or records.

Tuple types with a given number of components of any type are defined as:

|  |  |
| --- | --- |
|  |  |
| **tuple1d(***x***)** | **tuple1d\_of(***T***)** |
| **tuple2d(***x***,***y***)** | **tuple2d\_of(***T***)** |
| **tuple3d(***x***,***y***,***z***)** | **tuple3d\_of(***T***)** |
| **tuple4d(***x***,***y***,***z***,***t***)** | **tuple4d\_of(***T***)** |
| **tuple5d(***x***,***y***,***z***,***t***,***u***)** | **tuple5d\_of(***T***)** |
| **tuple6d(***x***,***y***,***z***,***t***,***u***,***v***)** | **tuple6d\_of(***T***)** |
| **tuple7d(***x***,***y***,***z***,***t***,***u***,***v***,***w***)** | **tuple7d\_of(***T***)** |

A type of the form **tuple2d\_of(***T***)** is the same as **tuple2d(***T,T***)**. All tuples conform to the type **tuple**.

PM provides syntactic sugar for tuple types:

'**[**' [ *type* ] { '**,**' [ *type* ] } '**]**'

A tuple value is defined using the tuple constructor:

'**[**' [ *expr* ] { '**,**' [ *expr* ] } '**]**'

If an element of a tuple constructor is missing, it is given the value **null**. A tuple constructor is syntactic sugar for a call to the **tuple** intrinsic procedure.

A tuple value conforms to a tuple type if the type has the same number of elements as the value and each element, in order, conforms to the corresponding element in the tuple type.

A tuple type *V* conforms to a tuple type *U* if *V* and *U* have the same number of elements and each element of *V*, in order, conforms to the corresponding element of *U*.

Elements of tuples are accessed using the the '**.**' operator and an element number:

var a=[1,2.5,"Hello"]

b=a.2 // has the value 2.5

a.1=123

c=a.1 // has the value 123

# Abstract types and Type declarations

## Type constraint expressions

Abstract types define arbitrary sets of concrete types and/or other abstract types. An abstract type is defined using a ***type constraint expression***. This describes a membership test determining whether a given value or type conforms to the abstract type defined by the expression.

## Universal type

The abstract ***universal type***, which includes all other types, is denoted using the keyword **any**. All types conform to **any** (including **any** itself). All values conform to **any**.

## Logical type expressions

Type constraint expressions may be combined using logical operators.

*type* **except** *type*

*type* **or** *type* { **or** *type* }

*type* **and** *type* { **and** *type* }

*type* **inc** *type*

**inc** *type*

In terms of a type *T*1  conforming to a logical type expression or a value with concrete type *T*1 conforming to a logical type expression:

*T*1 conforms to *T*2 **except** *T*3 if *T*1 conforms to *T*2 and *T*1 does not conform to *T*3

*T*1 conforms to *T*2 **or** *T*3 if *T*1 conforms to *T*2 or *T*1 conforms to *T*3 or both.

*T*1 conforms to *T*2 **and** *T*3 if *T*1 conforms to *T*2 and *T*1 conforms to *T*3

*T*1 conforms to *T*2 **inc** *T*3 if *T*1 conforms to *T*2 and *T*3 conforms to *T*1

*T*1 conforms to **inc** *T*2 if *T*2 conforms to *T*1

In terms of a logical type expression conforming to a type *T*1:

*T*2 **except** *T*3 conforms to *T*1 if *T*2 conforms to *T*1

*T*2 **or** *T*3 conforms to *T*1 if *T*2 conforms to *T*1 and *T*3 conforms to *T*1.

*T*2 **and** *T*3 conforms to *T*1 if *T*2 conforms to *T*1 or *T*3 conforms to *T*1 or both.

*T*2 **inc** *T*3 conforms to *T*1 if *T*2 conforms to *T*1

**inc** *T*2 conforms to *T*1 if *T*1 is **any**

When these logical operators are combined in a type constraint exprssion, they have the following precedence (lowest to highest) **except** **or** **and** **inc**.

## Type declarations

Abstract type declarations associate a name with a given type constraint expression.

*typedecl*::=

**type** *name* [ '**(**' *typeparams* '**)**' ] [ '**:**' *namelist* ] **is**  ( '**…**' *typelist* | *typelist* '**…**' | *typelist* )

typeparams::=

*name* [ '**:**' *type*] { '**,**' *name* [ '**:**' *type*] }

The simplest type declaration simply associates a name with a constraint expression:

type x is int or real or struct{x:int,y:int}

A type constraint expression constructed from **or** operators at the top level may be written as a comma-separated list:

type x is int,real, struct{x:int,y:int}

This is a form of syntactic sugar, although it is required when creating and extending open type definitions.

## Parameterised type declarations

A ***parameterised type*** declaration defines a template type constraint expression that may be parameterised by one of more ***type arguments*** in order to generate an ***actual type***. An actual type is derived from a type template by providing the correct number of type constraint expressions as type arguments.

type point(t:num) is struct{x:t, y:t}

type integer\_point is point(int)

type num\_point is point()

Type arguments used to create an actual type must conform, parameter by parameter, to the corresponding parameter type constraints, or be missing. Missing type arguments are assumed to be equal to the corresponding type parameter constraint or to **any** if no such constraint is present. Missing arguments can simply be omitted between commas, although trailing arguments may be omitted entirely and an argument list which consists entirely of omitted arguments my itself be omitted,

type foo(t:num,u:tuple,v:any) is …

foo(,,) // same as foo(num,tuple,any)

foo // same as foo(num,tuple,any)

foo(int,,) // same as foo(int,tuple,any)

foo(int) // same as foo(int,tuple,any)

foo(int,,real) // same as foo(int,tuple,real)

A parameterised **struct** or **rec** declaration operates very much like a regular type declaration, except that the parameters can be used in element constraint expressions.

type point(T:num) is rec{x:T,y:T}

Withing a corresponding new statement, the type name may be provided with or without parameters:

pointA = new point{x=1,y=1}

pointB = new point(real) {x=1.0,y=1.0}

pointC = new point {x=1.0,y=1} // conforms to point(num)

Note that in terms of implicit type conversions, a feature constraint containing a parameter is treated as is treated as if the parameter was equal to its own constraint expression (or to **any** if the parameter is unconstrained).

## Open type declarations

An ***open type declaration*** with trailing ellipses provides a potentially incomplete definition for a type which may be added to by other definitions. The type definition may be extended using a **type …** definition or by using by declaring another expression to be **in** that type:

type a is int, real, ...

type a is ..., string

type b in a is cpx // Type a is int or real or string or b

Augmenting type declarations do not have to be in the same module as the original declaration and do not have to precede that declaration in the module or program text.

Open type declarations may also be parameterised:

type point(T:num) is t2d(T,T),...

type point(T) is ..., t3d(T,T,T)

If the extending type definition is parameterised, then the type constraints associated with its parameters must either be absent or must conform, parameter by parameter, to the type constraints in the definition to which it refers. If type constraints are present then the extending definition only extends the type template when the arguments in the actual type expression conform to the type parameters in the extendingdefinition. For example:

type point(T:i8) is ..., int32

// point(int) matches t2d(int,int) and t3d(int,int,int)

// point(int8) matches: t2d(i8,i8)

// and t3d(i8,i8,i8)

// and i32

The above example enables points with coordinates that are i8 to be stored as a tuple or as a single **i32** (possibly as a packed value). Storage as a single i32 is not available for points with coordinates of other types.

## Recursive type declarations

A type may not conform to itself (and thus may not be a member of itself or a member of any type that conforms to it). However, type constraint expressions used to define a type *T*, or a type conforming to *T* may contain *T*. This enables the definition of recursive types.

type list is struct{head:int,tail:list},int

Note that while useful, directly recursive types scale poorly as each level of recursion introduces a new type. The above definition would therefore only be suitable for short lists.

PM type definitions may create infinite types:

type list is struct{head:int,tail:list}

Such declarations are legal since they may occur during program development (an implementation may decide to provide warnings). An infinite type is essentially a null type since no finite value will conform to it.

# Procedures

## Procedure definitions

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.

**proc** *procname* [ '**%**' ]  *params result* [*attr* ] [*block* ]

**proc** *procname* [ '**%**' ]  *params* [ *returns* ][*attr* ] [ **check** *exprlist* [ *whereclause* ] ] *block*

*params*::=

'**(**' [ *pars* [ '**,**' *keypars* ] | *keypars* ] '**)**'

*pars*::=

{ *param* '**,**' } ( *param* | **arg** '**...**' )

*param*::=

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

A procedure declaration defines the name of the procedure, parameters on which it operates and their associated type constraints and any values returned.

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

proc calc\_stats(data: real[])=mean,std\_dev {

......

mean=…

std\_dev=…

}

## Procedure calls

A procedure call invokes a procedure, supplying it with a list of values and/or objects to operate on.

*call*::=

*mname* [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

'**$**' *procname* [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

*arglist*::=

{ *arg* '**,**' } ( *arg* | **arg** '**...**') [ '**,**' *keyargs* ]

[ *keyargs* ]

*arg*::=

'**&**'*ref* |'**&**' '**&**'*ref* | *expr*

*keyargs::=*

*name* '**=**' *expr* { '**,**' *name* '**=**' *expr* }

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.

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

If a procedure call is used as an argument to another procedure call, then the nested call is assumed to return a single value. Thus the following are equivalent:

y = f(g(x))

y = f( \_anon ) where \_anon=g(x)

## Reference parameters and arguments

A procedure call may pass ***reference arguments*** to ***reference parameters***, both denoted by **&**. These indicate that the procedure may change the value stored in corresponding object. A procedure call with reference arguments must either return no values or form the complete right hand side of a definition, assignment or **where** clause, the complete conditional expression in a control statement or the complete expression in an iterator. They may not form part of a larger expression.

update(&x) // Allowed

y=f(&x) // Allowed

u,v,w=multi(&x,&y,2) // Allowed

if f(&x) { … } // Allowed

y=f(&x)+1 // Not allowed

y=g(f(&x)) // Not allowed

y=g(z) where z=f(&x) // Allowed

if f(&x)>0 { … } // Not allowed

if f>0 where f=f(&x) { … } // Allowed

## Reference argument aliasing

Given that (i) accessing and modifying sub-objects can be sometimes be a communicating (i.e. active) operation and (ii) that reference arguments may potentially alias each other or non-reference arguments, reference arguments are required to adhere to a strict set of rules:

1. Two reference arguments must not refer to the same object, to an object and a sub-object of that object or to overlapping sub-objects of the same object.
2. The reference must be capable of being the left hand side of a conventional rather than a **sync** assignment – that is, it must not invoke any communication and in particular it should not be an element of a distributed or shared object and must not include the **@** operator.

The following calls will therefor generate error conditions:

f(&a,&a) // Two reference arguments access the same object

f(&a,&a.b) // Overlapping sub-objects

f(&a.b[i],&a.b[j]) // Run-time check, error if i==j

If base object of a reference argument appears in the argument expressions for any non-reference argument to the call, then the argument value may be copied before being passed unless it is provably (at compile time) not aliasing a reference argument:

f(&a,a) // Second argument is copied before being passed

// - equivalent to \_temp=a;f(&a,temp)

f(&a.a,a.b) // No copying required – provable non-aliasing

In order to overcome these restrictions, there is the option of using && reference arguments. These have copy-in, copy-out semantics. Thus:

f(&&x,&&y[i])

Should be interpreted as equivalent to:

var temp1=x

var temp2=y[i]

f(&temp1,&temp2)

x=temp1

y[i]=temp2

## Variable-length argument lists

A procedure may accept a variable number of arguments:

proc set\_numbers(&dest:int[],arg:int...) { .... }

It is possible to pass optional arguments *en masse* to a procedure call (a ***pass through*** call):

proc process(a) {

print("Value is"++a)

}

proc process(a,arg...) {

process(a)

process(arg...)

}

process\_values(1,2,3,4)

## Keyword parameters and arguments

*keypars*::=

{ *name* [ '**:**' *type* ]'**=**' *expr* '**,**' } ( *name* [ '**:**' *type* ] '**=**' *expr* | **key** '**...**' )

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 do not form a part of the signature of the procedure.

procrun\_model(params : model\_params, iterations = 1000,

relaxation = 0.02) {

.......

}

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 **key…**:

proc process\_opts(&optarray,first\_opt=.false,second\_opt=33,key...) {

....

process\_other\_opts(key...)

}

The keyword argument value should be convertible to either (i) the concrete type of the default expression if no type constraint is provided or (ii) the provided type constraint.

It is possible to determine if a keyword argument **x** has actually been supplied using **present(x)**.

## Procedure calls using procedure values

A second form of procedure call is:

*expr* '**.**' [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

In place of a procedure name this form takes a procedure value (a value with a procedure name or procedure signature type). Since procedure values all have unique types and types may be determined statically during program analysis, then such a call should be identical in performance to any other procedure call (including the use of optimisations such an inlining). Care must be taken, however, to avoid producing excessive versions of code (this can be achieved using polymorphic types).

proc integrate(f:proc,a,b,interval)=result {

var s=0.0

foreach i in a + interval .. b - interval by interval {

s=s+f.(i)

}

result=interval\*(s+f.(a)/2.0+f.(b)/2.0)

}

print(integrate($sin,0.0,0.5,0.01))

Procedure calls using procedure values cannot pass keyword arguments.

## Procedure signatures

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

'**+**' | '**-**' | '**\***' | **and** | **or** | '**&**' | '**|**' | **not**

'**[]**' | '**#**' | **fmt** | '**..**' | **by** | **ortho**  | '**!**' | '**/=**'

'**==**' | '**>**' | '**>=**' | **in** | **inc |** '**/**' | '**\*\***' | **mod**

'**...**' '**\_**' | '**\_**' '**...**' | '**++**' | **xor**

Note the absence of **<** or **<=** from this list – these operators are defined with respect to **>** or **>=**.

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

* 1. The name of the procedure
  2. Whether the procedure is tagged with **%**
  3. The number of values the procedure will return.
  4. The number(s) of arguments the procedure will accept
  5. The type (and mode) constrains on the procedure's parameters
  6. Which parameters are declared to be reference parameters

Mode constraints are explained in the section on communicating procedures.

A procedure signature *P* is said to ***conform to*** procedure signature *Q* if:

1. The procedure names are the same.
2. Both *P* and *Q* have the **%** tag, or none
3. Either
   1. Both P and Q have the same number of parameters.
   2. P accepts a variable number of arguments and has fewer parameters than Q
   3. Q accepts a variable number of arguments and has fewer parameters than P
   4. Both P and Q accept a variable number of arguments
4. Reference parameters occur in the same position.
5. The type constraint for each parameter of *P* conforms to the type constraint for the equivalent parameter of *Q.* In either case the corresponding parameter may be in the variable-length part of the argument list, in which case the type constraint for **arg…** applies.

Signatures *P* and *Q* are said to ***conflict*** if both *P* conforms to *Q* and *Q* conforms to *P*. Two conflicting signatures may not be present in the same module, whether through direct definition, importing declarations from other modules or any combination of the two.

Signature *P* is ***strictly more specific*** than signature *Q* if *P* conforms to *Q* and type constraints are not strictly equal to one another for at least one parameter position.

## Matching procedure calls to procedure definitions

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

1. The procedure name is the same.
2. Both call and procedure has **%** or neither does.
3. 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.
4. All reference parameters are associated with a reference argument, starting with an '**&'**, in the same position.
5. The value supplied for each argument conforms to the type constraint (or for a ***communicating procedure call***, the ***moded type constraint***) for the corresponding parameter, or to the constraint for **arg…** if the argument corresponds to the variable-length portion of the parameter list.

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***, leading to an error.

If no procedure matches then PM will attempt to convert the arguments types to the parameter type constraints and then search again for a match. This is done in three passes. In each pass more type conversions are considered. Conversions are applied to all arguments before procedure matching is re-attempted.

For non-reference parameters:

Pass 1: Procedure signature conversion

Pass 2: As above + embedded type conversion

Pass 3: As above + polymorphic type conversion

For reference parameters:

Pass 1: No conversions

Pass 2: Embedded type conversion

Pass 3: Embedded type conversion

Once a given pass produces a matching procedure or fails due to ambiguity, then procedure selection is completed and no further conversion passes are considered. Note that implicit numeric type conversions are not considered for procedure call arguments.

Arguments passed using **arg…** pass-through form part of the signature of a procedure call – they are used in the matching process and do not necessarily have to correspond directly to the variable argument portion of the called procedures parameter list.

## Procedure specialisation

The PM system will construct a specialised implementation of the procedure body for the specific combination of concrete types associated with the argument values present in the procedure call.

## Type constraints between different arguments to a procedure

Unlike some other languages adopting ‘multi-method’ approaches to procedure selection, PM does not provide a mechanism within procedure selection to specify that the type of one argument (or component thereof) should match the type of a second argument. The **same\_type** function provides a similar functionality. This procedure returns **'false** its two arguments have different types and **'true** if the two arguments have the same type. A **test** statement or **check** clause will fail at compile time if given an expression which evaluates to **'false**. For those cases where the type matching should modify procedure selection, the **same\_type** procedure may be used as follows.

proc \_set\_table(&table:table,value,same:'true) {

\_set\_table\_from\_value(&table,value)

}

proc \_set\_table(&table:table,table2,same:'false)

check "Attempt to set tables of different types"=>same\_type(table,table2) {

\_set\_table\_from\_table(&table,table2)

}

proc set\_table(&table,value) {

\_set\_table(&table,value,same\_type(table.first\_element,value))

}

set\_table(&table1,0) // Fill with values

set\_table(&table2,table1) // Copy values between tables

## Procedure and procedure call attributes

Both procedure declarations and procedure calls may be associated with additional attributes that can modify their semantics. Most of these attributes are only applicable to communicating procedures or in a parallel context and therefore will be discussed in a following chapter. The attributes with universal applicability are listed below:

|  |  |
| --- | --- |
| **inline** | This attribute can be applied to either a procedure definition or a procedure call. Unless prevented by **no\_inline**, the presence of this attribute in either call or definition causes the procedure to be inlined. |
| **no\_inline** | This attribute can be applied to either a procedure definition or a procedure call. Its presence in either call or procedure definition will nullify the effect of the **inline** attribute. Note that it does not prevent inlining altogether, if this occurs as an optimisation. However, it does prevent the **inline** attribute forcing inlining. |

proc f(x)<<inline>> { … }

proc g(x){ … }

proc h(x)<<no\_inline>> { … }

f(x) // this call is inlined

f(x <<no\_inline>>) // inlining up to optimiser

g(x <<inline>>) // this call is inlined

g(x <<no\_inline>>) // inlining up to optimiser

h(x <<inline>>) // inlining up to optimiser

# Objects

## Variables and constants

PM stores values in ***named*** ***objects*** which may be ***variables*** or ***constants***. Variables may change the values stored in them, constants may not. Objects are created and associated with names and given values using a definition statement.

*definition::=*

( **var** | **const** ) *name* [ '**:**' *type* ] '**=**' *expr*

( **var** | **const** ) *lname* { '**,**' *lname* } [ '**:**' *type* ] '**=**' *mrhs*

*name* '**=**' *expr*

{ *lname* '**,**' } *name* { '**,**' *lname* } '**=**' *mrhs*

*lname*::=

*name* | '**\_**'

mrhs::=

*call* | *name* | *literal*

For example

pi=3.14179 // Simple constant definition

const message = "Hello World" // Explicit constant definition

var y = 3 // Variable definition

a,b = f(0) // f must return two values

var c,\_,d = f(0.0) // f must return three values

var e,f,g = 0 // All three variables initialised to 0

A name is defined until the end of the block of statements in which it is defined. Variable or constant names may not shadow variable or constant names defined in enclosing blocks or clash with the names of procedure parameters.

A variable or constant definition may contain a type constraint. This simply checks that the initialisation expression matches the given constraint and does not have any impact on the constant or variable created. Thus the following expression:

var x:num = 1

creates a variable holding values of type **int**, not a variable capable of holding any value conforming to type **num**.

Type conversions will be applied when the initialising expression does not match the type constraint.

var x:real = 1 // x is initalised to 1.0 (integer to real conversion)

It is possible to create variables and constants with a compile-time type. However, a variable holding such a value can only be assigned that same value (this seemingly useless assignment operation is permitted to enable generic code containing assignments to be able to be applied to name values.)

## Assigning new values to variables

The values stored in variables may change over time. Changing the value of a variable is achieved using an assignment statement.

*assignment*::=

*lhs* '**=**' *expr*

*lhs* '**,**' *lhs* { '**,**' *lhs* } '**=**' *mrhs*

*lhs*::=

*name qualifier* | '**\_**'

var x = 4 // Value stored in x is 4

x = x + 1 // Value changed to 5

Some procedures return more than one value. In this case, multiple left hand sides to the assignment are permitted. An underscore may be used as a place filler, allowing a given returned value to be ignored. For example:

x,y,\_, z = returns\_four\_args(a)

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

## Assignment operators

An assignment operator, as its name suggests, combines assignment with an operator.

*assignment*::=

*ref* *assnop* '**=**' *expr*

*lhs* '**,**' *lhs* { '**,**' *lhs* } *assnop* '**=**' *mrhs*

*assnop*::=

'**+**' | '**-**' | '**\***' | **and** | **or**  | '**&**' | **xor** | '**|**'

x+=2 // As x=x+2

x,y +=f(…) // Add first value returned to x, add second value to y

It is also possible to use an expression returning a function name as an assignment operator.

*ref* '**{**' *expr* '**}**' '**=**' *expr*

*lhs* '**,**'  *lhs* { '**,**' *lhs* } '**{**' *expr* '**}**' '**=**' *mrhs*

x {$max}= 0 // as x = max(x,0)

op=$+

x {op} = 1 // as x=x+1

In this case the procedure type must be a member of the open abstract type **assignment\_operator**. The notation **x+=y** can be taken as syntactic sugar for **x{$+}=y**. The exception is **x-=y** which is syntactic sugar for **x{$+}=-y**. PM assumes that any procedure conforming to **assignment\_operator** is associative and will give undefined results if this is not the case..

## Mixed initialisation/assignment

When a procedure call returns multiple values, it is occasionally useful to use these values to initialise or modify different kind of object. To enable this to happen, it is possible to combine several assignments/definitions into a comma separated list, followed by subexpressions.

( *definition* | *assignment*) { '**,**' ( *definition* | *assignment* ) } [ *subexpr* ]

var w:int=ww,const x=xx,y=y,z+=zz where ww,xx,yy,xx=f(…)

// w is an int variable

// x is an unconstrained constant

// The third value returned from the function is assigned to y

// The fourth value returned from the function is added to z

## Object resource management

In addition to holding one of a defined set of values, objects may also be associated with resources. Such resources may include, for example, the memory used to hold an array or a reference to an external file. These resources are obtained and attached to the object at the point of object creation. They are detached from the object and released when the object goes out of scope. Some resources may be moved between objects using intrinsic functions. However, a given resource may not be attached to more than one object.

As noted above, resource allocation occurs at object creation. Certain values are interpreted by the variable object creation process (invoked by **const =**, or **var** **=** ) as providing templates for resource allocation rather than a simple initial value. An example of this is the ***array\_template*** value. An array template value is returned by the **array** procedure and provides a template for array creation. The following statement:

var array= array(1,[1..3])

first creates an **array\_template** value containing the array shape and the value to be used to initialise the elements. The initialisationoperation creates an array object from the template, obtaining the memory required to store the array. This memory is relinquished when the array variable goes out of scope.

# Expressions

## Operator 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 | Priority | Equivalent call | Description |
| x[a,b … ] | 0 | $[].(x,a,b…) \* | Array subscript |
| # x | 1 | $#.(x) \* | Shape of x |
| x \*\* y | 2 | $\*\*.(x,y) | Power |
| x \* y | 3 | $\*.(x,y) | Multiplication (including matrix multiplication) |
| x / y | 3 | $/.(x,y) | Division |
| - x | 5 | $–.(x) | Minus |
| + x | 5 | $+.(x) | Plus (no-op on numerical values) |
| x # y | 6 | $#.(x,y) | Location in extent |
| x as y | 7 | $as.(x,y) | Type conversion |
| x shift y | 8 | $shift.(x,y) | Bitwise shift |
| x & y | 9 | $&.(x,y) | Bitwise and |
| x xor y | 10 | $xor.(x,y) | Bitwise exclusive or |
| x | y | 11 | $|.(x,y) | Bitwise or |
| x mod y | 12 | $mod.(x,y) | Modulo |
| x + y | 13 | $+.(x,y) | Addition |
| x – y | 13 | $-.(x,y) | Subtraction |
| x .. y | 14 | ($..).(x,y) | Range creation |
| ... y | 14 | ($\_...).(x,y) | Partial range creation |
| x ... | 14 | ($...\_).(x,y) | Partial range creation |
| by x | 15 | $by.(x) | Stride value creation |
| x by y | 15 | $by.(x,y) | Strided range creation |
| x == y | 16 | $==.(x,y) | Equals |
| x /= y | 16 | $/=.(x,y) | Not equal |
| x > y | 16 | $>.(x,y) | Greater than |
| x >= y | 16 | $>=.(x,y) | Greater than or equal |
| x < y | 16 | $>.(y,x) | Less than |
| x <= y | 16 | $>=.(y,x) | Less than or equal |
| x in y | 16 | $in.(x,y) | Membership |
| x inc y | 16 | $inc.(x,y) | Superset or equal |
| x is y | 16 | $is.(x,y) | Type membership test |
| not x | 17 | $not.(x) | Logical not |
| x and y | 18 | $and.(x,y) | Logical and |
| x or y | 19 | $or.(x,y) | Logical or |
| x except y | 20 | $except.(x,y) | Logical and not |
| x ortho y | 21 | $ortho.(x,y) | Orthogonal neighbourhood creation |
| ortho x | 21 | $ortho.(x) | Orthogonal neighbourhood creation |
| x fmt y | 22 | $fmt.(x,y) | Format as string |
| x ++ y | 23 | $++.(x,y) | String concatenation |

\* Exact semantics/call equivalence may vary in a parallel context

## Sub-expressions

*xexpr*::=

*expr subexp*

*subexpr*::=

[ **check** *exprlist* ] *whereclause*

*whereclause*::=

{ **where** *constdefs* }

Sub-expressions may be separated out 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)

It is also possible to include a **check** clause after an expression. This raises an error if the associated logical expression does not yield a **true** value. The check expression is evaluated after the main expression and after any assignment or variable creation:

x= my\_sqrt\_fn(x)

check x\*\*2==old\_x

where old\_x=x

A **check** clause may provide an associated error message (a string):

y = f(x) check "f returned –ve value: "++y => y>=0

Check expressions may optionally not be compiled or executed.

# Sequential control statements

## Blocks

'**:**' *statement*

'**{**' *statements* '**}**' [ '**--**' ( *name* | *mkeyword* ) ]

PM groups statements into blocks using braces. In addition to grouping statements, a block usually introduces a new object scope (the exceptions are labelled blocks, **any** and **nhd** statements):

do{

var x=0

x=x+1

print(x)

}

// x no longer defined outside of the block

There are two alternative syntactic forms for blocks. In the first, a single statement may be introduced using a colon:

if x<0 : print(x)

In order to clarify a complex code, a block may have a ***matching terminator***. This is a '**--'** symbol followed the keyword associated with the statement, the statement label (if present), or in the case of **for** or **foreach** statements only, the first iteration object defined:

do { … } -- do

while … { … } -- while

if … { … } else { … } -- if

main\_loop: while … { … } -- main loop

foreach i in …, … { … } -- i

## Proceed statement

A **proceed** statement is a placeholder that does nothing (no-operation).

## Do statement

**do** *block*

A **do** statement simply executes the statements in its associated block.

do {

var x=12

x=x\*x

print(x)

}

## If statement

**if** *xexpr block* { **elseif** *xexpr block* } [ **else** *block* ]

The **if** statement conditionally executes a list of statements. Only the final block in an **if** statement may have the '**::'** form or matching '**--'** terminator**.**

if x > y {

print("X is greater")

}

if x>y {

print("X is greater")

} else {

print("Y is greater or equal")

} -- if

## Switch statement

**switch** [*xexpr* ] '**{**'{ **case** *xexprlist* '**:**'*statements* } [ **default** '**:**' *statements* ] '**}**' [ '**--**' **switch** ]

The **select** statement tests an expression against lists of values and executes the first statement list to be associated with a value matching the expression.

switch digit {

case '1','3','5','7','9' : print("Odd")

case '2','4','6','8' : print("Even")

default : print("Not a digit!")

}

If the test expression is missing it is assumed to be equal to **true**.

## Until statement

**until** *xexpr block*

The **until** statement sequentially repeats a list of statements until an expression yields a true value (test at the end).

until x/=0 {

x = x/2

y = y+1

}

## While statement

**while** *xexpr**block*

The **while** statement executes a statement zero or more time while a given expression yields a true value (test at the start)

nbits=0

while x>0 {

x = x/2

nbits = nbits + 1

}

In each of these cases, conditional expressions must return a **bool** value. An error will result if they do not.

## Foreach statement

**foreach** *iter subexpr* [ **until** *xexpr*| **while** *xexpr*] *block*

The **foreach** statement executes its body for each element in order in a given ***iterable value*** (a sequence, array, domain, matrix or vector). Iteration order is defined by the shape of the value.

foreach i in [1..2,3..4] {

print(i.d1+10\*i.d2)

}

will output 31 32 41 and 42 in that order.

It is possible to iterate over more than one iterable value, providing they have conforming shapes. Iteration order is determined by the shape of the first (leftmost) iterable expression.

foreach i in shape(a),j in a {

print(i//"="//j)

}

If one of the iteration variables is assigned to, this will change the value for the corresponding element in the value being iterated over (assuming such assignment is possible – an error occurs if not).

foreach i in shape(a),j in a {

j=i\*\*2

! Assigning to i would generate an error

} -- foreach

A **while** clause tests for termination at the start of the loop body:

foreach i in 1..10 while i\*\*2<10 {

print(i)

} -- i

// prints out 1 2 3

## Test statement

The **test** statement checks if an expression yields a **true** or **'true** value and raises an error if this is not the case (in the case of a  **'false** value, during program analysis).

**test** [[*string*'**=>**' ]*expr*{'**,**' [*string*'**=>**' ]*expr* } ] [ *block* ]

For example:

test value /= null

test same\_type(a,b) // Compile-time error if types do not match

The **test** statement can be supplied with a literal string which will form part of the error report if the test condition fails.

test "x out of range" => 0<=x and x<=max\_thresh

A test statement may be associated with a block of statements which will be executed before the test expression(s) are evaluated. Test expressions may refer to variables and constants created in the associated block.

test success { success=run\_checks(…) }

test { run\_all\_tests(…) } -- test

A PM system will contain an option to disable run-time checks – disabling checks will also disable all statements in the associated block.

# Polymorphism and dynamic dispatch

## Polymorphic types

A ***polymorphic*** value has an internal representation that is capable of encoding any value that conforms to a given type constraint.

A polymorphic type is the concrete type of a polymorphic value. It can be described using the following type constraint expression.

'**\***' *type*

A polymorphic type **\****T* conforms to **\****U*if *T* conforms to *U*.

A value of a given polymorphic type is created by using a type conversion:

proc w(x:\*num)=x

x = 1 as \*num

var y : \*num = 1

z=w(1.0) // x, y and z all have concrete type \*num

A variable initialised to a polymorphic value may contain any value conforming to the given type constraint.

The binary '**|**' operator returns the value contained in the polymorphic value on its left providing it has the same type as the value given on the right. Otherwise the value on the right is returned.

type int\_or\_string is int,string

var a= 1 as <\*int\_or\_string>

a = "Hello" as <\*int\_or\_string>

print(a|"") // Outputs "Hello"

print(a|2) // Outputs 2

## Dynamic dispatch using the any statement

In PM, dynamic (runtime) dispatch is achieved using a control structure (the **any** statement) rather than being associated with each subroutine call as would be the case in some other languages. The **any** statement has the following form:

**any** *name* ['**=**' *xexpr* ] *block*

An **any** statement takes a named polymorphic object, extracts the object it contains and associates that extracted object with the original name for the scope of the statement. It then runs a body of statements.

var a = 3 as <\*num>

any a {

a = a+a

}

It is possible to specify an expression, rather than a polymorphic object (in which case the given name must not clash with existing constants, variables or parameters):

m= rec V { … , counter=0 as \*any\_int }

any x = m.counter : count=long(x)

Note that by design the block associated with an **any** statement does not introduce a new variable scope. Therefore in the above example it is possible for the **any** statement to define the constant **count** that may be accessed after the end of the statement block. If the expression is a direct reference to an object component, then the given name may be used to update that component.

any x = m.counter : x=x+1

It should normally be possible to for a PM system implement an **any** statement with no more execution overhead than a **switch** statement – the body of the statement being resolved for each possible type that could be extracted from the polymorphic value.

# Ranges, Extents, Sequences and Grids

## Ranges

Range types represent closed intervals. Range values are created using the '**..**', operator. For numerical ranges mixed numerical type balancing rules apply. *E.g*.:

range1= 0..5 // Contains 0, 1, 2, 3, 4, 5

range2= 0.0..5 // Contains { x: 0.0 ≤ x ≤ 5.0 }

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

It the lower bound is greater than or equal to the upper bound then the range contains zero values (although it may form the basis of a non-empty descending sequence, as described below)

The following intrinsic range type are defined:

**range\_base is any\_real,…**

**range**  **(** *T***: range\_base )**

Integer ranges may be interpreted as integer sequences with a unit stride. The function **step(***r***)** returns a value of one and **size(***r***)** returns **max(high(***r***)-low(***r***),0)** if *r* is a range of integer type.

## Extents

An extent is a tuple of integer ranges. It is primarily used to delineate a multidimensional rectangular subset of a multidimensional integer space, for example for specifying the bounds for indices for a given array.

The extent types are:

|  |  |  |
| --- | --- | --- |
| **extent1d** | **is** | **tuple1d(range(int))** |
| **extent2d** | **is** | **tuple2d(range(int))** |
| **extent3d** | **is** | **tuple3d(range(int))** |
| **extent4d** | **is** | **tuple4d(range(int))** |
| **extent5d** | **is** | **tuple5d(range(int))** |
| **extent6d** | **is** | **tuple6d(range(int))** |
| **extent7d** | **is** | **tuple7d(range(int))** |

All extent types conform to **extent**.

The size of an extent *x*, **size(***x***)** returns the total number of points in the extent while **dims(***x***)** returns a tuple of integers giving the size of each dimension of *x*.

Two extents, *x* and y, are said to conform if they have the same rank and same dimension sizes: **dims(***x***)==dims(***y***)**.

## Strided ranges

A strided range 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 (or stride) between sequence elements. Numeric balancing will apply between the low and high values of the range and the step argument.

integer\_sequence = 1..6 by 2 // 1, 3, 5

reverse\_sequence = 6..1 by -2 // 6, 4, 2

empty\_sequence = 1..6 by -1 // No values

real\_sequence = 3.2 .. 5.4 by 0.7 // 3.2, 3.9, 4.6, 5.3

Sequences may define values in ascending or descending order. An ascending sequence starts at the low bound of the range and successively adds the step (which must be positive) while the resulting value remains less than or equal to the second bound of the range. A descending sequence starts at the high bound of the range and successively adds the step (which must be negative) while the resulting value is greater than or equal to the low bound of the range.

Whatever the direction of the sequence, **low(***s***)** and **high(***s***)** define the lowest and highest values (**low(***x***)==***a* and **high(***x***)==**b when *s*>0 and **low(***x***)==***b*, **high(***x***)==***a* when *s*<0 for *x***=***a***..***b* **by** *s*). The first and last values of the sequence may be found using **first(***s***)** and **last(***s***)**. The step of a sequence may be found using **step(***s***)**. The number of elements in the sequence may be obtained using **size(***s***)**.

## Generic sequence type

The generic type **seq(***T***:range\_base)** contains **range(***T***:any\_int)** and **strided\_range(***T***).**

## Open ranges and sequences

An open range represents a set of values inclusively higher or lower than a given threshold. They are created using a prefix or suffix ... operator:

X= ... 3.0 // Contains { x: x ≤ 3.0 }, type range\_below(real)

Y= 2.0 ... // Contains { x: 2.0 ≤ x }, type range\_above(real)

An open range may also be strided:

xs= ... 3 by 2 // Type is strided\_range\_below(int)

ys= 2 ... by 3 // Type is strided\_range\_above(int)

It is also possible to have a strided infinite sequence created using a unary by operator

z= by 2 // Type is stride(int)

Open ranges and sequences are most commonly used in subscript expressions and in **switch** statements.

## Blocked sequences

A blocked sequence is a sequence of integer values that is defined by the following equivalence:

foreach j in block\_seq(low,high,step,width,align):print(j)

// is equivalent to

foreach i in low-align..high by step {

foreach j in max(i,low),i+width-1 {

print(j)

}

}

Therefore:

foreach j in block\_seq(3,32,10,5,2):print(j)

// Outputs 3 4 5 11 12 13 14 15 21 22 23 24 25 31 32

A blocked sequence conforms to the type **block\_seq**.

## Mapped sequences

A mapped sequence is an arbitrary strictly increasing or strictly decreasing sequence of integers. It can be created using the **map\_seq** procedure on an array whose elements meet the strictly increasing/decreasing criterion. Mapped sequences conform to the type **map\_seq**.

mseq = map\_seq({1,3,9,11,15})

## Grids

A tuple of sequences representing a multidimensional sequence (sequence of tuple values) with the first (.1) element varying most rapidly.

foreach i in [1..2,3..4] {

print(i.1++","++i.2)

}

// Outputs 1,2 2,3 1,4 2,4

A grid is a tuple of values, each conforming to **grid\_dim**. The **grid\_dim** type contains **seq**, **block\_seq** and **map\_seq**. All grids conform to **grid**. Grids with a given number of dimensions will conform to:

|  |  |  |
| --- | --- | --- |
| **grid1d** | **is** | **tuple1d(grid\_dim)** |
| **grid2d** | **is** | **tuple2d(grid\_dim)** |
| **grid3d** | **is** | **tuple3d(grid\_dim)** |
| **grid4d** | **is** | **tuple4d(grid\_dim)** |
| **grid5d** | **is** | **tuple5d(grid\_dim)** |
| **grid6d** | **is** | **tuple6d(grid\_dim)** |
| **grid7d** | **is** | **tuple7d(grid\_dim)** |

## Sequence and Grid Operations

There are a number of common operations defines on sequences and grids. The **dims** function gives the size along each dimension of the shape, expressed as a tuple of integers. The **size** function gives the number of points in the shape.

print(dims(array(3,5))) // Prints out [3,5]

print(dims(array(1..4,1..9))) // Prints out [4,9]

print(size(array(3,5))) // Prints out 15

A given element of a sequence may be obtained using a subscript expression. Subscripts for sequences start at zero.

x=(1..5)[2] // x is equal to 3

This operation can be reversed using the # operator, which returns the zero-based index of an element in a sequence (but not a mapped sequence):

x=(1..5) # 3 // x is equal to 2

These operations also apply to grids:

x=[1..5,1..5][2,2] // x is equal to [3,3]

x=[1..5,1..5]#[3,3] // x is equal to [2,2]

The **intersect** procedure returns the intersection of two **grid\_dim** values or two grids of the same rank,

x=intersect(6..10,1..20 by 2) // x is equal to 7..9 by 2

The **overlap** procedure operates similarly to **intersect**, but returns the intersection as zero based indices into the first argument.

x=overlap(6..10,1..20 by 2) // x is equal to 1..3 by 2

There **overlap** procedure can also return two values which provide the intersection as indices in the first argument and indices in the second argument respectively.

x,y=overlap(6..10,1..20 by 2) // x is equal to 1..3 by 2

// y is equal to 3..4 by 1

As noted above, intersect and overlap are defined for all types conforming to **grid\_dim**. The result types from these procedure depend on the argument types as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Range** | **Strided Range** | **Blocked Sequence** | **Mapped Sequence** |
| **Range** | *Range* | *Strided Range* | *Blocked Sequence* | *Mapped Sequence* |
| **Strided Range** | *Strided Range* | *Strided Range* | *Mapped Sequence* | *Mapped Sequence* |
| **Blocked Sequence** | *Blocked Sequence* | *Mapped Sequence* | *Mapped Sequence* | *Mapped Sequence* |
| **Mapped Sequence** | *Mapped Sequence* | *Mapped Sequence* | *Mapped Sequence* | *Mapped Sequence* |

# Parallel and distributed execution model

## Strands, nodes and node ownership

PM programs are assumed to be executed on a set of ***nodes***. Nodes are capable of efficiently executing a set of ***strands***. Strands are a much lighter weight concept than threads (even light threads) – in many cases a strand may correspond to no more than one iteration of a loop body in the final implementation of the PM code. The PM specification does not define how to translate the abstract concept of a node into hardware: a node can relate to anything from a single core to a sub-cluster. The primary requirement is that a node is capable of keeping its own hardware busy running available strands. This requires an efficient mechanism for strand migration within the node. Strand migration between nodes, however, is assumed to not to be possible at all in any acceptably efficient manner.

Each PM strand is said to ***own*** a set of nodes. Depending on the implementation, this may be interpreted in one of two ways which are equivalent in terms of PM semantics (if not necessarily in terms of performance):

1. The strand is running simultaneously on every node in its owned set.
2. Every node in a strand’s owned set is available to the strand to create child strands.

Ownership of a node can be shared among strands. In this case, the node will be responsible for scheduling the strands that own it.

PM assumes that the number of nodes available to the program is fixed by the implementation. When a PM program proceeds, this owned node set is divided up as follows:

* + 1. On start-up, the PM program runs a single strand. This strand owns all available nodes.
    2. When a parallel statement (**for**,or **par**) is encountered, a ***node topology*** is generated using the statement’s associated extent and the attributes of the parallel statement (if present). Elements of the shape are ***partitioned*** over this topology.
       - 1. A topology is a set of nodes with (optional) information concerned with their preferred connectivity.
         2. The extent is partitioned by assigning tiles to each node in the topology. Each tile is represented by a grid. The mapping from nodes to tiles is called a ***distribution***.
    3. A new strand is created for each point in the extent and assigned (possibly shared) ownership of a node in the topology according to its position in the partition.
    4. If the topology does not contain every node owned by the parent strand then ownership of the remaining nodes is given to some or all of the newly created child strands according to a ***work-sharing*** algorithm.
       - 1. The work-sharing algorithm associates each node *S* not in the topology with a node *P* that is in the topology. Any strand that owns *P* is then given (possibly shared) ownership of *S*.
    5. The parent strand is suspended and all newly created child strands are executed. Once all of the child strands have completed, ownership of the nodes is returned to the parent strand which is then resumed.

## Partitioned Global Address Space

PM adopts a hierarchical version of the Partitioned Global Address Space (PGAS) model for memory on a distributed system. A given strand may access any memory associated with a node owned by its parent strand. A distinction is made between accessing or modifying memory on a node owned by the strand (a ***non-communicating operation***) and accessing or modifying memory on node not owned by the current strand but owned by the parent strand (a ***communicating operation***). This model is manifest through the use of ***distributed arrays***. The extent of such an array is divided into tiles and assigned to nodes using a distribution in the same way that strands are assigned to nodes.

## Topologies and Work-sharing

When a strand owning multiple nodes encounters a parallel statement (**for** or **par**) it creates a new ***topology*** for those nodes. This is a particular *N*-dimensional arrangement of nodes that is suitable for efficiently partitioning a given *N*-dimensional extent. The default topology for an *N*-dimensional domain is determined by factoring the number of nodes owned by the current strand to yield a rectangular array of nodes with the required number of dimensions with as close as possible to the same number of elements along each dimension. It is also possible for parallel statements to override the default algorithm and provide their own topology.

The number of nodes required for a topology (either default or specified) may be less than the number of nodes owned by the current strand. In this event, nodes must be grouped using a work-sharing algorithm and the topology becomes an array of node-groups rather than an array or nodes.

The default work-sharing algorithm for *N* nodes onto an *M*-element topology assigns each node *i* to group *j* as follows:

1. For *i*≤*m* then assign node group *j*= *i*
2. For i>m then obtain j by solving the following:

Here *Ak* is an estimate of the work needed to be undertaken by group *k*. By default *Ak*=1 for all *k*, but other values may be specified in the attributes of a parallel statement. Groups are numbered in iteration order through the topology (first dimension changing most rapidly.) This option is directly inspired by *llc* (Reyes *et al*.,2009, 16th European PVM/MPI Users Group Meeting).

# Distributions

## Overview

A ***distribution***splits an extent into a set of tiles, each tile corresponding to one element of a topology. Each tile is represented by a grid. A distribution has type **distr** which is a tuple of **distr\_dim** values.

type distr is tuple(distr\_dim),...

type distr\_dim is no\_distr, direct\_distr,block\_distr, vblock\_distr,

cyclic\_distr,block\_cyclic\_distr,...

On most occasions, a distribution value is created from a ***distribution value template***. This is provides basic information about a distribution that, combines with a topology and extent, is sufficient to create a complete distribution value. In a similar manner to distributions, distribution templates consist of a tuple of distribution template dimension values. However, it is possible to just supply a single distribution template dimension value, which is then assumed to apply to all dimensions of the topology and extent.

type distr\_template is distr\_dim\_template,tuple(distr\_dim\_template),...

type distr\_dim\_template is null, direct\_template, block\_template,

vblock\_template,cyclic\_template, block\_cyclic\_template,...

## No distribution

A dimension of a distribution may be defined not to be distributed. A non-distributed distribution value for an extent dimension size *D* over a topology dimension size *T* is created using the procedure **no\_distr(***D*,*T***)** and conforms to type **nodistr**.

The distribution template dimension value for a no distribution is **null** (or in a tuple forming expression, a missing element).

## Direct distribution

A dimension of size *D* may be distributed over a topology dimension that also has size *D* using a direct distribution. A direct distribution value is created using the **direct\_distr**(*D*) procedure and conforms to **direct**.

The distribution template dimension value for a direct distribution is **DIRECT** (this is a unique value of type **direct\_template**).

## Fixed block distribution

A fixed block distribution divides a dimension size *D* into fixed-size blocks sized *B* over topology dimension size *T*. A valid block size *B* must satisfy *B*⋅*T* ≤ *D* < *B*⋅ (*T*+1). Tile *j*, *j*∈0..*T*-1 is defined by *j* ⋅ *B ..* (*j+1*)⋅ *B*

A fixed block distribution value is created using the **block\_distr(***extent\_size***,** *topology\_size*, *block\_size***)** procedure

Fixed block values conform to the type **block\_distr**.

The distribution template value for a fixed block distribution is **BLOCK(***b***)** where *b* is the block size (this is a procedure returning a value of type **block\_template**).

## Variable block distribution

A ***variable block distribution*** partitions a dimension of size *D* into *T*tiles: **floor**(*j\*D/T*)*..***floor**( (*j+1*) ⋅(*D-*1)/*T*), *j*∈*0..Ti*-1.

A block distribution value is created using the **vblock\_distr(***extent\_size***,** *topology\_size***)** procedure.

Variable block values conform to the type **vblock\_distr**.

The distribution template value for a variable block distribution is **VBLOCK** (this is a unique value of type **vblock\_template**),

## Cyclic distribution

A cyclic distribution divides a dimension size *D*into cyclically varying points over topology dimension size *T*. The tile for a given element *j*∈0..*T*-1, is given by *J* .. *D* by *T*.

A cyclic distribution value is created using the **cyclic\_distr(***extent\_size***,** *topology\_size***)** procedure.

Cyclic values conform to the type **cyclic\_distr**.

The distribution dimension template value for a cyclic distribution is **CYCLIC** (this is a unique value of type **cyclic\_template**).

## Block cyclic distribution

A ***block cyclic distribution*** divides a dimension sized *D* into cyclically varying fixed-size blocks of size *B* over topology *T*. The tile for a given element *j*∈0..*T*-1, is given by **block\_seq**( *j* ⋅ *B*, *D*, *T* ⋅ *B*, B ) .

A block cyclic distribution value is created using the **block\_cyclic\_distr(***extent\_size***,** *topology\_size, block\_size***)** procedure:

Block cyclic distributions conform to the type **block\_cyclic\_distr**.

The distribution dimension template value for a block cyclic distribution is **BLOCK\_CYCLIC(***b***)** where *b* is the block size (this is a procedure returning a value of type **block\_cyclic\_template**).

## User defined distributions

User defined distributions may be created by extending either the **distr** or **distr\_dim** types. The following four procedures then need to be defined for the new distribution type:

When extending **distr**:

dims(distribution:T)->tuple(int)

tile(distribution:T,node:tuple(int))->grid

node\_for(disribution:T,point:tuple(int))->tuple(int)

index(distribution:T,tuple(int))->tuple(int)

When extending **distr\_dim**:

dims(distribution\_dim:T)->int

tile(distribution\_dim:T,node:int)->grid\_dim

node\_for(disribution\_dim:T,point:int)->int

index(distribution\_dim:T,int)->int

The **dims** procedure must return the dimension sizes (or dimension size) of the underlying topology. The **tile** procedure must return the tile associated with a given point in the topology. Conversely, **node\_for** returns a position in the topology associated with a given point in the extent. Finally **index** returns the zero-based index of a given point within its associated tile.

Two procedures may be defined to improve performance under certain circumstances (although default implementations are available):

tile\_size(distribution:T)->tuple(int)

nodes\_for\_grid(distribution:T,grid\_value:grid)->grid

tile\_size(distribution\_dim:T)->int

nodes\_for\_grid(distribution\_dim:T,grid\_value:grid\_dim)->grid\_dim

Here the **tile\_size** procedure should give the same result as **size(tile(…))**. This may be useful if generating a full tile is computationally expensive. The **nodes\_for\_grid** procedure returns a conservative subset of the nodes in the topology whose tiles intersect with the given grid. It is used to limit the search for intersecting tiles in certain communicating operations. The default implementation for non-built-in distributions returns the complete topology.

For most purposes it will also be necessary to extend either **distr\_template** or **distr\_dim\_template** type and extend the **distribute** procedure that is used to create a distribution from a distribution template.

distribute(template:distr\_template,extent:tuple(int),topo:tuple(int))->distr

distribute(template:distr\_dim\_template,extent:int,topo:int)->distr\_dim

# Shapes

## Shapes

A shape is a combination of an extent and a distribution. A shape is used for two purposes: (i) to determine the association between strands and node-groups in parallel execution and (ii) to determine the node on which the memory for a given array element is stored. Shapes are often implicitly created by the PM system, for example in the creation of an array or the execution of a parallel statement.

A shape embeds its extent. It is therefore possible to use a shape as an argument for most procedures that take an extent using implicit conversion rules.

## Mirrored Shapes

A mirrored shape is a shape whose distribution is **no\_distr** in each dimension, in other words one in which execution or storage is mirrored on every node group.

A mirrored shape has one public element **.extent** which provides the underlaying extent of the shape.

## Distributed Shapes

A distributed shape is a shape that is not a mirrored shape, the distribution along at least one dimension not being **no\_distr**.

## Distributed Values

A value with a distributed shape, or which has an element or sub-element with a distributed shape, is known as a ***distributed value***. A value for which all elements or sub-elements have either no shape or a mirrored shape is known as a ***non-distributed value***.

## Values with shapes

A number of different types of PM value are associated with a shape, including ranges, strided ranges and arrays. The shape of a value is determined using the **#** operator.

The shape of any **grid\_dim** value is a one dimensional mirrored shape with a zero-based extent.

print((#(3..6)).extent) // Prints out 0..3

print((#(1.0..2.0 by 0.5)).extent) // Prints out 0..2

The shape of an *N*-dimensional **grid** value is an *N*-dimensional mirrored shape with a zero-based extent.

print((#[2..10,5..9]).extent) // Prints [0..8,0..4]

## Conformance of shapes

There exists a ***conformance*** relationship between shapes. This operates as follows:

1. A mirrored shape conforms to another shape (mirrored or distributed) is the two shapes have conforming extents.
2. A distributed shape conforms to another distributed shape is the two shapes have conforming extents and also have the same distribution.
3. A distributed shape cannot conform to a mirrored shape.

## Subscripting

Any value with a shape may be ***subscripted***. A subscript expression yields the value associated with a given point in the shape and has the following syntax:

'**[**' [ *expr* ]{ '**,** ' [ *expr* ] } '**]**'

For shapes, with *N* dimensions, subscript expressions may consist of *N* expressions yielding integer values. …

b=a[1,3]

c=a[s] where s=[2,2]

## Slicing

Most values with a shape be sliced. A slice yields the values associated with a subset of the points in the shape.

d=a[1..3,2..6 by 2]

A slice is applied along each dimension of the array as follows. If the shape of the original value has points {*x*: *start*≤*x*≤*finish*}then:

1. A slice value that is an integer range *low***..***high* restricts indices along the sliced dimension to those lying in {*low*≤*x*≤*high*}.
2. A slice value that is a strided integer range *low***..***high* **by** *step* restricts indices along the sliced dimension to {x: *x*=*low*+*n*×*stride*, integer *n*>0, *x*≤*high*}.
3. A slice value that is an open integer range **…***high* restricts indices along the sliced dimension to those lying in {x: *x*≤*high*}.
4. A slice value that is an open integer range *low***…** restricts indices along the sliced dimension to those lying in {x: *low*≤*x*}.
5. A slice value that is a strided open integer range **…***high* restricts indices along the sliced dimension to those lying in { x: x=*start* + *n*×*stride*, integer *n*>0, *x*≤*high* }.
6. A slice value that is a strided open integer range *low***…** restricts indices along the sliced dimension to those lying in {x :*x*=*low*+*n*×*stride*, integer *n*>0}.
7. A slice value that is a stride definition **by** *step* restricts indices along the sliced dimension to those lying in {x: x=start + n ×*stride*, integer *n*>0, *x*≤*finish* }
8. A **null** slice value does not restrict indices along the given dimension.

The shape of the sliced value is zero-based.

print(#shape(a[2..4,3..7)) // Prints out “[0..2,0..4]”

If a subscript expression contains a mixture of integer and range /sequence values, the resulting slice will have a reduced number of dimensions:

e=a[1..3,5]

When open ranges are used in subscripting, they act as if they were a closed range with the upper or lower bound as appropriate taken from the shape of the value being subscripted:

f=a[...7,2...]

A missing or null subscript keeps all points along the specified dimension:

g=a[2,]

# Arrays, Vectors and Matrices

## Array templates

An **array\_template** value is created using either the **array** or the **varray** procedures. This value contains the information needed to create and initialise an array. The first argument specifies an initialising value while the second provides an extent or the required shape while the third argument (if present) provides a distribution template.

array(0.0,[0..3,0..3])

varray(0.0,#A)

array(0.0,[0..100,0..100],BLOCK(10))

The difference between **array** and **varray** is that the **varray** procedure creates a template for a variable-length array whose shape can change dynamically while the **array** procedure creates a template for an array whose shape is fixed at creation and cannot be changed by subsequent assignments.

If an extent only is supplied to the array described by the template has a mirrored shape. If both an extent and distribution are supplied then the template describes an array with a distributed shape. If a shape is supplied then the described array has exactly that shape (mirrored or distributed).

For convenience, many of the procedures and operators applicable to fixed arrays are also applicable to array template values.

## Arrays

An array value stores a set of values corresponding to each element of a given domain.

Array values may be created by creating a variable or constant initialised by an array template dimension value.

var A = array(0.0,[1..4,1..4])

This works even if the dimension values are embedded in a larger structure.

var masked = new mask\_array{array(1,extent),array(false,extent)}

where extent=[1..3,1..3]

One and two dimensional array values with shapes **[0..N-1]** or **[0..N-1, 0..M-1]** may be defined using the following syntax:

'**{**' *list2d* '**}**'

*list2d*::=

*exprlist* {'**;**' *exprlist* } ['**;**']

For example:

array\_1d = { 1, 2, 3 } // shape is [0..2]

array\_2d = { 1, 2, 3 ; 4, 5, 6; 7, 8, 9} // shape is [0..2,0..2]

This notation creates an unnamed constant array object. Note that unlike statement blocks, semicolons are not optional.

Array assignment cannot change the type of the array, but can change the value of its elements and in the case of an array created from a **varray** template it may also change its shape, making it possible to resize an array through assignment. Arrays with zero size shapes are permitted for both **array** and **varray** and will have no elements.

## Vectors and matrices

A vector is an array defined over a 1-dimensional shape with a **vector** tag.

A matrix is an array defined over a 2-dimensional shape with a **matrix** tag

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

'**(**'*list2d* '**)'**

*list2d*::=

*exprlist* {'**;**' *exprlist* } ['**;**']

For example:

v = ( 1, 3, 1 )

a = ( 3, 1, 1;

0, 2, 1;

0, 0, 1 )

As with array generators, all elements must have the same type (numeric type balancing is not invoked.) Note that unlike in statement blocks, semicolons are not optional.

Matrix and vector values follow the usual rules for matrix multiplication using the '**\***' operator. Vectors act as row or column matrices as appropriate in the context of matrix multiplication.

## Accessing and updating array elements

An element or slice of an array may appear in an object element reference (left hand side of an assignment or **&** argument). In an assignment, a slice on the left hand side may be set to a conforming array value or to a single value which is set to all elements of the slice.

h[,2]=0

m[2,2]=m[2,2]+1

n[0..2,0..2]=m[3..5,3..5]

f(&m[0..5,])

# Parallel execution

## The for statement

The **for** statement executes its body using a separate strand for every corresponding element of one or more sequences, grids or arrays.

**for** *iter* [ *attr* ] [ *subexp* ]  *block*

*iter*::=

*name* (**in |** '**=**' ) *exp* { '***,***' *name* (**in |** '**=**' ) *exp* }

For example:

for pixel in image {

pixel = min(pixel,threshold)

}

It is possible to iterate over more than one sequence, grid or array, providing they all share conforming shapes:

for pixel in image1,pixel2 in image2 {

if pixel+pixel2>threshold {

pixel=pixel-threshold/2

pixel2=pixel2-threshold/2

}

}

The **for** statement determines the shape of the first supplied ***iteration element*** and then checks that all subsequent iteration elements have shapes conforming to this first shape. If the shape is mirrored, then a new topology and distributed shape are determined, using either the default topology and default distribution template (**VBLOCK**) or the values for these supplied by the attributes, if present.

## The par statement

Task parallelism is supported by the **par** construct.

**par** [ *attr* ]'**{**' *statements* { **task** *name* [ *attr* ] '**:**'*statements* } '**}**'

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

**par** *opts* **{**

*statements0*

**task** *name1* **:** *statements*1

**task** *name2* **:** *statements*2

…

**task** *nameN* **:** *statements*N

**}**

is semantically equivalent to:

**for***\_index* **in *0*..***N***-*1*** *attr*

**{**

*name1* **=** ***0***

*name2* **=** ***1***

**…**

*nameN* **=** *N****-1***

*statements*0

**switch \_***index* **{**

**case *0* :** *statements*1

**case *1* :** *statements*2

…

**case** *N-1* **:** *statements*N

**}**

**}**

## Parallel statement attributes

The partitioning and work-sharing processes may be modified by providing attributes to a **for** or **par** statement. The following attributes are defined:

**topo=***expr* Use the given topology in place of the default*.*

**distr=***expr* Generate the distribution using the supplied distribution template.

**work=***expr*Specifies an amount of work associated with each domain element. Used by work-sharing algorithms. Should be an integer array defined over the extent of the topology.

## Implicitly defined constants in parallel statement

All parallel statements implicitly define the following constant (ignoring the usual no-shadowing rule):

|  |  |
| --- | --- |
| ***region*** | The shape associated with the parallel statement |
| ***subregion*** | Initially **null** (this may be modified by an over statement) |
| ***here*** | The zero-based index of the current strand |

## The over statement

The **over** statement restricts computation to a given sub-region of the shape of the parallel statement:

An **over** statement is semantically equivalent to an **if** statement, but is designed to enhance both readability and to enable efficient translation to loop structures in the object code. An over statement also defines a new **subregion** constant (ignoring non-shadowing rules).

over [N..M] {… }

is equivalent to:

if here in [N..M] { subregion=[N..M]; … }

The subregion is specified according to zero based indices. In addition, it is possible to use unbounded sequence in the same manner as array slicing:

over [1..by 2,1..by2 ] { … } // Only execute for odd rows and columns

It is possible to nest over statements. However, an **over** statement itself may not otherwise be placed in a ***conditional context***.

## The forall statement

**forall** *iter* [ *subexp* ]  *block*

The **forall** statement is very similar to the **for** statement, except:

1. It does not apply a default distribution to mirrored shapes
2. It does not allow **topo**, **distr** or **work** attributes.

Apart from these differences, all of the features of the **for** statement apply – including creating a parallel context, creating new strands and allowing communicating operations between strands in its scope.

forall i in [1..10,1..10] {

// Creates 100 strands on each node owned by parent strand

// Each node mirrors the execution of the same code on the

// same data.

…

}

# Parallel contexts and value modes

## Parallel contexts

When a parallel statement (**for**, **forall** or **par**) is executed, it creates a new ***parallel context*** over its associated block of statements (for a **par** statement this is the entire block including both the initial set of statements and each task section.) The parallel statement creating the new parallel context will, of course, itself be situated within a parallel context – this existing parallel context is referred to as the ***directly enclosing parallel context***. A parallel context that encloses another parallel context, with or without intervening parallel contexts is known as an ***enclosing parallel context***.

The new parallel context will be associated with a number of newly created strands. The statement block associated with the parallel statement will be executed by each newly created strand associated with the newly created parallel context (which becomes the ***current parallel context*** for those statements). Statements in the block that are unconditionally executed (i.e. not inside any nested **if**, **switch**, or **over** statement) are said to reside in an ***unconditional context***. Statements that may only be executed by some strands due to the presence of a nested **if** or **switch** statement are said to reside in a ***conditional context***.

for h in … {

// An enclosing context

for i in … {

// Another enclosing context

for j in … {

// Directly enclosing context

for k in … {

// Current parallel context

// Unconditional context

if … {

// Conditional context

}

}

}

}

}

## Object modes

Every object created in a parallel context has a ***concrete mode***: **partial**, **coherent**, **chan**, **uniform** or **shared**. This provides information on the relationship between the value stored in the object, the current parallel context and the directly enclosing parallel context. In addition, there are five ***abstract modes***: **private**, **invar**, **complete**, **universal** and **local**, which describe related groups of concrete modes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **private** |  |  | **local** | **partial** | The object is defined in some of the strands in the current parallel context, but not necessarily all. The object may have different values in different strands. |
| **complete** | **universal** | **coherent** | The object is defined in all strands in the current parallel context with values that may differ between strands but are synchronised with other stands in that context. |
| **chan** | The object is defined in all strands in the current parallel context, with values that may differ between strands. Values may be communicated between strands in the current parallel context. |
| **invar** |  | **uniform** | The object is defined in all strands in the current parallel context and has exactly the same value in all of these strands. |
|  | **shared** | The object is defined in the directly enclosing parallel context. |
|

## Importing values into a parallel context

When the value of an object defined in an enclosing parallel context is used in the current parallel context then it will usually be ***imported*** into that context. Importing will usually reinterpret the value without having any specific runtime effect. However, in some situations (*e.g.*: the use of accelerator hardware) importing a value may have a runtime cost. Importing objects follows the following rules:

1. The mode of an imported object is **uniform**.
2. It is not possible to import a distributed value.

While a **shared** distributed value cannot be imported into a parallel context and used directly in an expression, it is permissible to subscript or slice a shared distributed array (which may itself by an element of a shared distributed value). The array is not imported. Rather the subscript or slice operation fetches the required values using appropriate inter-strand communication.

var a=darray(1.0,[1..10])

for i in … {

b=a[i] // This is allowed

c=a[2] // This is also allowed

d=a // This will raise an error

// - importing a distributed value

d=a[] // This is ok – will obtain the whole value of a

}

There are a number of other places where an object will not be imported:  ***synchronised assignments*** and arguments to ***shared procedure calls*** or ***communicating procedures***. These are all examples of ***communicating operations*** which are discussed later in this document.

## Determining the mode of an expression

The mode of a given expression may be determined by inspection. The following rules apply:

1. The mode of a literal constant is **uniform**.
2. The mode of an imported value is **uniform**.
3. The mode of a constant object defined in the current parallel context is equal to the mode of its initialising expression.
4. The mode of a value retrieved from a variable object defined in the current parallel context is
   1. **coherent** if the variable was defined in an unconditional context and the value is being retrieved in either an unconditional context or a labelled statement.
   2. Otherwise the mode of the retrieved value is **partial**.
5. For a the result of a standard procedure call:
   1. If the call is made in a conditional context then the mode of any result value is partial
   2. Otherwise, if any argument has a **partial** mode then the mode of any result value is **partial**.
   3. Otherwise, if any argument is **coherent** or **chan**, then the mode of any result value is **coherent.**
   4. Otherwise, the mode of any result value is **uniform**

## Assignment to variables created outside of the current parallel context

In order to guarantee the independent operation of strands, it is not generally permissible to assign to a variable defined outside of the current parallel context, either using conventional assignment statements or through a reference argument in a conventional procedure call. There are specialised statement and procedure call forms that enable this (including **sync** statements, ***shared procedure calls*** and ***communicating procedures***). These are discussed later in this document.

## Invariant control statements

The body of an **if**, **switch**, **while**, **until** or **foreach** statement is a conditional context. However, conditional contexts can be restrictive. If the controlling expression of such a statement is **uniform** or **shared**, then it can be advantageous to us an **if invar**, **switch** **invar**, **while invar**, **until invar** or **foreach invar** statement instead.

**if invar**  *xexpr block* { **elseif** *xexpr block* } [ **else** *block* ]

**switch** **invar** [*xexpr* ] '**{**'{ **case** *xexprlist* '**:**'*statements* } [ **default** '**:**' *statements* ] '**}**'

**while** **invar** *xexpr**block*

**until** **invar** *xexpr**block*

**foreach** **invar** *iter subexpr* [( **while** | **until** )*xexpr*]*block*

The body of an **invar** statement is an unconditional rather than a conditional context – if any strand runs this block of statements then all will. Note that **invar** statements may only be executed in unconditional statements.

for i in 1..3 {

foreach j in 1..10 {

v=f(j) // v has partial mode

}

foreach invar j in 1..10 {

v=f(j) // v has coherent mode

}

}

# Communicating operations

## Synchronised assignments: the sync statement

The full power of parallel statements becomes effective when different strands are available to communicate with each other using ***communicating operations***. The most commonly used communicating operation is the **sync** statement which, unlike ordinary assignments, can change values of variables defined in the directly enclosing parallel context. Variables modified by the **sync** statement must have a **private** mode in the directly enclosing context (which, when not imported, gives them a **shared** mode in the current parallel context).

The following ***synchronised assignment operations*** are permissible:

1. A **shared** variable may be assigned a **uniform** or **shared** value.

sync a=true

1. A **shared** variable may be updated using a synchronised operator assignment, providing the operator name conforms to the open abstract type **associative\_operators**. Each strand updates the value stored in the variable, but the order in which this occurs is not defined (hence the requirement that the assignment operators must be associative.)

sync a+=b // a is shared, b may have any mode

1. An element or sub-element of a **shared** array (distributed or non-distributed) may be assigned a **uniform** or **shared** value. For example:

sync a[i].b.c[j]=d // a is shared, is invariant

// i and j may have any mode

1. An element or sub-element of a **shared** array (distributed or non-distributed) accessed using at least one private subscript may be assigned a private value providing that the following rule is obeyed:

If two strands pass through the same **sync** statement and both assign a value to the same element of the array, they must assign exactly the same value (otherwise there could be a race condition between the two strands). This constraint may be tested for at run-time using code similar to the following (an option being available to automatically generate this test):

sync a[i].b.c[j]=d // Assignment where a is shared, d is private

// and at least one of i and j are private

// Synchronisation rules state that all strands

// strands have completed the assignment at this point

test a[i].b.c[j]==d

// Runtime test (automatically generated)

## Channel objects

While most inter-strand communication can be accomplished using shared arrays, it is sometimes useful to be able to communicate between locally defined private objects. This is accomplished by declaring a variable object at definition to be a ***channel object***.

Channel objects in most aspects behave exactly the same as normal variables or constants. However, there are some important differences.

* + 1. Channel objects have the mode **channel**
    2. The initialising expression for a channel object must have a **coherent** mode.
    3. Applying the **@** qualifier to the a variable or constant name treats that object as an element of a distributed array with the extent and distribution of the current parallel context (*i.e.*: the shape of the resulting array is equal to **region**).
    4. Assigning to a channel variable, unlike assigning to a regular private variable, is regarded as a synchronised assignment.

## Shared and complete procedure calls

A ***shared procedure call*** runs the procedure in the parent strand of the current strand, or in other words runs the called procedure in the directly enclosing context. All of the arguments passed to a shared procedure call must be invariant and the **shared** attribute must be present. Even if more than one strand invokes a shared procedure call, the procedure will only be run once. A shared procedure call can change shared variables passed as reference arguments.

A ***complete procedure call*** runs the procedure on all strands in the current parallel context, even when the procedure call itself takes place in an unconditional context. All of the arguments passed to a complete procedure call must have a **universal** mode and the complete attribute must be present.

If no strand invokes a shared or complete procedure call then the procedure will not be run. However, testing for this in a distributed context can be expensive (potentially, a node on which the call has not been made directly by any strand has to ask all other nodes in the current parallel context if they have made the call). For this reason, it is sometimes useful to always run the procedure call, even if no strand in the current context directly executes the call. This is achieved by adding an **always** attribute to the call.

## Indexed values

An indexed value is a value that is a simple function of the strand index **here**. Indexed values offer a considerable performance advantage over private values in that they do not need to be communicated between nodes – a node may simply compute their values as needed. An indexed value is an **invariant** value of type **indexed** which is a tuple of **index\_dim** values.

An **indexed** value may generated by using a *name***=***expr* iteration term in a **for** statement, instead of a *name* **in** *expr* iteration term where the value of the expression is a grid.

Indexed values may also be created by manipulating individual **indexed\_dim** values in the tuple and reconstructing an indexed value. The following arithmetic operations are available on an **indexed\_dim** value:

proc +(indexed\_dim)

proc –(indexed\_dim)

proc +(indexed\_dim,any\_int)

proc +(any\_int,indexed\_dim)

proc –(indexed\_dim,any\_int)

proc –(any\_int,indexed\_dim)

proc \*(indexed\_dim,any\_int)

proc \*(any\_int,indexed\_dim)

proc /(indexed\_dim,any\_int)

Note that an **indexed\_dim** value remembers the dimension that it was originally associated with. Therefore, the following expression will describe a transposed set of indices:

[ idx.2 , idx.1 ] // here idx is an indexed value

An indexed value for a given strand may be obtained using the unary operator \*. However, indexed values are primarily used for array indexing:

for i=[1..N,1..M] {

sync a[i.2,i.1]=… // Transpose

sync b[i.1,i.1,i.2]=… // Set a diagonal

c= d[N-i.1,M-i.2] // Flip both dimensions

}

# Structured parallel communication

## Synchronisation points

When separate strands, which may be running on different nodes, communicate there is, of course, a potential for race conditions to occur. For example, the order in which two different strands modify the same shared variable may become important. PM establishes a clear set of coherency rules to ensure such race conditions do not occur. These rules ensure that the PM compiler can always determine a logical sequence in which interacting parallel operations must occur. It is then the task of the compiler to enforce this logical ordering while placing minimal requirements on actual, physical, synchronisation between nodes.

Code in the body of any non-trivial parallel statement will contain a set of ***synchronisation points***. Synchronisation points can be ***active***, which directly change values accessible by other strands, or ***passive***, where an operation is sensitive to inter-strand ordering but does not necessarily communicate between strands.

***Active synchronisation points*** are associated with the following statements:

1. **sync** statement
2. **exit** statement
3. **nhd** statement

An ***active synchronisation point*** will also be associated with a conventional statement that does any of the following:

1. Modifies a value in a **chan**variable(either through assignment or passing as a reference argument)
2. Calls to a procedure using a ***coherent procedure call*** with at least one reference argument
3. Calls to a procedure using a ***shared procedure call*** with at least one reference argument.
4. Calls to a procedure using a ***communicating procedure call*** with at least one reference argument.

A ***passive synchronisation point*** is associated with a **coherent** or communicating procedure call that must be passed **coherent** arguments.

A statement containing a synchronisation point is a synchronised statement. Conceptually, strands passing through a given synchronisation point will wait for each other before proceeding. That is:

1. All strands will asynchronously execute up to the statement preceding the synchronised statement
2. The synchronised statement is simultaneously executed by all strands
3. All strands asynchronously execute statements following the synchronised statement.

In practice a synchronisation point does not have to impose a hard barrier between strands, providing the sequence in which values are modified is the same as the above rules would dictate.

If code is entirely straight-line and unconditional, then matching synchronisation points between strands is trivial – all the strands execute the same statements in the same sequence. The same is true for code in **invar** statements. However, the presence of non-**invar** conditional statements and loops makes the matching process more complex.

For non-invariant loops that are not themselves in a conditional context, synchronisation occurs as follows:

1. If the loop is active in all strands, then synchronisation takes place as for the straight-line case.
2. If the loop terminates on some strands, but not others then:
   1. strands with terminated loops wait at the end of the loop
   2. synchronised statements are matched in the usual order between active stands in the loop body
3. When all strands terminate the loop, they can all proceed to subsequent statements.

Labels are also required for passive synchronisation points. In a conditional context a value defined in the current parallel context will have a **partial** mode, even if it was itself defined in an unconditional context. However, in a labelled statement, a value defined in an unconditional context in the current parallel context will have a **coherent** mode.

## Synchronisation in conditional contexts – label matching

Conditional expressions introduce an additional level of complexity, since different strands may be executing different statement. For this reason synchronizing statements are not permitted in a conditional context – a restriction that may be overcome through statement labelling. A label may be applied to a single statement or to a block of statements.

A statement with the same label must appear on each non-empty path though conditional code and labels must be encountered in the same order along each path. Note that paths are more general than branches of the same conditional statement. Thus:

var a=0

var b=0

for i in 1..10 {

if i<5 {

change\_a: sync a+=2

change\_b: sync b+=a

} else {

if i< 7 {

change\_a: sync a-=1

change\_b: sync b-=a

}

}

}

print(a) // prints 0

print(b) // prints 2

In terms of side effects visible between strands, the textually first appearing labelled statement is executed before the second, and so on, with execution proceeding beyond the labelled statements only when all have completed. In the above code, synchronisation point **change\_a**, will effectively execute following code simultaneously on all strands in the current parallel context :

if i<5: sync a+=1

if i>=5: if i<7: sync a-=1

Similarly, at synchronisation point **change\_b**:

if i<5: sync b+=a

if i>=5: if i<7: sync b-=a

Since all strands must execute **change\_a** before any strand can proceed further, the value of a in **change\_b** will be the final result from the accumulated changes made by all strands.

Labelling a block of statements does not introduce a new variable scope.

Formally, the label-matching process operates as follows:

1. All stands will asynchronously execute up to the statement preceding a labelled statement
2. All strands must have encountered a statement with the same label name – otherwise a compile-time error results.
3. All labelled statements in strand executing in the first (textually in the source file) conditional path are executed with synchronisation-point matching applied only to these active strands.
4. Once all strands executing the first labelled statement have completed, the process is repeated for the textually second labelled statement.
5. This continues for all non-empty paths in textual order.
6. All strands asynchronously execute statements following their respective labelled statement.
7. Strands executing an empty path will only continue once all labelled statements in corresponding non-empty paths have completed on their respective strands.

## Merging synchronised loops

Statement labelling of sequential loops will force one loop to complete before the other commences. In many cases, the preferred behaviour is for loops to execute simultaneously, rather than sequentially. This is achieved by using merge-labelled loops. These operate as follows:

1. All strands execute asynchronously until they encounter a merge-labelled loop.
2. Each strand must encounter a loop with the same merge-label name, otherwise there is a compile-time error.
3. Different strands are not required to encounter the same kind of loop – **while**, **until** and **foreach** loops may be inter-matched freely.
4. The bodies of the respective loops are matched in the same manner as two different branches of a conditional statement.
5. No strand proceeds beyond the end of its merge-labelled loop until all strands have completed their respective merge-labelled loops.

LAST=…

var transfer=…

for i in 1..2 {

if i==1 {

pipeline> foreach i in 1..N {

pass\_value: if i<N {

transfer=next\_value(…)

} else {

transfer=LAST

}

}

} else {

pipeline> until transfer==LAST {

pass\_value: if transfer/=LAST: process\_value(transfer)

}

}

}

# Communicating procedures

## Overview

***Communicating procedures*** are user defined communicating operations that, unlike regular procedures, may incorporate communicating operations that operate within the context of parallel statement enclosing the procedure call.

**proc** *procname* '**%**'  *params result*  [*block* ]

**proc** *procname* '**%**'  *params* [ *returns* ][ **check** *exprlist* [ *whereclause* ] ] *block*

Communicating procedures can only be called in a parallel context (i.e.: inside **for** or **par** statement or inside the body of another conditional procedure). A communicating procedure call has the following syntax:

*mname* '**%**' '**(**' *arglist* [ *attr* ]'**)**'

*term* '**.**' '**%**' '**(**' *arglist* [ *attr* ]'**)**'

'**$**' *procname* '**%**' '**(**' *arglist* [ *attr* ]'**)**'

For example:

proc sum%(x) {

shared sum=convert(0,x)

sum+=x

return x

}

for value in array {

sum\_value=sum%(value)

// sum\_value has shared mode

}

Values returned by a communicating procedure take the mode of the expression in the **return** statement (or **=***expression*).

## Matching parameter and argument modes.

A communicating procedure may augment a parameter type constraint with a mode, yielding a ***moded type constraint***.

1. A moded type constraint matches a given value if the value has both a matching type and a matching mode.
2. A moded type constraint *mode1* *type1* is more specific than moded type constraint *mode2* *type2* if *mode1* is identical to *mode2* and *type2* includes *type1* or *mode1* is different to *mode2* and *mode2* includes *mode1*.

proc show\_mode%(x:shared num):print("Shared")

proc show\_mode%(x:universal int):print("Universal")

j=1

for i in 1..3 {

show\_mode%(j) // Prints Shared

show\_mode%(i) // Prints Universal

}

Note that the first definition in the above example is strictly more specific than the second, even though the type constraints do not conform in this way. For different mode constraints, mode inclusion takes precedence over type conformance.

Matching communicating procedures to communicating procedure calls may also be affected by the **cond** and **uncond** attributes and type constraints on ***implicit arguments*** as described below.

In an analogous manner to regular procedures, a communicating procedure will specialise to the concrete modes of the supplied arguments (including implicit arguments). It will also specialise according to whether it is called from a conditional or unconditional context.

## Communicating procedure definition attributes.

A communicating procedure definition may have the following attributes, over and above the attributes permitted for all procedure definitions:

|  |  |
| --- | --- |
| **shared** | The procedure body will run as if the procedure was called using a ***shared procedure call***. All parameter constraints must specify a mode, and that mode must conform to **invariant**. |
| **complete** | The procedure body will run as if the procedure was called using a ***complete procedure call***. All parameter constraints must specify a mode, and that mode must conform to **universal**. |
| **always** | This attribute modifies the **shared** or **complete** attribute in the same manner as for a procedure call. |
| **cond** | This procedure definition will only match a call in a ***conditional context*** |
| **uncond** | This procedure will only match a call in an ***unconditional context*** |

## Communicating procedure call attributes

Communicating procedure call attributes are restricted to **inline** and **no\_inline** – **shared**, **complete** and **always** attributes are not permitted (the basic rule is that these attributes can only appear in standard procedure calls and communicating procedure definitions).

## Implicit arguments to communicating procedures

Communicating procedures are automatically passed three additional arguments: **region**, **subregion** and **here**. These three parameters are always accessible within a communicating procedure. However, it is possible to specify them in the parameter list of a procedure definition in order to constrain their types. Type constraints on these parameters take a full part in procedure selection (by default they are unconstrained).

proc block\_comm%(region:shape(,block\_distr),x){ … }

// Only operates on a blocked distribution

These arguments must precede any other arguments and follow the order: **region**, **subregion**, **here**.

# Stencils: the nhd statement

## Overview

Modelling applications frequently require the computation of stencils, functions of a local neighbourhood surrounding a given point. To facilitate the efficient implementation of stencils, PM provides the **nhd** statement. Note that a statement is used instead of, for example, an expression in order to given the system the opportunity to reschedule strands within the body of the statement to perform such optimisations as delaying the computation of values at tile boundaries, appropriately blocking local computations, *etc*. The **nhd** statement has the following syntax:

**nhd** *nbhd* { '**,**' *nbhd* } [ **bounds** *expr* ] [ *attr* ] [ *subexpr* ] [ *block* ]

*nbhd*::=

*name* **of** *expr* { '**,**' *name* **of** *expr* }

In its most basic form, the **nhd** statement computes a neighbourhood value (type **nbhd**) for every point in the current region and then executes its accompanying block. Within that block, neighbourhood values may be sliced or subscripted in the same way as arrays, although in this case subscripts or slices indicate offsets from the current point and must not exceed the specified neighbourhood.

nhd [-1..1,-1..1] dx of x, dy of y {

…

a=dx[1,1] // This is equivalent to a=x[here+[1,1]]

b=dy[-2,-2] // Error – the offset lies outside the extent

// of the neighbourhood

}

Within the block of a **nhd** statement, communicating operations of any kind are not allowed (it is essentially a non-parallel context except that **over** statements are allowed).

It is possible to define different neighbourhoods for different values:

nhd [-1..1,-1..1] dx of x, dy of y

[-2..2,-2..2] da of a, db of b {

…

}

## Orthogonal neighbourhoods

Many stencils do not need the entirety of a rectangular neighbourhood to be retrieved. It is possible to define ‘orthogonal’ stencil neighbourhoods using the **ortho** operator.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | ⊕ |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

ortho [ -3..3, -3..3 ]

While the binary **ortho** operator allows the creation of a combined rectangular/orthogonal neighbourhood

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | ⊕ |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

[-2..2,-2..2] ortho [ -3..3, -3..3 ]

Orthogonal and combined neighbourhoods are used in the **nhd** statement in the usual way:

nhd ortho[-1..1,-1..1]dx of x {

…

jacobi= (x[-1,0]+x[1,0]+x[0,-1]+x[0,1])/4.0

}

## Boundaries

An offset subscript of an **nbhd** value cannot access a point outside of the current parallel region. This is typically achieved through the use of **over** statements to restrict computation to a sub-region for which no neighbourhood overlaps the edge of the region.

for point in [0..M+1,0..N+1] {

var cell = 0.0

over [0,]: cell=1.0 // boundary condition

until invar err<tol {

const diff

nhd ortho [-1..1,-1..1] dcell of cell {

over [1..N,1..M] {

cell= (dcell[-1,0]+dcell[1,0]+dcell[0,-1]+dcell[0,1])/4.0

diff=abs(cell-dcell[0,0])

}

}

err=reduce%($max,diff,0.0)

}

}

Given that the exclusion of boundary points is a common operation, there is a shorthand available to do this.

for point in [0..M+1,0..N+1] {

var cell = 0.0

over [0,]: cell=1.0 // boundary condition

until invar err<tol {

const diff

nhd [-1..1,-1..1] dcell of cell bounds EXCLUDED {

cell= (dcell[-1,0]+dcell[1,0]+dcell[0,-1]+dcell[0,1])/4.0

diff=abs(cell-dcell[0,0])

}

err=reduce%($max,diff,0.0)

}

}

The above code excludes boundary points according to the specified neighbourhood(s). It is also possible to specify the boundary extend directly.

for point in [0..M+1,0..N+1] {

var cell = 0.0

over [0..1,]: cell=1.0 // boundary condition

until invar err<tol {

const diff

nhd [-1..1,-1..1] dcell of cell bounds[-2..2,-2..2] {

cell= (dcell[-1,0]+dcell[1,0]+dcell[0,-1]+dcell[0,1])/4.0

diff=abs(cell-dcell[0,0])

}

err=reduce%($max,diff,0.0)

}

}

It is also possible to specify that one or more dimensions have cyclic boundary conditions. In this case, an off-edge access will wrap around to the other side of the region.

for cell in [1..N,1..M] bounds CYCLE { … }

// Cycle all dimensions

for cell in [0..N+1,1..M] bounds [,CYCLE] { … }

// Cycle only second dimension

Formally, a boundary is specified using a **boundary\_dim** value or a tuple of such values. These specify boundary handling along each dimension (or for all dimensions), as follows:

|  |  |
| --- | --- |
| **null** | No special boundary. Code must avoid accessing points beyond the boundary of the current region via **nbhd** values. |
| **range** | Specified boundary. The body of the **nhd** statement is not run on points with and index which is either **≤-low(range)** or **≥high(region)-high(range)** |
| **EXCLUDED** | As above, but the range is derived from the smallest envelope surrounding all of the **nbhd** values supplied to the **nhd** statement |
| **CYCLE** | Boundary is cyclic. The body of the **nhd** statement is run for all points in the current region. Access via a **nbhd** value beyond the edge of the current region will wrap around. |

# Input/output

## Overview

Input/output (I/O) in PM is complicated by the possibility that the program is running in parallel on distributed hardware. This introduces multiple possibilities for race conditions.

All I/O is conducted by accessing a **filesystem** object to open files. File reads and writes are than performed on the open file and the file is then closed.

The PM main program implicitly defines a variable of type **filesystem** called **filesystem.** This object defines the entirety of the external file system(s) available to the hardware running the program. A **filesystem** value is treated as distributed – it cannot be imported into a parallel context. A **filesystem** value also cannot be assigned.

Opening a file is regarded as modifying the filesystem variable, and for this reason a filesystem must be supplied as a reference argument to the **open** procedure:

file = open(&filesystem,"MyFile")

The following optional arguments control the operation of the open procedure (all take an argument of type **bool** which defaults to **false**):

|  |  |
| --- | --- |
| **read=***expr* | File is read only |
| **write=***expr* | File is write only |
| **create=***expr* | Create the file if it does not exist |
| **exclusive=***expr* | If creating a file, an existing file of that name cannot exist |
| **append=***expr* | Start at the end of the file |
| **temp=***expr* | Delete the file on close |

Similarly, read and write operations are regarded as modifying the file:

read(&file,&x)

write(&file,&y)

Random access is enabled though the pos procedure that returns the current location (in bytes) in the file as a large integer value and seek which moves the file pointer to a given location:

position=pos(file)

seek(&file,position+16)

Both of these operations advance the file pointer in the usual way. The type of value to be input or output is taken from the variable passed to the read or write procedure. Read and write operations are restricted to basic values (integers, reals or boolean values) or arrays of these (which may or may not be distributed).

As with a **filesystem** object, **file** objects are regarded as distributed and may not be imported into parallel statements. They also may not be assigned. All files manipulated in this way will have a binary representation.

## Input/output in a parallel context

The rules defined above do not generally permit the opening of files or reading to and writing from open files in a parallel context (with some exceptions such as ***shared procedure calls***). These actions may be achieved using a set of intrinsic communicating procedures.

The **partition%** procedure partitions a shared filesystem object into distinct sections, one for each strand in the current parallel context:

local\_filesystem=partition%(filesystem).

Each filesystem has its own set of files and cannot affect files in other partitions. How this is implemented in terms of actual file systems is (currently) implementation defined.

The **read%** and **write%** procedures are synchronised versions of read and write. These procedures will execute for each element in the current region, in standard iteration order (first dimension varying most rapidly).

var infile = open(…)

var outfile = open(…)

for i in [1..N,1..M] {

var value=0;

read%(&infile,&value)

process(&value)

write%(&outfile,&value)

}

## Input/output errors

Generally when an I/O operation fails, the PM program exits with an error condition. However, it is possible to obtain and check an error value from the procedure call. Error values are passed as a second result value for **open** and **pos** and as the sole result value for other I/O procedures. Error values have type **io\_error** which embeds **bool**. An embedded **true** value indicates the presence of an error. The error message may be obtained by applying the **string** procedure to an **io\_error** value.

file,err=open(…)

if err: print("Error on open: "++string(err)) else::

err=read(&file,&value))

if err: print("Error on read: "++string(err)) else::

# Syntax summary

*module***::=**

*program\_module* | *library\_module*

*program\_module*::=

[ *declarations* '**;**' ] *statements*

*library\_module*::=

*declarations*

*declarations*::=

{ *import* '**;**' OPT } ( *import* | *declaration* | *teststmt* ) { '**;**'OPT ( *declaration* | *teststmt* ) } [ '**;**' ]

*import*::=

**use** *module\_name* [ '**=>**' *name* ] [ *imports* ]

*module\_name*::=

[ '**.**'] *name* { '**.**' *name* }

*imports*::=

'**{**' *import* { '**;**'OPT *import* } '**}**'

*import*::=

[ **type** | **param** | **proc** ] *namelist*

*declaration*::=

*procdecl* | *typedecl* | *paramdecl* | *teststmt*

*procdecl*::=

**proc** *procname* [ '**%**' ]  *params* [ *attr* ] ['**...**' ] *result*  [*block* ]

**proc** *procname* [ '**%**' ]  *params* [ *returns* ][ *attr* ] ['**...**' ][ **check** *exprlist* [ *whereclause* ] ] *block*

**proc** *procname* [ '**%**' ]  *params* [ *returns* ][ *attr* ] '**...**'

*procname*::=

*name* | *opname* | **null**

*opname*::=

*assnop* | **not** | '**[]**' | '**#**' | **fmt** | '**..**' | **by** | **ortho**  | '**!**'

'**==**' | '**/=**' | '**>**' | '**>=**' | **in** | **inc |** '**/**' | '**\*\***' | **mod**

'**...**' '**\_**' | '**\_**' '**...**' | '**++**' | **xor**

*assnop*::=

'**+**' | '**-**' | '**\***' | **and** | **or**  | '**&**' | '**|**'

*params*::=

'**(**' *pars* [ '**,**' *keypars* ] '**)**'

'**(**' *keypars* '**)**'

*pars*::=

{ *par* '**,**' } ( *par* | **arg** '**...**' )

*par*::=

[ '**&**' ] *name* [ '**:**' [ *mode* ] *type* ]

*keypars*::=

{ *name* [ '**:**' [ *mode* ] *type* ] '**=**' *expr* '**,**' } ( *name* [ '**:**' [ *mode* ] *type* ] '**=**' *expr* | **key** '**...**' )

*returns*::=

'**->**' '**(**' [ *typelist* ] '**)**'

*result* ::=

'**=**' *expr* '{ '**,**' *expr* } [ *subexpr* ]

*namelist*::=

*name* { '**,**' *name* }

*typedecl*::=

**type** *name* [ *type\_params* ] [ **in** *namelist* ] **is** *structrec*

**type** *name* [ **in** *namelist* ] **is unique** ['**{**' *name* '**}**' ]

**type** *name* [ *type\_params* ] [ **in** *namelist* ] **is** [ '**...**' '**,**' ]*typelist* [ '**,**' '**...**' ]

*type\_params*::=

'**(**' *name* [ '**:**' *type* ] { '**,**' *name* [ '**:**' *type* ] } '**)**'

*typelist::=*

*type* { '**,**' *type* }

*opttypelist::=*

[ *type* ]{ '**,**'[ *type* ]}

*type*::=

*typeunion* [ **except** *typeunion* ]

*typeunion*::=

*typeconj* {**or** *typeconj* }

*typeconj*::=

*typeinc* { **and** *typeinc* }

*typeinc*::=

*typeunary* [ **inc** *typeunary* ]

*typeunary*::=

**inc** *typeunary*

**fix** *typeunary*

'**\***' *typeunary*

'**.**' *typeunary*

*typeval*

*typeval*::=

*mname* [ '**(**' *opttypelist* '**)**' ]

**any**

**null**

'**­\_**'

'**(**' *type* '**)**'

'**<**' *type* '**>**'

'**[**' *typelist* '**]**'

**proc** [ '**(**' *procarg* { '**,**' *procarg* } [ '**…**' ]'**)**' [ '-**>**' '**(**' *typelist* '**)**' ] ]

*ctval*::=

'**'**' ( *number* | **true** | **false** )

'**$**' *procname*

*procarg*::=

[ '**&**' ] [ *mode* ] *type*

*structrec*::=

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

*mode*::=

**shared** | **uniform** | **chan** | **coherent** | **partial**

**complete** | **invar** | **priv** | **universal** | **local**

*paramdecl*::=

**param** *name* '**=**' *xexpr*

*block*::=

'**:**' *statement*

'**::**' *statements*

'**{**' *statements* '**}**' [ '**--**' ( *name* | *matching\_keyword* ) ]

*matching\_keyword*::=

**if** | **while** | **until** | **switch** | **foreach** | **do** | **any** | **test** | **for** | **forall** | **par** | **nhd** | **over**

*statements***::=**

[ *statement* { '**;**'OPT *statement* } [ '**;**' ] ]

*statement***::=**

**if** [ **invar** ] *xexpr block* { **elseif** *xexpr block* } [ **else** *block* ]

**switch**  [ **invar** ] [*xexpr* ] '**{**'{ **case** *xexprlist* '**:**'*statements* } [ **default** '**:**' *statements* ] '**}**'

[*name* '**>**' ] **while** [ **invar** ] *xexpr**block*

[*name* '**>**' ] **until** [ **invar** ] *xexpr**block*

[*name* '**>**' ] **foreach** [ **invar** ] *iter subexpr* [( **while** | **until** )*xexpr*]*block*

[*name* '**>**' ] **do** *block*

**any** *name* ['**=**' *xexpr* ] *block*

( **for** | **forall** ) *iter* [ *attr* ] [ *subexpr* ] *block*

**par** [ *attr* ]'**{**' *statements* { **task** *name* [ *attr* ] '**:**'*statements* } '**}**'

**over** *xexpr* [ *attr* ] *block*

**nhd** *nbhd* { '**,**' *nbhd* } [ **bounds** *expr* ] [ *attr* ] [ *subexpr* ] [ *block* ]

( *definition* | *assignment*) { '**,**' ( *definition* | *assignment* ) } [ *subexpr* ]

( **var** | **const** ) *namelist* [ '**:**' *type* ]

[ **coherent** | **shared** ] *call* [ *subexpr* ]

[ **coherent** | **uniform** | **shared** | **chan** ] *definition*

*name block*

**sync** *ref* [ *assnop* | *ref* '**{**' *name* '**}**' ] '**=**' *expr* [ *subexpr* ]

**proceed**

**exit**

*teststmt*

*teststmt*::=

**test** ( *testexpr* *block* | *testexpr* | *block* )

*testexpr*::=

[ *string*'**=>**' ] *expr* { '**,**' [ *string*'**=>**' ] *expr* }

*iter*::=

[ '**\***' ] *name* **in** *expr* {'**,**' [ '**\***' ] *name*  **in** *expr* }

*nbhd*::=

( *tuple* [ **ortho** tuple] | **ortho** tuple | '**(**' expr '**)**' ) *name* **of** *expr* { '**,**' *name* **of** *expr* }

*definition::=*

( **var** | **const** ) *lname* [ '**:**' *type* ] '**=**' *expr*

( **var** | **const** ) *lname* { '**,**' *lname* } [ '**:**' *type* ] '**=**' *mrhs*

*lname*::=

*name* |'**\_**'

*mrhs*::=

*call* | *name* | *number* | *string*

*assignment::=*

*ref*  [ *assnop* | *ref* '**{**' *name* '**}**' ] '**=**' *expr*

*lhs* '**,**' *lhs* { '**,**' *lhs* } [ *assnop* | *ref* '**{**' *name* '**}**' ] '**=**' *mrhs*

*lhs*::=

*ref* |'**\_**'

*ref*::=

*name* [ *qualifier* ]

*qualifier*::=

[ '**@**' ] { '**.**' *name* | '**.1**' | '**.2**' | '**.3**' | '**.4**'| '**.5**'| '**.6**'| '**.7**'| [ '**.**' ] *slice* }

*xexprlist::=*

*exprlist subexp*

*exprlist ::=*

*expr* { '**,**' *expr* }

*sexprlist ::=*

[ *expr* ]{ '**,**' [ *expr* ] }

*xexpr*::=

*expr subexpr*

*subexpr*::=

[ **check** *testexpr* ] { *whereclause* }

*whereclause*::=

{ **where** *wdef* { '**,**' *wdef* } }

*wdef*::=

*name*  '**=**' *expr*

{ *lname* '**,**' } *name* { '**,**' *lname* } '**=**' *call*

*expr*::=

*lowest to highest precedence*

*expr* **'++'** *expr*

*expr* **fmt** *expr*

*expr* **ortho** *expr*

*expr* **except** *expr*

*expr* **or** *expr*

*expr* **and** *expr*

**not** *expr*

*expr* [ '**==**' | '**/=**' | '**>**'| '**<**' | '**>=**' | '**<=**' | **in** | **inc** ] *expr*

*expr* **by** *expr* | **by** *expr*

*expr* '**..**' *expr* | '**...**' *expr* | *expr* '**...**'

*expr* [ '**+**' | '**-**' ] *expr*

*expr* **mod** *expr*

*expr* '**|**' *expr*

*expr* **xor** *expr*

*expr* '**&**' *expr*

*expr* **shift** *expr*

*expr* **as** *expr*

*expr* '**#**' *expr*

[ '**+**' | '**-**' ] *expr*

*expr* [ '**\***' | '**/**' ] *expr*

*expr* '**\*\***' *expr*

'**#**' *expr*

*term* [ *qualifier* ]

*term*::=

*mname*

*literal*

'**(**'*expr* '**)**'

'**[**' *sexprlist* '**]**'

*call*

*array*

**new** *name* '**{**' *name* '**=**'*expr* { '**,**' *name* '**=**'*expr* } '**}**'

**if** '**(**' *expr* '**=>**'*expr* { '**,**' *expr* '**=>**' *expr* } '**,**' *expr* '**)**'

**switch** '**(**' *expr* '**,**' *expr* '**=>**'*expr* { '**,**' *expr* '**=>**' *expr* } '**,**' *expr* '**)**'

*literal*::=

**true**

**false**

**null**

*ctconst*

*number*

*string*

*call*::=

*mname* [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

*term* '**.**' [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

'**$**' *procname* [ '**%**' ] '**(**' *arglist* [ *attr* ]'**)**'

*arglist*::=

{ *arg* '**,**' } ( *arg* | **arg** '**...**') [ '**,**' *keyargs* ]

[ *keyargs* ]

*arg*::=

'**&**'*ref* | *expr*

*keyargs::=*

*name* '**=**' *expr* { '**,**' *name* '**=**' *expr* }

*slice* ::=

'**[**' *sexprlist* '**]**'

*array*::=

'**(**' *list2d* '**)**'

'**{**' *list2d* '**}**'

'**{**' *expr* '**:**' *iter* [ *attr* ]'**}**'

*list2d*::=

*exprlist* { '**;**' *exprlist* } [ '**;**' ]

*mname*::=

*name* [ '**'**' *name* ]

attr::=

{ '**<<**' atrb { '**,**' atrb } '**>>**' }

atrb::=

*name*

*name* '**=**' *expr*

*name* '**(**' *exprlist* '**)**'

# Intrinsic procedures and types

## Abstract numerical types

More general categories of numerical values are defined using the following types:

|  |  |  |
| --- | --- | --- |
| **any\_int** | **sint, int, lint, int8, int16, int32, int64** |  |
| **any\_real** | **sreal, real** |  |
| **any\_cpx** | **scpx, cpx** |  |
| **int\_num** | **any\_int** |  |
| **real\_num** | **any\_int, any\_real** |  |
| **cpx\_num** | **any\_int, any\_real, any\_cpx** |  |
| **num** | **cpx\_num** |  |

## Numeric type balancing

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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **sint** | **int** | **lint** | **int8** | **int16** | **int32** | **int64** | **sreal** | **real** | **scpx** | **cpx** |
| sint | sint | int | lint | int8 | int16 | int32 | int64 | sreal | real | scpx | cpx |
| int | int | int | lint | int8 | int16 | int32 | int64 | sreal | real | scpx | cpx |
| lint | lint | lint | lint | int8 | int16 | int32 | int64 | sreal | real | scpx | cpx |
| int8 | int8 | int8 | int8 | int8 | int16 | int32 | int64 | sreal | real | scpx | cpx |
| int16 | int16 | int16 | int16 | int16 | int16 | int32 | int64 | sreal | real | scpx | cpx |
| int32 | int32 | int32 | int32 | int32 | int32 | int32 | int64 | sreal | real | scpx | cpx |
| int64 | int64 | int64 | int64 | int64 | int64 | int64 | int64 | sreal | real | scpx | cpx |
| sreal | sreal | sreal | sreal | sreal | sreal | sreal | sreal | sreal | real | scpx | cpx |
| real | real | real | real | real | real | real | real | real | real | scpx | cpx |
| scpx | scpx | scpx | scpx | scpx | scpx | scpx | scpx | scpx | cpx | scpx | cpx |
| cpx | cpx | cpx | cpx | cpx | cpx | cpx | cpx | cpx | cpx | cpx | cpx |

## Arithmeric operations

|  |  |
| --- | --- |
| -(x:num) | Negate |
| +(x:num,y:num) | Add |
| -(x:num,y:num) | Subtract |
| \*(x:num,y:num) | Multiply |
| /(x:num,y:num) | Divide |
| \*\*(x:num,y:num) | Power |
| mod(x:num,y:num) | Modulo |
| max(x:real\_num,y:real\_num) | Maximum |
| min(x:real\_num,y:real\_num) | Minimum |
| abs(x:num) | Absolute value |
| &(x:any\_int,y:any\_int) | Bitwise and |
| |(x:any\_int,y:any\_int) | Bitwise or |
| xor(x:any\_int,y:any\_int) | Bitwise exclusive or |
| shift(x:any\_int,y:any\_int) | Bitwise shift |
| acos(x:num) | Arc cosine |
| asin(x:num) | Arc sine |
| atan(x:num) | Arc tangent |
| atan2(x:num,y:num) | Arc tangent of x/y |
| cos(x:num) | Cosine |
| cosh(x:num) | Hyperbolic cosine |
| exp(x:num) | Exponential |
| log(x:num) | Natural logarithm |
| log10(x:any\_real) | Logarithm base 10 |
| sin(x:num) | Sine |
| sinh(x:num) | Hyperbolic sine |
| sqrt(x:num) | Square root |
| tan(x:num) | Tangent |
| tanh(x:num) | Hyperbolic tangent |
| floor(x:any\_real) | Nearest integer ≤x as real of same type as x |
| ceil(x:any\_real) | Nearest integer ≥x as real of same type as x |
| re(x:std\_cpx) | Real component |
| im(x:std\_cpx) | Imaginary component |

1. The result type of these procedures is determined using ***numerical type balancing rules***.
2. The result of arithmetic overflow or underflow, division by zero or modulo zero is not defined by the language standard.
3. The modulo operation gives **mod(***a*,*p***)=***a-floor(a/p)\*p*

## Numerical comparisons

|  |  |
| --- | --- |
| >(x:real\_num,y:real\_num) | Greater than |
| >=(x:real\_num,y:real\_num) | Greater than or equal to |
| ==(x:real\_num,y:real\_num) | Equal to |
| /=(x:real\_num,y:real\_num) | Not equal to |

The result type of these procedures is **bool**.

## Numerical conversions

|  |  |
| --- | --- |
| int(x:num) | long(x:num) |
| int8(x:num) | int16(x:num) |
| int32(x:num) | int64(x:num) |
| sreal(x:num) | real(x:num) |
| scpx(x:num) | cpx(x:num) |

1. These procedures convert a numerical value to the type indicated by the name.
2. If a complex value is converted to a non-complex value then the real part is taken.

|  |  |
| --- | --- |
| xx,yy=balance(x:num,y:num) | Numerical balancing |

The balance procedure returns *x* and *y* converted to the type obtained by applying numeric balancing to the types of *x* and *y*

## General comparisons

|  |  |
| --- | --- |
| ==(x,y) check same\_type(x,y) | Equal to |
| /=(x,y) check same\_type(x,y) | Not equal to |

The result type of these procedures is **bool**.

## Logical operations

|  |  |
| --- | --- |
| and(x:bool,y:bool) | Logical and |
| or(x:bool,y:bool) | Logical or |
| not(x:bool) | Logical not |
| except(x:bool,y:bool) | Logical *x* and not y |

The result type of these procedures is **bool**.

## String operations

|  |  |
| --- | --- |
| ++(x,y) | Concatenate string |
| fmt(x,n:any\_int) | Format value as string with of width n |
| fmt(x,n:[any\_int,any\_int]) | Format value as string width n.d1 using n.d2 decimal places |
| string(x) | Convert value to a string |

1. The result type of these procedures is **string**.
2. Arguments to **++** are converted to **string** using the **string()** procedure.

## Array operations

|  |  |
| --- | --- |
| array(x,y:extent) | Spread value x over extent y to create an array value |
| #(x:array) | Shape of array x |
| sum(x:num) | Sum |
| prod(x:num) | Product |
| maxval(x:num) | Maximum value |
| minval(x:num) | Minimum value |
| allof(x:bool) | All values true |
| anyof(x:bool) | At least one value true |
| count(x:bool) | Number of values true |

## Type comparison

## 

|  |  |
| --- | --- |
| same\_type(x,y) | Type comparison |

1. The result type of this procedure is either **'true** or **'false**.
2. The result is **'true** if both arguments have the same concrete type. Otherwise the result is **'false**.

## Ranges and sequences

|  |  |  |
| --- | --- | --- |
| **Procedure** | **Action** | **Result conforms to** |
| ..(x:range\_base,y:range\_base) | Create range | range(t) |
| ...\_(x:range\_base) | Create infinite range below *x* | range\_below(t) |
| \_...(x:range\_base) | Create infinite range up to *x* | range\_above(t) |
| by(x:range) | Create a sequence | strided\_range(t) |
| by(x:seq) | Multiply stride of sequence | strided\_range{t} |
| by(x:range\_below) | Create infinite sequence below *x* | strided\_range\_below{t} |
| by(x:range\_above) | Create infinite sequence up to *x* | strided\_range\_above{t} |

The base type of the result, *t*, is the determined by applying numeric type balancing to the types (or for ranges and sequences to the base types) of the arguments.

|  |  |  |
| --- | --- | --- |
| **Procedure** | **Action** | **Result conforms to** |
| low(x:range\_base,y:range\_base) | Lowest point in range | range\_base |
| high(x:range\_base) | Highest point in range | range\_base |
| in(x:range\_base,y:seq) | Is point in the range? | bool |
| inc(x:range,y:range) | Does one range include another? | bool |
| low(x:range\_base,y:range\_base) | Lowest point in sequence (independent of direction) | range\_base |
| high(x:range\_base) | Highest point in sequence (independent of direction) | range\_base |
| step(x:range\_base) | Step of sequence | range\_base |
| start(x:range) | First point in sequence | range\_base |
| finish(x:seq) | Last point in sequence | range\_base |
| in(x:range\_base,y:seq) | Is point in sequence? | bool |
| inc(x:seq,y:seq) | Does one sequence include another? | bool |
| #(x:any\_seq) | Shape of range or sequence | shape |
| size(x:any\_seq) | Number of elements in sequence | long |

|  |  |
| --- | --- |
| expand(x:any\_seq,y:any\_seq) | Expand sequence *x* by extent of *y* |
| contract(x:any\_seq,y:any\_seq) | Contract sequence *x* by extent of *y* |
| convert(x:any\_seq,y:range\_base) | Convert sequence to have same base type as *y* |

## Distributions

|  |  |
| --- | --- |
| block(shape:tuple,topol:tuple) | Create block distribution |
| vblock(shape:tuple,topol:tuple,block:tuple) | Create a variable block distribution |
| cyclic(shape:tuple,topol:tuple) | Create a cyclic distribution |
| block\_cyclic(shape:tuple,topol:tuple,  block:tuple) | Create a block cyclic distribution |

All arguments must be values with the same number of dimensions

## Processor grouping

|  |  |
| --- | --- |
| sys\_nprc() | Number of processors available to the PM program |
| this\_prc() | Processor rank within set of processors used by current parallel statement |
| this\_nprc() | Number of processors used by current parallel statement |
| shared\_nprc() | Number of processors owned by current task |
| is\_par() | Does current parallel statement use more than one processor? (**bool** result) |
| is\_shared() | Does current task own more than one processor? (**bool** result) |

1. Processor ranks are numbered from zero
2. All result values are long integer unless otherwise stated.

## Communicating procedures

|  |  |
| --- | --- |
| sum%(x:num) | Sum |
| prod%(x:num) | Product |
| maxval%(x:num) | Maximum value |
| minval%(x:num) | Minimum value |
| allof%(x:bool) | All values true |
| anyof%(x:bool) | At least one value true |
| count%(x:bool) | Number of values true |