Generic Programming in Fortran

Martin Erwig Zhe Fu Ben Pflaum
School of EECS
Oregon State University
Corvallis, OR 97330, USA
[erwig|fuzh|pflaum]@eecs.oregonstate.edu

ABSTRACT

Parametric Fortran is an extension of Fortran that supports the construction of generic programs by allowing the parameterization of arbitrary Fortran constructs. A parameterized Fortran program can be translated into a regular Fortran program guided by values for the parameters. This paper describes the extensions of Parametric Fortran by two new language features, accessors and list parameters. These features particularly address the code duplication problem, which is a major problem in the context of scientific computing. The described techniques have been successfully employed in a project that implements a generic inverse ocean modeling system.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Fortran, Generic Programming, Scientific Computing, Software Maintenance, Reuse, Program Generation, Haskell

1. INTRODUCTION

Generic programming supports program adaptation through generalization since it allows many algorithms to be implemented independently of the choice of underlying data structures. Programming languages provide constructs to facilitate generic programming to different degrees. The probably best known example is the templates of C++ [17, 28].

Generic programming is particularly useful in scientific computing. For example, ocean scientists regularly develop simulation programs to evaluate ocean models. Since the model programs and the simulation programs usually have to deal with huge data sets (up to terabytes of data), they are often implemented in a way that exploits the given computing resources as efficiently as possible. In particular, the representation of the data in the model programs is highly specialized. If the simulation programs depend on the data structures used in the models, scientists have to rewrite the simulation programs for every ocean model, even though the algorithm behind the simulation programs is the same. Such simulation programs are usually written in Fortran for efficiency reason, for example, the Inverse Ocean Modeling (IOM) system [4, 14] and the Weather Research and Forecasting (WRF) model [23]. Re-implementing the simulation program for every model is very tedious and error-prone, and the programs will be very difficult to maintain.

One approach to solving this problem is to develop a software infrastructure that allows the definition of well-defined interfaces to implement composable and reusable components. This approach is pursued by the Earth System Modeling Framework (ESMF) collaboration [9, 5]. One disadvantage of this approach is that model developers have to re-implement their existing model programs against these newly defined interfaces, which is not trivial because refactoring a collection of Fortran programs (often consisting of hundreds of files and tens of thousands of lines of code) is a time-consuming and error-prone task. Furthermore, when their models want to apply another simulation program with a different interface, they have to rewrite their model programs against the new interface again. Therefore, model developers seem to prefer re-implementing simulation tools specifically targeted for their model. This software engineering problem in scientific computing shows a great opportunity for generic programming. Unfortunately, as a widely used programming language in scientific computing, Fortran lacks the support of generic programming. The only support for generic programming in Fortran 90 is ad-hoc polymorphism, which allows different subroutines to share the same name, but ad-hoc polymorphism is not enough to satisfy the needs of generic programming in scientific computing.

In this paper, we demonstrate an extension of Fortran to support generic programming, which is called *Parametric Fortran* [7]. The basic idea of Parametric Fortran is to parameterize a Fortran program with parameters that represent the varying aspects of data structures and other model-dependent information of the Fortran program. When the values of these parameters are provided, a program generator can translate the parameterized program into a Fortran program that fits the particular model. Scientists who develop the simulation programs can implement their algorithm in Parametric Fortran by using parameters to represent the model-dependent information. When the information about a specific model is given in the form of values for the parameters, a specialized simulation program can be generated for

*This work was supported by the National Science Foundation under the grant ITR/AP-0121542.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

this model. Therefore, developers of the simulation programs only need to implement their algorithm once and can generate different instances for different models automatically.

A first design of Parametric Fortran was presented in [7]. Parametric Fortran has been used for developing the IOM [4, 14] system. The IOM system developed in Parametric Fortran is currently intensively used with the ocean model PEZ at Oregon State University and the National Center for Atmospheric Research, and with the model KDV at Arizona State University. Other ocean modeling groups are currently in the process of adapting the IOM system, such as ROMS (developed at Rutgers and University of Colorado), ADCIRC (developed at Arizona State University and University of North Carolina), and SEOM (also developed at Rutgers). The Fortran source code of the IOM system consists of more than 10 thousand lines. With Parametric Fortran, the developers of the IOM system only need to maintain one copy of their programs, which increases the productivity greatly. When the IOM system needs to be applied to a new ocean model, they only need to provide the parameter values for that ocean model and the program generator will generate the IOM code for that particular ocean model automatically.

During the development of the IOM system in Parametric Fortran it became clear that the initial design of Parametric Fortran was limited in particular with respect to concisely expressing repeated code. In this paper, we present two new features, *accessors* and *list parameters*, which are extremely helpful in practice. In particular, with these two new language elements, Parametric Fortran has a great potential impact on other scientific computing projects. For example, by using Parametric Fortran to implement the model-dependent part, the WRF [23] system will be much easier to maintain. Moreover, using list parameters of Parametric Fortran can save about 30% of code for the WRF project. The Mercator system [1], the Data Assimilation Research Testbed (DART) developed by the NCAR Data Assimilation Initiative (DAI) [24], and the Hybrid Coordinate Ocean Model (HYCOM) system [2] can also utilize Parametric Fortran for being generic with respect to ocean models. The code size of all these systems can be greatly reduced by using list parameters and accessors of Parametric Fortran.

Three different groups of people are concerned with Parametric Fortran. First, scientists who implement generic algorithms in Fortran want to use the genericity provided by Parametric Fortran. Second, computer scientists who provide programming support in the form of new Fortran dialects for scientists by extending the Parametric Fortran compiler with new parameter types. Third, the clients/modelers provide the model information by parameter values and will use the generated Fortran program for their model. Rather than defining one particular extension of Fortran, our approach provides a framework for extending Fortran on a demand basis and in a domain-specific way. Figure 1 illustrates the principal interaction between the different groups of users. In the figure, `simulation.pf` is the generic program developed by scientists in Parametric Fortran, which implements their simulation algorithm. This simulation program needs to work for three different models named `M1`, `M2`, and `M3`. The clients of the simulation program, who are the developers of the models, provide the parameter values representing the information of their model. The program generator, that is, the Parametric Fortran compiler, generates the Fortran program `simulation.f` for each individual model and provides the generated simulation program to the clients. The Parametric Fortran compiler is written in Haskell [25] by computer scientists.

The rest of this paper is organized as follows. In Section 2 we illustrate the basic idea of Parametric Fortran by examples. A new language feature called *accessors* will be presented in Section 3. In Section 4, we will present another new language feature called *list parameters* and show how to use it to avoid duplicate code. In Section 5 we describe the implementation of the Parametric Fortran compiler. Related work is discussed in Section 6. Finally, we present some conclusions and future work in Section 7.

2. OVERVIEW OF PARAMETRIC FORTRAN

In this section, we will demonstrate the use of Parametric Fortran with two examples. We first introduce the syntax of Parametric Fortran in 2.1. A generic summation subroutine will be presented in 2.2. In Section 2.3, we illustrate how to define a parameter type. A generic array-slicing subroutine is described in Section 2.4.

2.1 Syntax

Parametric Fortran is an extension of Fortran by parameterization constructs. We distinguish two different kinds of parameterization annotations. One to propagate parameters into Fortran code and the other to stop the propagation of parameters. Braces denote the scope of parameterizations. The various parameterization constructs and their meanings are listed in Table 1.

Table 1: Parameterization Constructs

Construct	Meaning
`{p : ...}`	everything inside the braces is parameterized by `p`
`{p(v1,..,vn) : ...}`	only variables `v1`, ..., `vn` are parameterized by `p` inside the braces
`{#p : ...}`	the outermost syntactic object is parameterized by `p`
`!{ ... }`	nothing inside the braces is parameterized
`!v`	the variable `v` is not parameterized

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.

2.2 Generic Summation of All Array Elements

The following example shows how to write a Parametric Fortran subroutine to compute the sum of all the elements of an array of real numbers. For simplicity, we suppose that in this example the size of each dimension is fixed to 100.

```
1  subroutine arraySum (a, sum)
2    {dim: real :: a}
3    real :: sum
4    sum = 0.0
5    {#dim:
6      sum = sum + dim:a
7    }
8  end subroutine arraySum
```

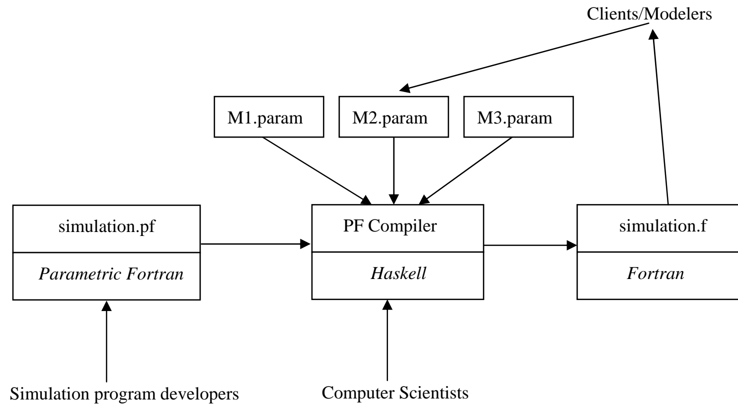

Figure 1: User Groups

An integer parameter `dim` is used in the example. The value of `dim` represents the number of dimensions of the input array and will guide the generation of the Fortran subroutine. The parameter `dim` is used at three places. In line 2, `dim` is used to parameterize the declaration of the input array `a`. This parameterization will generate an array declaration with `dim` dimensions. In line 5, `dim` parameterizes the whole assignment statement (line 6), which will generate `dim` loops over the assignment statement. The symbol `#` in front of `dim` indicates that the parameterization is not propagated into the subparts of the assignment statement. In the assignment statement (line 6), the variable `a` is parameterized by `dim`. In the generated program, this parameterization will cause appropriate array indices to be generated. For `dim = 2`, the following Fortran program will be generated.

```
subroutine arraySum (a, sum)
  integer :: i1, i2
  real, dimension (1:100, 1:100) :: a
  real :: sum
  sum = 0.0
  do i1 = 1, 100
    do i2 = 1, 100
      sum = sum + a(i1, i2)
    end do
  end do
end subroutine arraySum
```

We can observe that in the generated program, `a` is declared as a 2-dimensional array. The assignment statement that assigns `sum` is wrapped by 2 loops over both dimensions of `a`, and index variables are added to the array variable `a`. 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 will be shown in Section 2.3.

2.3 Defining a Parameter Type

Parameter types are represented by Haskell data types. The following shows the data type of the parameter type for `dim`.

```
data Dim = Dim Int
         deriving (Show, Typeable, Data)
```

Parameter types must be instances of type classes `Typeable` and `Data`, because we have to apply the functions `cast` and `everywhere'` [18] to values of these types, which enables a generic traversal of the syntax tree of Parametric Fortran programs.

The abstract syntax of Fortran programs is represented by a collection of Haskell data types that can represent only syntactically correct Fortran programs. Therefore, the syntax correctness of the generated Fortran programs is automatically guaranteed by the type system of Haskell. Below we show part of the data type definitions for Fortran indices, expressions and Fortran statements.

```
data Ind  = Ind [ExprP]
          deriving (Typeable, Data)

data Expr = Con   String
          | Var   VName   IndP
          | Bin   BinOp   ExprP ExprP
          | Unary UnaryOp ExprP
          deriving (Typeable, Data)

data Stmt = Assg ExprP    ExprP
          | Call SubNameP ArgListP
          | FSeq StmtP    StmtP
          deriving (Typeable, Data)
```

The data types `StmtP` and `ExprP` represent parameterized Fortran indices, statements and expressions.

```
data IndP  = forall p . Param p Ind  => I p Ind
data StmtP = forall p . Param p Stmt => S p Stmt
data ExprP = forall p . Param p Expr => E p Expr
```

A syntactic object may contain parameterized sub-objects, for example, a Fortran sequential statement, which is of type `Stmt`, may contain parameterized Fortran statements, which are of type `StmtP`.

For example, the data type `ExprP` can hold a parameter value and a Fortran expression. By using the `forall` quantifier in the data type definition we allow the parameter type `p` to be any type that can parameterize a Fortran expression. By defining a similar type for every Fortran syntactic category, any part of a Fortran program can be parameterized. A parameter type is represented by a Haskell data type. The program generation is controlled by the parameter values.

The multiple parameter type class `Param` defines a relation between parameter types and data types of Fortran syntactic categories. In the type class definition, `p` represents the parameter type, and `s` represents the syntactic category. The

generator function gen takes a parameter value and a Fortran syntactic object as input and generates a non-parameterized syntactic object.

```
class Data p => Param p s where
  gen :: p -> s -> s
  gen _ = id            -- default implementation
```

When a parameter type is provided, its instance for the type class Param must be defined for every syntactic category to guide the program generation. A parameter type usually only affects a few Fortran syntactic categories. For the syntactic categories that are not affected by the parameter type, the default implementation as shown above will be used. In this example, the parameter dim, which is represented by the Haskell data type Dim, affects types, statements, and expressions.

```
instance Param Dim Type where
  gen (Dim d) (BaseType bt)  = ArrayT (indx d) bt
  gen p       t              = t
      where indx d = replicate d [(a,b)]

instance Param Dim Stmt where
  gen (Dim d) s | d>0 =
    gen (Dim (d-1)) (For (newVar d) a b (F Void s))
  gen p       s       = s

instance Param Dim Expr where
  gen (Dim d) (Var v (I _ (Ind es))) =
    Var v (ind (es++map (var . newVar) [1..d]))
  gen p       e                       = e

a = i2e 1
b = i2e 100

i2e :: Int -> ExprP
i2e n = E Void (Con (show n))

ind :: [ExprP] -> IndP
ind es = I Void (Ind es)

var :: VName -> ExprP
var v = E Void (Var v (ind []))
```

The definition of the function gen for Dim extends a type by dimensions, adds loops over a Fortran statement, and extends a variable by index variables. The index variables used for extending array expressions and generating loops are generated by the function newVar::Int->VName. The names of these generated index variables are illegal Fortran names. The program generator just marks the places where a new variable is needed. After a program is generated, a freshNames function will go through the whole program and rename every marked place with an unused variable name and add declarations for these variables to the program. Although we could have implemented the generation of fresh variables with a state monad directly, we decided not to do so, because that would have complicated the interface for implementing new parameters.

The data type Void is a parameter type used in those cases in which no parameter is needed. Void can be used to parameterize any syntactic category. In Section 5.2 we will demonstrate how parameterized programs are transformed to programs parameterized by Void. Programs that are parameterized by Void are pretty-printed as plain Fortran programs. The Void parameter uses the default definition for the function gen.

```
data Void = Void
          deriving (Eq,Typeable,Data)

instance Show a => Param Void a where
```

In this definition for the parameter type Dim, we suppose that the size of every array dimension is fixed to 100. It is not difficult to extend the data type definition to allow the array dimensions to have different sizes. Instead of containing only an integer value, a value of the type Dim now contains an integer value and a list of integer pairs, which specify the lower and upper boundaries of all the dimensions.

```
data Dim = Dim Int [(Int, Int)]
         deriving (Show, Typeable, Data)
```

The function gen for the parameter type Dim can be changed as follows. The essential change happens in the instance definition for Fortran types and statements. The instance definition for expressions remains practically unchanged because the sizes of dimensions do not affect array indices.

```
instance Param Dim Type where
  gen (Dim d bs) (BaseType bt)  = ArrayT bs' bt
  gen p       t                 = t
      where bs' = map (\(l,u)->(i2e l, i2e u)) bs

instance Param Dim Stmt where
  gen (Dim d ((l,u):bs)) s | d>0 =
    gen (Dim (d-1) bs)
        (For (newVar d) l' u' (F Void s))
  gen p                  s      = s
      where (l', u') = (i2e l, i2e u)
```

Types parameterized by Dim are expanded by the boundaries of dimensions that are provided in the parameter value. The boundaries are also used for the generated loops over statements.

2.4 Generic Array Slicing

Array slicing means to project an n-dimensional array on the dth dimension to obtain an $(n-1)$-dimensional array. In the subroutine slice, a and b represent the argument and result array, respectively, and k is an index value on the dth dimension of a. In the body of the subroutine, two parameters are used: First, the parameter dim, which represents the number of dimensions of a, parameterizes the declaration of a. Second, the parameter slice is a pair of integers (n,d), representing the fact that the generated code slices the dth dimension of an n-dimensional array. b is parameterized by slice. This parameterization means that in the generated program b has (n-1) dimensions and that array indices are added such that the already existing index expression (in the example: k) will appear at the dth dimension. The parameter slice is not a Fortran variable, but a variable of Parametric Fortran. Therefore, it causes no conflict with the name of the subroutine.

```
subroutine slice(a, k, b)
    {dim:   real :: a}
    {slice: real :: b}
    integer :: k
    {slice: b = a(k)}
end subroutine slice
```

For dim=3 and slice=(3,2), the generated Fortran subroutine computes the kth slice on the second dimension of a 3-dimensional array. The same as in the first example, we assume for simplicity that the size of each dimension is 100.

```
subroutine slice(a, k, b)
    integer :: i1, i2
    real, dimension (1:100, 1:100, 1:100) :: a
    real, dimension (1:100, 1:100) :: b
    integer :: k
    do i1 = 1, 100
      do i2 = 1, 100
        b(i1, i2) = a(i1, k, i2)
      end do
    end do
end subroutine slice
```

Each parameter, such as `dim` or `slice`, requires a separate definition that explains how parameter values affect the parameterized Fortran constructs. We have seen how to define the parameter type for `Dim`. The following code shows how to define a parameter type for `slice`.

```
data Slice = Slice (Int,Int)
           deriving (Show, Typeable, Data)
```

The data type `Slice` contains a pair of integers. Like `Dim`, the parameter type `Slice` affects types, statements, and expressions. We can define the function `gen` as follows (`a`, `b`, `ind`, and `var` are defined as in the previous example).

```
instance Param Slice Type where
  gen (Slice (n, d)) (BaseType bt)  =
                  ArrayT (indx (n-1)) bt
  gen p             t               = t
      where indx m = replicate m [(a,b)]

instance Param Slice Stmt where
  gen (Slice (n, d)) s | n>1 =
    gen (Slice (n-1, d))
        (For (newVar (n-1)) a b (F Void s))
  gen    p       s           = s

instance Param Slice Expr where
  gen (Slice (n, d)) (Var v (I _ (Ind es))) =
    Var v (ind (map (var . newVar) [1..d-1]
              ++ es
              ++ map (var . newVar) [d..n-1]))
  gen p              e                       = e
```

If the parameter value is `Slice (n, d)`, the function `gen` adds `n-1` dimensions to a type, adds `n-1` loops over a statement, and insert `n-1` index variables to an array expression in the way that the existing index is at the `d`th dimension.

3. ACCESSORS

We have shown a generic array-slicing subroutine in Section 2.4. Requests from scientists for an even more general array slicing method led to the development of the subroutine presented in this section. This example demonstrates the new concept of *accessors* in Parametric Fortran.

The following code generalizes the `slice` subroutine from the previous section to enable array slicing on k dimensions simultaneously.

```
1  subroutine slice(a, p.inds, b)
2      {p.n: real :: a}
3      {p.o: real :: b}
4      integer :: p.inds
5      {#p.o:
6        {p.o:b} = {p:a(p.inds)}}
7  end subroutine slice
```

This program is parameterized by a parameter `p`. A parameter can be a plain value, such as an integer, or it can be a more complex record structure, which may contain some fields of which each can be used as a parameter. The value of each field can be accessed through *accessors* in the form of `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. In this example, the parameter `p` contains four fields, `n`, `o`, `dims`, and `inds`. `p.n` represents the number of dimensions of the input array, `p.o` represents the number of dimensions of the output array, `p.dims` is a list of numbers representing the dimensions to be sliced on, and `p.inds` represents the index variables on the dimensions. Similar to the previous examples, we assume for simplicity that the size of each dimension is 100. It is not difficult to add this information to the parameter. In the subroutine `slice`, `a` is the input n-dimensional array, `a`'s declaration is parameterized by `p.n`. The variable `b` is the result $(n-k)$-dimensional array and is parameterized by `p.o`. The field `p.inds` is used at three places. In the parameter list of the subroutine, `p.inds` means that the index variables will be in the parameter list of `slice` as input parameters. In line 4, `p.inds` is used to indicate that the index variables are integer variables. In line 6, `p.inds` is used at the right-hand side of the assignment statement which means that the index variables will be inserted as `a`'s indices. In line 5, `p.o` parameterizes the assignment statement to add loops. Also, `p.o` parameterizes the variable `b` to insert index variables. We use `p` to parameterize the right-hand side of the assignment, instead of a field of `p`, because in this parameterization, for inserting the index variables to the correct positions, both `p.dims` and `p.o` are needed. For example, when the value of `p` is provided in the following way, the generated Fortran subroutine will be able to compute the slice on the first and third dimensions of a 4-dimensional array.

```
p = {n=4, o=2, dims=[1,3], inds = [i,j]}
```

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

```
subroutine slice(a, i, j, b)
  integer :: i1, i2
  real, dimension (1:100,1:100,1:100,1:100) :: a
  real, dimension (1:100,1:100) :: b
  integer :: i, j
  do i1 = 1, 100
    do i2 = 1, 100
      b(i1, i2) = a(i, i1, j, i2)
    end do
  end do
end subroutine slice
```

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

The parameter types used by `p` can be defined as follows.

```
data SliceK = SliceK Dim Dim Dims Inds
data Dims   = Dims [Int]
data Inds   = Inds [VarName]
```

Every field of `p` has a parameter type that has already been defined as an instance of the type class `Param`. The field `n` and `o` have the parameter type `Dim`, `dims` has the type `Dims`, and the field `inds` has the type `Inds`, which contains a list of

variable names. Since Dims is not used independently, it just uses the default implementation for gen. The parameter inds only parameterizes empty syntactic objects, defining the gen function for the type Inds is straightforward. For example, we can define the function gen for indices as follows.

```
instance Param Inds Ind where
  gen (Inds vs) EmptyInd = Ind (map var vs)
  gen p         i        = i
```

The parameter p can be used to parameterize an expression to add index variables. We define the function gen as follows.

```
instance Param SliceK Expr where
  gen (SliceK _ o dims _) (Var v (I q i)) =
    Var v (I Void (insertInds o dims (gen q i)))
  gen p                   e               = e

insertInds :: Dim -> Dims -> Ind -> Ind
insertInds (Dim d) (Dims ds) i = ...
```

The function insertInds inserts the d index variables to an array expression in the way that the existing indices are at the positions specified by ds.

Every parameter type that has fields is an instance of the type class AccessClass and must define the member function access. The function access takes a parameter value and a field name as input, the return type is Maybe ParV for dealing with possible errors. When the field name passed to access is not defined, Nothing is returned. Otherwise, the field value is returned. The following code shows the definition for the data type ParV, which is used to wrap a parameter value, and the instance definition for the parameter type SliceK.

```
data ParV  = forall p . (Param p Expr,
                         Param p Stmt,
                         ...
                        )
                        => ParV p

class AccessClass p where
  access :: p -> String -> Maybe ParV
  access p s = Nothing    -- default implementation

instance AccessClass SliceK where
 access (SliceK n _ _ _)    "n"    = Just (ParV n)
 access (SliceK _ o _ _)    "o"    = Just (ParV o)
 access (SliceK _ _ dims _) "dims" = Just (ParV dims)
 access (SliceK _ _ _ inds) "inds" = Just (ParV inds)
 access _                   _      = Nothing
```

The fields n and o are redundant considering we know how many dimensions to slice by the length of the field dims. The following relationship holds.

```
p.n = p.o + length dims
```

Since only parameter names or field names can be used as parameters, but not expressions of parameters, we can remove neither p.n nor p.o to eliminate the redundancy. Furthermore, the above relationship between the field values is not guaranteed. If users provide field values for which the above relation does not hold, the generated program will contain type errors. We are currently investigating a dependent type system for Parametric Fortran to check the constraints that the parameter values should satisfy.

By using accessors, the readability of the Parametric Fortran program can be improved, especially when writing more complex Parametric Fortran programs with many parameters. Accessors alone are not an essential extension to Parametric Fortran– they can always be simulated by expanding the record to a plain collection of individual parameters. However, accessors prove to be very expressive when used together with *list parameters*, which are discussed next.

4. LIST PARAMETERS

In this section we illustrate a typical problem of *duplicated code* [11] in scientific computing applications. This example motivates the introduction of *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 simulation programs very difficult. By using Parametric Fortran, we can write a template of Fortran code fragment and parameterize it by the information about the state variables. Once the parameter value for a particular model is provided, the computation code for all the state variables can be generated automatically.

Apart from the varying number of state variables, similar code fragments in the simulation programs often lead Fortran programmers to duplicate code through "copy and paste", which can easily introduce errors. 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. Programs that contain duplicated code are known to be very difficult to maintain [11]. With Parametric Fortran, more specifically, with the combination of the two new language features of Parametric Fortran, *accessors* and *list parameters*, programmers only need to maintain one code fragment for duplicated code, which simplifies program maintenance.

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

```
program simulation
  {#stateVars:
    {stateVars.dim : real :: stateVars.name}
  }
  {#stateVars:
    {stateVars.dim : allocate(stateVars.name)}
    call readData(stateVars.name)
    call runComputation(stateVars.name)
    call writeOut(stateVars.name)
    deallocate(stateVars.name)
  }
end program
```

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. 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. Again for simplicity, the size of each dimension is fixed to 100. Another information in the parameter 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. If we want to generate the simulation program for a model with two state variables that represent temperature and velocity, and the arrays storing the two variables are 3-dimensional and 2-dimensional, respectively, the value for stateVars can be provided as follows.

```
stateVars = [temp, veloc]
temp      = {dim=3, name="temperature"}
veloc     = {dim=2, name="velocity"}
```

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
  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
```

The following code shows the definition of the parameter type for state variables.

```
data StateVar = SV Dim VarName
              deriving (Show, Typeable, Data)
```

The definition of the function `gen` for the type `[StateVar]` can be automatically derived from the definition of `gen` for `StateVar`. The details will be shown in Section 5.4.

List parameters are very helpful for reducing code size of scientific simulation programs. In the IOM [14] project, after using Parametric Fortran list parameters to remove duplicated code, the code size is reduced by almost 50%. In the WRF [23] project, which has a huge community of users and contains more than 100 thousand lines of code, by using Parametric Fortran, about 30% of code can be saved.

5. THE PARAMETRIC FORTRAN COMPILER

In this section, we describe the basic structure of the Parametric Fortran compiler. The Parametric Fortran compiler transforms the program through the following steps. First, the source program is parsed into an abstract syntax tree in which some nodes are parameterized by parameter names. Parameter values are read in from a file and placed in a list of name-value pairs. Next, parameter names in the abstract syntax tree are replaced with their values. Then a new syntax tree, in which no node is parameterized, is generated according to the values of the parameters. The effect of parameter values on the syntax tree is determined by Haskell definitions of syntax-tree transformations. Finally, the new Fortran program represented by the generated syntax tree is written to a file.

For the implementation of the frontend of Parametric Fortran we have used the Haskell scanner generator Alex [6] and the parser generator Happy [22]. The main part of the Parametric Fortran compiler is written in Haskell [25] using the type-directed recursion library by Lämmel and Peyton Jones [18] that is part of the GHC compiler [12].

5.1 Parameter Substitution

The Parametric Fortran parser produces an abstract syntax tree in which some nodes are parameterized by parameter names. We need to replace these parameter names with their values to apply the program generation. The name-value pairs of parameters are kept in a list of type `PList`.

```
type PList = [(VarName, ParV)]
```

The function `subst` performs a traversal on the abstract syntax tree returned by the parser; it finds syntactic objects parameterized by names and replaces the names with parameter values. The function `subst` is implemented using the `everywhere'` combinator [18] that applies a generic transformation function of type `forall a.a->a` to every node in a tree in a top-down manner. The generic transformation function is built with `extT`, which combines non-generic transformation functions into a generic one.

```
subst :: Data a => PList -> a -> a
subst param = everywhere' (id           `extT`
                           substS param `extT`
                           substE param `extT`
                           ...)
```

In the definition of the generic `subst` function, we need individual substitution functions for every syntactic category. For example, the function `substE` substitutes the parameter names with values for Fortran expressions. The function `substE` takes a list of name-value pairs and a parameterized expression as input. If that expression is parameterized by a name, it returns the expression parameterized by a value which is found in the list of name-value pairs.

We use the function `getName` to get the name of a parameter. Then we look up the value of that parameter in the name-value list and replace the parameter name with its value. The lookup is needed only when the parameter itself is a name. If it is not, for example, if the parameter is `Void`, `getName` just returns an empty string and `substE` leaves the parameter unchanged. The function `getName` is defined as follows.

```
getName :: Typeable a => a->VarName
getName = id `extQ` (\p->VarName "")

substE :: PList -> ExprP -> ExprP
substE param (E p e) =
  case getName p of
    VarName "" -> E p e
    v          -> case lookup v param of
      Just (ParV q) -> E q e
      Nothing       -> parameterNotDefinedError p
```

5.2 Code Generation

For each Fortran syntactic category we define a transformation function which takes a syntactic object parameterized by a parameter value and returns the generated Fortran object parameterized by `Void`. For example, the following code shows the transformation function for Fortran expressions and Fortran statements.

```
transE :: ExprP -> ExprP
transE (E p e) = E Void (gen p e)

transS :: StmtP -> StmtP
transS (S p s) = S Void (gen p s)
```

To transform a program we define the function `genF`. Again, we use `everywhere'` to traverse the syntax tree and to apply a transformation function to each node in a top-down manner.

```
genF :: Data g => g -> g
genF = everywhere' (id     `extT`
                    transE `extT`
                    transS `extT` ...)
```

5.3 Implementation of Accessors

A parameter field is used as a normal parameter in a Parametric Fortran program. For program generation, we have to substitute a parameter field by its value. In Section 5.1 we demonstrated how to replace a parameter name with its value. In this Section, we will show how to replace a parameter field `p.f` with its value.

The data type `Accessor` represents a parameter field. The type `VarName` represents the name of the parameter and the type `String` represents the field name. The parameter fields in a syntax tree are replaced by parameter values through a collection of functions.

```
data Accessor = Accessor VarName String
              deriving (Typeable,Data)
```

For example, the function `substE` maps an expression parameterized by a parameter field to an expression parameterized by the value of the parameter field. There are similar functions for each Fortran syntactic category.

```
substE :: PList -> ExprP -> ExprP
substE pList (E p e) =
  case getAccE p of
    Just (Accessor v f) ->
      case lookup v pList of
        Just (ParV q) -> E (AccessorV q v f) e
        Nothing       -> paramNotFoundError v
    Nothing -> ... -- not a parameter field

getAccE :: forall p . Param p Expr =>
           p -> Maybe Accessor
getAccE = (\p->Nothing) `extQ` Just
```

If an expression is parameterized by a parameter field `v.f`, `substE` first finds the value of the parameter `v` in the parameter list, then wraps the parameter value by the data type `AccessorV`. Both the data types `Accessor` and `AccessorV` represent an accessor. The difference is that `Accessor` contains the parameter name and the field name, while `AccessorV` also contains the parameter value. The type `AccessorV` is defined as an instance of the type class `Param`. For example, the following code shows the program generator function for `AccessorV` and Fortran expressions.

```
data AccessorV = forall p. AccessClass p =>
                 AccessorV p VarName String

instance Param AccessorV Expr where
  gen (AccessorV p v f) e =
    case access p f of
      Just (ParV w) -> gen w e
      Nothing       -> invalidFieldError v f
```

5.4 Implementation of List parameters

The value of a *list parameter* is a list of Parametric Fortran parameters. The program generator generates a sequence of syntactic objects when a Fortran syntactic object is parameterized by a list parameter. This is accomplished by making a list of parameters an instance of the type class `Param`. The following code shows the instance definition for the list type and the data type of Fortran statements.

```
instance (Param p Stmt) => Param [p] Stmt where
  gen []     s = NullStmt
  gen (p:ps) s = S p s `FSeq` S ps s
```

However, this definition of the program generator is *not* correct when we use accessors on list parameters. Consider, for example, the case when p is a list parameter whose value is $[p_1, p_2, ..., p_n]$. If we use $p.f$ to parameterize a syntactic object, we actually want to generate a list of syntactic objects which are parameterized by $p_1.f$, $p_2.f$, ..., $p_n.f$, respectively. To accomplish this behavior, we must extend the Parametric Fortran compiler in the following ways.

- Define the data type `RepList` to represents list parameters.
- Write a function, `repParam`, that replaces accessors on a list parameter with accessors on parameter values of the elements of the list.
- Create an instance of the `Param` type class for `RepList` to turn a block of code into a sequence of those blocks.

The type `RepList` is defined as follows.

```
data RepList = forall p . (Data p, AccessClass p)
             => RepList VarName [p]
```

The data type `RepList` is used to store the name and value of a list parameter. The value of a list parameter is a list of parameter values and is captured by the `[p]` in the data type definition. The name of a list parameter has to be remembered to solve the problem caused by the combination of list parameters and accessors.

The generation of a list of syntactic objects parameterized by $p_1.f$, $p_2.f$, ..., $p_n.f$ is performed by the function `repParam`. The function `repParam` replaces an accessor on a list parameter with a specific parameter value in the list. `repParam` takes a parameter value in a list and the name of the list parameter as arguments and does an `everywhere'` traversal on a subtree of a Parametric Fortran program. When `repParam` finds an accessor, it checks if the parameter name in the accessor matches the name of the list parameter. When it does, the accessor is replaced by the accessor whose parameter value is replaced with the specific parameter value.

```
repParam :: Data a => ParV -> VarName -> a -> a
repParam p v = everywhere'
               (id `extT`
               (\q@(AccessorV r w f) ->
                  if v == w then AccessorV p w f
                  else q))
```

The generator function tries to get the value of the parameter field by calling the function `access` and calls generator function for the field value. If the accessor function cannot get a value for the field, an error is reported.

The function `readParam` reads a parameter value from a pair of strings. The first element of the pair represents the parameter name and the second represents the parameter value. The value of a list parameter is transformed into a value of the type `RepList`. For each parameter type, `readParam` calls `readP`, stopping when there is a match and returning a parameter name and value. When there is no match, the compiler reports an error. If the parameter value is a list, we package it inside a `RepList` along with its parameter name.

```
readParam :: (String, String) -> ParV
readParam (name,v) =
  case (readP v :: Maybe [StateVar]) of
    Just x -> ParV (RepList (VarName name) x)
    Nothing -> ...  -- try next parameter type
```

The following code shows how to create an instance of the `Param` type class for `RepList` and Fortran statements.

```
instance Param RepList Stmt where
  gen (RepList v [])     s = NullStmt
  gen (RepList v (p:ps)) s =
    FSeq (S p (repParam (ParV p) v s))
         (S (RepList v ps) s)
```

The program generator creates a sequence of statements, of which each is parameterized by a value from the list. One thing to notice is that the result of the function `gen` for the type `RepList` contains parameters that are not `Void`. All parameters will be `Void` in programs generated by the function `genF` (refer to Section 5.2) because `genF` applies the `gen` function in a top-down manner.

6. RELATED WORK

Different programming languages support generic programming to a different degree and in different ways. C++ supports generic programming through function templates. In Java and C#, generic methods and interface are used for generic programming. Most functional programming languages also have support for generic programming. Haskell [25] supports generic programming through polymorphic functions and type classes. Generic Haskell [19] extends Haskell by allowing the definition of functions by induction on the structure of types. ML [29] uses functors and type signatures in addition to polymorphic functions to achieve generic programming. In the array processing languages APL [15] and J [16, 13] there are built-in mechanisms for computing with variable-dimensional arrays. Generally, generic programming means the ability to define functions that can work with different data types in these programming languages. In Parametric Fortran, the concept of generic programming is extended to being capable of implementing programs working with different *models*, which means that Parametric Fortran supports more complex genericity than most other programming languages because models are more complex entities than types. The type-based genericity supported in other programming languages is fully supported in Parametric Fortran. Furthermore, some interesting forms of genericity can only be supported in Parametric Fortran. For example, in each model, not only the data structures are different, but also the interfaces of subroutines/functions or the number of state variables that need to be processed in the simulation program are different. These differences can be represented by Parametric Fortran parameters easily but are generally not possible in type-based generic programming languages.

For example, the array summation and slicing examples can be expressed in C++. The C++ template library `Boost.MultiArray` provides a class template for declaring multidimensional arrays, using the number of dimensions as a parameter. A generic function could be implemented for slicing a multidimensional array on specific dimensions in C++. This function would be generic because it is parameterized by the array type to enable it to work with array types of different dimensions. However, the code duplication example cannot be easily realized in C++ or other programming languages since the parameterization requires parameters that are not types.

Some work has been done for removing duplicated code from programs. For example, CloRT [20] automatically rewrites duplicated code into programming abstractions. The approach of linked editing is used to manage duplicated Java code in [21].

Parametric Fortran was developed for use in scientific computing. Most scientific computing applications deal with huge data sets. Usually, these data sets are represented by arrays. The data structures of these arrays, such as the number of dimensions, are often different in different models to which the same simulation algorithm will be applied. Some languages, such as APL [15] and J [16, 13] provide built-in operations for computing with variable-dimensional arrays. However, APL and J only provide efficient operations for array processing, but not for efficient numerical computations, which prevents APL and J to be widely used in the scientific computing area.

In the Weather Research and Forecasting (WRF) system [23], generic Fortran programs are needed for communication with different models. A program generator in C has to be written for each model-generic module to be generated. The program generation is controlled by a so-called registry, which contains the model information. The approach is not general and since many modules need to be generated, the maintenance of these program generators is difficult.

Parametric Fortran is essentially a metaprogramming tool [26]. Existing Fortran metaprogramming tools include Foresys [27], whose focus is on the refactoring of existing Fortran code, for example, transforming a Fortran77 program into a Fortran90 program. Sage++ [3] is a tool for building Fortran/C metaprogramming tools. However, to express applications as the ones shown here a user has to write metaprograms in Sage++ for transforming Fortran, which is quite difficult and error prone and probably beyond the capabilities of scientists who otherwise just use Fortran. In contrast, Parametric Fortran allows the users to work mostly in Fortran and express generic parts by parameters; most of the metaprogramming issues are hidden inside the compiler and parameter definitions. Forge [8] is a program generator that transforms discrete equations into Fortran code. With Parametric Fortran we can create complete Fortran programs, whereas Forge is limited to the specification and generation of subroutines that will be part of larger simulation programs.

There has also been a lot of work on the generation of Fortran code for simulations in scientific areas. All of them are concerned with the generation of efficient code for *one particular* scientific model. For example, CTADEL [30] is a Fortran code-generation tool, which is applied to weather forecasting; its focus is on solving weather-forecast models, especially, solving partial differential equations. This and other similar tools do not address the problem of the modularization of scientific models to facilitate generic specifications.

The work reported in [10] is similar to Parametric Fortran in the sense that scientific computations are described in functional way and are then translated into lower-level efficient code. But again, the approach does not take into account model-dependent specifications.

7. CONCLUSIONS AND FUTURE WORK

Parametric Fortran extends Fortran by allowing the parameterization of Fortran code fragments. This approach increases the productivity of Fortran programmers and helps with the maintenance of Fortran programs. We have successfully applied Parametric Fortran in scientific computing to enable ocean scientists to implement model-generic algorithms [14, 7]. Beyond the IOM system, Parametric Fortran also has the potential to be applied to many other scientific computing projects, such as the WRF [23] system, the Mercator system[1], the DART system [24], and the HYCOM system [2].

In this paper we have presented two new features of Parametric Fortran called *accessors* and *list parameters*. By using accessors, we can use parameters as records and access

the field values of parameters. This feature improves the readability of Parametric Fortran programs and helps to reduce the number of parameters needed in applications. List parameters are an improvement that increases the expressiveness of Parametric Fortran and allows us to solve some practical problems, such as removing duplicated code, that could not be dealt with in the previous version.

Future work includes the development of a dependent type system for Parametric Fortran to allow deducing constraints for parameter values from Parametric Fortran programs. The constraints will be provided to the Parametric Fortran programmers. If the parameter values satisfy all the constraints, the generated Fortran programs will be guaranteed to be free of type errors.

8. REFERENCES

[1] Mercator Ocean System. `http://www.mercator-ocean.fr/html/mercator/`.

[2] Bleck, R. An oceanic general circulation model framed in hybrid isopycnic-cartesian coordinates. *Ocean Modelling*, 4:55–88, 2002.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, and S. Srinivas. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. In *OON-SKI'94, Second Object-Oriented Numerics Conference*, pages 122–138, April 1994.

[4] B. Chua and A. F. Bennett. An Inverse Ocean Modeling System. In *Ocean Modelling*, volume 3, pages 137–165, 2001.

[5] R. E. Dickenson, S. E. Zebiak, J. L. Anderson, M. L. Blackmon, C. DeLuca, T. F. Hogan, M. Iredell, M. Ji, R. Rood, M. J. Suarez, and K. E. Taylor. How Can We Advance Our Weather and Climate Models as a Community? *Bulletin of the American Meteorological Society*, 83(3):431–434, 2002.

[6] C. Dornan. Alex: A Lex for Haskell Programmers, 1997. `http://haskell.org/libraries/alex.tar.gz`.

[7] M. Erwig and Z. Fu. Parametric Fortran – A Program Generator for Customized Generic Fortran Extensions. In *6th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 3057, pages 209–223, 2004.

[8] M. Erwig and Z. Fu. Software Reuse for Scientific Computing Through Program Generation. *ACM Transactions on Software Engineering and Methodology*, 14(2):168–198, 2005.

[9] R. Ferraro, T. Sato, G. Brasseur, C. DeLuca, and E. Guilyardi. Modeling The Earth System. In *Int. Symp. on Geoscience and Remote Sensing*, 2003.

[10] S. Fitzpatrick, T. J. Harmer, A. Stewart, M. Clint, and J. M. Boyle. The Automated Transformation of Abstract Specifications of Numerical Algorithms into Efficient Array Processor Implementations. *Science of Computer Programming*, 28(1):1–41, 1997.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Canada, 2000.

[12] GHC. The Glasgow Haskell Compiler, 2004. `http://haskell.org/ghc`.

[13] R. K. W. Hui and K. E. Iverson. *J Dictionary.* Jsoftware, 1998.

[14] IOM. Inverse Ocean Modeling System. `http://iom.asu.edu/`.

[15] K. E. Iverson. *Introduction to APL.* APL Press, 1984.

[16] K. E. Iverson. *J Introduction and Dictionary.* Iverson Software Inc., Toronto, Canada, 1995.

[17] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[18] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

[19] A. Löh, D. Clarke, and J. Jeuring. Dependency-style generic haskell. In *8th ACM Int. Conf. on Functional Programming*, pages 141–152. ACM Press, 2003.

[20] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *6th Working Conference on Reverse Engineering*, pages 326–336, 1999.

[21] M. Toomim, A. Begel, S. L. Graham. Managing Duplicated Code with Linked Editing. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 173–180, 2004.

[22] S. Marlow and A. Gill. Happy User Guide, 2000. `http://www.haskell.org/happy/doc/html/happy.html`.

[23] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock. Development of a Next Generation Regional Weather Research and Forecast Model. In *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*, pages 269–276, 2001.

[24] NCAR. DART. `http://www.cgd.ucar.edu/DAI/`.

[25] S. L. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, Cambridge, UK, 2003.

[26] T. Sheard. Accomplishments and Research Challenges in Meta-Programming. In *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pages 2–44, 2001.

[27] Simulog, SA, Guyancourt, France. *FORESYS, FORtran Engineering SYStem, Reference Manual v1.5*, 1996.

[28] A. A. Stepanov and M. Lee. The Standard Template Library. 1994. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project.

[29] J. D. Ullman. *Elements of ML Programming (2nd ed.).* Prentice-Hall International, London, UK, 1998.

[30] R. van Engelen, L. Wolters, and G. Cats. The CTADEL Application Driver for Numerical Weather Forecast Systems. In *15th IMACS World Congress*, volume 4, pages 571–576, 1997.