Parametric Fortran – User Manual

Ben Pflaum and Zhe Fu
School of EECS
Oregon State University

September 14, 2006

1 Introduction

Parametric Fortran adds elements of generic programming to Fortran through a concept of *program templates*. Such a template looks much like an ordinary Fortran program, except that it can be enriched by parameters to represent the varying aspects of data and control structures. Any part of a Fortran program can be parameterized, including statements, expressions, declarations, and subroutines.

The Parametric Fortran compiler takes values for all the parameters used in a template and translates the program template into a plain Fortran program. Therefore, algorithms can be expressed in Parametric Fortran generically by employing parameters to abstract from details of particular implementations. Different instances can then be created automatically by the Parametric Fortran compiler when given the appropriate parameter values.

Parametric Fortran is, in fact, not simply a language extension, but more like a framework to generate customized Fortran extensions for different situations, because the way parameter values affect the code generation is not fixed, but is part of the definition of a particular parameter definition.

1.1 Array addition

The following example shows how to write a Parametric Fortran subroutine to add two arrays of arbitrary dimensions. An experienced Fortran programmer would see that this could easily be accomplished Fortran 90 array syntax. The purpose of this example is not to provide a better way to do array addition, but to show how to use parameter values to generate Fortran programs without the details of the program getting in the way.

```
{dim : subroutine arrayAdd(a, b, c)
  real :: a, b, c
  c = a + b
end subroutine arrayAdd }
```

For simplicity, we suppose that the size of each dimension is 100, because we want to use an integer, which represents the number of dimensions of the arrays, as the parameter. It is not difficult to lift this limitation by extending the parameter type with size information for every dimension. In this example, the program is parameterized by an integer `dim`. The value of `dim` will guide the generation of the Fortran subroutine. The braces `{` and `}` delimit the scope of the `dim` parameter, that is, every Fortran syntactic object in the subroutine is parameterized by `dim`. For `dim` = `SimpleDim 2`, the following Fortran program will be generated.

```
subroutine arrayAdd(a, b, c)
  real , dimension(1:100,1:100) :: a,b,c
  integer :: i1,i2
  do i1 = 1, 100, 1
    do i2 = 1, 100, 1
      c(i2,i1) = a(i2,i1)+b(i2,i1)
    end do
  end do
end subroutine arrayAdd
```

We can observe that in the generated program, `a`, `b`, and `c` are all declared as 2-dimensional arrays. When a variable declaration statement is parameterized by an integer `dim`, the variable will be declared as a `dim`-dimensional array in the generated program. The assignment statement that assigns the sum of `a` and `b` is wrapped by loops over their dimensions, and index variables are added to each array expression. The declarations for these index variables are also generated. This particular behavior of the program generator is determined by the definition of the parameter type for `dim`, which is implemented in Haskell by computer scientists as part of the Parametric Fortran compiler.

2 Installing and using the compiler

To install the Parametric Fortran compiler, follow these steps. Parametric Fortran requires GHC 6.4 or higher.

1. Untar the package.

   ```
   % gunzip -c pfc-2.0.tar.gz | tar xvf -
   ```

2. Go to the `src` directory.

   ```
   % cd pfc-2.0/src/
   ```

3. Run `compile.sh`

   ```
   % ./compile.sh
   ```

The compiler is now located at `pfc-2.0/bin/pfc`.

The Parameteric Fortran compiler can be invoked with a command of the following form:

```
pfc [-O] [-p paramVals.filename] source.pf [output.f]
```

`-O`	simplify expressions containing constants (*optional*)
`-p paramVals.filename`	read parameter values from `paramVals.filename` (*optional,* `paramVals` *is used by default*)
`source.pf`	input Parametric Fortran template filename
`output.f`	output Fortran source filename (*optional*)

3 Parameterization syntax

Parametric Fortran is an extension of Fortran that allows Fortran constructs to be parameterized. The various parameterization constructs and their meanings are listed in Table 1. Braces denote the scope of parameterizations. A parameterization construct must surround a complete Fortran syntactic object. When a parameterization construct begins at one kind of syntactic object, it must also end at the same kind. A parameterization construct can span multiple statements or declarations, but not a combination of both.

Table 1: Parameterization Constructs

`{p : ...}`	every syntactic object inside the braces is parameterized by `p`
`{p(v1,..,vn) : ...}`	only variables `v1`, ..., `vn` are parameterized by `p` inside the braces
`{#p : ...}`	only the outermost syntactic object is parameterized by `p`
`{#p(v1,..,vn) : ...}`	only variables `v1`, ..., `vn` and the outermost syntactic object are parameterized by `p`
`{ ... }`	everything inside the braces is protected from parameterization by an enclosing parameter

In this table, `p` is used as a generic parameter name, it could be either a parameter name or parameter accessor. – parameter accessor has the form – Parameter names and parameter fields are both strings starting with a letter and containing only letters, numbers, and underscores.

The value of each field can be accessed through *accessors*, written as `p.f`, where `p` and `f` represent the parameter name and the field name, respectively. When a field is used to parameterize a syntactic object `e` by `{p.f:e}`, the value of the field is used as a normal parameter. When a field is mentioned in a program without parameterizing anything, its value is used to parameterize an empty syntactic object.

4 Parameter values files

A parameter values file contains a list of one or more parameter definitions. A parameter definition has the form *parameter name* `=` *parameter value*. The syntax for parameter values varies by type, see Section 6. Each parameter name in a file must be unique.

Below is an example of a parameter values file. This example shows all the parameter values used in the examples in Section 6.

```
dim = SimpleDim 2,
rec = { d = SimpleDim 2, x = RVar "x", y = RNum 7 },
t = Cond True,
f = Cond False,
d = Loop [1:10,1:20] 2,
r1 = RCon "x",
r2 = RVar "x",
r3 = RSeq [RVar "x", RVar "y"],
slice = Slice 4 [3,4]
```

Figure 1: Example parameter values file

5 Examples

In this section we will show two examples of Parametric Fortran templates. The first is a generic array slicing subroutine, it shows how to use parameter accessors. The other example shows how to use lists of parameters to eliminate the need for duplicated code.

5.1 Array slicing

Array slicing means to project an n-dimensional array on k dimensions to obtain an $(n-k)$-dimensional array. With different combinations for n and k, we can obtain different versions of array slicing subroutines. In this section, we will show how to define a template in Parametric Fortran for array slicing. All the different versions of array slicing can be generated from the template automatically.

```
subroutine slice(a, slice.inds, b)
  {slice.n: real :: {a}}
  {slice.o: real :: {b}}
  integer :: slice.inds
  {slice.o: b = {slice: a(slice.inds)}}
end subroutine slice
```

Figure 2: `slice.pf`

In this example, `slice` is used as both the subroutine name and as a parameter name. It should be noted that parameter names can be the same as Fortran keywords and variable names with out causing any conflicts. The parameter `slice` contains three accessors, `n`, `o`, and `inds`, which have the following effects. `slice.n` represents the number of dimensions of the input array, `slice.o` represents the number of dimensions of the output array, and `slice.inds` represents the index variables that will be used for the sliced dimensions. For simplicity we suppose that the size of each dimension is 100.

In the subroutine `slice`, `a` is the input n-dimensional array, `a`'s declaration is parameterized by `slice.n`. The variable `b` is the result $(n - k)$-dimensional array and is parameterized by `slice.o`. The field `slice.inds` is used at three places. In the parameter list of the subroutine, `slice.inds` has the effect of inserting the index variables in the parameter list of `slice` as input parameters. In line 4, `slice.inds` is used to declare the type of the index variables to be integers. In line 6, `slice.inds` is used in the right-hand side of the assignment statement where it means that the index variables will be inserted as `a`'s indices. In line 5, `slice.o` parameterizes the assignment statement to add loops. Also, `slice.o` parameterizes the variable `b` to insert index variables. We use `slice` to parameterize the right-hand side of the assignment, instead of a accessor of `slice`, because in this parameterization, for inserting the index variables to the correct positions, both `slice.inds` and `slice.n` are needed. When the values for all the fields of `slice` are provided, one specific array slicing subroutine can be generated. For example, the following value for `slice` describes the generation of a Fortran subroutine that computes the slice on the first and third dimensions of a 4-dimensional array.

```
slice = Slice 4 [1, 3]
```

Figure 3: `slice.param`

A Fortran program can be generated with the following command. The option `-p slice.param` tells the compiler where to find the parameter values file. `slice.pf` is the input Parametric Fortran template and `slice.f` is the filename of the Fortran subroutine to be generated.

```
pfc -p slice.param slice.pf slice.f
```

The following code shows the Fortran subroutine, which is automatically generated by the Parametric Fortran compiler.

```
1  subroutine slice(a, j1, j2, b)
2    real , dimension(1:100,1:100,1:100,1:100) :: a
3    real , dimension(1:100,1:100) :: b
4    integer :: j1, j2
5    integer :: i1,i2
6    do i1 = 1, 100, 1
7      do i2 = 1, 100, 1
8        b(i2,i1) = a(j1,i2,j2,i1)
9      end do
10   end do
11 end subroutine slice
```

Figure 4: `slice.f`

We can observe that in the generated program, `a` is a 4-dimensional array and `b` is a 2-dimensional array. In line 8, the index variables `j1` and `j2` are inserted to the array expression of a at the first and third position, which is specified by `slice.dims`. The assignment statement is wrapped by 2 additional loops because the output array is 2-dimensional.

5.2 Removing duplicated code

In this section we demonstrate how Parametric Fortran can be used to solve a typical problem of *duplicated code*. This example explains a feature of Parametric Fortran called *list parameters*.

In scientific computing, simulation programs are often used to perform computations on some state variables representing the measurements in scientific models. In different models both the number and the meanings of the state variables may be different, which makes writing generic simulation programs very difficult. This problem can be solved using Parametric Fortran by representing the information about the state variables in parameters. Once the parameter values for a particular model are provided, the computation code for all the state variables can be generated automatically.

Similar code fragments in the simulation programs often lead Fortran programmers to duplicate code through "copy and paste", which can easily introduce errors when the copied parts are not adapted properly to the new context. Moreover, when a change is required in one part of the computation, all the copies of the code fragment have to be changed in the same way, which is also prone to errors. With Parametric Fortran, only one code fragment for duplicated code is maintained, which simplifies the program maintenance.

In the following example, we show how to write a simple simulation program in Parametric Fortran to avoid duplicated code.

```
program simulation
  implicit none
  {#stateVars :
    {stateVars.dim : real, allocatable :: stateVars.name}
  }
  {#stateVars:
    allocate(stateVars.name(stateVars.dim.bounds))
    call readData(stateVars.name)
    call runComputation(stateVars.name)
    call writeOut(stateVars.name)
    deallocate(stateVars.name)
  }
end program simulation
```

Figure 5: `simulation.f`

In this simulation program, we have a *list parameter* `stateVars` containing a list of Parametric Fortran parameters of which each contains the information about one single state variable.

```
stateVars = [{dim=SimpleDim 3, name=RVar "temperature"},
             {dim=SimpleDim 2, name=RVar "velocity"}]
```

Figure 6: `simulation.f`

In this simple example, we suppose that every state variable is stored in an array whose number of dimensions is specified by the parameter field `dim`. Another information for a state variable is its name, which can be accessed through the parameter field `name`. The declaration and body part of the simulation program are parameterized separately since they belong to different Fortran syntactic categories. In Parametric Fortran, a parameterization construct can span multiple statements or declarations, but not a combination of both. The parameter value for `stateVars` shown in this example is used for generating the simulation program for a scientific model that has two state variables representing temperature and velocity, and the arrays storing the two variables are 3-dimensional and 2-dimensional, respectively.

The following command will generate a Fortran program based on the information about the state variables contained in `simulation.param`.

```
pfc -p simulation.param simulation.pf simulation.f
```

The following simulation program will be generated for this model. In the generated program, a declaration statement and a code fragment for the computation are generated for both state variables.

```
program simulation
  implicit none
  real , dimension(:,:,:), allocatable :: temperature
  real , dimension(:,:), allocatable :: velocity
  allocate(temperature(1:100, 1:100, 1:100))
  call readData(temperature)
  call runComputation(temperature)
  call writeOut(temperature)
  deallocate (temperature)
  allocate(velocity(1:100, 1:100))
  call readData(velocity)
  call runComputation(velocity)
  call writeOut(velocity)
  deallocate (velocity)
end program simulation
```

Figure 7: `simulation.f`

6 Built–in parameter types

This section describes the parameter types that come with this version of the Parametric Fortran compiler.

6.1 `Cond`

Parametric Fortran code that is parameterized by `Cond True` will remain in the generated Fortran program. Code that is parameterized by `Cond False` will be removed from the generated code. This parameter can be used when code should only exist if some condition is met. For instance, in the IOM convolution program calls to time convolution routines are only made if a variable has a time dimension. Calls to time convolution subroutines are parameterized by a list of all variables, but by using a `Cond` parameter the subroutine call are only created when necessary.

6.1.1 Syntax

cond `::=` `Cond True`
    `|` `Cond False`

6.1.2 Accessors

`not` logical not

Example:
`t` = `Cond True`
`f` = `Cond False`
`t.not` → `Cond False`
`f.not` → `Cond True`

6.2 `IndexDim`

`IndexDim` is an extended dim type that allows for specification of the bounds of each dimension and the manual selection of the first index variable.

6.2.1 Syntax

indexdim `::=` `ILoop [`*bexpr*`:`*bexpr*`,...]` *timepos start*
    `|` `IInside [`*bexpr*`:`*bexpr*`,...]` *timepos start*
bexpr `::=` *integer* `|` *varname*
timepos `::=` the position of the time dimension, `0` if there is no time dimension
start `::=` integer representing the first index to use (if *start* = 3, the first index will be `i3`)

6.2.2 Accessors

none

6.3 `IOMDim`

`IOMDim` type for specifying the dimensionality of arrays, extend for use by the IOM system. Extensions include making the time dimension significant and the introduction of accessors for covariances.

6.3.1 Syntax

iomdim `::=` `Loop` `[`*bexpr*`:`*bexpr*`,...]` *timepos*
`|` `Inside` `[`*bexpr*`:`*bexpr*`,...]` *timepos*
bexpr `::=` *int* `|` *varname*
timepos `::=` the position of the time dimension, `0` if there is no time dimension

6.3.2 Accessors

In the examples we use `d` = `Loop [1:10, 1:20] 2`

`bounds`	sequence of upper and lower bounds Example: `d.bounds` → `RSeq [RBnd (RNum 1, RNum 10), RBnd (RNum 1, RNum 20)]`
`cov`	creates an `IOMDim` with twice the number of dimensions where the shapes of the first half of the bounds are equal to the shapes of the second half. Example: `d.cov` → `Loop [1:10,1:20,1:10,1:20] 0`
`cov_ind`	creates and `IndexDim` that has the same size and shape and which indices start at size+1 Example: `d.cov_ind` → `ILoop [1:10,1:20] 2 3`
`has_space`	`Cond True` when the is at least one space dimension, `Cond False` otherwise Example: `d.has_space` → `Cond True`
`has_time`	`Cond True` when there is a time dimension, `Cond False` otherwise Example: `d.has_time` → `Cond True`
`lower_bounds`	a sequence of the lower bounds Example: `d.lower_bounds` → `RSeq [RNum 1, RNum 1]`
`size`	the number of dimensions Example: `d.size` → `RNum 2`
`space`	an `IOMDim` with the time dimension removed, valid only if `has_space` is true Example: `d.space` → `Loop [1:10] 0`
`strip`	an `IOMDim` of the same size but with all bounds information stripped off Example: `d.strip` → `Loop [:,:]  0`

`time`	an `IOMDim` with only the time dimension, valid only if `has_time` is true Example: `d.time` → `Loop [1:20] 1`
`timepos`	the position of the time dimension Example: `d.timepos` → `RNum 2`
`upper_bounds`	a sequence of the upper bounds Example: `d.upper_bounds` → `RSeq [RNum 10, RNum 20]`

6.4 `Record`

The `Record` types provides a way of grouping related parameter values. This can be useful when using lists of parameters as shown in Section 5.2. A record may contain many fields. A parameter value in a field can be accessed by using its field name as a parameter accessor.

6.4.1 Syntax

record `::=` { *field name* `=` *pval*`,` *field name* `=` *pval*`,` ...`,` *field name* `=` *pval* }
field name `::=` a string
pval `::=` a parameter value

6.4.2 Accessors

field name	returns the parameter value associated with the field name
	Example:
	`rec` = { `d = SimpleDim 2, x = RVar "x", y = RNum 7` }
	`rec.d` → `SimpleDim 2`
	`rec.x` → `RVar "x"`
	`rec.y` → `RNum 7`
	{`rec.d :  y`} `= rec.x * rec.y` → `y(i2,i1) = x * 7`

6.5 `Replace`

The `Replace` type is used to enter constant values and variables into a program. There are several different kinds of `Replace` parameters, `RVar` for variable names, `RCon` for string constants, `RNum` for integer constants, `RSeq` for a sequence of any other kind `Replace` parameter, and `RBnd` for array boundaries.

6.5.1 Syntax

replace `::=` `RVar` *qstr*
 `|` `RCon` *qstr*
 `|` `RNum` *int*
 `|` `RSeq [`*replace*`]`
 `|` `RBnd (`*replace*`,`*replace*`)`
qstr `::=` quoted string

6.5.2 Accessors

`var` turns a `RCon` into an `RVar`, this can be used to get a variable name that matches a string without needing to add another parameter value

Example:
`r1` = `RCon "x"`
`r1.var` → `RVar "x"`
`r1.var = {r1 : }` → `x = "x"`

`right` append to a parameterized variable

Example:
`r2` = `RVar "x"`
`{r2.right : y}` → `yx`

`left` prepend to a parameterized variable

Example:
`r2` = `RVar "x"`
`{r1.left: y}` → `xy`

`size` length of an `RSeq`

Example:
`r3` = `RSeq [RVar "x", RVar "y"]`
`r3.size` → `RNum 2`

6.6 `SimpleDim`

The `SimpleDim` type allows code to be written independent of the number of dimensions of arrays. `SimpleDim` is meant to be used only for examples. Only the number of dimensions is given. The bounds are all assumed to be 1 to 100. For a more advanced dim type see `IOMDim`.

6.6.1 Syntax

simpledim `::=` `SimpleDim` *dims*
dims `::=` an integer value specifying the number of dimensions

6.6.2 Accessors

`bounds` returns and `RSeq` with the bounds of the array.

Example:

`dim` = `SimpleDim 2`

`dim.bounds` = `RSeq [RBnd (RNum 1, RNum 100), RBnd (RNum 1, RNum 100)]`

6.7 `Slice`

A parameter value for generic array slicing.

6.7.1 Syntax

slice `::=` `Slice` *size* `[`*ind*`,` ...`]`
size `::=` integer size of the array to slice
ind `::=` number of an index to fix

6.7.2 Accessors

In the examples we use `s = Slice 4 [3, 4]`

`inds`	A sequence of index variables that set the fixed part of the array to slice Example: `s.inds` → `RSeq [RVar "j3", RVar "j4"]`
`n`	A `SimpleDim` for the array to slice. Example: `s.n` → `SimpleDim 4`
`o`	A `SimpleDim` for the sliced array. Example: `s.o` → `SimpleDim 2`

6.8 `Void`

The `Void` parameter type does not change the input source in any way. Anything that is parameterized by `Void` is essentially not parameterized.

6.8.1 Syntax

void `::=` `Void`

6.8.2 Accessors

none

7 References

More information about Parametric Fortran can be found in the following papers.

Generic Programming in Fortran, Martin Erwig, Zhe Fu and Ben Pflaum. *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*, 130-139, 2006.
`http://web.engr.oregonstate.edu/~erwig/papers/GenericFortran_PEPM06.pdf`

Parametric Fortran - A Program Generator for Customized Generic Fortran Extensions, Martin Erwig and Zhe Fu. *6th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 3057, 209-223, 2004.
`http://web.engr.oregonstate.edu/~erwig/papers/ParametricFortran_PADL04.pdf`