JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Parametric Fortran: Program Generation in Scientific Computing

Martin Erwig[1,*], Zhe Fu[2], Ben Pflaum[1]

[1] *School of EECS, Oregon State University, Corvallis, OR*
[2] *Microsoft Corp., Redmond, WA*

SUMMARY

Parametric Fortran is an extension of Fortran that supports defining Fortran program templates by allowing the parameterization of arbitrary Fortran constructs. A Fortran program template can be translated into a regular Fortran program guided by values for the parameters. This paper describes the design, implementation, and applications of Parametric Fortran. Parametric Fortran is particularly useful in scientific computing. The applications include defining generic functions, removing duplicated code, and automatic differentiation. The described techniques have been successfully employed in a project that implements a generic inverse ocean modeling system.

KEY WORDS: Automatic Differentiation, Fortran, Generic Programming, Haskell, Ocean Modeling, Program Generation, Scientific Computing, Software Maintenance

1. Introduction

Fortran is widely used in scientific computing for efficiency reasons. For example, scientists usually write simulation programs to evaluate scientific models in Fortran. Two examples are the Inverse Ocean Modeling (IOM) system [1, 2] and the Weather Research and Forecasting (WRF) model [3]. Since these simulation programs 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 simulation programs is highly specialized for each model. Unfortunately, this high degree of specialization causes significant software engineering problems that impact the advance of scientists in evaluating and comparing their models. One particular problem is that the

*Correspondence to: Martin Erwig, School of EECS, Oregon State University, Corvallis 97331, OR, USA
Contract/grant sponsor: National Science Foundation; contract/grant number: ITR/AP OCE-0121542

Received
Revised
Accepted

simulation programs have to be rewritten for every individual scientific model, even though the underlying algorithms are principally the same for all models. 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 [4, 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 modify 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. However, Fortran still lacks suitable supports for generic programming. Fortran 90 offers 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.

We provide a solution to this problem by an extension of Fortran, called *Parametric Fortran*, which allows the creation of generic programs. Instead of writing normal Fortran programs, scientists can write Fortran program templates in which parameters are used to represent the varying aspects of data structures and other model-dependent information of the simulation programs. Any part of a Fortran program can be parameterized, such as statements, expressions, or subroutines. When the model-dependent information is provided in the form of values for these parameters, the Parametric Fortran compiler can translate the program template into a specialized simulation program that fits the particular model. Therefore, developers of the simulation programs only need to implement their algorithm once and can generate different instances for different models automatically.

Parametric Fortran has been used for developing the IOM [1, 2] system. The IOM system 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. By using Parametric Fortran, the developers of the IOM system only need to maintain one version of their code in Parametric Fortran. Fortran code specialized for different models is generated fully automatically, 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 Parametric Fortran compiler will generate the IOM code for that particular ocean model automatically.

Three different groups of people are concerned with Parametric Fortran. First, scientists who implement simulation algorithms in Fortran want to use the genericity provided by

Parametric Fortran. Second, computer scientists or scientists with knowledge of Haskell [6] extend the Parametric Fortran compiler with new parameter types. Third, the clients/modelers provide the model information in the form of parameter values and will use the generated Fortran programs 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 interactions between the different groups of users and the Parametric Fortran system.

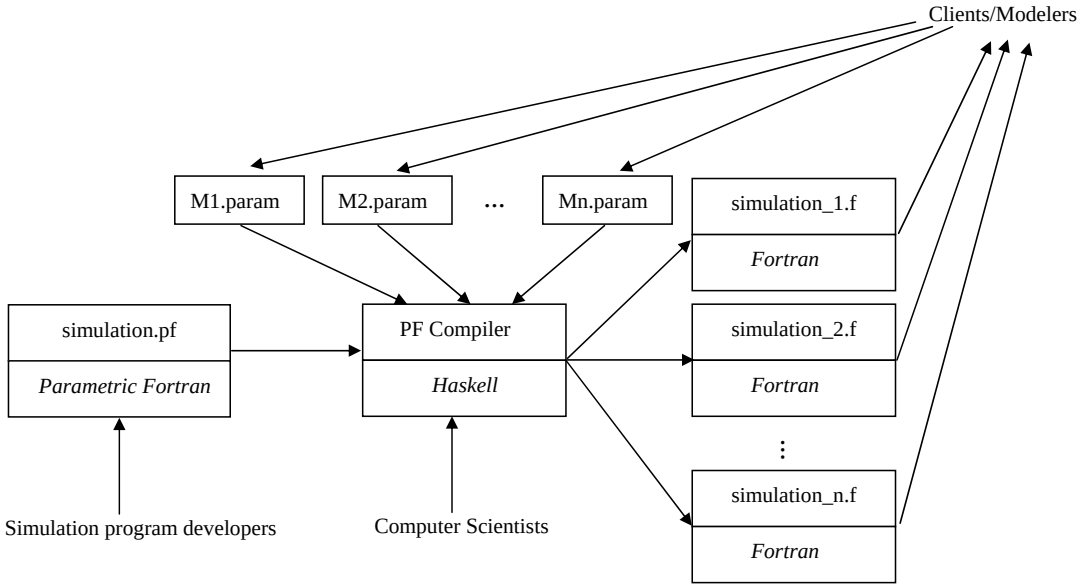

Figure 1. User Groups

In the left box, `simulation.pf` is the simulation program template developed by scientists in Parametric Fortran, which implements their simulation algorithm. This simulation program needs to work for different models named `M1` through `Mn`. The clients of the simulation program, who are the developers of the models, provide the parameter values representing the information of their models. These parameter values can range from simple numbers to whole Fortran program fragments. The program generator, that is, the Parametric Fortran compiler, generates the different Fortran program `simulation_1.f` through `simulation_n.f` for all the models and provides the generated simulation program to the clients. Note that the Parametric Fortran compiler does not automatically create instructions for parallel program execution. Any parallelizing is the responsibility of the user who can supply parallelized Fortran code via parameters. Parallelization can also be implemented through MPI calls parameterized

by Parametric Fortran parameters, such as the number of dimensions. The Parametric Fortran compiler is written in Haskell [6] by computer scientists.

The following table shows how different kinds of users interact with the Parametric Fortran framework. Scientists developing the simulation programs write program templates in Parametric Fortran to implement their algorithms. All the model-dependent information is represented by parameters used in the templates. Model developers who want to run the simulation programs for their model provide their model information in the form of parameter values. They need to know how to provide parameter values in a correct way. Model developers can be provided with a graphical user interface, as in the IOM [1, 2] system to facilitate the input of parameter values. The parameter values are represented as Haskell values so they can be used by the Parametric Fortran compiler directly. If a graphical user interface is not available for model developers, they need to know how to define parameter values in Haskell, which usually happens using a constructor on plain values, such as integers or strings.

The rest of this paper is organized as follows. In Section 2 we illustrate the use of Parametric Fortran by examples. We will describe how Parameter Fortran is used in developing an inverse ocean modeling system in Section 3. In Section 4 we describe another application of Parametric Fortran for automatic differentiation. We discuss related work in Section 5 and present some conclusions in Section 6.

2. Overview of Parametric Fortran

In this section, we will demonstrate the use of Parametric Fortran with examples. We first introduce the syntax of Parametric Fortran in 2.1. A generic array addition subroutine will be presented in 2.2. In Section 2.3 we described how the program generation is realized. A generic array-slicing subroutine is described in Section 2.4 where a feature called *parameter accessors* is introduced. In Section 2.5 we will show how to remove duplicated code using another feature of Parametric Fortran called *list parameters*. We conclude this section by commenting on the expressiveness and scope of Parametric Fortran in Section 2.6.

2.1. 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 I. 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 I. Parameterization Constructs

Syntax	Efffect
`{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`
`!{ ... }`	everything inside the braces is protected from parameterization by an enclosing parameter
`!v`	the variable `v` is not parameterized

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

Figure 2. Dimension-independent array addition in Parametric Fortran

2.2. Array Addition for Arbitrary Dimensions

The example in Figure 2 shows how to write a Parametric Fortran subroutine to add two arrays of arbitrary dimensions.† The dimensionality of the arrays is not specified in the Parametric Fortran code and has to be provided through parameter values.

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` = 2, the Fortran program shown in Figure 3 will be generated.

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

†This example is meant for illustration. Dimension-independent array addition is already supported in Fortran 90 by array syntax.

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

Figure 3. Generated Fortran program

of the parameter type for `dim`, which is implemented in Haskell by computer scientists as part of the Parametric Fortran compiler.

In the following we will explain the principles behind the Parametric Fortran compiler that enable the described program generation.

2.3. Implementation of the Parametric Fortran Compiler

Fortran programs are generated from Parametric Fortran templates. The generation process is performed by the Parametric Fortran compiler whose source language is Parametric Fortran and whose target language is Fortran. The compiler is implemented in Haskell [6]. In this section, we will briefly describe the implementation of the compiler in an abstract way.

Every Parametric Fortran program is represented as a Fortran abstract syntax tree, in which some nodes are annotated with one parameter. Figure 4 shows an example Parametric Fortran syntax tree.

In the syntax tree, every square represents a Fortran syntactic object, and `p1` and `p2` are two parameters. Three nodes are parameterized by `p1`, two nodes are parameterized by `p2`, and three other nodes are not parameterized at all. In a Parametric Fortran syntax tree, every parameter is a name. Program generation is based on the parameter values which are stored in a file. When a template is input to the Parametric Fortran compiler, the compiler will retrieve the values for parameters used in the template and replace the parameter names with their values in the syntax tree. Every parameter has a type. In this example, we suppose that `p1` has the type `T1` and `p2` has the type `T2`. Each parameter type has an associated program generation function *gen* that takes a parameter value of that type and a Fortran syntactic object as input and produces a Fortran syntactic object, which must be in the same syntactic category as the argument. The generation function of a parameter type is implemented in Haskell employing pattern matching for syntax tree transformations. The process is illustrated abstractly by an example in the following. The details of the Haskell implementation are beyond the scope of this paper and can be found in [7, 8].

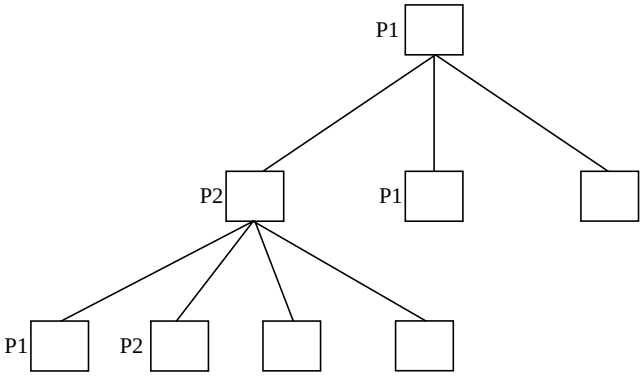

Figure 4. Parametric Fortran syntax tree

The Parametric Fortran compiler takes a Parametric Fortran syntax tree as input, performs a top-down traversal on the syntax tree, and applies the generation function for a particular parameter type to every node parameterized by that type. In this example, the Parametric Fortran compiler traverses the syntax tree and applies the generation function for the parameter type `T1` to every node that is parameterized by `p1`, and applies the generation function for `T2` to every node that is parameterized by `p2`. This type-based generic traversal is implemented using the so-called "Scrap Your Boilerplate" approach introduced in [9], which allows one to only write the code transformation rules for the interesting cases and have the generic traversal parts for the remaining syntactic cases be inferred automatically. Details are given in [8]. The output of the Parametric Fortran compiler will be a Fortran syntax tree in which no node is parameterized. The resulting syntax tree represents the generated program and will be converted to a textual representation by a pretty printer.

We can take the `arrayAdd` program in Section 2.2 as a concrete example. In `arrayAdd`, we use a parameter `dim` containing an integer value. The parameter type for `dim` is defined as a Haskell data type `Dim` as follows.

```
data Dim = Dim Int
```

The generation function for the parameter type `Dim` can be defined by pattern matching for every Fortran syntactic object, such as expressions and statements. For example, when a variable expression is parameterized by a parameter of the type `Dim` with an integer n, the generated expression will be an n-dimensional array expression. The function *gen* can be defined as follows for this case. We use the syntax $gen_p[e]$ to express the application of the *gen* function to the parameter value p and expression e.

$$gen_{\texttt{Dim } n}[v] = v\texttt{(i1, i2,...,i}n\texttt{)} \tag{1}$$

The index variables `i1` through `i`n are new variables generated by the Parametric Fortran compiler. These variables will be only used as array indices and loop variables.

When a `Dim` parameter of value n parameterizes a Fortran statement s, if n is a positive number, the generation function *gen* will generate n loops over array dimensions around the statement; otherwise, the generation function leaves s unchanged. In this example, we suppose that all dimension have the same size `100`. If the sizes of different dimensions are different, we can extend the parameter type with the size information of array dimensions. The definition of the function *gen* is shown below.

$$gen_{\texttt{Dim } n}[s] = \begin{cases} gen_{\texttt{Dim } (n\texttt{-1})}[\texttt{do } i n\texttt{=1,100 } s \texttt{ end do}] & \text{if } n > 0. \\ s & \text{otherwise.} \end{cases} \quad (2)$$

The process of program generation can be illustrated by how the Parametric Fortran compiler transforms the following Parametric Fortran statement into a Fortran statement.

```
{dim: c = a + b}
```

This statement is represented by the syntax tree shown in Figure 5.

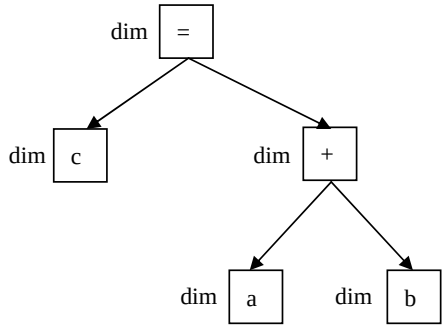

Figure 5. Syntax tree of `{dim: c = a + b}`

Since the parameterization is recursively propagated, every node in the syntax tree is parameterized by the parameter `dim`. When the value for `dim` is provided, for example, `Dim 1`, a Fortran statement can be generated from this parameterized one. The Parametric Fortran compiler traverses the syntax tree top-down and applies the generation function for the parameter type `Dim` to every node. Figure 6 shows the transformation for each single node.

The root node, which is an assignment statement, is transformed into a loop statement. The node `c` is transformed into an array expression `c(i1)`, and the nodes `a` and `b` are also variables and transformed like `c`. The parameterization does not affect the "`+`" node which causes the Parametric Fortran compiler to leave the node `+` unchanged.

After applying the *gen* function for the type `Dim` to all the nodes, the Parametric Fortran compiler outputs the Fortran syntax tree in Figure 7, that represents the following generated Fortran statement. The resulting syntax tree is obtained by composing all the generated nodes.

The syntax tree represents the generated Fortran program shown in Figure 8.

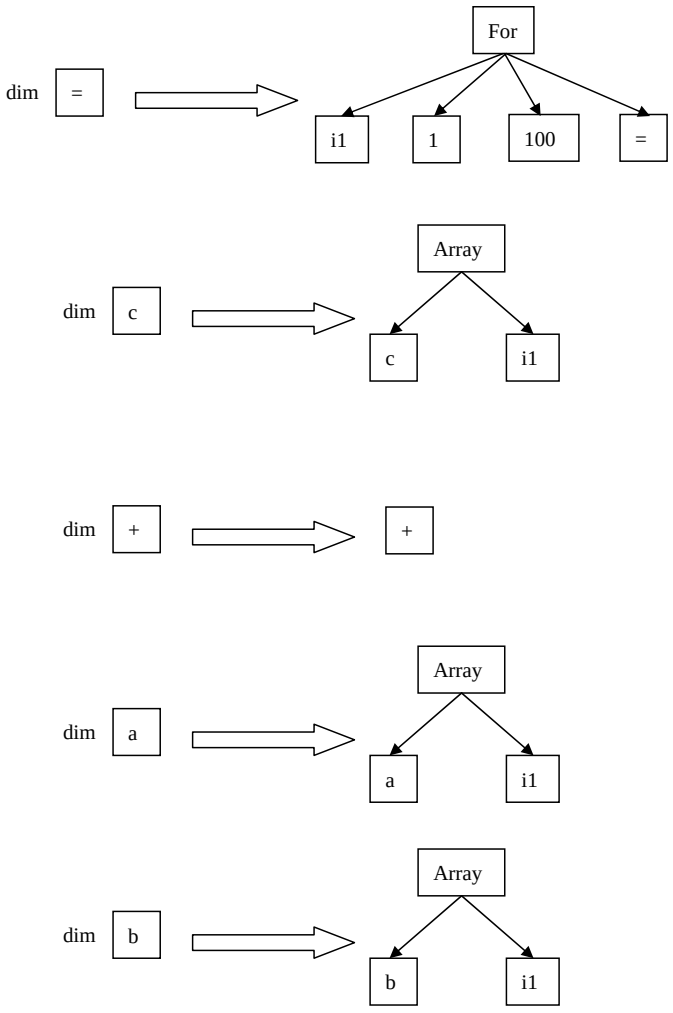

Figure 6. Transformation of nodes

The shown example included only one parameter and one simple form of parameterization. In the following we will illustrate the potential of Parametric Fortran's code generation capabilities with a more sophisticated example.

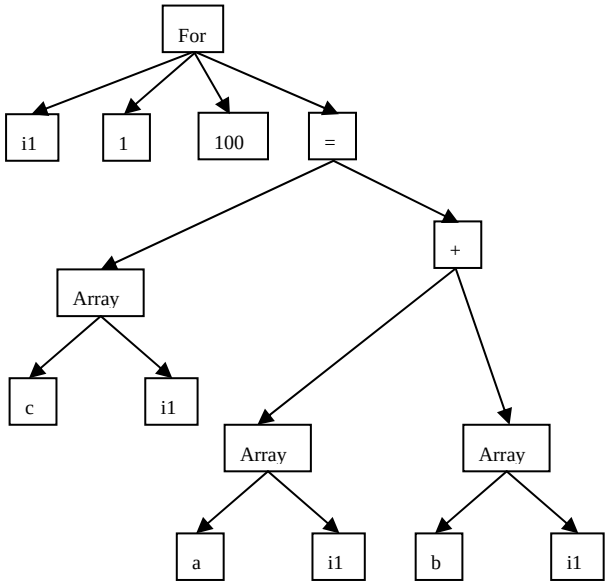

Figure 7. Syntax tree of generated for loop

```
do i1 = 1, 100
  c(i1) = a(i1) + b(i1)
end do
```

Figure 8. Generated Fortran loop

2.4. Generic 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.

In this program template shown in Figure 9, the parameter `p` is not a plain value, but a record structure, which contains several fields of which each can be used as a parameter. 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. In this example, the parameter `p` contains four fields, `n`, `o`, `dims`, and `inds`, which have the following effects. `p.n` represents the number of dimensions of the

```
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
```

Figure 9. Template for array slicing

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 that will be used for the sliced dimensions. Similar to the example in Section 2.2, we assume for simplicity 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 `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` has the effect of inserting the index variables in the parameter list of `slice` as input parameters. In line 4, `p.inds` is used to declare the type of the index variables to be integers. In line 6, `p.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. `p.o` is used at two places. 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. When the values for all the fields of `p` are provided, one specific array slicing subroutine can be generated. For example, the following value for `p` describes the generation of a Fortran subroutine that computes 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 code in Figure 10 shows the Fortran subroutine, which is automatically generated by the Parametric Fortran compiler.

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 `i` and `j` are inserted to the array expression of `a` at the first and third position, which is specified by `p.dims`. The assignment statement is wrapped by 2 additional loops because the output array is 2-dimensional.

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

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

Figure 10. Generated array slicing Fortran subroutine

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 the possibility to allow users to input constraints about parameter values. The Parametric Fortran compiler then could check if the parameter values satisfy the constraints before generating programs.

2.5. Avoiding Duplicated Code

In this section we demonstrate how Parametric Fortran can be used to solve a typical problem of *duplicated code* [10] in scientific computing applications. This example motivates the introduction of 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. Programs that contain duplicated code are known to be very difficult to maintain [10]. With Parametric Fortran, only one code fragment for duplicated code is maintained, which simplifies the program maintenance.

In Figure 11, we show how to write a simple simulation program in Parametric Fortran to avoid 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
```

Figure 11. Simulation template

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, as shown in Figure 12.

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

Figure 12. State variables

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 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 simulation program shown in Figure 13 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.

List parameters are very helpful for reducing code size of scientific simulation programs. In the IOM project (see next section), after using list parameters to remove duplicated code, the code size could be reduced by almost 50%.

```
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
```

Figure 13. Generated simulation program

2.6. Summary and Scope of Parametric Fortran

The design of Parametric Fortran essentially consists of two parts. The first is an extension of the Fortran syntax to annotate arbitrary parts of Fortran programs by parameters, which are simply placeholders for values to be provided later. The second part is a mechanism to define the meaning of these parameter annotations. The effect that values provided for parameters have on an annotated Fortran program is defined with rewriting rules on syntax trees given in Haskell. The details of this aspect are given elsewhere [7, 8].

As we have illustrated, the ability to annotate different parts of a Fortran program with different parameters that can be defined to have arbitrary effects provides great flexibility. In fact, Parametric Fortran does not just provide a fixed set of extensions for Fortran, but rather provides a framework for defining arbitrary program generation extensions to Fortran. The shown parameter types and their effects are not built into the Parametric Fortran compiler, but are extensions built using the framework.

This generality has both advantages and disadvantages. A major benefit is the ability to implement any program generation/manipulation through a suitable encoding as a parameter type. The use of Haskell [6] as a metalanguage to formulate syntax transformations principally offers Turing-complete expressiveness, although this full generality will probably not be used very often. A downside of this expressiveness is that static analyses concerning, for example, the consistency of parameter values are principally limited. For example, constraints among parameter values like the ones for the array slicing example cannot be detected, in general, automatically. These observations mean a responsibility for the designer of Parametric Fortran parameters to define parameters in a way to balance expressiveness against ease of use and proneness to errors.

3. Inverse Ocean Modeling

Ocean models have been invented to simulate and to predict the state of oceans. Most ocean models are typically represented by equations of motion. The equations need to be solved by numerical approximation. For efficiency reasons, ocean models are usually implemented in Fortran. Model developers use data assimilation systems, such as the Inverse Ocean Modeling (IOM) system [1, 2], to combine their models with real observations of the ocean. The output of the IOM is a weighted least-squared best-fit to the equations of motion and to the data. The IOM can provide important information about the quality of data produced by ocean and weather forecasting models. The accuracy of forecasting models is important for the successful planning of flight or ship routes, navy operations, and many other applications.

The IOM consists of tools that are used for computing the equations of best fit from ocean models. Models usually use arrays over time and space to store the state values of oceans, such as velocity or temperature. Different ocean models may use different data structures to describe the ocean, for example, different dimensional arrays, which makes writing the IOM system very difficult. The problem is that the genericity that is inherent in this problem cannot be expressed succinctly in Fortran and requires low-level work-arounds. In this section we will illustrate how different components used in the IOM system can be implemented in a model-generic way in Parametric Fortran.

3.1. An Example Tool: Markovian Convolution in Time

Convolution is essentially the process of averaging over weighted values. The basic idea is the value of one point is computed by averaging over the weighted values of its neighbors. The weights of the neighbors depend on the distances from the point. The number of different possible convolutions is unlimited since each new weighting functions defines a new convolution. Every convolution has the following form.

$$b(x) = \int_0^X F(x, x')a(x')\mathrm{d}x' \tag{3}$$

$F(x, x')$ is the weighting function, x and x' can range over either time or space from 0 to X, which is an upper boundary of time or space. The field a contains the initial values, and b contains the result values. A similar kind of convolution is also used in image processing to reduce noise in images [11].

The IOM tool described here is *Markovian Convolution in Time*, which is formally defined by the following continuous equation.

$$b(t) = \int_0^T \exp(-|t - t'|/\tau)a(t')\mathrm{d}t' \tag{4}$$

The weighting function is $\exp(-|t - t'|/\tau)$, and the variables t and t' range over time, which means that the convolution is in time. The coefficient τ is the correlation time scale, which is

provided by the ocean modelers when they use the tool. The smaller τ is, the more quickly the values of the old points will be forgotten.

The above formula is a continuous equation. Computer simulations are based on the following corresponding discrete equations that can be derived from the continuous one through techniques developed in [1].

$$h_L = 0 \tag{5}$$

$$\frac{h_n - h_{n-1}}{\Delta t} + \tau^{-1} h_{n-1} = -2\tau^{-1} a_n \tag{6}$$

$$b_U = -(\tau/2) h_U \tag{7}$$

$$\frac{b_{n+1} - b_n}{\Delta t} - \tau^{-1} b_{n+1} = h_n \tag{8}$$

In fact, the shown equations represent a slight generalization of the continuous formula that offers flexibility in how arrays are indexed, for example, starting at 0 or 1. Moreover, the equations also support the parallel execution of the subroutine. With $L = 0$ and $U = T$ we obtain as a particular instance the equations that correspond exactly to the continuous formula.

When implementing these discrete equations in Fortran, a, b, and h are defined as arrays, and L (U) is the lower (upper) boundary on the time dimension of the arrays. The array h is used to hold temporary values. All these arrays are over time and space. In different models, the number of space dimensions may be different, and time may be stored in a different array dimension. Therefore, the IOM has to provide different subroutines for all the possible data structures used in all the models. Moreover, the IOM should also be able to provide tools for any new model that uses data structures in a completely new way.

3.2. Expressing Markovian Convolution in Parametric Fortran

For parameterizing Markovian Convolution, we first have to find the parameters representing the model-dependent information. In this example, the model-dependent information is the number of space dimensions of the arrays and the size (lower bound and upper bound) of each dimension. To simplify the following description, we assume that the time dimension is always the first dimension of the arrays in all the models. In the convolution tool that is actually implemented for the IOM system we also parameterize the position of the time dimension. We can define a Haskell data type `Space` to represent the space dimensions of the arrays as follows. Therefore, we can use a parameter of the type `Space` to parameterize the Markovian convolution.

```
data Space = Space Int [(Bound, Bound)]
```

The type `Bound` is defined as follows.

```
data Bound = BCon  Int | BVar VName
```

The information of space dimensions is parameterized by an integer representing the number of space dimensions, and a list of pairs of array boundaries, which can be either an integer constant of a variable. For example, the parameter value `Space 2 [(BVar "X", BVar "Y"), (BCon 1, BCon 100)]` specifies that the number of dimensions of the arrays in the model is 2, that the first dimension is bounded by variables `X` and `Y`, (where `X` is the lower bound and `Y` is the upper bound), and that the second dimension is bounded by `1` and `100`.

We use a parameter `s` of the type `Space` to parameterize the time convolution subroutine as follows. The body of the subroutine shown in Figure 14 implements the discrete equations in Section 3.1.

```
1  subroutine timeConv {s: (L, U, dt, tau, a, b)}
2    integer :: L, U
3    real :: dt, tau
4    {s: real, dimension (L:U) :: a, b, h}
5    integer :: n
6    {#s(a,b,h):
7      h(L) = 0.0
8      do n = L+1, U
9        h(n) = h(n-1) - dt*(h(n-1)/tau + 2.0*a(n)/tau)
10     end do
11     b(U) = -0.5*h(U)/tau
12     do n = U-1, L, -1
13       b(n) = b(n+1) - dt*(h(n) + b(n+1)/tau)
14     end do
15   }
16 end subroutine timeConv
```

Figure 14. Time convolution template

The parameter `s` is used at 3 places. The meanings of the parameterizations are described below.

- In line 1, the parameter list of the Fortran subroutine is parameterized by `s`. This parameterization tells the program generator to add the new variables used in `s`'s value as dimension boundaries to the parameter list of the generated Fortran subroutine.
- In line 4, the declarations of the array variables is parameterized by `s` to append the space dimensions to the current time dimension of these arrays.
- The body of the subroutine is parameterized by `s` to add loops over space dimensions to the body. The meaning of the symbol `#` is that `s` parameterizes whole body of the subroutine, but not the single statements inside the body. Therefore, in the generated program, the loops over space dimensions are added outside the body, no loops will be generated for each single statement inside the body. Furthermore, this parameterization will add index variables to particular array variables used in the program body. These array variables are specified in parenthesis after the parameters. In this example, the

syntax `{s(a,b,h):...}` expresses that the parameter `s` only parameterizes the variables `a`, `b`, and `h`, or array expressions using these names, such as `h(n)`.

To generate the Fortran subroutine `timeConv` for the model in which the arrays have 1 space dimension and the boundaries of that dimension is `(X:Y)`, we use the following value for `s` of type `Space`.

```
s = Space 1 [(BVar "X", BVar "Y")]
```

The Fortran program shown in Figure 15 can then be generated automatically by the Parametric Fortran compiler.

```
subroutine timeConv (X, Y, L, U, dt, tau, a, b)
  integer :: X, Y
  integer :: L, U
  real    :: dt, tau
  real, dimension (L:U, X:Y) :: a, b, h
  integer :: n
  integer :: i1
  do i1 = X, Y
    h(L,i1) = 0.0
    do n = L+1, U
      h(n,i1) = h(n-1,i1) - dt*(h(n-1,i1)/tau + 2.0*a(n,i1)/tau)
    end do
    b(U,i1) = -0.5*h(U,i1)/tau
    do n = U-1, L, -1
      b(n,i1) = b(n+1,i1) - dt*(h(n,i1) + b(n+1,i1)/tau)
    end do
  end do
end subroutine timeConv
```

Figure 15. Generated Fortran subroutine

The program generator performed the following actions. (1) `X` and `Y` are added to the parameter list of the generated subroutine. (2) The arrays `a`, `b`, and `h` are declared as 2-dimensional arrays with boundaries specified in the parameter value. (3) The body of the subroutine is wrapped by a loop over the space dimension. (4) Every array expression is extended by an additional index variable `i1`. (5) `i1`'s declaration is generated. How a parameter type affects the program generation is defined in Haskell. We have seen the details of defining the behavior of a parameter type in Section 2.3.

3.3. A Program Generating System for IOM

We have implemented a program generating system that can generate IOM instances for different models automatically. The IOM modules, such as the Markovian convolution, are written as Parametric Fortran templates, in which parameters are used to capture aspects that are specific to individual ocean models. Values for these parameters have to be provided by each model for which an IOM instance is to be created.

In Section 3.2 we have seen how an individual tool can be generated when model information is provided. Individual tools will be combined into larger inversion programs. This process is controlled by a graphical user interface that presents ocean modelers with a variety of inversion options, such as the selection of inversion outputs and the choice of different inversion algorithm options. The graphical user interface also controls system configurations, such as Makefile and Fortran compiler options, the values of model parameters, and different execution options which are particularly important for efficient parallel execution on supercomputers. Currently, the GUI supports three different execution options: The IOM module and the model module run in one single executable with either of them being the main module and calling the other, or the IOM module and the model module run as two separate executables that communicate via shared data files. We plan to support more fine-grained parallelism in future versions.

The graphical user interface is implemented in Java and runs together with the Parametric Fortran and Fortran compilers on Windows, Mac OS, and Solaris. A snapshot of the current system is shown in Figure 16.

The first 3 fields in the "Parameters" section of the graphical user interface correspond the parameters that we have seen in Section 3.2. For example, when a user inputs "number of space dimensions" as `1`, "position of the time position" as `1`, and "sizes of space dimensions" as `(X,Y)`, the Parametric Fortran compiler will generate the Fortran program for Markovian convolution shown in Section 3.2. The parameter "number of inner iterates" is used for other convolution tools. Users can also select other options on the graphical user interface to customize the simulation program they want to run. The ability to provide model parameters, to select a particular combination of inversion outputs, and to customize the inversion algorithm offers to ocean modelers a flexible customization of inversion programs.

The use of Parametric Fortran had significant impact on the development and maintenance of the IOM system. Instead of writing different instances of the IOM for all the models, IOM developers only need to keep one copy of Parametric Fortran templates. The Parametric Fortran compiler can generate instances for all the models automatically. Table illustrates the benefits obtained from using Parametric Fortran by showing the code savings achieved in the development of the IOM system. We list 5 models that the IOM is currently applied to, and 5 modules in the IOM that are automatically generated. The numbers are the number of lines of the code for a particular module and a particular model. For example, the main convolution module for the model `PEZ` has 502 lines of code. The rightmost 3 columns show the sums of lines of code of the IOM modules for all the models, the number of lines of the Parametric Fortran templates for the IOM modules, and the code saving, respectively.

In the tools listed in Table II, the Markovian time convolution has been explained in detail in Section 3.1, and the Bell-shaped space convolution is a similar convolution tool. In some ocean models, the IOM generates multiple instances of these two convolution tools to convolve state

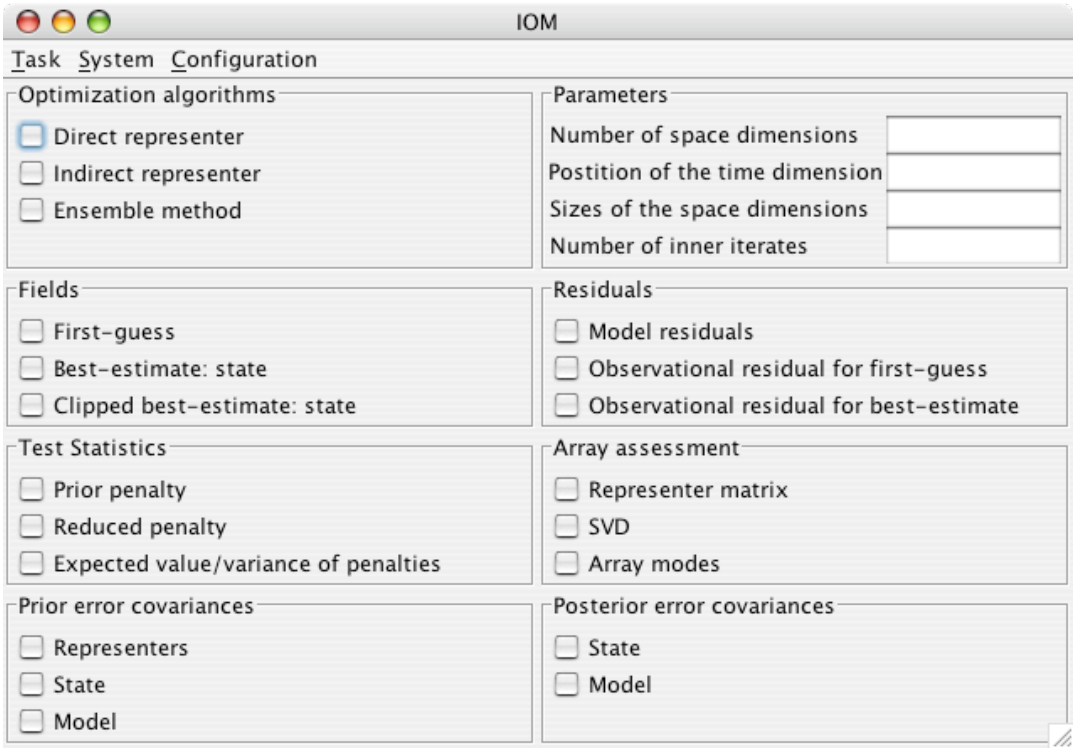

Figure 16. Graphical User Interface.

variables of different dimensions. The main convolution module convolves all state variables in an ocean model. The combination module is an IOM tool that converts a vector into an array whose shape is parameterized, and the measurement module performs the opposite conversion. The models listed in Table II use different data structures for storing the ocean data and the number and meanings of state variables are different in these models. The IOM is not limited to these listed models, but can be applied to other models as well. From the table we can observe that the sizes of the Parametric Fortran templates are small compared to the generated code. For all the generates modules, the code saving is at least 88%, which makes the development and maintenance much easier.

4. Application of Parametric Fortran to Automatic Differentiation

Automatic differentiation (AD) [12] refers to the technique to compute the derivatives of a model defined by a computer program. Two kinds of derivatives are computed, the tangent linear model and the adjoint model. A model can be considered as a mapping of a vector

Table II. Code savings achieved by Parametric Fortran

	SW2D	**KDV**	**PEZ**	**ADCIRC**	**ROMS**	Σ	**PF**	**1-PF/Σ**
Markovian Time Convolution	20	38	60	60	84	262	16	94%
Bell-Shape Space Convolution	27	58	85	85	120	375	27	93%
Main Convolution Module	374	390	502	390	578	2234	245	89%
Combination Module	241	257	321	257	377	1453	177	88%
Measurement Module	183	199	263	199	307	1151	127	89%

Explanation of model acronyms:

SW2D	Shallow Water 2D Model
KDV	Korteweg-de Vries Model
PEZ	Primitive Equation Z-coordinate Model
ADCIRC	Advanced Circulation Model
ROMS	Regional Ocean Modeling System

of control variables X to a vector of predictions Y. For example, in a model for analyzing the relationship between people's income and education level, the control variable could be people's education level, and the prediction is their annual income. The tangent linear model maps variations of the control variable δX of a model to variations of the model prediction δY. Therefore, tangent linear models can be used to quantify how changes of the control variables influence the model predictions. In contrast, the adjoint model maps in the reverse direction and computes the influence of the control variables on a given anomaly of the model predictions. Adjoint models can be used to analyze the origin of any anomaly of a model prediction.

4.1. Implementing Automatic Differentiation in Parametric Fortran

The tangent linear model is constructed by applying the chain rule shown in Figure 17, which can be realized by using pattern matching and syntax tree transformation.

The approach for constructing the adjoint model is to apply rules for each code fragment of the original model to obtain the adjoint code fragment. The adjoint code fragments will then be composed in reverse order, compared to the original model code. The result of the composition is the adjoint model. Giering proposed the rules in for obtaining the adjoint code fragment from each kind of Fortran statements in [13], including assignments, loops, conditionals, and

$$
\begin{aligned}
c' &= 0, \quad c \textit{ is a constant} \\
(u^k)' &= ku^{k-1}u' \\
(e^u)' &= u'e^u \\
(\sin(u))' &= u'\cos(u) \\
(\cos(u))' &= -u'\sin(u) \\
(\log(u))' &= u'/u \\
(u+v)' &= u'+v' \\
(u-v)' &= u'-v' \\
(u*v)' &= u'v+uv' \\
(u/v)' &= (u'v-uv')/v^2
\end{aligned}
$$

Figure 17. Function Derivatives for Computing the Tangent Linear Model

subroutine calls. For example, since the adjoint model reverses the data-flow in the original model, the adjoint code of a loop statement is just the loop with a reverse order. As a concrete example, the adjoint correspondent of the following loop

```
do i=1, 100, 1 ... end do
```

is

```
do i=100, 1, -1 ... end do
```

Rules for other statements are not that trivial. For example, the rules for getting adjoint code from assignments are achieved through the manipulations of the Jacobian matrix. Interested readers can find the details in [13].

We have implemented the chain rule in Figure 17 and the algorithm proposed in [13] as the generation function of a parameter type `Diff`.

```
data Diff = TL [VNames] | AD [VNames]
```

The parameter value of the type `Diff` can be either `TL` or `AD`, followed by a list of the variable names representing the *active variables* of the model, which are either the control variables or the predictions. For example, when `TL ...` is used to parameterize a Fortran subroutine, the Parametric Fortran compiler will generate the tangent linear model of that subroutine. When the parameter value is `TL ...`, the generation function applies the chain rule in Figure 17 to the right-hand side of the assignment statements whose left-hand side is an active variable. The active variable at the left-hand side is renamed to distinguish from the original variable. When the parameter value is `AD ...`, the generation function applies the rules proposed in [13]

for all the statements that assign a value to an active variable or the loops containing such assignments. The active variable at the left-hand side of assignments is also renamed. The following code shows part of the definition of the generation function for the parameter type `Diff` for generating tangent linear code. Below we show the implementation of the fourth and seventh case from Figure 17.

$$gen_p[x\texttt{=sin(}u\texttt{)}] = \texttt{tl\_}x \texttt{ = } gen_p[u] \texttt{ * cos(}u\texttt{)} \quad \textbf{where } p = \texttt{TL [...,}x\texttt{,...]} \tag{9}$$

$$gen_p[x\texttt{=}u\texttt{+}v] = \texttt{tl\_}x \texttt{ = } gen_p[u] \texttt{ + } gen_p[v] \quad \textbf{where } p = \texttt{TL [...,}x\texttt{,...]} \tag{10}$$

When the parameter value is `AD ...`, the following code generates the adjoint code for a loop statement. The generation function reverses the loop boundaries and negates the loop step.

$$gen_{\texttt{AD }vs}[\texttt{do } i\texttt{=}l\texttt{,}u\texttt{,}step\ s\ \texttt{end do}] = \begin{cases} \texttt{do } i\texttt{=}u\texttt{,}l\texttt{,-}step\ s\ \texttt{end do} & \text{if } s \text{ changes any active variables in } vs \\ \texttt{do } i\texttt{=}l\texttt{,}u\texttt{,}step\ s\ \texttt{end do} & \text{otherwise} \end{cases} \tag{11}$$

Generating the adjoint code for assignments is more complicated. The algorithm can be described as follows. If the left-hand side of the assignment is an active variable, the tangent linear code of the right hand side is first calculated. The Jacobian matrix [14] is constructed from the tangent linear code. The adjoint matrix is the transposed Jacobian. From this matrix the adjoint assignments are generated. Details of constructing the Jacobian matrix can be found in [13]. As a concrete example, if `x`, `y`, and `z` are all active variables, the adjoint code of the assignment

```
z = x * sin(y*y)
```

is shown in Figure 18.

```
ad_y = ad_y + ad_z * x * cos(y*y) * 2 * y
ad_x = ad_x + ad_z * sin(y*y)
ad_z = 0
```

Figure 18. Generated adjoint code

In the remainder of this section, we will demonstrate how to use the AD tool built in Parametric Fortran to generate differential programs for a practical example. We have applied the AD tool to generating differential programs for the Primitive Equation Z-coordinate Model (PEZ), which is a variant of Bryan-Cox-Semtner class model [15], developed jointly at Oregon State University and the National Center for Atmospheric Research.

4.2. An Example: The Inviscid Burger's Model

In this section and Section 4.3 we show how to use the AD tool to generate the tangent linear and adjoint code for a simple model, called *the inviscid Burger's model* [16], which is widely used in physics [17, 18]. The inviscid Burger's model can be represented by the following equation. In the equation, u is the function to be calculated, and x and t represent space and time, respectively.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{12}$$

The above formula is a continuous equation. Computer simulations are based on the corresponding discrete equations that can be derived from the continuous one. The corresponding mathematics background can be found in [16]. Below we show the discrete equations of the inviscid Burger's model.

$$u_0^1 = u_0^0 - \frac{\Delta t}{2\Delta x}u_0^0(u_1^0 - u_X^0) \tag{13}$$

$$u_x^1 = u_x^0 - \frac{\Delta t}{2\Delta x}u_x^0(u_{x+1}^0 - u_{x-1}^0), \text{ for } x = 1, 2, \ldots, X-1 \tag{14}$$

$$u_X^1 = u_X^0 - \frac{\Delta t}{2\Delta x}u_X^0(u_0^0 - u_{X-1}^0), \tag{15}$$

$$u_0^{t+1} = u_0^{t-1} - \frac{\Delta t}{\Delta x}u_0^t(u_1^t - u_X^t), \text{ for } t = 1, 2, \ldots, T-1 \tag{16}$$

$$u_x^{t+1} = u_x^{t-1} - \frac{\Delta t}{\Delta x}u_x^t(u_{x+1}^t - u_{x-1}^t), \text{ for } t = 1, 2, \ldots, T-1 \;\; x = 1, 2, \ldots, X-1 \tag{17}$$

$$u_X^{t+1} = u_X^{t-1} - \frac{\Delta t}{\Delta x}u_X^t(u_0^t - u_{X-1}^t), \text{ for } t = 1, 2, \ldots, T-1 \tag{18}$$

The discrete equations describe how the value of u changes over time. Time ranges from 0 to T, and space ranges from 0 to X. Δt and Δx represent the length of a time step and the size of a spatial grid, respectively. The values of u_x^0, for $0 \le x \le T$ form the initial condition and should already exist before the calculation. The periodic boundary condition is used, which means the right neighbor of the rightmost point is the leftmost point and the left neighbor of the leftmost point is the rightmost point.

4.3. Differentiating the Inviscid Burger's Model Using Parametric Fortran

The discrete equations can be directly translated into the Parametric Fortran subroutine, shown in Figure 19, which is parameterized by `diff`. In the subroutine, we perform computations on array indices so that all the discrete equations can be represented by one assignment statement in line 24. The value for `diff` can be `TL [u]` or `AD [u]` since `u` is the only active variable in the inviscid burger's model. `TL` and `AD` will lead the Parametric Fortran compiler to generate the tangent-linear code and the adjoint code of the inviscid Burger's model, respectively.

```
{diff:
  subroutine burger(X,T,dx,dt,u)
    integer :: X, T
    real :: dx, dt, c
    real, dimension(0:X,0:T) :: u
    integer :: x, t, xm1, xp1, tm1, tp1
    c = dt / (2 * dx)
    do t = 0, T-1
      tp1 = t + 1
      tm1 = t - 1
      if (t == 0) then
        tm1 = 0
      else
        c = dt / dx
      end if
      do x = 0, X
        xp1 = x + 1
        xm1 = x - 1
        if (x == 0) then
          xm1 = X
        else if (x == X) then
          xp1 = 0
        end if
        u(x,tp1) = u(x,tm1)-u(x,t)*(u(xp1,t)-u(xm1,t))*c
      end do
    end do
  end subroutine burger
}
```

Figure 19. Inviscid Burger's model template

When the value for `diff` is `TL [u]`, the tangent-linear program of the inviscid Burger's model will be generated. The following code shows the generated Fortran subroutine for calculating the tangent-linear derivative. The program generator changes the output variable name of the generated subroutine to `tl_u` to distinguish from `u`. In addition to this name change, the only difference from the original subroutine is in the assignment statement for `u` since that is the only place where the value of the output variable is changed. The program generator applies the chain rule to the right-hand side of this assignment, and the new assignment in the code fragment shown in Figure 20 is generated.

```
subroutine tl_burger(X,T,dx,dt,u,tl_u)
...
      tl_u(x,tp1) = tl_u(x,tm1)-(tl_u(x,t)*(u(xp1,t)-u(xm1,t))
                    +u(x,t)*(tl_u(xp1,t)-tl_u(xm1,t)))*c
...
end subroutine tl_burger
```

Figure 20. Part of generated subroutine

When the value for `diff` is `AD [u]`, the program for the adjoint model of the inviscid Burger's model will be generated as shown in Figure 21.

```
1  subroutine ad_burger(X,T,dx,dt,u,ad_u)
2  ...
3    do t = T-1, 0 ,-1
4      ...
5      do x = X, 0 ,-1
6        ...
7        ad_u(x,tm1) = ad_u(x,tm1)+ad_u(x,tp1)
8        ad_u(x,t)   = ad_u(x,t)-(c*(u(xp1,t)-u(xm1,t)))*ad_u(x,tp1)
9        ad_u(xp1,t) = ad_u(xp1,t)-(c*u(x,t))*ad_u(x,tp1)
10       ad_u(xm1,t) = ad_u(xm1,t)+(c*u(x,t))*ad_u(x,tp1)
11       ad_u(x,tp1) = 0
12     end do
13   end do
14 end subroutine ad_burger
```

Figure 21. Generated adjoint subroutine

Similar to the tangent-linear program, a new variable `ad_u` is added to hold the adjoint values. The original assignment statement is replaced by a group of assignment statements in the loop body. These generated assignments are obtained from the rule in [13] for assignment statements. Following the rule for loop statements shown in Section 4.1, the loops over time and space are reversed.

5. Related Work

Parametric Fortran provides a framework to allow developing simulation programs for scientific models. A different approach was chosen by the Earth System Modeling Framework (ESMF), which defines an architecture that allows the composition of applications using a component-based approach [4, 5]. The focus of the ESMF is to define standardized programming interfaces and to collect and provide data structures and utilities for developing model components. One disadvantage of the ESMF approach is that developers of the scientific models need to refactor their existing code to fit the interface, which is usually not easy. In contrast, using Parametric Fortran to develop the simulation programs will not require model developers to change their code. The simulation programs for their models can be generated automatically.

Parametric Fortran was developed for the 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 changed in different models to which the same algorithm will be applied. The programming languages APL [19] and J [20] have built-in mechanisms for computing with variable-dimensional arrays. However, since APL and J only provide efficient operations for array processing, they have not been widely used in scientific computing area, which also requires efficient numerical computations.

Parametric Fortran is essentially a metaprogramming tool. For a comprehensive overview over the field, see [21]. Existing Fortran metaprogramming tools include Foresys [22], whose focus is on the refactoring of existing Fortran code, for example, transforming a Fortran77 program into Fortran90 program. Sage++ [23] 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 that are implemented by computer scientists. Macrofort [24] can generate Fortran code from Maple programs and does not provide the mechanism to deal with generic, model-dependent code. Forge [25] 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. Rosing and Yubasaki present a customizable preprocessor for Fortran in [26]. Similar to Parametric Fortran, their preprocessor works by taking annotated Fortran code and transforming it into a new Fortran program. The preprocessor uses a C-like language called DL to describe how the Fortran program should be changed, whereas Parametric Fortran uses Haskell. The preprocessor is only meant to be used to parallelize programs, whereas Parametric Fortran is meant to be more general and extensible by adding new parameter types. In particular, their preprocessor library could be implemented as Parametric Fortran parameter types, and directives could be represented by parameter values.

Parametric Fortran can be used for removing duplicated code. Some work has been done for removing duplicated code from programs. For example, CloRT [27] automatically rewrites duplicated code into programming abstractions. The approach of linked editing is used to manage duplicated Java code in [28].

Another application of Parametric Fortran described in this paper is automatic differentiation (AD). Much has been done in this area, and some tools have been developed, such as TAMC [29], COSY [30], and TAPENADE [31]. All these tools have developed their own algorithm for doing source program transformation. Their algorithms can be integrated into Parametric Fortran as parameter types. For example, the AD tool described in Section 4 implements the algorithm used in the TAMC system.

6. Conclusions

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 [8, 7]. Beyond the IOM system, Parametric Fortran also has the potential to be applied to many other scientific computing projects, such as the WRF [3] system, the DART system [32], and the ESMF [4, 5]. Parametric Fortran is also applied to automatic differentiation (AD). The

AD tool built in Parametric Fortran can incorporate different AD algorithms and is easy to use for Fortran programmers.

REFERENCES

1. B. Chua and A. F. Bennett. An Inverse Ocean Modeling System. *Ocean Modelling*, 2001, 3:137–165.
2. IOM. Inverse Ocean Modeling System. `http://iom.asu.edu/` [29 Jan 2007].
3. Michalakes J., S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock. Development of a Next Generation Regional Weather Research and Forecast Model. *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*, World Scientific Publishing Co., Hackensack, NJ, 2001. 269–276.
4. R. Ferraro, T. Sato, G. Brasseur, C. DeLuca, and E. Guilyardi. Modeling The Earth System. *Int. Symp. on Geoscience and Remote Sensing*, IEEE Computer Society, Los Alamitos, CA, USA, 2003. 630–633.
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*, 2002, 83(3):431–434.
6. S. L. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003, 270 pp.
7. M. Erwig, Z. Fu, and B. Pflaum. Generic Programming in Fortran. *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ACM Press, New York, NY, USA, 2006. 130–139.
8. M. Erwig and Z. Fu. Parametric Fortran – A Program Generator for Customized Generic Fortran Extensions. *6th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 3057, Springer-Verlag, New York Berlin Heidelberg, 2004. 209–223.
9. R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, ACM Press, New York, NY, USA, 2003. 26–37.
10. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 2000, 464 pp.
11. K. R. Castleman. *Digital Image Processing*. Prentice-Hall International, Englewood Cliffs, NJ, 1996, 667 pp.
12. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Computer and Information Science. Springer-Verlag, New York Berlin Heidelberg, 2001, pp. Selected papers from the AD2000 conference, Nice, France, June 2000.
13. R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Transactions on Mathematical Software*, 1998, 24(4):437–474.
14. F.H. Clarke. *Optimization and nonsmooth analysis*. Wiley-Interscience, New York, 1983, 320 pp.
15. Pacanowski and Griffies. *MOM 3.0 Manual*. NOAA/GFDL, 2000. `http://www.gfdl.noaa.gov/smg/MOM/web/guide_parent/guide_parent.html` [8 Feb 2007].
16. D. R. Durran. *Numerical Methods for Wave Equations in Geophysical Fluid Dynamics*. Springer-Verlag, New York Berlin Heidelberg, 1999, 482 pp.
17. Y. B. Zel'Dovich. Gravitational instability: an approximate theory for large density perturbations. *Astronomy and Astrophysics*, 1970, 5:84–89.
18. M. Kardar, G. Parisi, and Y.C. Zhang. Dynamical scaling of growing interfaces. *Physical Review Letters*, 1986, 56:889–892.
19. K. E. Iverson. *Introduction to APL*. APL Press, 1984, 110 pp.
20. K. E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995. 411 pp.
21. Tim Sheard. Accomplishments and Research Challenges in Meta-Programming. *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, Springer-Verlag, New York Berlin Heidelberg, 2001. 2–44.
22. Simulog, SA, Guyancourt, France. *FORESYS, FORtran Engineering SYStem, Reference Manual v1.5*, 1996. 80 pp.
23. F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Srinivas, N. Srinivas, and B. Winnicka. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. *OON-SKI'94, Second Object-Oriented Numerics Conference*, Computer Science Department, Indiana University, Bloomington, IN, April 1994. 122–138.

24. C. Gomez and P. Capolsini. Macroc and Macrofort, C and Fortran Code Generation Within Maple. *Maple Technical Newsletter*, 1996, 3(1):14–19.
25. M. Erwig and Z. Fu. Software Reuse for Scientific Computing Through Program Generation. *ACM Transactions on Software Engineering and Methodology*, 2005, 14(2):168–198.
26. Matt Rosing and Steve Yabusaki. A programmable preprocessor for parallelizing Fortran-90. *ACM/IEEE Conf. on Supercomputing*, ACM Press, New York, NY, USA, 1999. 3.
27. M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. *6th Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, 1999. 326–336.
28. M. Toomim, A. Begel, and S. L. Graham. Managing Duplicated Code with Linked Editing. *IEEE Symposium on Visual Languages and Human-Centric Computing*, IEEE Computer Society, Los Alamitos, CA, USA, 2004. 173–180.
29. R. Giering and T. Kaminski. Using TAMC to generate efficient adjoint code: Comparison of automatically generated code for evaluation of first and second order derivatives to hand written code from the Minpack-2 collection. In C. Faure, editor, *Automatic Differentiation for Adjoint Code Generation*, 31–37. INRIA, Sophia Antipolis, France, 1998.
30. L. M. Chapin, J. Hoefkens, and M. Berz. The cosy language independent architecture: Porting cosy source files. *7th Int. Conf. on Computational Accelarator Physics*, volume 175 of *Institute of Physics Conference Series*. IOP Publishing Ltd., 2002, 37–45.
31. L. Hascoet, R.-M. Greborio, and V. Pascual. Computing Adjoints by Automatic Differentiation with TAPENADE. In B. Sportisse and F.-X. LeDimet, editors, *HGP05*. Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France, 2005. `http://www-sop.inria.fr/tropics/Laurent.Hascoet/papers/HGP05.pdf` [2 Feb 2007].
32. National Center for Atmospheric Research. The Data Assimilation Research Testbed – DART. `http://www.image.ucar.edu/DAReS/DART/` [29 Jan 2007].

AUTHORS' BIOGRAPHIES

Martin Erwig is an Associate Professor of Computer Science at Oregon State University. He received his M.S. in Computer Science from the University of Dortmund, Germany, in 1989, and his Ph.D. and Habilitation from the University of Hagen, Germany, in 1994 and 1999, repsectively. His research interests include functional programming, domain-specific languages, and visual languages.

Zhe Fu received his Ph.D. from Oregon State University in 2006. Before joining Oregon State University, he received his B.S. from Peking University in 1998 and his M.S. from the Software Institute of the Chinese Academy of Sciences in 2001. He is now working at Microsoft on software development for the C# language.

Ben Pflaum received his B.S. degree in Computer Science from Oregon State University in 2006. He is currently an M.S. student at Oregon State University working on automatic program generation for scientific computing.