Parametric Fortran – A Program Generator for Customized Generic Fortran Extensions*

Martin Erwig and Zhe Fu

School of EECS
Oregon State University
[erwig|fuzh]@cs.orst.edu

Abstract. We describe the design and implementation of a program generator that can produce extensions of Fortran that are specialized to support the programming of particular applications. Extensions are specified through parameter structures that can be referred to in Fortran programs to specify the dependency of program parts on these parameters. By providing parameter values, a parameterized Fortran program can be translated into a regular Fortran program.
We describe as a real-world application of this program generator the implementation of a generic inverse ocean modeling tool. The program generator is implemented in Haskell and makes use of sophisticated features, such as multi-parameter type classes, existential types, and generic programming extensions and thus represents the application of an advanced applicative language to a real-world problem.

Keywords: Fortran, Generic Programming, Program Generation, Haskell

1 Introduction

Fortran is widely used in scientific computing. For example, scientific models to predict the behavior of ecological systems are routinely transformed by scientists from a mathematical description into simulation programs. 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. Alas, 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 simulation programs currently have to be rewritten for each individual forecasting model, even though the underlying algorithms are principally the same for all models. Since Fortran lacks the flexibility to apply the same subroutine on different data structures, scientists have to rewrite programs for every model.

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

We provide a solution to this problem by an extension of Fortran, called *Parametric Fortran*, which is essentially a framework that allows the creation of domain-specific Fortran extensions. A Parametric Fortran program is a Fortran program in which parts, such as statements, expressions, or subroutines, are parameterized by special variables. Every syntactic element of Fortran can be parameterized. When the values of these parameters are given, a program generator can create a Fortran program from the parameterized program, guided by these values. Scientists can implement their algorithm in Parameterized 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 Fortran program can be generated for this model. Therefore, scientists only need to write their programs once and can generate different instances for different models automatically.

The program generator, which is implemented in Haskell [15], takes a Parametric Fortran program and parameter definitions and their values as inputs and produces a Fortran program, see Figure 1.

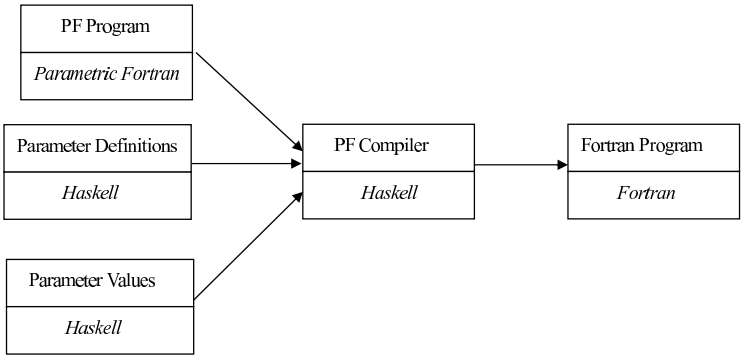

Fig. 1. System architecture

Two different groups of people are concerned with Parametric Fortran. First, scientists who write programs 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 Parametric Fortran with new parameter types. Rather than defining one particular extension of Fortran, our approach is to provide a framework for extending Fortran on a demand basis and in a domain-specific way.

In the remainder of this Introduction we outline the approach by two simple examples. In Section 2 we illustrate the use of Parametric Fortran in a real-world application. In Section 3 we describe the implementation of Parametric Fortran. We discuss related work in Section 4 and present some conclusions in Section 5.

1.1 Array Addition for Arbitrary Dimensions

The following example shows how to write a Parametric Fortran subroutine to add two arrays of arbitrary dimensions.[1] For simplicity, we suppose that the size of each dimension is 100.

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

The program is parameterized by an integer `dim`. The value of `dim` will guide the generation of the Fortran subroutine. The brackets `{` and `}` delimit the scope of the `dim` parameter, that is, every Fortran syntactic object in the scope is parameterized by `dim`. For `dim` = 2, the following Fortran program will be generated.

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

We can observe that in the generated program, `a`, `b`, and `c` are all 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 2 loops over both dimensions, and index variables are added to each array expression. The declarations for these index variables are also generated. This particular behavior of the program generator is determined by the definition of the parameter type for `dim`.

1.2 Dimension-Independent Array Slicing

We can also define an array slicing subroutine, which slices an n-dimensional array on the dth dimension. In the subroutine `slice`, `a` is the input n-dimensional array, `b` is the result $(n-1)$-dimensional array, and `k` is the index on the dth dimension of the input array. This program is parameterized by two parameters. The parameter `dim` is an integer representing the number of dimensions of `a`. It only parameterizes the declaration of `a`. The parameter `slice` is a pair of integers of the form `(n,d)`, representing that the generated code slices the `d`th dimension

[1] This example is meant for illustration. Dimension-independent array addition is already supported in Fortran by array syntax.

of an `n`-dimensional array. 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 example, when `dim` is `3` and `slice` is `(3,2)`, the generated Fortran subroutine will be able to compute the `k`th slice of the second dimension of a 3-dimensional array. 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
```

The declaration of `a` is parameterized by an integer, which has the same meaning as in the first example. The declaration of `b` is parameterized by the parameter `slice`, which is a pair of integers `(n,d)`. 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 `d`th dimension. How the different types of parameters guide the program generation has to be defined as part of the parameter definition, which specifies the program generator. Since `n` represents the number of the dimensions and `d` represents one of the `n` dimensions, `d` must be in the range `1` to `n`. The program generator can report an error if the value of `slice` is, for example, `(2,3)`. We also have to ensure that `n` must be equal to `dim`. Otherwise, the generated Fortran program will contain a type error.

2 An Application in Scientific Computing

We consider as a real-world application the generation of inversion tools in the domain of ocean modeling. The Inverse Ocean Modeling (IOM) system [3, 9] is a data assimilation system, which enables the developers of ocean models 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 consists of tools that are used for solving the equations of best fit.

Since every ocean model uses its own data structure to describe the ocean, it is impossible to write a system in Fortran that works with different ocean models, although the algorithm for inverse ocean modeling is the same for all models. The genericity that is inherent in the problem cannot be expressed by Fortran.

With Parametric Fortran the IOM tools can be written in a generic way so that different Fortran programs can be generated automatically as needed. Using an extension of Fortran supports the idea of "gentle slope" [14] and allows the developers of the IOM, who are familiar with Fortran, to start writing the tools without first having to learn a completely new language. Moreover, most of their current programs can be modified to Parametric Fortran programs with reasonable effort. The main task is to identify the parameters for representing the model-dependent information.

2.1 An Example Tool: Markovian Convolution in Time

The IOM tool described here is an example of a convolution tool used for averaging over weighted values. The basic idea is to compute the value of one point by averaging over the weighted values of its neighbors. The weights of its neighbors depend on the distances from the point. The *Markovian Convolution in Time* is formally defined by the following continuous equation.

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

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 [3].

$$\begin{aligned} h_L &= 0 & \frac{h_n - h_{n-1}}{\Delta t} + \tau^{-1} h_{n-1} &= -2\tau^{-1} a_n \\ b_U &= -(\tau/2) h_U & \frac{b_{n+1} - b_n}{\Delta t} - \tau^{-1} b_{n+1} &= h_n \end{aligned}$$

In the above equations, h represents a temporary array, and L (U) is the lower (upper) boundary of the arrays. The arrays are over time and space where different models may use different array dimensions for space. The number of time dimension is always one. However, different models may have different numbers of space dimensions. 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.

2.2 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. We suppose 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, but for simplicity we do not do that in this example. We can define a Haskell data type `Dim` to represent the model-dependent information as follows.

```
data Space = Space Int [(IExpr, IExpr)]
data IExpr = ICon Int | IVar VName | ...
```

For example, the parameter value `Dim 2 [(IVar "X", IVar "Y"), (ICon 1, ICon 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`.

Since the only dependent information is given by the space dimensions of the arrays, including the number of dimensions and the boundaries of each dimension, we only need one parameter type `Space`. Every array has the same type, which means that the array variable declarations can be parameterized by the same parameter. However, one parameter is not sufficient to parameterize the body of the subroutine because we want to be able to group assignment statements in the same loop, instead of getting a separate loop for each assignment. This distinction of different "loop groups" requires additional information to indicate if a loop over space dimensions needs to be generated, which is captured by two tags.

```
data Pos a = Loop a | Inside a
```

We use `Pos Space` to parameterize both declarations and assignments. If an assignment is parameterized by `Loop s`, a loop over the space dimensions is generated. However, if an assignment is parameterized by `Inside s`, no loop will be generated, the program generator just adds index variables to each array expression. The Parametric Fortran code for the Markovian Convolution is shown below.

```
subroutine timeConv {d: (L, U, a, b)}
  real :: dt, tau
  {d: real, dimension (!L:!U) :: a, b, h}
  integer :: n
  {d:
    {x: h(!L) = 0.0}
    {x(a,b,h):
      do n = L, U
         h(n) = h(n-1) - dt*(h(n-1)/tau + 2.0*a(n)/tau)
      end do
      b(U) = -0.5*h(U)/tau
      do n = L, U
         b(n) = b(n+1) - dt*(h(n) + b(n+1)/tau)
      end do
    }
  }
end subroutine timeConv
```

The value of parameter `d` is `Loop s`, where `s` represents the space dimensions of the arrays. The parameter `d` is used to parameterize 3 parts of the subroutine.

- The parameter list of the generated Fortran subroutine to add the new variables used in `s`'s value as dimension boundaries
- The declarations of the array variables to append the space dimensions to the current time dimension of these arrays
- The body of the subroutine to create loops.

The symbol `!` is used to stop the downward propagation of a parameter for a particular part of the syntax tree. For example, since `L` is a non-array variable, we do not want to parameterize it by `x` in the first assignment.[2] The parameter `x` has the value of `Inside s` since the assignments parameterized by `x` do not need their own loops to be generated. However, all the array expressions in the assignment statements parameterized by `x` will be filled by index variables of the space dimensions. We also provide a form of applicability constraint for parameters by allowing a list of variables to be added after the parameter. For instance, the syntax `{x(a,b,h):...}` expresses that the parameter `x` only parameterizes the variables `a`, `b`, and `h`, or array expressions using these names, such as `h(n)`. When we 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 can assign `d` and `x` the following values.

```
d, x :: Pos Space
s    :: Space
s = Space 1 [(IVar "X", IVar "Y")]
d = Loop s
x = Inside s
```

[2] Since every node in the syntax tree can be annotated only by one parameter, we could not use `d` and `x` to annotate the whole body. Therefore, we have parameterized the first assignment and the remaining sequence of assignments separately.

With these parameter values the program generator creates the following subroutine.

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

The program generator had the following effects. (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. (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.

3 Implementation of the Parametric Fortran Compiler

3.1 The Parametric Fortran Parser

For the implementation of the front end of Parametric Fortran we have used the Haskell scanner generator Alex [5] and the parser generator Happy [13]. The parser deals with a subset of Fortran language and ignores some esoteric "features". For example, we do not allow spaces inside identifiers. Figure 2 shows part of the syntax of Parametric Fortran. It illustrates for the syntactic categories *Stmt* and *Expr* how every syntactic category of Fortran is extended by two forms of parameterization.

We use `{p:`e`}` to represent that the syntactic object e is parameterized by the parameter `p`. The semantics of such a parameter annotation is that e itself is parameterized by `p` and also that `p` is propagated downward to all (parameterizable) descendants in the syntax tree. We use `!`e to stop the effect of a possibly propagated parameter on e, which means e and all of its subexpressions are not parameterized at all. A parameter can be extended by a list of variables that restrict the downward propagation to just this set of variables. For example, in `{p(v):`e`}` only variable occurrences of `v` and array expressions `v(...)` inside e will be parameterized by `p`. This form of parameter applicability constraints is currently limited to variables, a language for allowing the specification of tree patterns to be parameterized is part of future work.

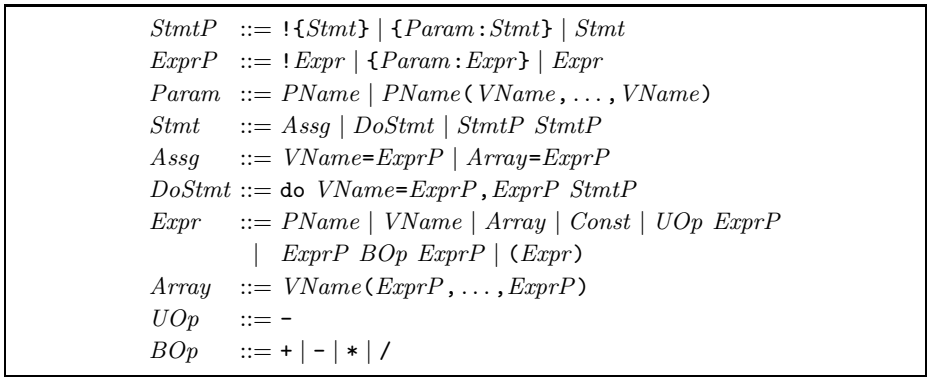

Fig. 2. Syntax of Parameterized Fortran.

3.2 Abstract Syntax

For lack of space, we only show an excerpt of the abstract Fortran syntax. For example, `Stmt` only contains assignment, loop, and sequential statements. The Haskell definitions of the data types for Fortran statements and expressions are shown below.

```
data Stmt = Assg VarName [ExprP] ExprP
          | For  VarName ExprP ExprP StmtP
          | Sequ StmtP StmtP
            deriving (Typeable,Data)

data Expr = IntCon Int
          | RealCon Float
          | Var VarName [ExprP]
          | Bin BinOp ExprP ExprP
            deriving (Typeable,Data)
```

The data types `StmtP` and `ExprP` represent parameterized Fortran statements and expression. A syntactic object may contain parameterized sub-objects, for example, a Fortran assignment statement, which is of type `Stmt`, may contain parameterized Fortran expressions, which are of type `ExprP`. These data types must be instances of type classes `Typeable` and `Data`, because we have to apply the functions `cast` and `everywhere` (described below) to elements of these types.

Since every Fortran syntactic category can be parameterized, we need a data type for each parameterized Fortran syntactic category. The following code shows the data type definitions for parameterized Fortran statements and expressions.

```
data StmtP = forall p . Param p Stmt => F p Stmt
data ExprP = forall p . Param p Expr => E p Expr
```

We use existential types to facilitate the use of parameters of different types at different nodes in the syntax tree.

3.3 A Type Class for Program Generation

We define a multiple parameter type class `Param` to represent the relation between a parameter type and the data type of a syntactic category. Again, the context `Data p` is needed because we want to use the function `everywhere` to implement traversals of the syntax tree together with a type-based selective application of a generator function.

```
class (Show p,Data p,Show e) => Param p e where
  showP  :: p -> e -> String
  gen    :: p -> e -> e
  check  :: p -> e -> Bool
  -- default implementations
  gen   _   = id
  check _ _ = True
```

In the above definition, `p` represents the parameter type and `e` represents the syntactic category. The member function `showP` is used for pretty-printing the parameterized program. The program generator `gen` takes a parameter value and a Fortran syntactic object as input and generates a non-parameterized syntactic object. The member function `check` can be used to implement validity checks for parameterized program. For lack of space we cannot describe the details of this part of the program generator.

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 3.5 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 by `showP`. The `Void` parameter uses the default definition for `gen` and `check`.

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

instance Show Void where
   show Void = ""

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

Since parameters in Parametric Fortran programs are variables, we define the type of variable names `VarName` as an instance of the type class `Param`. Similar to `Void`, `VarName` can be used to parameterize any syntactic category and its `gen` function does nothing. The Parametric Fortran parser is a function of type `String->FortranP`. In the parsing result, all parameters are of type `VarName`. For generating the Fortran programs, we have to supply values for the parameters so that every parameter name in the abstract syntax tree can be replaced by the parameter's value. We have defined a derived type class `Par p` to be able to succinctly express that `p` is a valid parameter type of Parametric Fortran.

```
class (Param p Expr, Param p Stmt, ...) => Par p where
```

In other words, only if the type `p` can be used to parameterize all the Fortran syntactic categories, it is an instance of the type class `Par`. We use a heterogeneous list `param` of type `PList` to store parameters of different types.

```
data ParV  = forall p . Par p => ParV p
type PList =  [(VarName, ParV)]
```

Parameter names are replaced by parameter values through a collection of functions that look up the values of parameters in the list `param`. For example, the function `substE` maps expressions parameterized by parameter names to expressions parameterized by the parameters' values.

```
substE :: ExprP -> ExprP
substE (E p e) = case lookup (getName p) param of
                      Nothing        -> E Void e
                      Just (ParV p') -> E p' e

getName :: forall p . Par p => p -> VarName
getName = (\p->(VarName "")) `extQ` id
```

We have similar functions for all other other syntactic categories. The function `getName` in the above definition is a generic function that works on any parameter type, but only gets a valid value for the type `VarName`. This genericity is realized by the function `extQ`, which was introduced in [12] to extend generic queries of type `Typeable a => a->r` by a new type-specific case.

3.4 Defining Parameter Types as Instances of `Param`

In the array slicing example, we have two parameter types, `Dim` and `Slice`. The type `Dim` specifies the number of dimensions of an array. The type `Slice` is used for parameterizing the dimension to be sliced.

```
data Dim   = Dim Int
data Slice = Slice Int Int
```

To use these types to parameterize Fortran programs, we have to make them instances of the `Param` type class for all Fortran syntactic categories. Figure 3 shows how to define the `gen` function for the parameter type `Dim` for every Fortran syntactic category. Most instance declarations use the default definition of `gen`. We only have to consider the cases of extending a type by dimensions, adding loops around a statement, and filling an expression with 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, the function `freshNames` traverses the complete program and renames 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.

```
instance Param Dim Type  where
  gen (Dim d) (BaseType bt)  = ArrayT (indx d) bt
  gen (Dim d) (ArrayT rs bt) = ArrayT (rs++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 es) = Var v (es++map (var . newVar) [1..d])
  gen p       e          = e

a = E Void (IntCon 1)
b = E Void (IntCon 100)

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

Fig. 3. Example Parameter Implementation

3.5 Transformation Functions

For every Fortran syntactic category, a transformation function is defined. Whereas `gen` has to be implemented for every new parameter type, the transformation functions are defined once and for all. They transform a parameterized syntactic object into a Fortran object parameterized by `Void`. The following code shows how these transformation functions are defined for Fortran statements and Fortran expressions. The approach is to extract the parameter `p` from the node in the syntax tree and apply the function `gen` with this parameter to the syntax constructor under consideration. Other Fortran syntactic categories are dealt with similarly.

```
transF :: StmtP -> StmtP
transF (F p f) = F Void (gen p f)

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

We can observe that every transformation function has the type `a->a`. That enables us to apply the function `everywhere` [12] to build a generic transformation function `genF`, which is basically the program generator. The function `everywhere` is a generic traversal combinator that applies its argument function to every node in a tree. The argument function is a generic transformation function of type `forall b. Data b => b->b`. We can lift a non-generic transformation function `f`, which has the type `Data t => t->t`, into a generic transformation function `g` by the function `extT` as follows.

```
g = id `extT` f
```

Therefore, `extT` allows the composition of different behaviors on different types into one generic function. In the definition of `genF`, we use `extT` to compose different transformation functions for different syntactic categories. By applying the function `everywhere` to a generic transformation function `g`, we obtain another generic transformation function, which applies `g` to every node in a tree. Therefore, `genF` can be defined in the following way.

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

The function `genF` takes a parameterized Fortran program as input and outputs a Fortran program in which every syntactic object is parameterized by `Void`. The source code of the generated Fortran program can be obtained by calling the `showP` function.

4 Related Work

The Earth System Modeling Framework (ESMF) defines an architecture that allows the composition of applications using a component-based approach [6, 4]. The focus of the ESMF is to define standardized programming interfaces and to collect and provide data structures and utilities for developing model components.

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 [10] and J [11] 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. Although Matlab [2] is popular for testing simulations on small examples, it is too inefficient for large data sets.

Parametric Fortran is essentially a metaprogramming tool. For a comprehensive overview over the field, see [16]. Existing Fortran metaprogramming tools include Foresys [18], whose focus is on the refactoring of existing Fortran code, for example, transforming a Fortran77 program into Fortran90 program. Sage++ [1] 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. Macrofort [8] can generate Fortran code from Maple programs and does not provide the mechanism to deal with generic, model-dependent code. In metaprogramming systems like MetaML [19] or Template Haskell [17] the metaprogramming language is an extension of the object language. In Parametric Fortran, metaprogramming happens in two different languages: (i) parameters are defined in Haskell and (ii) parameter-dependencies in object programs is expressed in Parametric Fortran syntax.

There has also been a lot of work on the generation of Fortran code for simulations in all areas of science. All this work is concerned with the generation of efficient code for *one particular* scientific model. For example, CTADEL [20] 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 [7] is similar to ours 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.

5 Conclusions

Parametric Fortran provides a framework for defining Fortran program generators through the definition of parameter structures. The system offers a two-level architecture to Fortran program parameterization and generation. On the first level, the definition of parameter structures creates a Fortran dialect for a particular class of applications. On the second level, programs for different problems can be implemented within one Fortran dialect employing parameters in similar or different ways. Any such Parametric Fortran program can be translated into different ordinary Fortran programs by providing different sets of parameter values.

We have successfully applied Parameteric Fortran in the area of scientific computing to enable the generic specification of inverse ocean modeling tools. The general, two-level framework promises opportunities for applications in many other areas as well. In one tool (space convolution) we had to employ a parameter that allows the parameterization by Fortran subroutines and provides limited support for higher-order subroutines. This extension turned out to be a straightforward task. From the experience we have gained so far and from the feedback from ocean scientists we conclude that Parameteric Fortran is a flexible and expressive language that provides customized genericity for Fortran through a pragmatic approach. The use of advanced Haskell features was essential in this approach.

References

1. F. Bodin, P. Beckman, D. Gannon, J. Gotwals, and S. Srinivas. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. In *Second Object-*

Oriented Numerics Conference (OON-SKI'94), pages 122–138, 1994.
2. S. J. Chapman. *MATLAB Programming for Engineers*. Brooks Cole, 2001.
3. B. Chua and A. F. Bennett. An Inverse Ocean Modeling System. *Ocean Modelling*, 3:137–165, 2001.
4. 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), 2002.
5. C. Dornan. Alex: A Lex for Haskell Programmers, 1997. `http://haskell.org/libraries/alextar.gz`.
6. R. Ferraro, T. Sato, G. Brasseur, C. DeLuca, and E. Guilyardi. Modeling The Earth System. In *Int. Symp. on Geoscience and Remote Sensing*, 2003.
7. 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.
8. C. Gomez and P. Capolsini. Macroc and Macrofort, C and Fortran Code Generation Within Maple. *Maple Technical Newsletter*, 3(1), 1996.
9. IOM. `http://iom.asu.edu/`.
10. K. E. Iverson. *Introduction to APL*. APL Press, 1984.
11. K. E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.
12. 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.
13. S. Marlow and A. Gill. Happy User Guide, 2000. `http://www.haskell.org/happy/doc/html/happy.html`.
14. B. Myers, S. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3–28, 2000.
15. S. L. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.
16. 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.
17. T. Sheard and S. L. Peyton Jones. Template Metaprogramming for Haskell. In *Haskell Workshop*, 2002.
18. Simulog, SA, Guyancourt, France. *FORESYS, FORtran Engineering SYStem, Reference Manual v1.5*, 1996.
19. W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.
20. 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.