AN ABSTRACT OF THE DISSERTATION OF

Zhe Fu for the degree of Doctor of Philosophy in Computer Science presented on July 6, 2006.

Title: Automatic Program Generation for Scientific Computing

Abstract approved: _____________________________________________

Martin Erwig

The code reuse problem is a common software engineering problem in scientific computing. As a prevailing programming language in many scientific fields, Fortran does not provide support to address this problem. One particular reason is that Fortran lacks the support for generic programming. By applying program-generation techniques, we developed two approaches to address the code reuse problem. The first approach is to design a program generator for the equation-based specification of subroutines that can be generic in the dimensions of arrays, parameter lists, and called subroutines. We apply that approach to a real-world problem in scientific computing, which requires the generic description of inverse ocean modeling tools. In addition to a compiler that can transform generic specifications into efficient Fortran code for models, we have also developed a type system that can identify possible errors already in the specifications. The second approach is to extend Fortran with the support for generic programming. The result is the language Parametric Fortran, which 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. Parametric Fortran is particularly useful in scientific computing. The applications include defining generic functions, removing duplicated code, and automatic differentiation. The described Fortran extension has also been successfully employed implementing the generic inverse ocean modeling system.
Automatic Program Generation for Scientific Computing

by

Zhe Fu

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented on July 6, 2006
Commencement June 2007
Doctor of Philosophy dissertation of Zhe Fu presented on July 6, 2006.

APPROVED:

________________________________________
Major Professor, representing Computer Science

________________________________________
Director of the School of Electrical Engineering and Computer Science

________________________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

________________________________________
Zhe Fu, Author
ACKNOWLEDGEMENTS

Foremost I especially thank my advisor Martin Erwig for suggesting the wonderful thesis topic. His wide knowledge and logical way of thinking have been of great value for me. His understanding, encouraging and personal guidance have provided a good basis for the present thesis. I also thank professor Andrew Bennett. His summer school in 2002 helped me to understand the area of inverse ocean modeling, which is the major application of the approaches presented in this thesis. My thanks also go to Boon Chua and Ben Pflaum, who helped me with the implementation of the IOM system and Parametric Fortran. I am really grateful to my family, especially to my parents for their continuous support and encouragement. At last, I want to express my special thanks to my wife, Yuanyuan Wang. I would not have been able to completed my Ph.D thesis without her support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Code Reuse Problem in Scientific Computing</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Solving the Code Reuse Problem by Automatic Program Generation</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>11</td>
</tr>
<tr>
<td>2  Literature Review</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Generic Programming</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1 C++ Templates</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2 Generic Programming in Haskell</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Array Programming Languages</td>
<td>22</td>
</tr>
<tr>
<td>2.2 Metaprogramming Tools for Fortran</td>
<td>25</td>
</tr>
<tr>
<td>2.2.1 CTADEL</td>
<td>25</td>
</tr>
<tr>
<td>2.2.2 FORESYS</td>
<td>30</td>
</tr>
<tr>
<td>3  The Inverse Ocean Modeling System</td>
<td>32</td>
</tr>
<tr>
<td>3.1 Markovian Time Convolution</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Bell-Shaped Space Convolution</td>
<td>35</td>
</tr>
<tr>
<td>3.3 A Graphical User Interface for the IOM</td>
<td>38</td>
</tr>
<tr>
<td>4  Forge: A Domain-Specific Language for Defining Inverse Ocean</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Overview of Forge</td>
<td>42</td>
</tr>
<tr>
<td>4.1.1 Using Forge to Generate IOM Tools</td>
<td>45</td>
</tr>
<tr>
<td>4.1.2 The Forge Compiler</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Generating Fortran from Tool Descriptions</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1 Expressing Markovian Time Convolution in Forge</td>
<td>49</td>
</tr>
<tr>
<td>4.2.2 Interfaces and Dependent Indices</td>
<td>52</td>
</tr>
<tr>
<td>4.2.3 Summary: The Complete Tool Specification for the Markovian Convolution</td>
<td>56</td>
</tr>
<tr>
<td>4.2.4 Expressing the Bell-Shaped Space Convolution in Forge</td>
<td>57</td>
</tr>
<tr>
<td>4.2.5 Model Specifications</td>
<td>59</td>
</tr>
<tr>
<td>4.3 The Forge Type System</td>
<td>60</td>
</tr>
</tbody>
</table>
## TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1 The Typing Rules of Forge</td>
<td>62</td>
</tr>
<tr>
<td>4.3.2 Declaration Typing</td>
<td>70</td>
</tr>
<tr>
<td>4.3.3 Expression Typing</td>
<td>72</td>
</tr>
<tr>
<td>4.3.4 Statement Typing</td>
<td>74</td>
</tr>
<tr>
<td>5.1 Overview of Parametric Fortran</td>
<td>78</td>
</tr>
<tr>
<td>5.1.1 Array Addition for Arbitrary Dimensions</td>
<td>79</td>
</tr>
<tr>
<td>5.1.2 Realization of the Program Generation</td>
<td>80</td>
</tr>
<tr>
<td>5.1.3 Generic Array Slicing</td>
<td>86</td>
</tr>
<tr>
<td>5.1.4 Removing Duplicated Code</td>
<td>89</td>
</tr>
<tr>
<td>5.2 Implementing the IOM System in Parametric Fortran</td>
<td>92</td>
</tr>
<tr>
<td>5.2.1 Expressing Markovian Time Convolution in Parametric Fortran</td>
<td>92</td>
</tr>
<tr>
<td>5.2.2 Implementing the Bell-Shaped Space Convolution in Parametric Fortran</td>
<td>96</td>
</tr>
<tr>
<td>5.2.3 Benefits of Using Parametric Fortran in the IOM</td>
<td>98</td>
</tr>
<tr>
<td>5.3 Application of Parametric Fortran to Automatic Differentiation</td>
<td>100</td>
</tr>
<tr>
<td>5.3.1 Implementing Automatic Differentiation in Parametric Fortran</td>
<td>101</td>
</tr>
<tr>
<td>5.3.2 An Example: The Inviscid Burger’s Model</td>
<td>105</td>
</tr>
<tr>
<td>5.3.3 Differentiating the Inviscid Burger’s Model Using Parametric Fortran</td>
<td>106</td>
</tr>
<tr>
<td>5.4 Implementation of the Parametric Fortran Compiler</td>
<td>109</td>
</tr>
<tr>
<td>5.4.1 Representing Fortran Syntax in Haskell</td>
<td>110</td>
</tr>
<tr>
<td>5.4.2 A Type Class for Program Generation</td>
<td>112</td>
</tr>
<tr>
<td>5.4.3 Defining Parameter Types as Instances of \texttt{Param}</td>
<td>115</td>
</tr>
<tr>
<td>5.4.4 Transformation Functions</td>
<td>118</td>
</tr>
<tr>
<td>5.4.5 Implementation of Accessors</td>
<td>120</td>
</tr>
<tr>
<td>5.4.6 Implementation of List Parameters</td>
<td>123</td>
</tr>
<tr>
<td>6 Conclusion and Future Work</td>
<td>128</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

Bibliography 132

Appendices 138

Appendix A: The Syntax of Forge ........................................... 138
Appendix B: The Fortran Code of Markovian Time Convolution Generated by Forge for PEZ ........................................... 140
Appendix C: The Haskell Implementation of the Parameter Type \texttt{diff} . 141
Appendix D: The Installation of Parametric Fortran ................. 147
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Fortran code for the 2D simple convolution in time.</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>The Fortran code for the 3D simple convolution in time.</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>User Groups</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Summation of array elements.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Using <code>timeConv</code> on 3-D arrays.</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>The Fortran statement generated by CTADEL.</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>The FORESYS Graphical User Interface</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>The discrete equations of Markovian time convolution.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>The Fortran program of Markovian Time Convolution.</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Graphical User Interface</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>System architecture.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Using model parameters in tools and library function calls</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Fortran translation of discrete equations.</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>The Forge specification for Markovian time convolution.</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>The specification for the bell-shaped convolution in space.</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>Rules for environment judgments.</td>
<td>64</td>
</tr>
<tr>
<td>4.7</td>
<td>Array representations.</td>
<td>66</td>
</tr>
<tr>
<td>4.8</td>
<td>Typing rules for expressions.</td>
<td>67</td>
</tr>
<tr>
<td>4.9</td>
<td>Partial order on types</td>
<td>68</td>
</tr>
<tr>
<td>4.10</td>
<td>Type rules for Forge statements.</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Array addition for 2D arrays.</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Parametric Fortran syntax tree</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Syntax tree of {dim: c = a + b}</td>
<td>84</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4</td>
<td>Transformation of nodes</td>
<td>85</td>
</tr>
<tr>
<td>5.5</td>
<td>Syntax tree of generated for loop</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Parametric Fortran Template of an Array Slicing Subroutine</td>
<td>87</td>
</tr>
<tr>
<td>5.7</td>
<td>Removing Duplicated Code with Parametric Fortran</td>
<td>90</td>
</tr>
<tr>
<td>5.8</td>
<td>The Parametric Fortran Template for the Markovian Time Convolution</td>
<td>94</td>
</tr>
<tr>
<td>5.9</td>
<td>The Parametric Fortran Template for the Bell-Shaped Space Convolution</td>
<td>97</td>
</tr>
<tr>
<td>5.10</td>
<td>The Fortran Program for the Bell-Shaped Space Convolution</td>
<td>98</td>
</tr>
<tr>
<td>5.11</td>
<td>The Chain Rule for Computing the Tangent Linear Model</td>
<td>102</td>
</tr>
<tr>
<td>5.12</td>
<td>The Parametric Fortran Template for the Inviscid Burger’s Model</td>
<td>107</td>
</tr>
<tr>
<td>5.13</td>
<td>The adjoint program for the inviscid Burger’s model</td>
<td>109</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Parameterization Constructs</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Code savings achieved by Parametric Fortran</td>
<td>99</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>Syntax of specifications.</td>
<td>138</td>
</tr>
<tr>
<td>2</td>
<td>Syntax of interfaces.</td>
<td>139</td>
</tr>
<tr>
<td>3</td>
<td>Syntax of dependent-index declarations.</td>
<td>139</td>
</tr>
<tr>
<td>4</td>
<td>Forge types.</td>
<td>139</td>
</tr>
<tr>
<td>5</td>
<td>Syntax of parameter definitions.</td>
<td>139</td>
</tr>
</tbody>
</table>
Code reuse refers to the technique that allows a partial or complete computer program written at one time to be used in another program written at a later time. The goal of code reuse is to save time and energy of programmers by reducing redundant work. Code reuse is critical especially in those areas in which the involved software fragments are large and thus costly to reproduce. A very common example of code reuse is using software libraries. Common operations, such as accessing data in a specific format, processing data in a specific way (sorting, searching, etc), or I/O operations, are needed by many different applications. Developers of new programs can use the code in the software library to accomplish these tasks, instead of actually writing new code directly to perform the operations. Using libraries is a very effective form of code reuse. However, the use of libraries is often limited by the fact that library functions work only on specific data structures.

The most direct way to achieve code reuse is copying some or all of the code from an existing program into a new one. This approach is not preferable because code from the original program usually needs to be modified before being used in the new program, for example, variable names may need to be changed. Such modifications can easily introduce errors in the new program.

Code reuse is highly desirable in scientific computing. However, as a widely
used programming language in scientific computing, Fortran still lacks the support to achieve code reuse. This thesis makes contributions to the Fortran code reuse problem in scientific computing by applying program generation techniques.

In this introduction we first give a description of the code reuse problem in scientific computing. We also describe the program-generation approach to addressing this problem and then outline the rest of the thesis.

1.1 The Code Reuse Problem in Scientific Computing

In the area of scientific computing, the inability to properly reuse software has a particularly severe impact on the effective use of computers to support the scientific activities. Although there has been considerable work in scientific computing on efficiency issues, such as compilation and parallelization, the software engineering problems caused by large legacy software systems have been largely ignored so far.

Fortran is still the prevailing programming language in many areas of science. For example, ocean scientists have implemented ocean models in Fortran to simulate and predict the state of oceans, such as the Regional Ocean Model System (ROMS) [32] and the Advanced Circulation (ADCIRC) model [2]. Major reasons for using Fortran are the existence of efficient Fortran compilers for supercomputers that are needed to run large-scale simulations and the reliance on legacy software packages that are required for particular subsystems. At the same time, the reliance on Fortran causes severe software engineering problems, in particular,
the inability to reuse software. Although many projects incorporate previously written Fortran code, a closer look reveals that this “reuse” is coupled with high costs for adapting old Fortran programs. Moreover, the code reuse is performed in an ad hoc way that makes future reuse even more difficult.

Scientists often write data assimilation programs in Fortran to analyze scientific models with real observation data. Two examples are the Inverse Ocean Modeling (IOM) system [4, 17] and the Weather Research and Forecasting (WRF) model [20]. The algorithm of a particular data assimilation program is the same for all the scientific models it analyzes. However, in different scientific models, the representation of data may be different. For example, in the ROMS [32] and the ADCIRC model [2], data are stored in 4-dimensional arrays that contain 3 space dimensions and the time dimension. In the 2D Shallow Water model [27], data are represented as 3 dimensional which is 2-dimensional space plus the time dimension. Furthermore, even though the number of dimensions of the arrays storing the data is the same in the ROMS model and the ADCIRC model, the time dimension in these two models is located in different dimensions of the arrays. In the ROMS model, the time dimension is the first dimension of the arrays, whereas in the ADCIRC model, the time dimension is at the last dimension.

Since these data assimilation programs need to work for different scientific models, they have to be implemented in the way that can work with all the different data representations in the scientific models. For example, convolution tools that compute the average of weighted values are commonly used in data assimilation programs. The following formulas show a very simple form of con-
volution in the time dimension that computes the average of nearest neighbors in the forward direction (from 0 to $T$) for only one iteration.

$$b_0 = a_0$$

$$b_t = (a_{t-1} + a_t + a_{t+1})/3 \quad t = 1, 2, \ldots, T - 1$$

$$b_T = a_T$$

In the equations, the input array $a$ contains the original data. After the convolution the output $b$ contains the computed average values. The variable $t$ ranges over time, and the average is computed over only the direct neighbors of the current time. The lower bound and upper bound of the time dimension are 0 and $T$, respectively. The data stored in the arrays $a$ and $b$ contain a time dimension and space dimensions. In the convolution equations, only the time dimension of the arrays is shown. The space dimensions are implicit because they are not used for the computation.

This simple convolution tool can be applied to the data of different scientific models. However, in different scientific models, the data representation for the arrays can be different. For example, as mentioned above, the arrays can have different number of space dimensions in different scientific models. Figure 1.1 shows the Fortran implementation of the convolution tool for a 1-dimensional space.

For simplicity, we suppose that the elements of $a$ and $b$ are float numbers, the
subroutine timeConv2d (T, a, b)
  integer :: T
  real, dimension (0:T, 1:100) :: a, b
  integer :: t
  integer :: i1
  do i1 = 1, 100
    b(0,i1) = a(0,i1)
    do t = 1, T-1
      b(t,i1) = (a(t-1,i1)+a(t,i1)+a(t+1,i1))/3
    end do
    b(T,i1) = a(T,i1)
  end do
end subroutine timeConv2d

Figure 1.1: The Fortran code for the 2D simple convolution in time.

size of the space dimension is fixed to 100, and the time dimension is always the first dimension. Therefore, the only varying part in this convolution program is the number of space dimensions. In practice, the position of the time dimension and the size of each space dimension can also be different in different scientific models. We can observe that in line 3, the a and b are declared as 2-dimensional arrays that contain a time dimension and a space dimension. The statements from line 7 to line 11 implement the convolution equations and are wrapped by a loop over the space dimension. In line 5, the loop variable i1 is declared. All array expressions in the program contain indices for both time and space dimensions.

When the number of space dimensions of the array a is different, the convolution program is also different. Figure 1.2 shows the convolution program for
arrays that have two dimensions for space and one time dimension.

```fortran
subroutine timeConv3d (T, a, b)
    integer :: T
    real, dimension (0:T, 1:100, 1:100) :: a, b
    integer :: t
    integer :: i1, i2
    do i2 = 1, 100
        do i1 = 1, 100
            b(0,i1,i2) = a(0,i1,i2)
            do t = 1, T-1
                a(t,i1,i2) = (a(t-1,i1,i2)+a(t,i1,i2)+a(t+1,i1,i2))/3
            end do
            b(T,i1,i2) = a(T,i1,i2)
        end do
    end do
end subroutine timeConv3d
```

Figure 1.2: The Fortran code for the 3D simple convolution in time.

In the convolution program for 2-dimensional space the arrays a and b are declared as 3-dimensional. In line 5, two loop variables are declared. The assignments from line 8 to line 12 are wrapped by two loops over space dimensions. All the array expressions used in the program have three indices.

If the data assimilation system containing such a convolution tool needs to work for scientific models that have different number of space dimensions, it has to provide different versions of the convolution program, such as those shown in Figures 1.1 and 1.2, to different models. One way of implementing the data assimilation system is to rewrite the data assimilation programs for every individual scientific model, even though the underlying algorithm of the data assimilation
is principally the same for all scientific models. The disadvantage is that re-implementing the programs for every model is very tedious and error-prone, and the programs will be very difficult to maintain. This software engineering problem shows a great need for code reuse.

One approach to solving this code reuse 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 [13, 7]. One disadvantage of this approach is that developers of the scientific models 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 a scientific model wants to apply another data assimilation program with a different interface, the developers of the scientific model have to modify their model programs against the new interface again. Therefore, developers of scientific models seem to prefer re-implementing simulation tools specifically targeted for their model.

The code reuse problem of the data assimilation programs is not only caused by different data representations in scientific models, but can also be caused by other differences in models. In scientific models, state variables are used to represent the data of different measurements. For instance, in ocean models, state variables can include velocity and temperature of the ocean. Data assimilation programs often need to perform the same computation on all the state variables
in the scientific model they are applied to. In different scientific models, the number and meanings of state variables can be different, which makes writing the reusable data assimilation programs very difficult.

1.2 Solving the Code Reuse Problem by Automatic Program Generation

We provide two solutions to the code reuse problem in scientific computing by automatic program generation. The first solution is to design a high-level specification language for defining data assimilation tools, such as the convolution tool presented in Section 1.1. The tool specification can be generic with respect to the model-dependent information, such as the number of space dimensions for arrays. The model-dependent information is represented by parameters used in the generic specification. The compiler for the specification language, which is also a program generator, takes the specification of a data assimilation tool and the parameter values that represent the information of a scientific model as input, and generates a Fortran program that implements the data assimilation tool for that particular scientific model. A type system is also implemented for the specification language to ensure the generation of syntax- and type-correct Fortran programs. This approach is domain specific because the specification language is targeted for defining data assimilation tools.

The second approach is to design a language extension for Fortran to allow scientists to write Fortran program templates. In the program templates, For-
tran constructs, such as expressions or statements, can be parameterized. The parameters represent the varying aspects of data types, such as the dimension information of arrays, and other model-dependent information, such as the number and meanings of state variables in scientific models. When the model-dependent information is provided in the form of values for these parameters, a program generator can translate the program template into a specialized Fortran program that fits the particular scientific model. The language extension is not domain specific and can be applied to other applications other than data assimilation.

Therefore, by using the provided program generation techniques, developers of the data assimilation programs only need to implement their algorithm once and can generate different instances for different scientific models automatically. Three different groups of people are involved in the program-generation process. First, developers of data assimilation programs use the specification language or the language extension of Fortran to write their programs. Second, computer scientists implement the compiler of the specification language, that is, the program generator, and define how parameters affect the program generation. Third, the developers of scientific models provide the model information in the form of parameter values and will use the generated Fortran programs for their model. Figure 1.3 illustrates the principal interactions between the different user groups and the program generator.

In the left box, assimilation.hl is the tool specification or the program template for a data assimilation program. This data assimilation program needs to work for scientific models named M1 through Mn. The developers of these scientific
models provide the parameter values representing the information of their models. The program generator creates the different Fortran programs \texttt{assimilation\_1.f} through \texttt{assimilation\_n.f} for all the models and provides the generated data assimilation programs to the model developers. The program generator is written in Haskell \cite{29} by computer scientists. Note that the structure shown in Figure 1.3 is valid for both approaches. In the first approach, the \texttt{assimilation.tl} is defined in the specification language. In the second approach, \texttt{assimilation.tl} is implemented as a Fortran program template.
1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 gives a literature review of related work in programming languages. In Chapter 3, we describe an inverse ocean modeling system, called the IOM system, which is the major application of our program-generation approaches. Chapter 4 presents a domain-specific language that specifically targets the code reuse problem in the development of the IOM system. In Chapter 5, we describe the design and implementation of Parametric Fortran, a language extension of Fortran for supporting generic programming. Chapter 5 describes the application of Parametric Fortran to the IOM system and to automatic differentiation. Finally, we present some conclusions and future work in Chapter 6.
Chapter 2 – Literature Review

This dissertation contributes to software reuse in scientific computing, generic programming, and meta-programming in Fortran. In this chapter, we review the prior work in these areas that is most related to the thesis research. In Section 2.1, we review the support for generic programming in other programming languages. In Section 2.2, we review existing meta-programming tools for Fortran.

2.1 Generic Programming

Generic programming refers to the techniques that allow defining functions or writing programs that can work with different data types. Generic programming is an increasingly popular and important paradigm for software development and is meant to enable a higher level of code reuse. Generic programming supports program adaptation through generalization since it allows many algorithms to be implemented independently of the choice of the underlying data structures. Many programming languages provide support for generic programming. Perhaps the best known examples are the templates of C++ [21, 36].

Generic programming can be used for solving the code reuse problem in scientific computing. For example, the convolution tool presented in Section 1.1 depends on the data type of arrays it processes. By applying generic programming
techniques, that type-dependent convolution tool can be implemented. However, even though these generic programming techniques solve the code reuse problem to some extent, they are not dealing with Fortran, which is required by many scientific computing projects, such as the IOM system [4, 17].

The support for generic programming in Fortran is basically non-existent. The only support in Fortran 90 is ad-hoc polymorphism, which allows different subroutines to share the same name. In the recently published specification for Fortran 2003, the object-oriented programming paradigm is introduced. The support for template programming is also expected in future versions of Fortran. However, as we will discuss in Section 2.1, even with a C++-like template mechanism, the code reuse problem that is not caused by data type differences cannot be solved.

2.1.1 C++ Templates

C++ templates [21, 36] provide an effective way for defining generic functions. Suppose we want to write a function \texttt{sum} to sum up the elements in a 1-dimensional array. We require the function to be generic with respect to the element type so that it can be applied to arrays of different types. By using C++ templates, the generic function \texttt{sum} can be defined in Figure 2.1.

In this function definition, the type parameter \( T \) represents the element type of the array whose elements need to be summed up. The elements in the array are addressed by pointers. The function \texttt{sum} takes two arguments, \texttt{first} and \texttt{last},
template <class T>
T sum(T* first, T* last) {
    T s = 0;
    while (first != last)
        s = s + *first++;
    return s;
}

Figure 2.1: Summation of array elements.

specifying the start location and end location of the array, respectively. Both arguments have the type T*. The function returns the sum of all the elements between these two locations, and the return type is T. When the function sum is applied to an array, the C++ compiler generates a specialized version of the generic function that is instantiated by the type parameter T at compile time.

The sum function illustrates how parametric polymorphism can be realized through C++ templates. However, parametric polymorphism is not expressive enough to capture the genericity that is required for the reusable implementation of the convolution tool shown in Section 1.1.

To address this limitation, some C++ template libraries targeted at solving the code reuse problems in scientific computing provide class templates for multi-dimensional arrays. For example, the Boost.MultiArray template library [31] and Blitz++ [39] can be used to define array types parameterized by the element type and the number of dimensions. Generic functions can then be defined on the parameterized array types. For example, Blitz++ provides an array template class, called Array<T_numtype,N_rank>. This array class provides a dynamically
allocated N-dimensional array. The \texttt{Array} class takes two template parameters, \texttt{T\_numtype} and \texttt{N\_rank}. \texttt{T\_numtype} is the element type of the array. \texttt{N\_rank} represents the number of dimensions of the array, which should be an integer constant that must be known at compile time. The space for an array is allocated when the array is constructed. For example, the following code constructs a 2-dimensional array \texttt{a} of float numbers. The size of each dimension of that array is \texttt{N}.

\begin{verbatim}
Array<float,2> a(N,N);
\end{verbatim}

Array elements can be referred to by indices. For example, the expression \texttt{a(0,0)} represents the first element of the array \texttt{a}. The expression \texttt{a(0)} represents the first row of the 2-dimensional array.

The array template class provided by Blitz++ can be used to implement generic algorithms that depend on array types. For example, the convolution program introduced in Section 1.1 can be implemented using Blitz++ by the function \texttt{timeConv} as follows.

\begin{verbatim}
#include <blitz/array.h>
using namespace blitz;

typedef Array(float, DIM) A;

timeConv (int T, const A& a, A& b){
    b(0) = a(0);
    for (i=1; i<T; i++)
        b(i) = (a(i-1)+a(i)+a(i+1))/3;
    b(T) = a(T);
}
\end{verbatim}
To use the `Array` class, we need to include the header file `<blitz/array.h>` and use the C++ namespace `blitz`. In the program we define a type synonym `A` for the array type that contains float numbers and has `DIM` dimensions. The value for the integer constant `DIM` must be known at compile time. The function `timeConv` implements the convolution tool introduced in Section 1.1. The function takes an integer argument `T` that specifies the upper bound of its first dimension. Both arguments `a` and `b` have the same array type `A`. For performance reasons, the array arguments are passed by reference. The argument `a` is declared as constant since it is the input array and its value will not be changed. The array `b` is the output array and will contain the computed average values after the function is executed.

The following code shows how the function `timeConv` can be used for arrays that have one time dimension and one space dimension.

```cpp
const int DIM = 2;
typedef Array(float, DIM) A;
A a(T+1,100), b(T+1,100);
// initialization of the array a
...
timeConv(T, a, b);
```

In the above program fragment, the value for the constant `DIM` is 2. The arrays `a` and `b` have two dimensions. The size of the first dimension, which is the time dimension, is `T+1`. The size of the second dimension, which is the space dimension, is 100. The function `timeConv` can then be called to perform the convolution.

If the arrays `a` and `b` have two space dimensions, the code in Figure 2.2 performs the convolution. We only need to change the value of `DIM` and the
declaration of a and b. The implementation of the function timeConv does not need to be changed.

```c++
const int DIM = 3;
typedef Array(float, DIM) A;
A a(T+1,100,100), b(T+1,100,100);
//initialization of the array a
...
timeConv(T, a, b);
```

Figure 2.2: Using timeConv on 3-D arrays.

The Blitz++ implementation of the convolution tool is independent of the number of space dimensions of the input and output arrays. However, the array template class in Blitz++ cannot parameterize the position of the time dimension. Moreover, C++ templates are an example of the type-based generic programming. C++ templates are able to implement the generic algorithms that only depend on data representations. The algorithms that not only depend on data types but also on the other model-dependent information are impossible to express by C++ templates. For example, C++ templates cannot implement the data assimilation program that perform the same computation on varying number of state variables.

2.1.2 Generic Programming in Haskell

The Haskell programming language [29] has parameterized types, parametric polymorphism, and type classes, which together support a style of programming
somewhat similar to what is possible C++ templates. Haskell also has some more unusual generic programming features, which have inspired programmers and language designers to strive for even more genericity and code reuse than polymorphism can provide. For example, the “Scrap Your Boilerplate” approach [24] and Generic Haskell [25] allow defining generic functions that can work on any data type.

Haskell supports array types that can be parameterized by the index type and element type. If \( i \) is an index type and \( e \) is any type, the type of arrays with indices in \( i \) and elements in \( e \) is written as \( \text{Array} \ i \ e \). An array can be created by the function \( \text{array} \). The first argument of \( \text{array} \) is a pair of bounds, each of the index type of the array. These bounds are the lowest and highest indices in the array, in that order. For example, an array of length 3 has bounds \((1,3)\), and a 3 by 3 matrix has bounds \(((0,0),(2,2))\). The second argument of \( \text{array} \) is a list of index-value pairs. A pair \((i,x)\) defines the value of the array at index \( i \) to be the value \( x \).

For example, the following code shows the definitions of two arrays.

\[
\text{a1} :: \text{Array} \ \text{Int} \ \text{Float} \\
\text{a1} = \text{array} \ (1,3) \ [(1,1.1),(2,5.3),(3,6.0)]
\]

\[
\text{a2} :: \text{Array} \ (\text{Int,Int}) \ \text{Int} \\
\text{a2} = \text{array} \ ((1,1), (3,3)) \ [(\text{i,j}, \text{i*j}) \mid \text{i<-}[1..3], \text{j<-}[1..3]]
\]

The array \( \text{a1} \) is a one-dimensional array and contains three float numbers. The array elements can be referenced by an integer number. For example, \( \text{a1!2} \) refers to the second element of \( \text{a1} \), which is \( 5.3 \). The 2-dimensional array \( \text{a2} \) contains nine integers and represents a multiplication table that maps each index pair
(i, j) to its product i*j. The index of type of a2 is (Int, Int) which means elements in a2 are referenced by a pair of integers. For example, a2!(2, 3) is 6.

Generic functions that are independent of array types can be defined by using the parameterized Array type. For example, we can define a function sumA to sum up the elements of an array as follows. The function can be applied to arrays with different dimensionality and element types. Dimensionality of an array means its number of dimensions and the size of each dimension.

```
sumA :: (Ix i, Num e) => Array i e -> e
sumA = sum . elems
```

The function sumA takes an array whose index type is represented by the type variable i and the element type is represented by the type variable e. The index type must be an instance of the built-in type class Ix. The Ix class is used to map a continuous subrange of values in an instance type onto integers. For example, Int and (Int, Int) all are instances of the Ix class and can be used as the array index type. The element type must be an instance of the type class Num so that we can perform the sum function on the array elements. For example, Float and Int are all instances of the Num class. The function sum computes the sum of the numbers in a list and has the following type.

```
sum :: Num a => [a] -> a
```

In the definition of sumA, we first apply the function elems to the input array to obtain the list of its elements. The function elems is a built-in Haskell function and has the following type.
elems :: (Ix a) => Array a b -> [b]

The sum of the array elements is then computed by applying the \texttt{sum} function to the resulting list.

The function \texttt{sumA} is generic with respect to the element type and the dimensionality of the input array. For example, the function \texttt{sumA} can be applied to the previously defined arrays \texttt{a1} and \texttt{a2} without any adaptations. The key part of defining this \texttt{sumA} function is the generic function \texttt{elems} that can transform arrays of any dimensionality into a list. The function \texttt{elems} can be defined because a Haskell array is constructed by a list of index-value pairs. The \texttt{elems} function just returns a list containing all the values from the index-pair list by omitting the indices.

Generally, the \texttt{Array} type does not support defining functions that are generic with respect to array dimensionality. For example, the generic convolution tool introduced in Section 1.1 cannot be implemented with the \texttt{Array} type alone. The following shows the Haskell version of the convolution tool for arrays that have one time dimensions and one space dimension.

```
type Array2D = Array (Int,Int) Float

conv2d :: Int -> Array2D -> Array2D
conv2d t a = array ((0,1), (t,100))
        [([(0,x), a!(0,x)) | x<-[1..100]] ++
         [((t,x), a!(t,x)) | x<-[1..100]] ++
         [((i,x), (a!(i-1,x)+a!(i,x)+a!(i+1,x))/3) | x<-[1..100], i<-[1..t-1]])
```

The convolution tool is implemented by the function \texttt{conv2d}, which takes an
integer argument representing the upper bound of the time dimension, and a 2-dimensional array as input, and returns a 2-dimensional array containing the averaged values. The result array is constructed by using the function \texttt{array}. The size of the space dimension is 100 for simplicity. Therefore, the index of the result array ranges from \((0,1)\) to \((t,100)\). The second argument of the function \texttt{array} is the list containing the index-value pairs of the result array. Similarly, the function \texttt{conv3d} implements the convolution tool for arrays with two space dimensions and is shown below.

\begin{verbatim}
type Array3D = Array (Int,Int,Int) Float
conv3d :: Int -> Array3D -> Array3D
conv3d t a = array ((0,1,1), (t,100,100))
  [((0,x,y), a!(0,x,y)) | x<-[1..100], y<-[1..100]] ++
  [((t,x,y), a!(t,x,y)) | x<-[1..100], y<-[1..100]] ++
  [((i,x,y), (a!(i-1,x,y)+a!(i,x,y)+a!(i+1,x,y))/3)
   | x<-[1..100], y<-[1..100], i<-[1..t-1]]
\end{verbatim}

The \texttt{Array} type has two limitations for implementing a generic version of the convolution tool. First, array elements can only be accessed with the full index value. For example, for a 2-dimensional array of the type \texttt{Array2D}, the elements can only be referenced by a pair of integers. The \texttt{Array} type does not support using a single integer as index for a 2-dimensional array to get a 1-dimensional array, as in the C++ Blitz library shown in Section 2.1.1. Second, mathematical operations are not supported at the array level. For example, if \(a\) and \(b\) are two arrays with the same element type and dimensionality, \(a+b\) is not supported. These limitations prevent us from implementing the generic convolution tool with respect to the dimensionality of arrays. Although generic programming for array
dimensionality can be achieved by extending the \texttt{Array} type, the changes are not trivial.

Another disadvantage of using Haskell in scientific computing is that Haskell’s performance is difficult to predict due to lazy evaluation.

2.1.3 Array Programming Languages

The fundamental idea behind \textit{array programming} \cite{35} is that operations apply at once to an entire set of values. Therefore, array programming is a high-level programming model as it allows the programmer to think and operate on whole aggregates of data, without having to resort to explicit loops of individual scalar operations. Array programming languages, such as APL \cite{18} and J \cite{19, 16}, provide built-in operations for computing with variable-dimensional arrays. For example, the function that sums all elements in an array can be simply expressed by the following code in J.

\[
\text{sum} =: +/
\]

The operator \(+\) is called a \textit{verb}, which corresponds to functions in other programming languages. The operator \(/\) is an \textit{adverb} that applies to its verb argument to produce a new verb. Applying \(/\) to the verb \(+\) causes the \(+\) operator to be applied to all the elements in an array. The expression \(+/ \ 1 \ 2 \ 3 \ 4\) is equivalent to \(1+2+3+4\). Similarly, the expression \(*/ \ 1 \ 2 \ 3 \ 4\) is equivalent to \(1*2*3*4\).

A unary verb such as \(+/\) takes only one argument. Two unary verbs can be combined by a binary verb. For example, the binary verb \(\%\) performs the division
operation on its two arguments, 2%5 yields 0.4. The operator # is another unary verb that applies to an array and returns the number of elements in the array. We can define a verb computing the average of an array as follows.

\[
\text{mean} =: +/%# 
\]

When the verb +/%# applies to an array, it first computes the sum of the array elements using the first unary verb +/, then uses the sum to divide the number of elements in the array.

The convolution tool introduced in Section 1.1 can be implemented in J as follows.

\[
\text{timeConv} =: 3 : '(3 (+/%#)"1 y.)(<<0 _1)}"1 y.'
\]

The syntax may look a little weird because it is very terse. timeConv is defined as a unary verb. The representation

\[ 3 : '...' \]

represents that the script quoted by ’s is used to define a verb. In the quoted part,

\[ (3 (+/%#)"1 y.) \]

performs the computation of the moving average which averages successive 3 elements of the right argument. A unary verb can have its argument on either the left side or the right side. The variable \( y \) represents the right argument of timeConv. The notation "1 used in the definition represents that the computation is performed on the first dimension of the right argument. The remaining part
assigns the first and last element of the result array with the first and last element of the right argument of \texttt{timeConv}, respectively. The defined unary verb \texttt{timeConv} can be applied to arrays with any number of dimensions and performs the convolution on the first dimension. For example, we can define a 2-dimensional array \texttt{a} as follows.

\begin{verbatim}
a =: 90 47 58 29 22 32
  55 5 55 73 58 50
  40 5 69 46 34 40
  46 84 29 8 75 97
  24 40 21 82 77 9
\end{verbatim}

We can then apply the verb \texttt{timeConv} to the array \texttt{a} and store the result in \texttt{b} as follows.

\begin{verbatim}
b =: timeConv a
\end{verbatim}

The following shows the content of the array \texttt{b}.

\begin{verbatim}
90 65 44.6667 36.3333 27.6667 32
55 38.3333 44.3333 62 60.3333 50
40 38 40 49.6667 40 40
46 53 40 3333 37.3333 60 97
24 28.3333 47.6667 60 56 9
\end{verbatim}

In the result array, the first column and last column are the same as the input array \texttt{a}. The four middle columns are replaced with the average values.

Array programming languages, such as APL and J, provide the support for implementing generic algorithms that depend on the dimensionality of arrays.
However, APL and J only provide efficient operations for array processing, but not for efficient numerical computations, which prevents them to be widely used in the scientific computing area. Moreover, the dense syntax of array programming languages is generally not considered very user friendly.

2.2 Metaprogramming Tools for Fortran

Program generation is essentially a metaprogramming technique. For a comprehensive overview over the field, see [33]. Although Fortran is a widely used programming language in scientific areas and has a large user community, little work has been done in metaprogramming for Fortran. In the existing metaprogramming tools for Fortran, no one can be used for solving the Fortran code reuse problem.

In this section, we present two existing metaprogramming tools for Fortran. In Section 2.2.1, we describe the CTADEL system that is used for automatic generation of Fortran code solving partial differential equations. We review the Fortran transformation tool FORESYS in Section 2.2.2.

2.2.1 CTADEL

The CTADEL system [38, 37] is a code generation tool for applications based on partial differential equations (PDEs). The CTDEL system can automatically generate efficient Fortran programs solving the PDEs that are defined using a
high-level language specification. The translation of a PDE specification into Fortran code consists of three main phases.

1. The PDE defined in the high-level specification is discretized. The resulting discrete equations are input to the next phase.

2. The discrete equations obtained in the first phase are transformed into an intermediate representation that are mainly assignments. The assignments are then simplified and optimized by employing global common subexpression elimination.

3. Finally, Fortran code is generated from the intermediate representation obtained from the second phase.

For using the CTADEL specification language to define a PDE, domains need to be defined first. A domain defines a set of possible values for variables. A domain basically consists of an index variable and a range of integer values for that variable. For example, the latitudinal domain can be specified by the variable \( j \) and the range \( 1..J \). Some example domain definitions are shown below.

\[
\begin{align*}
\text{longitudinal: domain} & := i = 1 .. I. \\
\text{latitudinal : domain} & := j = 1 .. J. \\
\text{vertical} & := k = 1 .. K.
\end{align*}
\]

Domains can be composed to form new domains. For example, the horizontal domain is formed by composing the longitudinal and latitudinal domains.

\[
\begin{align*}
\text{horizontal} & := \text{longitudinal by latitudinal}. \\
\text{global} & := \text{horizontal by vertical}.
\end{align*}
\]
Domains can also be composed by concatenating domains from right to left where the bounds of a domain on the left may depend on the index variable of a domain on the right. For example, the following represents a triangular domain.

\texttt{hori\_triangular = i=j..n by j=1..m}

Variables can be associated with fields. A field of a variable is defined by associating coordinates with domains. The following code defines three variables with their associated fields.

\texttt{u: field(x(half),y(grid),z(grid)) on global.}
\texttt{v: field(x(grid),y(half),z(grid)) on global.}
\texttt{p: field(x(grid),y(grid),z(grid)) on global.}

In the above definitions, \(x\), \(y\), and \(z\) are coordinates. The coordinates are parameterized with \texttt{grid} or \texttt{half} to represent that the field is placed on a whole or half grid point.

Next we illustrate CTADEL by an example. The following partial differential equation is used for the surface pressure tendency in the High Resolution Limited Area Model [22].

\[ \frac{dp}{dt} = - \int_{\text{vertical}} \nabla \cdot (V \frac{\partial p}{\partial z}) \]

In the above equation, \(V = [u, v]^T\) is a horizontal vector, and \(\nabla\) is the two-dimensional horizontal divergence operator. The variables \(u\), \(v\), and \(p\) are defined previously on the \texttt{global} domain. Therefore they will be represented by 3-dimensional arrays in Fortran programs. The variable \(z\) in the equation represents the coordinate on the vertical direction. More details about the meaning
of this equation can be found in [22]. The PDE can be specified in CTADEL as follows.

\[
\text{vector } V := [u, v].
\]
\[
dps: \text{field}(x(\text{grid}), y(\text{grid})) \text{ on horizontal.}
\]
\[
dps = -\text{int}(\nabla(V * df(p, z)), \text{vertical}).
\]

The specification can then be automatically discretized by CTADEL and transformed into an intermediate representation that is not shown to the user. It would look as follows.

\[
t_{1i,j,k} = \frac{1}{4} u_{i,j,k}
\]
\[
t_{2i,j,k} = p_{i,j,k} t_{1i,j,k}
\]
\[
t_{3i,j,k} = p_{i+1,j,k} t_{1i,j,k}
\]
\[
du_{i,j,k} = 
\begin{cases} 
(t_{2i,j,k} + t_{3i,j,k})/h_k & \text{if } k \leq 1 \\
(t_{2i,j,k-1} + t_{3i,j,k-1})/h_k & \text{if } k \geq K \\
(t_{2i,j,k} + t_{3i,j,k})/h_k + t_{2i,j,k-1} + t_{3i,j,k-1})/h_k & \text{otherwise}
\end{cases}
\]

The variables \(t_{1i,j,k}, t_{2i,j,k}, \) and \(t_{3i,j,k}\) are temporary variables generated in the process of discretization. \(du\) is used for computing the derivative of \(V\) by \(dV = [du, dv]^T\).

There are also similar equations for calculating \(dv\). The variables \(du\) and \(dv\) are declared automatically by CTADEL as follows.

\[
du: \text{field}(x(\text{half}), y(\text{grid}), z(\text{grid})) \text{ on global.}
\]
\[
dv: \text{field}(x(\text{grid}), y(\text{half}), z(\text{grid})) \text{ on global.}
\]
Details about the discretization and transformation can be found in [38].

The intermediate representation will finally be translated into Fortran statements. Every equation in the intermediate representation is automatically translated into a Fortran assignment statement or a loop statement. For example, the equation

\[ t_{1_{i,j,k}} = \frac{1}{4} u_{i,j,k} \]

is translated into the Fortran statement in Figure 2.3.

```fortran
    do i = 1, I
        do j = 1, J
            do k = 1, K
                t1(i,j,k) = u(i,j,k)/4
            end do
        end do
    end do
```

Figure 2.3: The Fortran statement generated by CTADEL.

By allowing users to define the PDEs in a high-level specification language and generating the Fortran programs automatically, the CTADEL system supports scientists in programming. We have followed similar approach by designing a domain-specific language for inverse ocean modeling, which will be described in Chapter 3. On the other hand, the CTADEL system focuses on generating efficient Fortran code for solving PDEs and does not deal with the code reuse problem in scientific computing.
2.2.2 FORESYS

The Fortran Engineering System (FORESYS) [34] provides metaprogramming tools whose focus is on the refactoring of existing Fortran code. FORESYS provides a set of tools towards analysis, restructuring, and transforming existing Fortran 77 and Fortran 90 programs. The provided tools can be used for code refactoring, automatic parallelization and for transformation of Fortran 77 programs into Fortran 90 programs. However, FORESYS does not provide any support for generic programming in Fortran and cannot provide a solution for the code reuse problem.

The tools in FORESYS are based on a sophisticated data dependency analysis and are currently targeting Fortran 90 array syntax and High Performance Fortran (HPF). Program transformations are performed through a graphical user interface. For example, Figure 2.4 shows the transformation of Fortran 77 code into Fortran 90 code.

The Fortran 77 program is in the left window in Figure 2.4. The FORESYS transformation tool performs a data dependency analysis on the program and generates the data dependency graph in the right window. Users can browse and modify the generated graph and use it to generate a Fortran 90 program.

The FORSYS system provides the transformation tool for the automatic parallelization of Fortran code. For example, the following Fortran code fragment shows a loop that performs a simple computation on array a.
do i=start, end
    a(i) = d + a(i)
end do

Using the FORESYS tool, the OpenMP version of the above loop can be generated automatically.

!$OMP PARALLEL DO
!$OMP& PRIVATE(i) SHARED(a)
do i = start, end
    a(i) = d + a(i)
end do
!$OMP END PARALLEL DO

In the parallel program the OpenMP directives around the loop to specify that the array a is shared by different processors and the loop variable i is private to each processor.
Chapter 3 – The Inverse Ocean Modeling System

Ocean scientists use ocean models to simulate and to predict the state of oceans. The Inverse Ocean Modeling (IOM) system [4, 17] is a data assimilation system that can be used to analyze and improve ocean models by combining the ocean models with real observations of the ocean. The output of the IOM system is a weighted least-squared best-fit to the equations of motion and to the data. The IOM system can provide important information about the quality of data produced by ocean and weather forecasting models. The accuracy of ocean and weather forecasting models is important for the successful planning of flight or ship routes, navy operations, and many other applications.

The IOM system consists of tools that are used for solving the equations of best fit to an ocean model and simultaneously to a set of observations. Most IOM tools depend on the data structures used in ocean models. Therefore, the IOM system needs to provide different programs implementing the same tool to different ocean models. The IOM tools are composed into inversion programs that perform the data assimilation. One particular problem is that the IOM tools and the inversion programs have to be rewritten for each individual ocean model, although the process of inversion is principally the same for all models (at least for one chosen inversion algorithm).

In Sections 3.1 and 3.2 we present two examples of IOM tools. The presented
tools are all model dependent and therefore need to be re-implemented for different ocean models. In Section 3.3 we describe a graphical user interface for the IOM system. The graphical user interface allows users of the IOM system to input parameters describing ocean models to control the generation of IOM programs, to configure the environment for running the IOM software, and to customize the output of data assimilation programs.

3.1 Markovian Time Convolution

Convolution is essentially the process of smoothing a sequence of weighted values. We have shown a very simple example in Section 1.1. 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 function defines a new convolution. Every convolution has the following form.

\[ b(x) = \int_{0}^{X} F(x, x') a(x') dx' \]

\( F(x, x') \) 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 [3]. The convolution tool described in Section 1.1 can also be represented in this form. The function
$F$ is then a non-continuous function.

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

$$b(t) = \int_0^T \exp(-|t-t'|/\tau) a(t') dt'$$

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 $\tau$ is the correlation time scale, which is provided by the ocean modelers when they use the tool. The smaller $\tau$ 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 corresponding discrete equations shown in Figure 3.1, that can be derived from the continuous one through techniques developed in [4]. 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
\[
\begin{aligned}
    h_L &= 0 \\
    \frac{h_n - h_{n-1}}{\Delta t} + \tau^{-1}h_{n-1} &= -2\tau^{-1}a_n \\
    b_U &= -(\tau/2)h_U \\
    \frac{b_{n+1} - b_n}{\Delta t} - \tau^{-1}b_{n+1} &= h_n
\end{aligned}
\]

Figure 3.1: The discrete equations of Markovian time convolution.

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.

The code in Figure 3.2 shows the Fortran subroutine of the Markovian time
convolution for the model in which the space has one dimension, the size of the
space dimension is 100, and the time dimension is the first dimension. In the
program, the arrays \( a \), \( b \), and \( h \) all have two dimensions. The first dimension
of these arrays is the time dimension and the second one represents space. The
body of the subroutine implements the discrete equations and is wrapped by a
loop over the space dimension.

3.2 Bell-Shaped Space Convolution

We consider another IOM tool that depends not only on the data structure in
ocean models, but also on the algorithms used in different ocean models. Similar
to the Markovian convolution, the \textit{bell-shaped space convolution} is also a convo-
subroutine timeConv (L, U, dt, tau, a, b)
    integer :: L, U
    real    :: dt, tau
    real, dimension (L:U, 1:100) :: a, b, h
    integer :: n
    integer :: i1
    do i1 = 1, 100
        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 3.2: The Fortran program of Markovian Time Convolution.

In the above continuous formula, $a$ contains the input values, and $b$ contains the result values. The weighting function $F(x,x')$ is $\exp(-(x-x')^2/L^2)$ where $x$ and $x'$ range over space. $L$ is the correlation length scale, a parameter to be specified by users of the tool. The smaller $L$ is, the more quickly the values of far neighbors will be omitted. The bell-shaped convolution can be discretized using the explicit
time-stepping scheme [9] to the following discrete equations.

\[
\theta_i^0 = a_i
\]

\[
\theta_i^{n+1} = \theta_i^n + \Delta s \cdot RHS_i^n \quad \text{for } 0 \leq n \leq N - 1
\]

\[
b_i = \theta_i^N
\]

\(\theta\) is an intermediate array for solving the final result \(b\). More specifically, \(\theta_i^n\) represents the \(i\)th element of the array \(\theta\) in the \(n\)th iteration. The final value of the output array is calculated in the \(N\)th iteration, where \(N\) is the number of iterations for computing the final result and given by \((L^2/4)/\Delta s\). \(\Delta s\) is the time step used in the explicit time-stepping method for computing the next \(\theta\) from the current \(\theta\). \(RHS_i^n\) is computed by a Laplacian operator. In different ocean models, the Laplacian operator can be implemented differently. Developers of ocean models prefer to implement their own Laplacian operator, instead of letting the IOM generate the Laplacian operator for them, because the implementation of the Laplacian operator usually involves optimizations that are very specific to each individual ocean model. In the IOM implementation for the bell-shaped space convolution, the Laplacian operator is represented by a Fortran subroutine provided by the users of the IOM system, that are also the developers of ocean models.

It is not really important to understand why the shown discrete equations are a solution for the continuous one, and even less how they can be derived. The
two important aspects this example is intended to demonstrate are:

- Discrete equations can differ considerably from the continuous equations.
- We need a mechanism for including model-dependent code in IOM tools, such as the Laplacian operator used in the bell-shaped space convolution.

These aspects of IOM tools ask for a design of a mechanism that is (1) expressive to specify all the tools the IOM provide and (2) flexible to allow for customizations that can take into account parameters for particular ocean models.

### 3.3 A Graphical User Interface for the IOM

We have implemented a Graphical User Interface (GUI) that can generate IOM instances for different ocean models automatically. In the early version of the GUI, IOM tools are defined in a domain-specific language called Forge [11]. In the specifications of IOM tools, parameters are used to represent model-dependent information, such as the number of space dimensions and the subroutine implementing the Laplacian operator. Developers of ocean models input the values for parameters through the GUI, and the GUI generates the IOM tools for their ocean model automatically. Although Forge can be used to generate IOM tools, it is designed to only deal with tools that can be represented by discrete equations. Some IOM programs cannot be represented by discrete equations, such as the IOM main program, and the program that runs the convolution tools on all state variables of an ocean model. Therefore, Forge is not expressive enough to
specify the whole IOM system in a generic way.

In the current version of the GUI, we have replaced Forge with an extension of Fortran that supports template programming, called Parametric Fortran [10, 12]. The IOM tools, such as the Markovian convolution, and other model-dependent programs, are written as program templates in Parametric Fortran, 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. The whole IOM system is now implemented in Parametric Fortran and can be generated automatically.

The GUI also presents ocean modelers with a variety of inversion options, such as the selection of inversion outputs and the choice of different inversion algorithm options, to guide the combination of the generated IOM programs into complete data assimilation programs. The GUI also controls system configurations, such as Makefile and Fortran compiler options, for compiling and running the generated programs. Previously, the GUI supported three different execution options of the IOM system: 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. In the current version, only the last execution option is supported because the other two options require developers of ocean models to use the parallel infrastructure of the IOM, which most of the current users do not want to use.

The graphical user interface is implemented in Java and runs on Windows,
Mac OS, and Sun Solaris. A snapshot of the current system is shown in Figure 3.3.

<table>
<thead>
<tr>
<th>Task</th>
<th>System</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization algorithms</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>- Direct representer</td>
<td>- Number of space dimensions</td>
<td></td>
</tr>
<tr>
<td>- Indirect representer</td>
<td>- Position of the time dimension</td>
<td></td>
</tr>
<tr>
<td>- Ensemble method</td>
<td>- Sizes of the space dimensions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- ComputeRHS file</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fields</th>
<th>Residuals</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- First-guess</td>
<td>- Model residuals</td>
<td></td>
</tr>
<tr>
<td>- Best-estimate: state</td>
<td>- Observational residual for first-guess</td>
<td></td>
</tr>
<tr>
<td>- Clipped best-estimate: state</td>
<td>- Observational residual for best-estimate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Statistics</th>
<th>Array assessment</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Prior penalty</td>
<td>- Representer matrix</td>
<td></td>
</tr>
<tr>
<td>- Reduced penalty</td>
<td>- SVD</td>
<td></td>
</tr>
<tr>
<td>- Expected value/variance of penalties</td>
<td>- Array modes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prior error covariances</th>
<th>Posterior error covariances</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Representers</td>
<td>- State</td>
<td></td>
</tr>
<tr>
<td>- State</td>
<td>- Model</td>
<td></td>
</tr>
<tr>
<td>- Model</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Graphical User Interface.

The fields in the “Parameters” panel of the GUI specify parameters used in program templates for generating IOM programs. The values for the parameters represent information of ocean models. For example, the first field in the panel represents number of space dimensions of arrays in ocean models. The “Parameters” panel is scrollable which means there are more parameters than shown in this Figure. How the parameters can affect the program generation will be shown
in Chapter 5.

Other options shown in Figure 3.3 can customize the data assimilation program that users of the IOM want to run. For example, users can select different optimization algorithms for the generated data assimilation program from the panel on the upper-left. Selecting a particular optimization algorithm makes the GUI generate IOM tools required by that optimization algorithm and combine the generated tools into the data assimilation program. The automatic customization/generation of IOM data assimilation programs is achieved by defining the data assimilation programs as Parametric Fortran templates. The ability to provide parameters representing model information and to customize the algorithm of data assimilation offers to users of the IOM a flexible customization of data assimilation programs.
Chapter 4 – Forge: A Domain-Specific Language for Defining
Inverse Ocean Modeling Tools

In this chapter, we present a program-generation approach to address the software-reuse challenge in the area of scientific computing. More specifically, we describe the design of a domain-specific language called *Forge* [11] for the specification of subroutines that can be generic in the dimensions of arrays, parameter lists, and called subroutines. We also present the application of that approach to the IOM system described in Chapter 3. In addition to a compiler that can transform generic specifications into efficient Fortran code for models, we have also developed a type system that can identify possible errors in the specifications. This type system is important for the acceptance of the program generator among scientists because it prevents a large class of errors in the generated code.

4.1 Overview of Forge

In Chapter 3 we described the IOM system that faces the code reuse problem. In particular, the IOM project poses the following challenges.

1. IOM tools should be implemented or specified only once.

---

1 This chapter is mainly based on [11].
2. Tool specifications should be readily usable, that is, scientists should not be required to modify the inversion tools.

3. Existing programs for ocean models that the IOM tools apply to should be reusable without needing any changes.

The approach we have taken is driven by the goal to leave existing programs for ocean models basically unchanged. To this end, we capture the specifics of each model by a collection of parameters that are sufficient to guide the adaptation of the IOM tools to that particular ocean model. We have developed a language called Forge (an acronym for “Fortran generator”), which allows the specification of Fortran subroutines in a generic way that fixes the general structure of the subroutine, but may leave open some details, such as the dimensions of arrays, the number of nested levels of loops, or parameters of the subroutine. By using model parameters that represent the information of ocean models in these specifications, the dependency of the corresponding program parts on the ocean-model specifics can be expressed. Given a concrete set of values for the parameters, a Fortran program can be generated from the Forge specification.

Although the program generator was specifically designed to be used with inversion tools for ocean modeling, the scope and applicability of Forge are much larger. Inversion includes medical imaging, seismic exploration, retrieval of atmospheric profiles from satellite soundings, assimilation of initial data into operational weather forecast, and, in our example application, the testing of hypotheses about the dynamics of ocean circulation. Although inversions can be used for dif-
ferent objectives, they are mathematically identical. Therefore, Forge can also be applied to other scientific areas for defining inversion tools.

Another important aspect of our program-generator approach is that it has to integrate the work of people who work in different areas. This was achieved by a modular design, which plays a key role, because Forge actually consists of three sublanguages, a language for specifying the tools, a very simple one for specifying characteristics of particular ocean models, and a language for defining program combinators that can be used in the tool specifications. Each sublanguage is targeted at a different user group, namely developers of ocean models, inverse ocean-modeling software developers, and computer scientists. Each sublanguage offers each group of users the abstractions needed to express their part of inverse ocean modeling tools. The fact that these languages are integrated into a program generating system allows these three user groups to work together while still being able to focus on their particular professional skills.

In the remainder of this section, we briefly describe the background of the application area and the program-generation approach. In Section 4.2 we show how to generate Fortran code for IOM tools through two examples. We will describe a type system to prevent type errors in the generated Fortran programs in Section 4.3.
4.1.1 Using Forge to Generate IOM Tools

IOM tools can be specified by tool specifications in Forge. Tool specifications are parameterized by variables that capture aspects that are specific to individual ocean models. Values for these parameters have to be provided by each model for which a tool is to be created. A program generator creates Fortran code that implements a tool for any specific ocean model. The program generator is basically a compiler whose source language is Forge and whose target language is Fortran. Figure 4.1 illustrates the architecture of the system.

![Figure 4.1: System architecture.](image)

The developers of the IOM system (this is the group of people developing inversion tools) define tool specifications, which are written in Forge, to describe the tools that the IOM provides. The tool specifications can use library functions, which are mainly basic Fortran program transformers, for example, for generating loops. Library functions can also be employed to specify array index types, see
Section 4.2.2. The library functions are written in Haskell [29] and are provided by computer scientists who develop these functions in close collaboration with the ocean-modeling software developers who need them. A Fortran subroutine is generated for each tool specification. The generation of Fortran code depends on a model specification that contains the information that is specific to a particular ocean model, such as the dimensions of the array that is used to store the state values of the ocean. These model specifications are written by the users of the IOM tools, the ocean model developers, who only have to understand the parameters they define and do not otherwise have to be concerned about the implementation of tools and the specification language Forge. Model specifications are essentially \((name, value)\) pairs. However, the “values” can be complex entities, such as Fortran subroutines. The Forge compiler takes a tool specification, a model specification, and library function definitions as its input and produces a Fortran subroutine implementing an inversion tool for a particular ocean model.

![Diagram](image)

Figure 4.2: Using model parameters in tools and library function calls
Figure 4.2 shows a schematic instance of Figure 4.1 to illustrate how the components of the Forge system interact. In the shown example, the tool specification tool is parameterized by a model parameter $p$, which is used, for example, in the body to control the effect of a library function $f$. The library function $f$ is taken from an extensible library of auxiliary program transformations. The value for $p$ is taken from a model specification for which the Fortran program is to be generated.

4.1.2 The Forge Compiler

For the implementation of the Forge front end we have used the Haskell scanner generator Alex [8] and the parser generator Happy [26]. The main part of the Forge compiler is written in Haskell [29]. There are two advantages of using a Haskell to implement the compiler.

First, the abstract syntax of Fortran programs is represented by a collection of Haskell data types that can represent only syntactically correct Fortran programs. Therefore, the syntax correctness of the generated Fortran programs is automatically guaranteed by the type system of Haskell. This property runs under the slogan “type-correct meta programs ensure syntax-correct object programs”, which is discussed in more detail by Tim Sheard in [33].

Second, a Fortran program is obtained from an abstract syntax value of type $T$ through a function show that is defined for the data type $T$. This approach of defining the abstract syntax of Fortran Haskell data types and then developing
a `show` function for them gives us the flexibility to easily generate programs written in other high-level programming languages, such as C or C++. Since the abstract syntax of those languages is similar to Fortran, we simply have to re-implement the `show` function, so that the rest of the compiler can be reused. In fact, with regard to the subset of Fortran used by the program generator, the abstract syntax can be reused almost without change for C or C++. Multilanguage support has not been an explicit goal of Forge, but the abstract syntax representation supports it well in case this issue should become important in the future.

4.2 Generating Fortran from Tool Descriptions

In this section we illustrate the elements of the specification language Forge and their translation into Fortran through examples. We consider two convolution tools of the IOM system introduced in Section 3.1 and Section 3.2. Even though the mathematical continuous definition for all convolutions can be captured by a single equation that is parameterized by a weighting function, the discrete forms of the convolution tools, which have to be used in practical systems, differ considerably. Since there is no algorithm known that could automatically transform arbitrary convolutions into efficiently implementable discrete forms, this process still has to be performed by scientists. Therefore, the input to the program generator cannot be the single continuous convolution equation, but has to be instead the mathematically derived set of discrete equations. Since these sets of discrete
equations differ very much for different convolutions, Forge is useful in describing each of them in a generic form that can be translated into Fortran programs based on model parameter values.

Even though convolution tools are very important, Forge is not limited to the definition of convolution tools. Since Forge provides a general mechanism for representing model-dependent implementations of discrete equations, any such set can be represented in Forge, no matter what form the corresponding continuous equation has. However, Forge is not a general-purpose program generator.

An essential ingredient of the covered tool descriptions are discrete equations. Therefore, Fortran programs that cannot be represented by discrete equations cannot be generated by Forge.

4.2.1 Expressing Markovian Time Convolution in Forge

In Section 3.1 we have described the IOM convolution tool called *Markovian Time Convolution*. In this section, we illustrate how this convolution tool can be specified in Forge.

Each IOM tool is defined by a Forge specification. Every specification has a name and possibly some parameters. These parameters are called *model parameters* since they refer to model-specific information that is used to guide the translation of the specification. The model parameters are given in the form `Type::PName` through the specification's parameter list.

For example, the specification for the Markovian convolution `timeConv` has
one integer model parameter \( \text{dim} \) to represent the number of space dimensions of the underlying ocean model. The corresponding representation in Forge is as follows.

\[
\text{timeConv}(\text{integer}::\text{dim})
\]

The body of a specification consists of an interface, which is discussed in Section 4.2.2, and a definition of its function, given by a list of statements, which can be assignments, subroutine calls, loops, or applications of library functions. For example, in the definition of the tool \text{timeConv} we use a definition that is similar to the following.

\[
\text{genLoops}(\text{dim} \mid h[L] = 0.0)
\]

Here \text{genLoops} is a library function that generates \( \text{dim} \) nested loops over a Fortran assignment statement that is derived from the equation \( h[L] = 0.0 \) and whose purpose is to assign 0 to a specific array location. Why can we not just use a Fortran statement directly here? The reason is that the Fortran statement depends on the value of \( \text{dim} \). For example, if \( \text{dim} = 0 \), the Fortran statement derived from \( h[L] = 0.0 \) will be simply the following.

\[
h(L) = 0.0
\]

In this case, \text{genLoops} has no effect. In contrast, for \( \text{dim} = 2 \), the derived Fortran statement (without the surrounding \text{for} loops) will be:

\[
h(L, \text{dim1i}, \text{dim2i}) = 0.0
\]
where \( \text{dim1i} \) and \( \text{dim2i} \) are integer variables that are generated to be used as loop variables. In this case, the application of \text{genLoops} creates the following Fortran code.

```fortran
do dim2i = X2, Y2, 1
  do dim1i = X1, Y1, 1
    h(L,dim1i,dim2i) = 0.0
  end do
end do
```

The boundary variables of the \text{for} loops are parameters that are created for the subroutine \text{timeConv} from the dependent-index declaration \text{index space} = \text{X:Y[1:dim]}, which is explained in the next subsection.

For the definition of the Markovian convolution in Forge we have to translate each discrete equation into a Forge assignment statement and apply to the sequence of assignment statements the library function \text{genLoops} with the model parameter \text{dim} as follows.

```fortran
\text{genLoops}\{\text{dim} | \\
h[L] = 0.0; \\
h[n] = h[n-1] - dt*(h[n-1]/\tau + 2.0*a[n]/\tau); \\
b[U] = -0.5*h[U]/\tau; \\
b[n] = b[n+1] - dt*(h[n] + b[n+1]/\tau)
\}
```

For \text{dim} = 2, the Fortran program shown in Figure 4.3 will be generated.

In the generated program, a Fortran assignment statement is generated for each discrete equation, and a loop is generated for each dimension. The assignments that correspond to the second and fourth discrete equations are enclosed by an additional loop over the first dimension since \text{n} is an index variable on the
do dim2i = X2, Y2, 1  
do dim1i = X1, Y1, 1  
    h(L,dim1i,dim2i) = 0.0  
do n = L+1, U, 1  
    h(n,dim1i,dim2i) = h(n-1,dim1i,dim2i) &  
    & -dt*(h(n-1,dim1i,dim2i)/tau+2.0*a(n,dim1i,dim2i)/tau)  
end do  
b(U,dim1i,dim2i) = -0.5*h(U,dim1i,dim2i)/tau  
do n = U-1, L, -1  
    b(n,dim1i,dim2i) = &  
    & b(n+1,dim1i,dim2i)-dt*(h(n,dim1i,dim2i) &  
    & +b(n+1,dim1i,dim2i)/tau)  
end do  
end do

Figure 4.3: Fortran translation of discrete equations.

first dimension, whereas \( L \) and \( U \) are boundary values for the first dimension and denote single values in it. Therefore, the assignment statements for the first and third discrete equations are not placed inside of these additional loops.

4.2.2 Interfaces and Dependent Indices

The main purpose of an interface is to declare local variables and input variables for the generated Fortran subroutine. Both kinds of variables will appear in the generated Fortran subroutine. While input variables will be translated into the parameters of the generated Fortran subroutine, local variables will be translated into Fortran declarations of local variables. In our example, \( \Delta t \), \( \tau \), \( L \), \( U \), \( a \),
and b will be input parameters of the generated Fortran subroutine, while the variables h and n are only used locally.

Variable declarations also have to introduce the variables’ types. However, as we have already seen, the types of array variables might depend on model parameters. Therefore, we must be able to define such dependent types [40] in Forge, which is achieved through the concept of a dependent index. A dependent index is the model-dependent part of an array type. The following statement defines the dependent index representing the model-dependent space dimensions of the arrays in the Markovian convolution.

\[
\text{index space} = \text{X:Y}[1:\text{dim}];
\]

A dependent-index expression introduces two variable names, X and Y in the example, and attaches a range to them, here 1:dim. This range guides the Forge compiler to create just as many pairs of Fortran variables, that is, X1, Y1, X2, Y2, ..., Xdim, Ydim. Each pair of variables corresponds to the lower and upper boundary variables of a dependent dimension. These created boundary variables will be input parameters of the generated subroutine. It is important that the upper and lower bound of the range can be only a statically evaluable integer expression or a model parameter. This constraint is necessary to ensure that array dimensions are known at compile time for the specifications (and therefore also at Fortran compile time). The validity of the values of the upper and lower bound of the range is checked at compile time. For example, the created boundary variables must be yet unused variables and the two range values must be integer expressions.
Array types can make use of dependent indices either directly or by referring to a dependent-index definition. In general, the index type of an array is composed of a fixed, non-dependent part and a dependent part. Fixed index types can be merged with dependent-index types by applying a library function that can also make use of model parameters. If only one index part is needed (fixed or dependent) to describe an index type of an array, a library function call is not required.

For example, in the interface part of the Markovian convolution the library function cons places the fixed index \( L:U \) at the beginning of a’s and b’s index types, which are otherwise given by the dependent index space.

```fortran
  index space = X:Y[1:dim];
  param real :: dt, tau;
  integer :: L, U;
  real, dimension (cons{L:U,space}) :: a, b;
  local real, dimension (cons{L:U,space}) :: h;
  integer :: n;
```

In a more general version of the timeConv tool, we actually use the library function insertAt to place the fixed index at a particular position, which is represented by the model parameter timePos, in the list of variables represented by a dependent-index expression.

```fortran
  real, dimension (insertAt{timePos | L:U,space}) :: a;
```

Suppose we generate a Fortran program for an ocean model in which the value of model parameter dim is 2 and timePos is 2. The above code will then be translated into the following Fortran code.
integer :: L
integer :: U
integer :: X1
integer :: Y1
integer :: X2
integer :: Y2
real, dimension (X1:Y1, L:U, X2:Y2) :: a

In the generated Fortran program, the array \( a \) has three dimensions, which means one dimension of its fixed part plus two dimensions of its dependent part. The value 2 for the model parameter \texttt{timePos} has caused that the fixed index part is created as the second dimension index of \( a \).

The main task of library functions is to control the code generation for those parts of the generated Fortran program that depend on values for model parameters. For example, the library functions \texttt{cons} or \texttt{insertAt} allow a fixed index part to be combined with dependent indices in array declarations. In Section 4.2.1 we have seen the library function \texttt{genLoops} that generated nested loops and appropriate array indexing. Library functions like \texttt{genLoops} therefore combine several tasks in the translation of discrete equations, guided by the value for the model parameter \texttt{dim}.

The dependent part of an array index depends on model parameters of type \texttt{integer}, which are used to calculate the number of the dimensions. By using dependent types, we can use a generic form to represent array types used in different models, which have different number of dimensions. If the value for the model parameter specifying the number of dimensions of a dependent part is 0, the array does not contain any dependent dimension. In this case, any
reference to the dependent dimensions will cause type errors in the generated Fortran program. Therefore, the dependent part of an array is not allowed to be referenced in discrete equations, but it is necessary for generating Fortran programs.

4.2.3 Summary: The Complete Tool Specification for the Markovian Convolution

The Forge specification of `timeConv` is shown in Figure 4.4. The line numbers are shown to simplify the discussion of type checking that will be presented in Section 4.3.

```fortran
1 timeConv(integer::dim)
2   index space = X:Y[1:dim];
3   param real   :: dt, tau;
4       integer :: L, U;
5       real, dimension (cons{L:U,space}) :: a, b;
6       local real, dimension (cons{L:U,space}) :: h;
7           integer :: n;
8   genLoops{dim |
9       h[L] = 0.0;
10      h[n] = h[n-1] - dt*(h[n-1]/tau + 2.0*a[n]/tau);
11      b[U] = -0.5*h[U]/tau;
12      b[n] = b[n+1] - dt*(h[n] + b[n+1]/tau)
13   }
```

Figure 4.4: The Forge specification for Markovian time convolution.

The specification refers to one model parameter `dim`. The dependent-index
space has dim dimensions. The index type of the array a, b, and h is defined to depend on space, adds to space exactly one dimension, L:U. Therefore, a, b, and h have dim+1 dimensions. The equations in the specification body correspond exactly to the discrete equations of the Markovian time convolution. The library function genLoops takes an integer as input parameter to generate a corresponding number of nested loops over the Fortran code fragment translated from the argument equations. In the example, genLoops generates exactly dim nested loops for all the equations. Since the index of h[L] and b[U] are a boundary variable, no additional loops need to be generated for the first and third assignment. An additional loop over the time dimension is generated for the other two equations since the arrays on the left-hand side are indexed by a variable that is not an array boundary.

We have deliberately retained much of Fortran’s syntax, in particular, with regard to type, range, and array notation, to accommodate the users of Forge, who know Fortran very well. The complete syntax of Forge is shown in Appendix A.

4.2.4 Expressing the Bell-Shaped Space Convolution in Forge

We consider another IOM tool to demonstrate another feature of Forge, subroutine parameters, by showing how the convolution tool described in 3.2 can be expressed in Forge. The tool specification for the bell-shaped space convolution is shown in Figure 4.5.
Figure 4.5: The specification for the bell-shaped convolution in space.

In this example, computeRHS is a Fortran subroutine that has to be provided by the ocean modelers as a model parameter. This subroutine is used for computing the $RHS_i^n$ in the discrete equations of the bell-shaped convolution. This subroutine has a parameter space, which is the dependent index because the boundaries of space dimensions are needed by this subroutine. For example, if the value of model parameter dim is 2, then the statement for calling the subroutine in the generated Fortran program will be as follows.

```
call computeRHS(X1, Y1, X2, Y2, theta, rhs)
```
Note that the type system cannot guarantee the type correctness for the Fortran programs that were generated from tool descriptions involving subroutine parameters since we have generally no access to the code of the called subroutine at the time the tool description is processed. We can only check whether the called Fortran subroutine has been declared as a model parameter. In practice, this limitation means that the generated Fortran programs might contain type errors. However, these errors will be contained in the subroutines provided by the user who is responsible for delivering correct code.

4.2.5 Model Specifications

A model specification is given by a model name followed by model-parameter definitions. Each definition has the form $PName=Expr$ or $TName\cdot PName=Expr$. The former is for global model-parameter definitions, which means the model parameter can be used by any specification. The latter is for tool-specific model-parameter definitions. $TName$ is a tool name in which the parameter is used, whereas $PName$ is the model-parameter name used by that tool, which can be either a variable or a Fortran subroutine name. Tool-specific model parameters cannot be used by other tools. $Expr$ is either a constant or a Fortran file name, which contains the Fortran subroutine whose name is the model parameter.

Below we give a simple example of a model specification for the so-called “PEZ” model, the Primitive Equation Z-coordinate Model, which is a variant of Bryan-Cox-Semtner class model [28]. Two model parameters are defined in the
model specification. The first, \texttt{dim}, is a global model parameter that has the value 3. The second, \texttt{computeRHS}, is a model parameter that can be only used by the tool \texttt{spaceConv}, and refers to a Fortran subroutine provided by the modelers who want to generate Fortran code for \texttt{spaceConv}. The code of \texttt{computeRHS} is contained in the Fortran source file "\texttt{laplace.f90}".

\begin{verbatim}
model PEZ;
dim=3;
spaceConv.computeRHS = "laplace.f90"
\end{verbatim}

We can compile a tool specification into a Fortran program by calling the Forge compiler \texttt{forge}. In addition to the specification, we have to provide the name of the model specification for which we want to generate code. For example, we can generate the Fortran subroutine for the Markovian time convolution and the model \texttt{PEZ} by the following command.

\begin{verbatim}
forge timeConv PEZ
\end{verbatim}

The generated Fortran program is written to a file \texttt{timeConvPEZ.f90} whose content is shown in Appendix B.

4.3 The Forge Type System

The generation of Fortran programs that contain no syntax and type errors is of great importance in the context of scientific computing because the creation of programs that cause compiler errors would disturb users, in particular for program parts they have not written themselves. A principal problem with generated
code is that users cannot, in general, understand why the error occurred and, even worse, how to correct it. Such a situation should be avoided because it could make users lose trust in the system and eventually not use it anymore. Therefore, we have put a lot of effort in designing a type system that can prevent errors in generated Fortran programs. The type system captures possible errors at Forge compile time so that inversion tools to be provided by the IOM system can be checked in advance, before they are released, and thus will always compile smoothly when the system is in use. Of course, since the users themselves also provide Fortran code that is combined with the generated code, we cannot rule out syntax or type errors completely, but when the compiler complains about errors, this will happen in user-supplied code, and it should be clear then to the users that it is their responsibility to correct the mistakes in their code.

The syntactical correctness of generated Fortran programs is automatically guaranteed by the type system of the host language, Haskell. Because Fortran programs are represented by a Haskell data type that can represent only valid abstract Fortran syntax, any syntax error produced by the Forge translator (or by a library function) would be caught by the Haskell type system. Thus, the program generator will ensure that any syntactically correct discrete equations used in specifications will always be translated to syntactically correct Fortran code. Furthermore, the library functions can never introduce syntax errors since they are Haskell functions on the Haskell data type representing only syntactically correct Fortran programs.

The goal of the Forge type system is to guarantee (as much as possible) the
type correctness of the generated Fortran programs, which means that any type-
correct specification will be translated into a syntax-correct and type-correct
Fortran program.

All expressions and statements, including library function calls and Fortran
subroutine calls, are checked by the type system. The formal typing rules are
first given in Section 4.3.1. In Sections 4.3.2, 4.3.3, and 4.3.4, we demonstrate
how the type system can prevent type errors in the generated Fortran programs
through several examples.

4.3.1 The Typing Rules of Forge

In the following rules, we use metavariables \( e, f, vd, p, r, s, \) and \( v \) to range over
\( Expr, FName, VarDecl, PName, Range, Stmt \) and \( VName \) (refer to Appendix A), respectively. We use \( t \) to range over the Forge types. For convenient use in
the typing rules, we represent array types by quadruples \((f, \text{FixInd}, \text{DepInd}, \text{Bty})\).
In this representation, \( f \) represents the library function used for combining the
fixed part and dependent part of array dimensions. \( \text{FixInd}, \text{DepInd}, \) and \( \text{Bty} \)
represent the fixed part, the dependent part, and the element type of an array
type, respectively.

This style of defining typing judgments by rules is explained in detail, for
example, in Benjamin Pierce’s book [30]. Simply said, a rule consists of zero
or more premises \( P_1, \ldots, P_n \) written above the bar and one conclusion \( C \), and
has the meaning that the conclusion is true if all the premises are fulfilled. If
$n = 0$, that is, if no premises are given, the conclusion is always true, and the corresponding rule just states a fact. For example, all the rules in Figure 4.7 are facts, but the rules in Figure 4.6 contain premises. Premises and conclusions are expressed in form of judgments, which are essentially relations between the participating objects. For example, the judgment $Aty \Downarrow t$ in Figure 4.7 expresses the relationship between an array type $Aty$ and its quadruple representation.

The rules in Figure 4.6 define how to construct two typing environments $\Delta$ and $\Gamma$ from the interface part of a specification. A typing environment is a set of pairs $(v, t)$, expressing that variable $v$ has the type $t$. The typing environment $\Delta$ contains the type information for all model parameters, including external Fortran subroutines to be called, whereas $\Gamma$ contains the type information for constants, variables and library functions. $\Gamma_0$ is the initial typing environment, which contains the type information of constants and library functions. The judgment $\Delta; \Gamma \vdash d \triangleright \Delta'; \Gamma'$ expresses that the declaration $d$ changes the current typing environments from $\Delta$ and $\Gamma$ to $\Delta'$ and $\Gamma'$. The judgment $\Delta; \Gamma \vdash t$ expresses that $t$ is a valid array type under the typing environments. The judgment $\Delta; \Gamma \vdash e : t$ expresses that the expression $e$ has the type $t$ under the typing environments. We maintain two separate typing environments $\Gamma$ and $\Delta$, because model parameters can only be used in expressions for dependent indices and the parameters of library functions can only be model parameters.

The rule $MP_r$ and $MPS_r$ show how $\Delta$ is constructed from model parameter declarations. If a model parameter declaration $t :: p$ appears in the parameter list of a specification, the pair $(p, t)$ will be added into the typing environment.
### Figure 4.6: Rules for environment judgments.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPr</td>
<td>$p \notin \text{dom}(\Delta)$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash t::p \triangleright \Delta \cup {(p, t)}; \Gamma$</td>
</tr>
<tr>
<td>MPsr</td>
<td>$\Delta; \Gamma \vdash t_1::p_1 \triangleright \Delta_{i+1}; \Gamma$</td>
</tr>
<tr>
<td></td>
<td>$\ldots$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash t_n::p_n \triangleright \Delta_{n+1}; \Gamma$</td>
</tr>
<tr>
<td>DIXE</td>
<td>$\Delta; \Gamma_0 \vdash e_1: \text{integer} \quad \Delta; \Gamma_0 \vdash e_2: \text{integer}$</td>
</tr>
<tr>
<td></td>
<td>$l_i, h_i \notin \Gamma \quad i \geq 1$</td>
</tr>
<tr>
<td>DIX</td>
<td>$IName \notin \text{dom}(\Gamma) \quad \Delta; \Gamma \vdash \text{IExpr} \triangleright \text{depix}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{IExpr}=\text{DepExpr} \triangleright \Delta; \Gamma \cup {(IName, \text{depix})}$</td>
</tr>
<tr>
<td>DIXS</td>
<td>$\Delta; \Gamma_1 \vdash \text{IExpr} \triangleright \Delta; \Gamma_{i+1}$</td>
</tr>
<tr>
<td></td>
<td>$1 \leq i \leq n$</td>
</tr>
<tr>
<td>BVS</td>
<td>$v_i \notin \text{dom}(\Gamma) \quad 1 \leq i \leq n \quad i \neq j \Rightarrow v_i \neq v_j$</td>
</tr>
<tr>
<td>RANGE</td>
<td>$\emptyset; \Gamma \vdash e_1: \text{integer} \quad \emptyset; \Gamma \vdash e_2: \text{integer}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash e_1::e_2: \text{range}$</td>
</tr>
<tr>
<td>FIXr</td>
<td>$\emptyset; \Gamma \vdash r::\text{range}$</td>
</tr>
<tr>
<td></td>
<td>$\emptyset; \Gamma \vdash \text{FixInd}::\text{range}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{FixInd}::\text{range}$</td>
</tr>
<tr>
<td>ATY</td>
<td>$\emptyset; \Gamma_0 \vdash f::t_1 \rightarrow \ldots \rightarrow t_k \rightarrow \text{range} \rightarrow \text{depix} \rightarrow \text{range}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{Bty}, \text{dimension}(f(p_1, \ldots, p_k) \text{FixInd}, \text{DepInd})$</td>
</tr>
<tr>
<td>FATY</td>
<td>$\Delta; \Gamma \vdash \text{FixInd}::\text{range}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{Bty}, \text{dimension}(\text{FixInd})$</td>
</tr>
<tr>
<td>DATY</td>
<td>$\Delta; \Gamma \vdash \text{DepInd}::\text{depix}$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{Bty}, \text{dimension}(\text{DepInd})$</td>
</tr>
<tr>
<td>AVS</td>
<td>$v_i \notin \text{dom}(\Gamma) \quad 1 \leq i \leq n \quad \Delta; \Gamma \vdash \text{Aty} \downarrow t$</td>
</tr>
<tr>
<td></td>
<td>$i \neq j \Rightarrow v_i \neq v_j$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma \vdash \text{Aty}::v_1, v_2, \ldots, v_n \triangleright \Delta; \Gamma \uplus {(v_1, t), (v_2, t), \ldots, (v_n, t)}$</td>
</tr>
<tr>
<td>PS</td>
<td>$\Delta; \Gamma_1 \vdash \text{param} \vd{v_1}; \vd{v_2}; \ldots; \vd{v_n} \triangleright \Delta; \Gamma_{n+1}$</td>
</tr>
<tr>
<td></td>
<td>$1 \leq i \leq n$</td>
</tr>
<tr>
<td>LS</td>
<td>$\Delta; \Gamma_1 \vdash \text{local} \vd{v_1}; \vd{v_2}; \ldots; \vd{v_n} \triangleright \Delta; \Gamma_{n+1}$</td>
</tr>
<tr>
<td></td>
<td>$1 \leq i \leq n$</td>
</tr>
<tr>
<td>ENV</td>
<td>$\emptyset; \Gamma_0 \vdash \text{MList} \triangleright \Delta; \Gamma_0\Delta; \Gamma_0 \vdash \text{DepLx} \triangleright \Delta; \Gamma$</td>
</tr>
<tr>
<td></td>
<td>$\emptyset; \Gamma \vdash \text{Param} \triangleright \Delta; \Gamma \prime\Delta; \Gamma \prime \vdash \text{Local} \triangleright \Delta; \Gamma \prime \prime$</td>
</tr>
<tr>
<td></td>
<td>$\emptyset; \emptyset \vdash \text{TName}(\text{MList}) \text{DepLx}; \text{Param}; \text{Local}; \triangleright \Delta; \Gamma \prime \prime$</td>
</tr>
</tbody>
</table>
\(\Delta\). \texttt{DIXE}\_\texttt{E}\_\texttt{L}\) says that a dependent-index expression is valid only if the two expressions specifying the number of dependent dimensions have integer types and only contain model parameters and constants. Fortran variables must not appear in the expressions because the number of dependent dimensions has to be known at compile time. Therefore, we use \(\Delta\) and \(\Gamma\_0\), which do not contain type information for Fortran variables, to infer the type of the two expressions in the rule \texttt{DIXE}\_\texttt{L}\. The presence of Fortran subroutine names in \(\Delta\) causes no problem, since their result type is \texttt{subroutine}, which means that they can never successfully contribute to a well typed integer expression. The rule \texttt{DIX}\_\texttt{L}\ shows that if a dependent-index expression is valid and is bound to a name, then the name has the type \texttt{depix} in the typing environment \(\Gamma\). We do not have to distinguish different dependent-index types in the type system, because the type \texttt{depix} is only used to check if a library function call for constructing a dependent array type is type correct by the rule \texttt{ATY}\_\texttt{E}\ and \texttt{DATY}\_\texttt{E}\. As long as a dependent index is type correct, library functions can be applied to it—the exact type does not matter. Therefore, all dependent-index expressions and names have the same type \texttt{depix}\.

The rules \texttt{BVS}\_\texttt{E}\ and \texttt{AVS}\_\texttt{E}\ check variable declarations. If a variable has a base type, the rule \texttt{BVS}\_\texttt{E}\ adds the pair \((v, t)\) into \(\Gamma\). If a variable has an array type, the rule \texttt{AVS}\_\texttt{E}\ first checks if the array type is valid, in which case the name and the quadruple representing the array type is added into \(\Gamma\). The judgment \(\text{Aty} \downarrow t\) used in the rule \texttt{AVS}\_\texttt{E}\ expresses that \(t\) is the quadruple corresponding to the array type \(\text{Aty}\). The judgment \(\text{Aty} \downarrow t\) is defined in Figure 4.7 by inference rules.

The rule \texttt{RANGE}\_\texttt{E}\ expresses that a range consists of two integer expressions.
Since the two expressions of a range can only contain Fortran variables, we use $\emptyset$ instead of $\Delta$ in the premises of the rules $\text{range}_\varphi$ and $\text{fix}_\varphi$. The rules $\text{faty}_\varphi$ and $\text{daty}_\varphi$ check if an array type is valid when there is no library function involved. $\text{faty}_\varphi$ says that if an array type only contains fixed indices, the fixed indices must have the type range. $\text{daty}_\varphi$ says that if an array type only contains dependent indices, the dependent indices must have the type depix. When an array type is constructed by a library function, the rule $\text{aty}_\varphi$ needs to check if the library function call is type correct. Every library function is a Haskell function whose type signature is contained in $\Gamma_0$. A library function used for merging array indices must have the type $t_1 \to \ldots \to t_k \to \text{range} \to \text{depix} \to \text{range}$. The type of every parameter of the library function has to match the type signature.

The rules $\text{ps}_\varphi$ and $\text{ls}_\varphi$ add the typing information of all the parameter variables and local variables into the typing environment $\Gamma$. The rule $\text{env}_\varphi$ shows how the type environments are constructed from the interface part of a specification. We start from the initial typing environments $\emptyset$ and $\Gamma_0$. The result typing environment $\Delta$ contains the type information for all model parameters. The type information for all the dependent-index names, parameter variables, and
local variables will be subsequently added, and the result typing environment is
\( \Gamma'' \).

The rules in Figure 4.8 define the typing rules of Forge expressions. The
determination \( \Delta; \Gamma \vdash e : t \) means the expression \( e \) has the type \( t \) under the typing
assumptions \( \Delta \) and \( \Gamma \).

\[
\begin{array}{|c|}
\hline
\text{CON}_\tau & e \text{ is a constant of } Bty \\
\hline
\Delta; \Gamma \vdash e : Bty & \Delta; \Gamma \vdash (v, t) \in \Gamma \\
\hline
\text{VAR}_\tau & (v, t) \in \Gamma \\
\hline
\text{ARRAY}_\tau & \Delta; \Gamma \vdash (f, (r_1, r_2, \ldots, r_n), \text{DepInd}, Bty) \\
\hline
\Delta; \Gamma \vdash e [e_1, e_2, \ldots, e_k] : (f, (r_{k+1}, r_{k+2}, \ldots, r_n), \text{DepInd}, Bty) \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{UOP}_\tau & \Delta; \Gamma \vdash e : t \\
\hline
\Delta; \Gamma \vdash UOp e : t \\
\hline
\text{BOP}_\tau & \Delta; \Gamma \vdash e_1 : t_1 \\
\hline
\Delta; \Gamma \vdash e_2 : t_2 \\
\hline
\Delta; \Gamma \vdash t_1 \sim t_2 \\
\hline
\end{array}
\]

Figure 4.8: Typing rules for expressions.

The rule \( \text{ARRAY}_\tau \) shows that we only allow referencing fixed indices in Forge.
This restriction is necessary because the number of dimensions of dependent
indices is different in different models, it might even be 0 in some models. The
rule \( \text{UOP}_\tau \) is trivial since we currently have negation as the only unary operator
in our abstract syntax; see Figure 1. The notation \( t_1 \sim t_2 \) is used to express that
\( t_1 \) and \( t_2 \) are compatible in the sense that binary operations can well operate on
arguments of both types. The relation \( \sim \) is defined as follows.

\[
t_1 \sim t_2 \text{ iff } t_1 \prec t_2 \text{ or } t_2 \prec t_1
\]

In the definition, we use a partial order relation on types, \( \prec \), which is defined in
Figure 4.9. The rule \( \text{BAP}_\sim \) essentially allows the initialization of an array with a
constant or to add a constant to every element of an array, cf. rule \texttt{ASSGN}_n in Figure 4.10. This rule overloads constants of base types with array types.

\begin{center}
\begin{tabular}{llll}
\texttt{BASE}_\prec & \texttt{REAL}_\prec & \texttt{REFL}_\prec & \texttt{ARRAY}_\prec \\
\hline
\texttt{integer} \prec \texttt{real} & \texttt{t} \prec \texttt{t} & \texttt{Bty}_1 \prec \texttt{Bty}_2 & \\
\hline
\texttt{FixInd}_1 = \texttt{FixInd}_2 & \texttt{DepInd}_1 = \texttt{DepInd}_2 & \texttt{Bty}_1 \prec \texttt{Bty}_2 & \\
& \texttt{Bty}_2 \prec (f, \texttt{FixInd}, \texttt{DepInd}, \texttt{Bty}_2) & \\
\end{tabular}
\end{center}

\begin{center}
Figure 4.9: Partial order on types.
\end{center}

The inclusion of the ordering $\prec$ may not seem necessary since we currently have only two base types, \texttt{integer} and \texttt{real}; but this relation will be very useful when we extend the type system to include more base types. The function \texttt{max} returns the larger of two types with respect to $\prec$ if the two types are comparable, that is $\texttt{max}(t_1, t_2)$ returns $t_1$ if $t_2 \prec t_1$, otherwise it returns $t_2$. Note that the application of \texttt{max} in rule \texttt{BOP}_n is always well defined since \texttt{max} will be applied only if its argument are compatible.

The typing rules in Figure 4.10 define the typing of Forge statements. The judgment $\Delta; \Gamma \vdash s$ means that the statement $s$ is valid under the typing environments $\Delta$ and $\Gamma$. The rule \texttt{ASSGN}_n says that the assignment statement is valid if the type of the left-hand side is upward compatible with the type of the right-hand side. For example, we can assign a real number to a real array, which has the effect that all the elements in the array will be initialized to the same value. The use of $\emptyset$ in the two premises ensures that $v$ and all variables used in $e$ are not confused with model parameters and that have valid types defined in $\Gamma$. The rule \texttt{SEQU}_n expresses that a sequence of two statements is valid only if
both statements in the sequence are valid. In the rule \( \text{SUB}_\text{L} \) we can only check whether the called Fortran subroutine \( \text{SubName} \) has been declared as a model parameter. We cannot check the type consistency of the arguments because we do not have access to the code of the called subroutine at this time. The rule \( \text{LIB}_\text{L} \) checks a library function call in a similar way as the rule \( \text{ATY}_\text{L} \) in Figure 4.6. The typing rule \( \text{SPEC}_\text{L} \) expresses that if the body of a specification is type correct under the typing environments constructed from its interface, the whole specification is type correct.
4.3.2 Declaration Typing

In the following, we refer to the specification for the Markovian time convolution in Figure 4.4. Declarations in a specification are used to construct two typing environments, which are used in statements and expressions to check whether or not variables are used in a type-consistent way. One typing environment $\Delta$ contains the type information for all model parameters, including external Fortran subroutines, and another typing environment $\Gamma$ contains the type information for variables and library functions. A typing environment is a set of pairs $(v, t)$ expressing that name $v$ has the type $t$. Each declaration statement adds one or more pairs into the typing environment. Model parameter declarations in the parameter list of a specification add such pairs into the typing environment $\Delta$. For example, the following model parameter declaration (line 1) adds $(\text{dim}, \text{integer})$ into $\Delta$.

\begin{verbatim}
  integer :: dim
\end{verbatim}

Local variable declarations and parameter declarations are used to declare Fortran variables. For example, the following declaration (line 3) adds $(\text{dt, real})$ and $(\text{tau, real})$ into the environment $\Gamma$.

\begin{verbatim}
  real :: dt, tau
\end{verbatim}

In the typing environment $\Gamma$ an array type is represented as a quadruple which consists of the name of the library function used to construct the array type, the fixed index, the dependent index, and the base type of the array. For ex-
ample, the following declaration (line 5) adds \((a, (\text{cons}, (L, U), \text{space}, \text{real}))\) and 
\((b, (\text{cons}, (L, U), \text{space}, \text{real}))\) into \(\Gamma\).

\[
\text{real, dimension } (\text{cons}\{L:U, \text{space}\}) :: a, b
\]

A dependent-index declaration also adds a (name, type) pair into the typing environment \(\Gamma\). In contrast to Fortran variables declared in a specification, all the dependent-index names have the type \text{depix}. Consider, for example, the following dependent-index declaration (line 2).

\[
\text{space} = x:Y[1: \text{dim}];
\]

The type checker adds \((\text{space}, \text{depix})\) into the typing environment. Dependent-index names cannot be used in any expressions, they are only used to construct array types, which is ensured by their type \text{depix}. We need the type information for dependent-index names for checking the library function calls to construct array types. The rules for constructing typing environments are defined in Figure 4.6. Array types can be constructed by calling library functions, where the validity of the library function calls is checked by the rule \(\text{ATy}_\text{depix}\). Consider again the declaration in line 5. Since \(L\) and \(U\) have type \text{integer}, \(L:U\) is a valid fixed index. Because the name \text{space} has the type \text{depix}, it is a valid dependent index. In the typing environment \(\Gamma\), we store the type of the library function \text{cons}, which is \text{range}\rightarrow\text{depix}\rightarrow\text{range}, where \text{range} represents fixed index types. The type environment \(\Gamma\) is initialized with the types for all available library functions, which are derived from the type signatures of the Haskell functions. In the above declaration, the library function \text{cons} has two arguments which are a valid fixed
index and a valid dependent index, respectively. Therefore, the library function call in the declaration statement is type correct.

4.3.3 Expression Typing

The expressions in `timeConv` include constants, such as 0.0 (line 9), simple variables, such as `tau` and `U` (line 11), array expressions, such as `h[n-1]` (line 10), and expressions such as `-0.5*h[U]/tau` (line 11). Determining the type of constants and variables is trivial because the type information for constants and variables can be obtained from the typing environment directly. Array expressions are more complex since arrays may have dependent indices. Forge does not allow referencing dependent indices in array expressions since it could cause type errors in the generated Fortran program. For example, suppose we have an expression `a[i,1]`, where 1 refers to the first dimension of the dependent index. If the model parameter `dim` in the user model is 0, in which case there is no dependent index at all in the array `a`, then `a[i,1]` would cause a type error in the generated program because `a` is only a one-dimensional array. The rule in Figure 4.8 for array expressions expresses that an expression `e` can be indexed by index expressions `e_1, e_2, ...`, and `e_k` if the type of `e` is an array type with fixed ranges `r_1, r_2, ...,` and `r_n` and `n ≥ k` (see rule `ARRAY`).

Applications of operations are type correct if all the subexpressions are type correct and have types that are compatible with the operator. For example, `real` and `integer` are compatible, so `tau*2` is type correct. The type of `real` arrays
is compatible to \texttt{real}, so the expression \(-0.5\cdot h[U]/\tau\) in line 11 is also type correct. The compatibility of types is defined by a relation \(\sim\). Two array types are compatible only when (1) they have the same fixed and dependent dimensions and (2) the applied library functions for constructing the arrays are the same. We define a rigid form of equality for the dimensions, which requires for two dimensions to be equal that both, the number of dimensions and the boundaries of each dimension, are the same. For example, since \(\texttt{b}\) and \(\texttt{h}\) are declared by the same statement in line 5 and 6 in Figure 4.4, they have the same type. Therefore, the expression \(\texttt{b[n+1]} - \texttt{dt*(h[n] + b[n+1]/\tau)}\) in line 12 is type correct. In contrast, the expression \(\texttt{a + b[n]}\) would cause a type error since \(\texttt{a}\) and \(\texttt{b[n]}\) do not have compatible types.

Since we use library functions to construct array types when declaring array variables, we require that these library functions have the following two properties.

1. Library functions do not change the relative order of fixed or dependent dimensions.

2. Library functions do not change fixed or dependent dimensions, including the number of dimensions and the name or value of the boundaries of each dimension.

These conditions guarantee that after applying a library function the newly constructed array type still has the same fixed and dependent dimensions as before. These two conditions restrict the effect of library functions essentially to merging
dependent and fixed indices in their given order. Unfortunately, these properties of library functions cannot be checked by the compiler. Therefore, only carefully checked, “hand-certified” library functions will be made available to guarantee the soundness of the type checker. Since library functions are, like the type system itself, implemented by the Forge developers and cannot be changed by the IOM developers or ocean modelers, they have the same sensitive status as the implementation of the type checker itself, which could also principally contain errors. A careful selection and implementation of library functions together with a careful implementation of the type checker therefore can guarantee the soundness of results reported by the type checker.

4.3.4 Statement Typing

The type system also checks the validity of all statements. For example, an assignment statement is valid only if the type of the left-hand side is upward compatible with the type of the right-hand side. The upward compatibility is defined by a relation \( \prec \). If a type \( t_1 \) is upward compatible with a type \( t_2 \), then an expression of type \( t_2 \) can be assigned to a variable or an array expression of type \( t_1 \). For example, in line 9 of Figure 4.4, the left-hand side of the assignment has the type of array of real numbers; the right-hand side has the type of \texttt{real}. Since a \texttt{real} array is upward compatible with \texttt{real}, this assignment is valid. Similar reasoning shows that all the assignments in Figure 4.4 are valid.

For a Fortran subroutine call we need to check if the types of the arguments
match the types of the subroutine’s parameters. Unlike all other typing constraints, we cannot perform this test independently of a particular model, that is, without knowing the values of the model parameters for that model, because we have to know the subroutine’s parameter definitions to judge whether or not the call is correct. Therefore, we have to defer this test to the time when a Fortran subroutine is generated from the tool specification for a particular model.

For example, in line 13 of Figure 4.5, a Fortran subroutine computeRHS is called. At this point all we know is that this user-provided subroutine must have 2*dim integer parameters (for the array boundaries) and two more array parameters of the type specified in the local declaration. However, the correctness of the generated subroutine spaceConv cannot be decided at this point and depends critically on the availability of a user-provided Fortran subroutine computeRHS of the corresponding type. Thus, along with the generation of the subroutine spaceConv we check whether this condition is fulfilled. The formal definition can be found in Figure 4.10 in Section 4.3.1 in the typing rule sub.

Finally, the library function calls for program transformation have to be type checked. When checking a library function call statement, we first obtain the type of the corresponding Haskell function from the type environment Γ. For example, the type of the library function genLoops is integer->fortran->fortran, where the type fortran represents Fortran programs. Looking at the application of genLoops in line 8 of Figure 4.4, we can observe that the first parameter dim is a model parameter of type integer, and the second parameter is a sequence of statements which is translated to Fortran statements. Therefore, the library
function application in line 8 is valid. The typing rule for checking library function call statements is defined in Figure 4.10 in Section 4.3.1 in the rule LIBr.
Chapter 5 – Parametric Fortran: An Extension of Fortran for Generic Programming

In Chapter 4, we described a domain-specific language, Forge, that can be used to solve the code reuse problem in the IOM system. However, Forge is limited to specifying tools that can be represented by discrete equations. In this chapter, we present a more general approach to solving the code reuse problem in Fortran.

Parametric Fortran [10, 12] 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. We describe the design, implementation, and some applications of Parametric Fortran. Parametric Fortran is particularly useful in scientific computing for solving the code reuse problem. The applications of Parametric Fortran include defining generic functions, removing duplicated code, and automatic differentiation. The described techniques have been successfully employed in the IOM project.

This chapter is organized as follows. In Section 5.1 we give an overview of Parametric Fortran through examples. We then describe in Section 5.2 how Parametric Fortran is employed in the IOM project for solving the code reuse problem. In Section 5.3 we describe another important application of Parametric Fortran,

\footnote{This chapter is mainly based on [10, 12].}
automatic differentiation. Finally, we describe the Haskell implementation of the Parametric Fortran compiler in Section 5.4.

5.1 Overview of Parametric Fortran

Parametric Fortran is an extension of Fortran that allows Fortran constructs to be parameterized. The various parameterization constructs and their meanings are listed in Table 1.

<table>
<thead>
<tr>
<th>Parameterization Construct</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{p : ...}</code></td>
<td>every syntactic object inside the braces is parameterized by p</td>
</tr>
<tr>
<td><code>{p(v1,..,vn) : ...}</code></td>
<td>only variables v1, ..., vn are parameterized by p inside the braces</td>
</tr>
<tr>
<td><code>{#p : ...}</code></td>
<td>only the outermost syntactic object is parameterized by p</td>
</tr>
<tr>
<td>`{ ... }</td>
<td>everything inside the braces is protected from parameterization by an enclosing parameter</td>
</tr>
<tr>
<td><code>!v</code></td>
<td>the variable v is not parameterized</td>
</tr>
</tbody>
</table>

In Parametric Fortran, 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.

In the remainder of this section, we will demonstrate the use of Parametric Fortran with examples. A generic array addition subroutine will be presented in 5.1.1. In Section 5.1.2 we described how the program generation is realized.
A generic array-slicing subroutine is described in Section 5.1.3 where a feature called \textit{parameter accessors} is introduced with an example of array slicing. In Section 5.1.4 we will show how to remove duplicated code using another feature of Parametric Fortran called \textit{list parameters}.

5.1.1 Array Addition for Arbitrary Dimensions

The following program shows how to write a Parametric Fortran subroutine to add two arrays of arbitrary dimensions.\footnote{This example is meant for illustration. Dimension-independent array addition is already supported in Fortran 90 by array syntax.}

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

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 \texttt{dim}. The value of \texttt{dim} will guide the generation of the Fortran subroutine. The braces \{ and \} delimit the scope of the \texttt{dim} parameter, that is, every Fortran syntactic object in the subroutine is parameterized by \texttt{dim}. For \texttt{dim} = 2, the Fortran program in Figure 5.1 will be generated.

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

5.1.2 Realization of the Program Generation

Fortran programs are generated from Parametric Fortran templates. The gen-
eration process is performed by the Parametric Fortran compiler whose source
language is Parametric Fortran and whose target language is Fortran. The comp-
iler is implemented in Haskell [29]. 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 5.2 shows an example Parametric Fortran syntax tree.

In the syntax tree, every square represents a Fortran syntactic object, and \( p_1 \) and \( p_2 \) are two parameters. Three nodes are parameterized by \( p_1 \), two nodes are parameterized by \( p_2 \), 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 \( p_1 \) has the type \( T_1 \) and \( p_2 \) has the type \( T_2 \). 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 through pattern matching and syntax tree transformation. 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 \( T_1 \) to every node that is parameterized by \( p_1 \), and applies the generation function for \( T_2 \) to every node that is parameterized by \( p_2 \). This type-based generic traversal is implemented using the “Scrap Your Boilerplate” approach introduced in [24]. 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 \texttt{arrayAdd} program in Section 5.1.1 as a concrete example. In \texttt{arrayAdd}, we use a parameter \texttt{dim} containing an integer value. The parameter type for \texttt{dim} is defined as a Haskell data type \texttt{Dimension} as follows.

\begin{verbatim}
data Dimension = Dim Int
\end{verbatim}

The generation function for the parameter type \texttt{Dimension} 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 \texttt{Dimension} with an integer \( n \), the generated expression will be an \( n \)-dimensional array expression. The function \texttt{gen} can be defined as follows
The variables $i1$ through $in$ are new variables generated by the program generator. These variables will be only used as array indices and loop variables.

When a Dimension parameter of positive value $n$ parameterizes a Fortran statement $s$, 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 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.

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

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

$$\{\text{dim: } c = a + b\}$$

This statement is represented by the syntax tree shown in Figure 5.3.
Figure 5.3: Syntax tree of \{\text{dim}: c = a + b\}

Since the parameterization is recursively propagated, every node in the syntax tree is parameterized by the parameter \text{dim}. When the value for \text{dim} is provided, for example, \text{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 \text{Dimension} to every node. Figure 5.4 shows the transformation for each single node. The root node, which is an assignment statement, is transformed into a loop statement. The node \text{c} is transformed into an array expression \text{c(i1)}, and the nodes \text{a} and \text{b} are also variables and transformed like \text{c}. The parameterization does not affect the \text{“+”} node which causes the Parametric Fortran compiler to leave the node \text{“+”} unchanged.

After applying the \text{gen} function for the type \text{Dimension} to all the nodes, the Parametric Fortran compiler outputs the Fortran syntax tree in Figure 5.5, that represents the following generated Fortran loop statement.
\begin{figure}
\centering
\begin{tikzpicture}
\node[coordinate] (a) at (0,0) {\texttt{dim}};
\node[coordinate] (b) at (1,0) {\texttt{i1}};
\node[coordinate] (c) at (2,0) {\texttt{l}};
\node[coordinate] (d) at (3,0) {\texttt{100}};
\node[coordinate] (e) at (4,0) {\texttt{=}};
\node[coordinate] (f) at (2,-1) {\texttt{Array}};
\node[coordinate] (g) at (2,-2) {\texttt{c}};
\node[coordinate] (h) at (3,-2) {\texttt{i1}};
\node[coordinate] (i) at (1,-2) {\texttt{+}};
\node[coordinate] (j) at (0,-3) {\texttt{dim}};
\node[coordinate] (k) at (1,-3) {\texttt{c}};
\node[coordinate] (l) at (2,-3) {\texttt{a}};
\node[coordinate] (m) at (3,-3) {\texttt{b}};
\node[coordinate] (n) at (2,-4) {\texttt{Array}};\n\node[coordinate] (o) at (2,-5) {\texttt{a}};
\node[coordinate] (p) at (3,-5) {\texttt{i1}};
\node[coordinate] (q) at (1,-5) {\texttt{b}};
\node[coordinate] (r) at (2,-6) {\texttt{Array}};
\node[coordinate] (s) at (2,-7) {\texttt{b}};
\node[coordinate] (t) at (3,-7) {\texttt{i1}};
\draw[->] (a) -- (b);
\draw[->] (b) -- (c);
\draw[->] (c) -- (d);
\draw[->] (d) -- (e);
\draw[->] (f) -- (g);
\draw[->] (g) -- (h);
\draw[->] (i) -- (j);
\draw[->] (j) -- (k);
\draw[->] (k) -- (l);
\draw[->] (l) -- (m);
\draw[->] (n) -- (o);
\draw[->] (o) -- (p);
\draw[->] (p) -- (q);
\draw[->] (r) -- (s);
\draw[->] (s) -- (t);
\end{tikzpicture}
\caption{Transformation of nodes}
\end{figure}

\begin{verbatim}
do i1 = 1, 100
  c(i1) = a(i1) + b(i1)
end do
\end{verbatim}

The resulting syntax tree is obtained by composing all the generated nodes.
5.1.3 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 the program template shown in Figure 5.6, 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
Figure 5.6: Parametric Fortran Template of an Array Slicing Subroutine

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 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 5.1.1, 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

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

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

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

```fortran
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
```

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.

5.1.4 Removing Duplicated Code

In this section we demonstrate how Parametric Fortran can be used to solve a typical problem of duplicated code [14] 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 [14]. With Parametric Fortran, only one code fragment for duplicated
code is maintained, which simplifies the program maintenance.

The Parametric Fortran program in Figure 5.7 shows 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 5.7: Removing Duplicated Code with Parametric Fortran

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.

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

In this 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 sim-
plicity, 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
The declaration and body part of the simulation program are parameterized separately since they belong to different Fortran syntactic categories. This is necessary since although a parameterization construct can span multiple statements or declarations, it cannot span a combination of both.

The parameter value for `stateVars` shown in this example is used for generating the simulation program for a scientific model that has two state variables representing temperature and velocity, and the arrays storing the two variables are 3-dimensional and 2-dimensional, respectively. The following simulation program will be generated for this model. In the generated program, a declaration statement and a code fragment for the computation are generated for both state variables.

```fortran
program simulation
    real, dimension (:,:,100, 1:100, 1:100), 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
```
5.2 Implementing the IOM System in Parametric Fortran

We have introduced the IOM system in Chapter 3 and discussed how IOM tools can be expressed and generated in a domain-specific language in Section 4. In this section we will illustrate how the convolution tools described in Chapter 3 can be implemented in Parametric Fortran. In Section 5.2.1, we show how the Markovian time convolution is implemented in Parametric Fortran. The Parametric Fortran template for the bell-shaped space convolution is shown in Section 5.2.2. In Section 5.2.3 we give a summary about the benefits of using Parametric Fortran to implement the IOM system.

5.2.1 Expressing Markovian Time Convolution in Parametric Fortran

For parameterizing the Markovian time 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 \texttt{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.

```haskell
data Space = Space Int [(Bound, Bound)]
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 shown in Figure 5.8. In the program, the body of the subroutine just implements the discrete equations of Markovian time convolution shown in Section 3.1. 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 declaration 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
subroutine timeConv {s: (L, U, dt, tau, a, b)}
  integer :: L, U
  real :: dt, tau
  {s: real, dimension (L:U) :: a, b, h}
  integer :: n
  {#s(a,b,h):
    h(L) = 0.0
    do n = L+1, 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 = U-1, L, -1
      b(n) = b(n+1) - dt*(h(n) + b(n+1)/tau)
    end do
  }
end subroutine timeConv

Figure 5.8: The Parametric Fortran Template for the Markovian Time Convolution

dimensions to the body. The meaning of the symbol # is that s parameterizes the whole body of the subroutine, but not the single statements inside the body. Therefore, in the generated program, while 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 \texttt{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 \texttt{s}.

\[
\texttt{s :: Space}
\]
\[
\texttt{s = Space 1 [(BVar "X", BVar "Y")]}\]

The following program can then be generated automatically by the Parametric Fortran compiler.

\begin{verbatim}
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
   end do
   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 subroutine timeConv
\end{verbatim}

The program generator performed the following actions. (1) \texttt{X} and \texttt{Y} are added to the parameter list of the generated subroutine. (2) The arrays \texttt{a}, \texttt{b}, and \texttt{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 \texttt{i1}. (5) \texttt{i1}'s
declaration is generated. How a parameter type affects the program generation is defined in Haskell. We have seen an abstract description of defining the behavior of a parameter type in Section 5.1.2.

5.2.2 Implementing the Bell-Shaped Space Convolution in Parametric Fortran

Figure 5.9 shows the Parametric Fortran template for the bell-shaped space convolution described in Section 3.2.

Similar to the Markovian time convolution, we use the parameter $s$ of the type $\text{Space}$ in the Parametric Fortran template. In line 1, the parameter list of the Fortran subroutine is parameterized by $s$. This parameterization adds the new variables used in $s$'s value as dimension boundaries to the parameter list of the generated Fortran subroutine. Similarly, in line 12, $s$ parameterizes the argument list of the subroutine call to add the new variables used in $s$'s value. In line 4, the declaration of the array variables is parameterized by $s$ to append the space dimensions to the current time dimension of these arrays. In lines 6, 13, and 19, $s$ is used to add loops over space dimensions to the parameterized statements and add index variables to the specified array variables. We use the same parameter value for $s$ as in Section 5.2.1 to generate the Fortran subroutine $\text{spaceConv}$ for the model in which the arrays have 1 space dimension and the boundaries of that dimension is $(X:Y)$.

$$s = \text{Space 1 ([BVar "X", BVar "Y"])}$$
subroutine spaceConv {s: (L, U, N, ds, a, b)}
  integer :: L, U, N
  real :: ds
  {s: real, dimension (L:U) :: a, b, theta, rhs}
  integer :: n, i
  {#s(a,theta):
    do i = L, U
      theta(i) = a(i)
    end do
  } do n = 1, N
    call computeRHS{s: (theta, rhs)}
    {#s(theta,rhs):
      do i = L, U
        theta(i) = theta(i) + ds*rhs(i)
      end do
    }
  end do
  {#s(b,theta):
    do i = L, U
      b(i) = theta(i)
    end do
  }
end subroutine spaceConv

Figure 5.9: The Parametric Fortran Template for the Bell-Shaped Space Convolution

The Fortran program in Figure 5.10 can then be generated automatically by the Parametric Fortran compiler.
subroutine spaceConv (X, Y, L, U, N, ds, a, b)
    integer :: L, U, N
    real :: ds
    real, dimension (L:U, X:Y) :: a, b, theta, rhs
    integer :: n, i
    integer :: i1
    do i1 = X, Y
        do i = L, U
            theta(i,i1) = a(i,i1)
        end do
    end do
    do n = 1, N
        call computeRHS(X, Y, theta, rhs)
        do i1 = X, Y
            do i = L, U
                theta(i,i1) = theta(i,i1) + ds*rhs(i,i1)
            end do
        end do
    end do
    do i1 = X, Y
        do i = L, U
            b(i,i1) = theta(i,i1)
        end do
    end do
end subroutine spaceConv

Figure 5.10: The Fortran Program for the Bell-Shaped Space Convolution

5.2.3 Benefits of Using Parametric Fortran in the IOM

The use of Parametric Fortran has 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 5.2 shows the benefits obtained from using Parametric Fortran to develop the IOM system. We list 5 models that the IOM is currently applied to, and 5 modules in the IOM that are automatically generated. Each number is 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.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Model</th>
<th>SW2D</th>
<th>KDV</th>
<th>PEZ</th>
<th>ADCIRC</th>
<th>ROMS</th>
<th>Σ</th>
<th>PF</th>
<th>1-PF/Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Markovian Time Convolution</td>
<td></td>
<td>20</td>
<td>38</td>
<td>60</td>
<td>38</td>
<td>60</td>
<td>202</td>
<td>16</td>
<td>93%</td>
</tr>
<tr>
<td>Bell-Shape Space Convolution</td>
<td></td>
<td>25</td>
<td>62</td>
<td>93</td>
<td>62</td>
<td>93</td>
<td>335</td>
<td>24</td>
<td>93%</td>
</tr>
<tr>
<td>Main Convolution Module</td>
<td></td>
<td>374</td>
<td>360</td>
<td>502</td>
<td>390</td>
<td>578</td>
<td>2234</td>
<td>245</td>
<td>89%</td>
</tr>
<tr>
<td>Combination Module</td>
<td></td>
<td>147</td>
<td>147</td>
<td>251</td>
<td>147</td>
<td>273</td>
<td>965</td>
<td>177</td>
<td>82%</td>
</tr>
<tr>
<td>Measurement Module</td>
<td></td>
<td>98</td>
<td>98</td>
<td>162</td>
<td>98</td>
<td>162</td>
<td>618</td>
<td>102</td>
<td>83%</td>
</tr>
</tbody>
</table>

In the tools listed in Table 5.2, the Markovian time convolution and the bell-shaped space convolution have been explained in detail in Section 5.2.1 and 5.2.2. In some ocean models, the IOM generates multiple instances of these two convolution tools to convolve state 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 conver-
sion. The models listed in Table 5.2 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 generated modules, the code saving is at least 82%, which makes the development and maintenance much easier. One could question the savings because hand-written code could be shorter than the generated one. Apart from the quantitative considerations, we can also make the qualitative argument that no matter how many ocean models the IOM system will work with, the developers of the IOM only need to maintain one copy of their program. Instances of the IOM can be generated automatically for all the ocean models.

5.3 Application of Parametric Fortran to Automatic Differentiation

A model can be considered as a mapping of a vector 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. Automatic differentiation (AD) [6] 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. The tangent linear model maps small variations of the control variable $\delta X$ of a model to small variations of the
model prediction $\delta Y$. Therefore, tangent linear models can be used to quantify how small 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.

5.3.1 Implementing Automatic Differentiation in Parametric Fortran

The tangent linear model is constructed by applying the chain rule shown in Figure 5.11, 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 [15], including assignments, loops, conditionals, and 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

```fortran
  do i=1, 100, 1
```
\[c' = 0, \ c \text{ is a constant}\]
\[(u^k)' = ku^{k-1}u'\]
\[(e^u)' = u'e^u\]
\[(\sin(u))' = u'\cos(u)\]
\[(\cos(u))' = -u'\sin(u)\]
\[(\log(u))' = u'/u\]
\[(u + v)' = u' + v'\]
\[(u - v)' = u' - v'\]
\[(u*v)' = u'v + uv'\]
\[(u/v)' = (u'v - uv')/v^2\]

Figure 5.11: The Chain Rule for Computing the Tangent Linear Model

... end do

is

\[
\text{do i=100, 1, -1}
\]
\[
\text{...}
\]
\[
\text{end do}
\]

Rules for other statements are not that trivial. For example, the rules for calculating the adjoint code from assignments are achieved through the manipulations of the Jacobian matrix. The details can be found in [15].

We have implemented the chain rule in Figure 5.11 and the algorithm proposed in [15] as the generation function of a parameter type \texttt{Diff}. 
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 5.11 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 [15] 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 chain rule from Figure 5.11.

\[
\text{gen}_{p}[x = \sin(u)] = \text{tl}_x = \text{gen}_{p}[u] \times \cos(u) \\
\text{where } p = \text{TL } [..., x, ...]
\]

\[
\text{gen}_{p}[x = u + v] = \text{tl}_x = \text{gen}_{p}[u] + \text{gen}_{p}[v] \\
\text{where } p = \text{TL } [..., x, ...]
\]
When the parameter value is $\text{AD} \ldots$, the following code generates the adjoint code for a loop statement. The generation function reverses the loop boundaries and negates the loop step.

\[
\text{gen}_{\text{AD}} \{\text{do } i=l, u, \text{step } s \text{ end do}\}
= \begin{cases} 
\text{do } i=u, l, -\text{step } s \text{ end do} & \text{if } s \text{ changes any active variables in } us \\
\text{do } i=l, u, \text{step } s \text{ end do} & \text{otherwise}
\end{cases}
\]

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 [5] 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 [15]. As a concrete example, if $x$, $y$, and $z$ are all active variables, the adjoint code of the assignment

\[
z = x \times \sin(y^2)
\]

is shown below.

\[
\begin{align*}
\text{ad}_y &= \text{ad}_y + \text{ad}_z \times x \times \cos(y^2) \times 2 \times y \\
\text{ad}_x &= \text{ad}_x + \text{ad}_z \times \sin(y^2) \\
\text{ad}_z &= 0
\end{align*}
\]

We have applied the AD tool to generating differential programs for the Primitive Equation Z-coordinate Model (PEZ), which is a variant of the Bryan-Cox-Semtner class model [28], developed jointly at Oregon State University and the
National Center for Atmospheric Research. In the remainder of this section, we will demonstrate how to use the AD tool for a practical example. The Haskell code implementing the parameter type `diff` is shown in Appendix C.

5.3.2 An Example: The Inviscid Burger’s Model

In this section and Section 5.3.3 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 [9], which is widely used in physics [41, 23, 1]. 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. For example, $u$ represents the turbulence in a dynamic fluid system in [1].

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

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 [9]. Below we show the discrete equations of the inviscid Burger’s model.

$$u_x^0 = u_x^0 - \frac{\Delta t}{2\Delta x} u_x^0 (u_x^1 - u^0_{X-1})$$

$$u_x^1 = u_x^0 - \frac{\Delta t}{2\Delta x} u_x^0 (u_{x+1}^0 - u_{x-1}^0), \text{ for } x = 1, 2, \ldots, X - 1$$
\[ u^1_X = u^0_X - \frac{\Delta t}{2\Delta x} u^0_X (u^0_0 - u^0_{X-1}), \]

\[ u^{t+1}_0 = u^{t-1}_0 - \frac{\Delta t}{\Delta x} u^t_0 (u^t_1 - u^t_X), \text{ for } t = 1, 2, \ldots, T - 1 \]

\[ u^{t+1}_x = u^{t-1}_x - \frac{\Delta t}{\Delta x} u^t_x (u^t_{x+1} - u^t_{x-1}), \text{ for } t = 1, 2, \ldots, T - 1 \quad x = 1, 2, \ldots, X - 1 \]

\[ u^{t+1}_X = u^{t-1}_X - \frac{\Delta t}{\Delta x} u^t_X (u^0_0 - u^t_{X-1}), \text{ for } t = 1, 2, \ldots, T - 1 \]

The discrete equations describe how the value of \( u \) changes over time. Time ranges from 0 to \( T \), and space ranges from 0 to \( X \). \( \Delta t \) and \( \Delta x \) represent the length of a time step and the size of a spatial grid, respectively. The values of \( u^0_x \), for \( 0 \leq x \leq T \) 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.

5.3.3 Differentiating the Inviscid Burger’s Model Using Parametric Fortran

The discrete equations can be directly translated into the following Parametric Fortran template shown in Figure 5.12, which is parameterized by the parameter \texttt{diff} of type \texttt{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 values for \texttt{diff} can be \texttt{TL [u]} or \texttt{AD [u]} since \( u \) is the only active variable in
Figure 5.12: The Parametric Fortran Template for the Inviscid Burger’s Model

the inviscid burger’s model. The constructors 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.
When the value for \texttt{diff} is \texttt{TL} \, \texttt{[u]}, the tangent-linear program of the inviscid Burger's model will be generated. The following code fragment illustrates the generated Fortran subroutine for calculating the tangent-linear derivative.

```fortran
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
```

The program generator changes the output variable name of the generated subroutine to \texttt{tl\_u} to distinguish from \texttt{u}. In addition to this name change, the only difference from the original subroutine is in the assignment statement for \texttt{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 a new assignment is generated.

When the value for \texttt{diff} is \texttt{AD} \, \texttt{[u]}, the program for the adjoint model of the inviscid Burger's model will be generated as shown in Figure 5.13.

Similar to the tangent-linear program, a new variable \texttt{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 [15] for assignment statements. Following the rule for loop statements shown in Section 5.3.1, the loops over time and space are both reversed.
subroutine ad_burger(X,T,dx,dt,u,ad_u)  
...  
do t = T-1, 0, -1  
...  
do x = X, 0, -1  
...  
ad_u(x,tm1) = ad_u(x,tm1)+ad_u(x,tp1)  
ad_u(x,t) = ad_u(x,t)-(c*(u(xp1,t)-u(xm1,t)))*ad_u(x,tp1)  
ad_u(xp1,t) = ad_u(xp1,t)-(c*u(x,t))*ad_u(x,tp1)  
ad_u(xm1,t) = ad_u(xm1,t)+(c*u(x,t))*ad_u(x,tp1)  
ad_u(x,tp1) = 0  
end do  
end do  
end subroutine ad_burger

Figure 5.13: The adjoint program for the inviscid Burger’s model

5.4 Implementation of the Parametric Fortran Compiler

The Parametric Fortran compiler is implemented in Haskell. Every Fortran syntactic construct is represented by a Haskell data type. The Parametric Fortran compiler takes a Parameterized Fortran program as input, performs a generic traversal on the syntax tree of the program using the “Scrap Your Boilerplate” approach proposed in [24], and applies a generation function to every parameterized Fortran construct in that program with the provided parameter values. The output of the compiler is the generated Fortran program. The generation function is defined for any parameter type and for any Fortran construct in a Haskell type class.

This section is organized as follows. In Section 5.4.1 we show the Haskell data types for the Fortran and Parametric Fortran syntactic constructs. The
type class for program generation is defined in Section 5.4.2. In Section 5.4.3 we demonstrate how to define a parameter type for the array addition example shown in Section 5.1.1. The transformation function that performs the program generation for Parametric Fortran programs is defined in Section 5.4.4. The implementation of accessors and list parameters is shown in Sections 5.4.5 and 5.4.6, respectively.

5.4.1 Representing Fortran Syntax in Haskell

The abstract syntax of Fortran programs is represented by a collection of Haskell data types that can represent only syntactically correct Fortran programs. Therefore, the syntax correctness of the generated Fortran programs is automatically guaranteed by the type system of Haskell. Below we show part of the data type definitions for Fortran statements, expressions, and array indices. For example, the data type Stmt contains assignments, subroutine calls, and sequential statements. These data types must be instances of type classes Typeable and Data, because we have to apply the functions cast and everywhere’ [24] to values of these types.

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

data Expr = Con String
          | Var VName IndP
          | Bin BinOp ExprP ExprP
```
For example, the const expression 2 in Fortran can be represented by the Haskell expression \texttt{Con "2"}. A syntactic object may contain parameterized sub-objects. For example, a Fortran sequential statement, whose type is \texttt{Stmt}, may contain parameterized Fortran statements, that have the type \texttt{StmtP}. If \texttt{s1} and \texttt{s2} are the Haskell representation of two Fortran statements, the sequence of \texttt{s1} and \texttt{s2} can be represented by the following Haskell value of the the type \texttt{Stmt}.

\[
\texttt{FSeq (S Void s1) (S Void s2)}
\]

In this expression, \texttt{S Void s1} represents a Parametric Fortran statement that is the same as \texttt{s1} and is parameterized by nothing. The parameter value \texttt{Void} will be introduced in Section 5.4.2.

The code below shows the data types that represent parameterized Fortran statements, expressions, and array indices.

\[
\begin{align*}
\texttt{data IndP} &= \forall p . \texttt{Param p Ind} \Rightarrow \texttt{I p Ind} \\
\texttt{data StmtP} &= \forall p . \texttt{Param p Stmt} \Rightarrow \texttt{S p Stmt} \\
\texttt{data ExprP} &= \forall p . \texttt{Param p Expr} \Rightarrow \texttt{E p Expr}
\end{align*}
\]

For example, the data type \texttt{ExprP} can hold a parameter value and a Fortran expression. By using the \texttt{forall} quantifier in the data type definition we allow different expressions in one syntax tree to be parameterized by parameters of
different types. For example, in the syntax tree shown in Figure 5.2, the parameters p1 and p2 have different types. Both of them can parameterize a Fortran expression, which has the type \texttt{Expr}. Although the expressions are parameterized by parameters of different types, the parameterized ones have the same type \texttt{ExprP}. By defining a similar type for every Fortran syntactic category, any part of a Fortran program can be parameterized.

As another example, consider the following Parametric Fortran assignment.

\begin{verbatim}
{dim(a,b): a(i) = 2 * b(i)}
\end{verbatim}

This statement is represented in the abstract syntax as follows (by \texttt{s'}).

\begin{verbatim}
i = Var (VName "i") (Ind []) -- the variable i
a_i = Var (VName "a") (Ind [E Void i]) -- a(i)
a_i' = E (VarName "dim") a_i -- \{dim: a(i)}
b_i = Var (VName "b") (Ind [E Void i]) -- b(i)
a_i'' = E (VarName "dim") b_i -- \{dim: b(i)}
bi2 = Bin Mul (E Void (Con "2")) b_i' -- 2*\{dim: b(i)}
s = Assg a_i' (E Void bi2) -- \{dim: a(i)} = 2*\{dim: b(i)}
s' = S (VarName "dim") s -- \{dim : s}
\end{verbatim}

In this Parametric Fortran statement, the parameter \texttt{dim} is used to parameterize the \texttt{a(i)}, \texttt{b(i)}, and the assignment statement that is represented by \texttt{s}.

5.4.2 A Type Class for Program Generation

We define a multiple parameter type class \texttt{Param} to represent the relation between a parameter type and the data type of a syntactic construct. Again, the context \texttt{Data p} is needed because we want to use the function \texttt{everywhere'} [24].
to implement generic traversals of the syntax tree together with a type-based
selective application of a generation function.

```haskell
class (Show p, Data p, Show e) => Param p e where
  showP :: p -> e -> String
  gen :: p -> e -> e
  -- default implementations
  gen _ = id
  showP p e | isVoid p = e'
             | otherwise = '{' ++ p ++ ++ e ++ '}'
             where (p', e') = (show p, show e)
```

In the above definition, \( p \) represents the parameter type and \( e \) represents the
Fortran syntactic construct. The member function \( \text{showP} \) is used for pretty-
printing the parameterized program. For example, if the Fortran array expression
\( a(0) \) is parameterized by a parameter \( p_1 \), it will be pretty-printed by the function
\( \text{showP} \) as follows.

\[
\{p_1: a(0)\}
\]

The program generator \( \text{gen} \) takes a parameter value and a Fortran syntactic
object as input and generates a non-parameterized syntactic object.

The data type \( \text{Void} \) is a parameter type used in those cases in which no
parameter is needed. If a Fortran object is parameterized by \( \text{Void} \), it is pretty-
printed by the function \( \text{showP} \) as plain Fortran.

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

\( \text{Void} \) can be used to parameterize any Fortran syntactic category. In Section 5.4.4
we will demonstrate how parameterized programs are transformed to programs
parameterized by `Void`. For example, the following code shows the instance declaration of the type class `Param` for `Void` and Fortran expressions.

```haskell
instance Param Void Expr where

The body of the instance declaration is empty because the type `Void` just uses the default definitions for the functions `gen` and `showP`. Such an instance declaration is defined for `Void` and every Fortran syntactic category.

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`, the parameter type `VarName` can be used to parameterize any syntactic category and its `gen` function does nothing.

```haskell
data VarName = VarName String
deriving (Eq, Typeable, Data, Show)
```

The Parametric Fortran parser takes a Parametric Fortran program as input and outputs an abstract syntax tree represented by the Haskell type `ProgramP`. In the parsing result, all parameters have the type `VarName`. For generating Fortran programs, we replace every parameter name in the abstract syntax tree with the value for that parameter. We have defined a derived type class `Par p` to succinctly express that `p` is a valid parameter type of Parametric Fortran.

```haskell
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 the 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 [24] to extend generic queries of type Typeable a => a->r by a new type-specific case.

5.4.3 Defining Parameter Types as Instances of Param

Parameter types are represented by Haskell data types. The following code shows the data type of the parameter type for the parameter dim in the array addition example in Section 5.1.1.

    data Dimension = Dim Int
                    deriving (Show, Typeable, Data)
When a parameter type is provided, its instance for the type class \texttt{Param} must be defined for every syntactic category to guide the program generation. A parameter type usually only affects a few Fortran syntactic categories. For the syntactic categories that are not affected by the parameter type, the default implementation of the function \texttt{gen} is used, which leaves the syntactic construct unchanged.

In this example, the parameter \texttt{dim}, which is represented by the Haskell data type \texttt{Dimension}, affects types, statements, and expressions.

\begin{verbatim}
instance Param Dimension Type where
  gen (Dim d) (BaseType bt) = ArrayT (indx d) bt
  gen p t = t
  where indx d = replicate d [(a,b)]

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

a = i2e 1
b = i2e 100

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

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

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

In this definition for the parameter type \texttt{Dimension}, we suppose that the size of every array dimension is fixed to 100 (see definition of \texttt{b}). It is not difficult to extend the data type definition to allow the array dimensions to have different sizes. Instead of containing only an integer value, a value of the type \texttt{Dimension} now contains a list of integer pairs, which specify the lower and upper boundaries of all the dimensions. The number of dimensions is the length of the list.

\begin{verbatim}
data Dimension = Dim [(Int, Int)]
deriving (Show, Typeable, Data)
\end{verbatim}

The function \texttt{gen} for the parameter type \texttt{Dimension} can be changed as follows. The essential change happens in the instance definition for Fortran types and statements. The instance definition for expressions remains practically unchanged because the sizes of array dimensions do not affect indices of array expressions.
instance Param Dimension Type where
    gen (Dim bs) (BaseType bt) = ArrayT bs' bt
    gen p t = t
    where bs' = map (\(l,u)\)\>(i2e l, i2e u)) bs

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

Types parameterized by Dimension are expanded by the boundaries of dimensions provided in the parameter value. The boundaries are also used as loop boundaries of the generated loops for Fortran statements.

5.4.4 Transformation Functions

For every Fortran syntactic construct, 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 Fortran object under consideration. Other Fortran syntactic categories are dealt with similarly.

\[ \text{transS :: StmtP} \rightarrow \text{StmtP} \]
\[ \text{transS (S p s)} = \text{S Void (gen p s)} \]
transE :: ExprP -> ExprP
transE (E p e) = E Void (gen p e)

We can observe that every transformation function has the type a->a, which enables us to apply the function everywhere’ [24] 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 that has the following 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

The function 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 constructs. 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 in a top-down manner. Therefore, genF can be defined in the following way.

genF :: Data g => g -> g
genF = everywhere’ (id ‘extT‘
transS ‘extT‘
transE ‘extT‘
...
)
The function \texttt{genF} takes a parameterized Fortran program as input and outputs a Fortran program in which every syntactic object is parameterized by \texttt{Void}. The source code of the generated Fortran program can be obtained by calling the \texttt{showP} function.

5.4.5 Implementation of Accessors

A parameter field is used as a normal parameter in a Parametric Fortran program. For program generation, we have to substitute a parameter field by its value. In Section 5.4.2 we demonstrated how to replace a parameter name with its value. In this Section, we will show how to replace a parameter field \texttt{p.f} with its value.

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

\begin{verbatim}
  data Accessor = Accessor VarName String
                   deriving (Typeable,Data)
\end{verbatim}

For example, the function \texttt{substAccE} maps an expression parameterized by a parameter field to an expression parameterized by the value of the parameter field. There are similar functions for each Fortran syntactic category.

\begin{verbatim}
  substAccE :: ExprP -> ExprP
  substAccE (E p e) =
    case getAccE p of
        Just (Accessor v f) ->
            case lookup v param of
\end{verbatim}
Just (ParV q) -> E (AccessorV q v f) e
Nothing -> paramNotFoundError v
Nothing -> substE (E p e) -- p is not a parameter field,
-- do the normal substitution

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

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

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

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

Every parameter type that has fields is an instance of the type class AccessClass and must define the member function access. The function access takes a parameter value and a field name as input, the return type is Maybe ParV for dealing with possible errors. The type definition of ParV was introduced in Section 5.4.2.
When the field name passed to `access` is not defined, `Nothing` is returned. Otherwise, the field value is returned.

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

The following code shows the definition of the parameter type `Slice`, which is the type of the parameter `p` in the array slicing example in Section 5.1.3.

```haskell
data Slice = Slice Dimension Dimension Dimensions Inds
data Dimension = Dim Int
data Dimensions = Dims [Int]
data Inds = Inds [String]
```

The instance definition of the type class `AccessClass` for the parameter type `Slice` is shown below.

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

The member function `access` returns the value for a field name of the parameter. If the input field does not exist, a `Nothing` value is returned.

For example, the parameter `p` used in the array slicing example in Section 5.1.3 has the following value of the type `Slice`.

```haskell
p :: Slice
p = Slice (Dim 4) (Dim 2) (Dims [1,3]) (Inds ["i", "j"])
```
When the parameter field \( p.n \) is used to parameterize an expression in a program, the parameter field is represented by the following value of the type \texttt{Accessor} in the syntax tree of the Parametric Fortran program.

\[
\text{Accessor (VarName "p") "n" -- representation of } p.n
\]

The function \texttt{substAccE} then substitutes the above value by a value of the type \texttt{AccessorV}, which has the following value.

\[
\text{AccessorV } p \text{ (VarName "p") "n"}
\]

This value also contains the value of the parameter \( p \). When the \texttt{gen} function of the type \texttt{AccessorV} is called on this value, the function \texttt{gen} calls

\[
\text{access } p \text{ "n"}
\]

and obtains the following value of the type \texttt{Maybe ParV}.

\[
\text{Just (ParV (Dim 4))}
\]

Finally, the \texttt{gen} function of the type \texttt{Dimension} is called on the parameter value \texttt{Dim 4} and the Fortran expression that is parameterized by \( p.n \).

5.4.6 Implementation of List Parameters

The value of a list parameter is a list of Parametric Fortran parameters. The program generator generates a sequence of syntactic objects when a Fortran syntactic object is parameterized by a list parameter. This is accomplished by making a list of parameters an instance of the type class \texttt{Param}. The following code shows the instance definition for the list type and the data type of Fortran statements.
instance (Param p Stmt) => Param [p] Stmt where
  gen [] s = NullStmt
  gen (p:ps) s = S p s `FSeq` S ps s

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

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

The type \texttt{RepList} is defined as follows.

\[
data \texttt{RepList} = \text{forall } p . (\text{Data } p, \text{AccessClass } p) \\
\quad \Rightarrow \text{RepList VarName } [p]
\]

The data type \texttt{RepList} is used to store the name and value of a list parameter. The value of a list parameter is a list of parameter values and is captured by the list type \([p]\) in the data type definition. The name of a list parameter has to be remembered to solve the problem caused by the combination of list parameters and accessors.
The generation of a list of syntactic objects parameterized by $p_1.f$, $p_2.f$, ..., $p_n.f$ is performed by the function `repParam`. The function `repParam` replaces an accessor on a list parameter with a specific parameter value in the list. `repParam` takes a parameter value in a list and the name of the list parameter as arguments and performs an `everywhere` traversal on a subtree of a Parametric Fortran program. When `repParam` finds an accessor, it checks if the parameter name in the accessor matches the name of the list parameter. If the names match, the accessor is replaced by the accessor whose parameter value is replaced with the specific parameter value.

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

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

The function `readParam` reads a parameter value from a pair of strings. The first element of the pair represents the parameter name and the second represents the parameter value. The value of a list parameter is transformed into a value of the type `RepList`. For each parameter type, `readParam` calls `readP`, stops when there is a match and returns a parameter name and value. When there is no match, the compiler reports an error. If the parameter value is a list, we package it inside a `RepList` along with its parameter name.
The following code shows how to create an instance of the `Param` type class for `RepList` and Fortran statements.

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

The program generator creates a sequence of statements, each of which is parameterized by a value from the list. One thing to notice is that the result of the function `gen` for the type `RepList` contains parameters that are not `Void`. In the Fortran programs generated by the function `genF` (refer to Section 5.4.4), all parameters will be `Void` because `genF` applies the `gen` function in a top-down manner and the function `gen` will be called on the non-`Void` parameters produced by the generation function of `RepList`.

For example, in the example shown in Section 5.1.4, the list parameter `stateVars` is used to parameterize the declaration in the Parametric Fortran program as follows.

```haskell
{-#stateVars:
  {stateVars.dim : real :: stateVars.name}
#
```

In the declaration statement parameterized by `stateVars`, two parameter fields of `stateVars` are used. Consider the following value for `stateVars`. 
stateVars = [temp, veloc]
temp = {dim=3, name="temperature"}
veloc = {dim=2, name="velocity"}

The original declaration is transformed to the following two declarations by the gen function of the type RepList.

{temp.dim : real :: temp.name}
{veloc.dim : real :: veloc.name}

Finally, the following two Fortran declarations are generated from the above two statements.

real, dimension (:,:,,:), allocatable :: temperature
real, dimension (:,:), allocatable :: velocity
Chapter 6 – Conclusion and Future Work

This thesis presents two approaches to automatically generating Fortran programs for addressing the code reuse problem in scientific computing. In the first approach, we have designed a domain-specific language, Forge, for specifying inverse ocean modeling tools that can be represented by discrete equations. The domain-specific language specifically deals with the code reuse problem caused by representation of arrays, such as number of array dimensions and meaning of dimensions. The information about array representation is represented by parameters that are used in Forge specifications of inverse ocean modeling tools. A program generator can generate a Fortran subroutine automatically from the Forge specification of an inverse ocean modeling tool when values for the parameters used in the specification are provided. The generated Fortran subroutine implements the tool particularly for arrays represented by the parameters. A type system for Forge is also developed to prevent the generated Fortran programs from containing type errors.

The second approach is to extend Fortran with support for generic programming. The extension is called Parametric Fortran and allows Fortran programmers to define program templates. The program templates can be used to implement algorithms whose actual implementation depends on other information, such as the number of dimensions of arrays the Fortran programs deal with. The
information is represented by parameters used in the templates. When values for the parameters are available, a program generator can translate such program templates into normal Fortran programs.

At first, Forge was used to specify the inverse ocean modeling tools in the IOM system. However, since Forge is targeted at solving the code reuse problem caused by array representations, it is not expressive enough to define the IOM inversion programs that depend on other model-dependent information, such as the number of state variables used in ocean models. In contrast, Parametric Fortran is expressive enough to implement the whole IOM system in a generic way. All the model-dependent information in the IOM system can be represented by parameters in program templates implementing the IOM system. When developers of an ocean model want to use the IOM system to analyze/improve their model, they can provide their model information in the form of parameter values, and the program generator can generate an instance of the IOM specifically for their model. Therefore, we have replaced Forge with Parametric Fortran for implementing the IOM.

The successful application of the presented approaches to the IOM system also demonstrates how computer scientists can help scientists of other areas through techniques of programming language design. Another important aspect of the presented program-generation approaches is that they have successfully integrated the work of people who work in different areas. Three groups of people, namely ocean scientists who develop ocean models, ocean scientists who develop data assimilation programs, and computer scientists, work collaboratively in both
approaches in a very modular way. The GUI of the IOM system is the best example to show this collaboration. Developers of ocean models provide their model information through the GUI and customize the data assimilation programs they want to run for their model. The data assimilation programs are implemented by developers of the IOM system as program templates. When the model information is provided, the GUI is able to automatically generate the customized data assimilation program for the particular model. Computer scientists maintain the GUI program and realize the program generation by defining parameter types.

Although the whole IOM system has already been implemented in Parametric Fortran successfully, Parametric Fortran is not only limited to implementing the IOM system. This thesis also presents a successful application of Parametric Fortran to automatic differentiation.

Parametric Fortran provides a mechanism to support template programming in Fortran, which makes the generic programming paradigm possible in Fortran. Furthermore, the generic programming support provided by Parametric Fortran is more expressive than the counterparts in other programming languages, such as C++ templates and array programming languages. Especially, Parametric Fortran provides solutions for the code reuse problem in scientific computing that cannot be dealt with in those programming languages.

Parametric Fortran does not provide a mechanism to prevent generated Fortran programs from type errors. For example, in the array slicing example shown in Section 5.1.3, the parameter fields \( n \) and \( o \) are redundant considering we know how many dimensions to slice by the length of the field \( \text{dims} \). The following
invariant should hold.

\[ p.n = p.o + \text{length dims} \]

Since only parameter names or field names can be used as parameters, but not expressions of parameters, we can remove neither \( p.n \) nor \( p.o \) to eliminate the redundancy. Furthermore, the above relationship between the field values is not guaranteed. If users provide field values for which the above relation does not hold, the generated program will contain type errors.

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


[33] T. Sheard. Accomplishments and Research Challenges in Meta-
Programming. In 2nd Int. Workshop on Semantics, Applications, and Im-

[34] Simulog, SA, Guyancourt, France. FORESYS, FORtran Engineering SYS-

[35] K. Smillie. A lecture on array languages. SIGAPL APL Quote Quad,

Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++
Project.

[37] P. van der Mark, R. van Engelen, K. Gallivan, and W. Dewar. A Case Study
for Automatic Code Generation on a Coupled Ocean-Atmosphere Model. In
2002 International Conference on Computational Science, pages 419–428,
2002.

[38] R. van Engelen, L. Wolters, and G. Cats. The CTADEL Application Driver
for Numerical Weather Forecast Systems. In 15th IMACS World Congress,

tional Scientific Computing in Object-Oriented Parallel Environments (IS-

Appendix A: The Syntax of Forge

A specification \((Spec)\) has a name followed by a list of model parameter declarations \((MPList)\) and consists of an interface part \((Interface)\) and a body, which is given by a statement \((Stmt)\). A Forge statement can be an assignment, a call to a Fortran subroutine, a loop statement, a sequential statement, or a call to a library function, see Figure 1.

| Spec | ::= | TName \((MPList)\) Interface Stmt |
| MPList | ::= | Type::PName ; . . . ; Type::PName |
| Stmt | ::= | FName\{PNames \mid \textbf{Stmt} \mid \textbf{SubCall} \mid \textbf{DoStmt} \mid \textbf{Stmt} ; \textbf{Stmt}\} |
| PNames | ::= | PName , . . . , PName |
| SubCall | ::= | call SubName\((FParam , . . . , FParam)\) |
| DoStmt | ::= | \textbf{do} VName=Expr , Expr Stmt |
| FParam | ::= | Expr | IName |
| Assg | ::= | VName=Expr | Array=Expr |
| Array | ::= | VName\[Expr , . . . , Expr\] |
| Expr | ::= | PName | VName | Array | Const | UOp Expr | Expr BOp Expr |
| UOp | ::= | - |
| BOp | ::= | + | - | * | / |

Figure 1: Syntax of specifications.

The syntax of interfaces is summarized in Figure 2, which is built on the syntax for dependent-index declarations shown in Figure 3.

The syntax of types is shown in Figure 4. \texttt{Bty} is used for types of variables in tool specifications and model parameters. \texttt{Aty} is only used for array variables used in tool specifications. The type \texttt{subroutine} is for subroutine parameters. All other types are used only internally for type checking and cannot be used
Interface ::= DepIx; Param; Local;
DepIx ::= index IxDecl; . . . ; IxDecl
Param ::= param VarDecl; . . . ; VarDecl
Local ::= local VarDecl; . . . ; VarDecl
VarDecl ::= Type::VName, . . . , VName

Figure 2: Syntax of interfaces.

IxDecl ::= IName=DepExpr
DepExpr ::= VName::VName[Expr::Expr]

Figure 3: Syntax of dependent-index declarations.

explicitly in tool specifications and will be illustrated in Section 4.3.

Type ::= Bty | Aty | subroutine | depix | range | fortran
     | Type->Type
Bty ::= integer | real
Aty ::= Bty, dimension (Ind)
Ind ::= FName{PNames | FixInd, DepInd} | DepInd | FixInd
DepInd ::= IName | DepExpr
FixInd ::= Range | Range, FixInd
Range ::= Expr::Expr

Figure 4: Forge types.

Finally, the syntax of model specifications is defined in Figure 5.

Model ::= model Name; PDef; . . . ; PDef
PDef ::= PName=Expr | TName.PName=Expr

Figure 5: Syntax of parameter definitions.
Appendix B: The Fortran Code of Markovian Time Convolution Generated by Forge for PEZ

subroutine timeConv(X1, Y1, X2, Y2, X3, Y3, L, U, dt, tau, a, b)
  integer :: dim3i
  integer :: X3
  integer :: Y3
  integer :: dim2i
  integer :: X2
  integer :: Y2
  integer :: dim1i
  integer :: X1
  integer :: Y1
  real :: dt, tau
  integer :: L, U
  real , dimension (L:U, X1:Y1, X2:Y2, X3:Y3) :: a, b
  real , dimension (L:U, X1:Y1, X2:Y2, X3:Y3) :: h
  integer :: n
  do dim3i = X3, Y3, 1
    do dim2i = X2, Y2, 1
      do dim1i = X1, Y1, 1
        h(L,dim1i,dim2i,dim3i) = 0.0
        do n = L+1, U, 1
          h(n,dim1i,dim2i,dim3i) = h(n-1,dim1i,dim2i,dim3i) &
            -dt*(h(n-1,dim1i,dim2i,dim3i)/tau+2.0*a(n,dim1i,dim2i,dim3i)/tau)
        end do
        b(U,dim1i,dim2i,dim3i) = -0.5*h(U,dim1i,dim2i,dim3i)/tau
        do n = U-1, L, -1
          b(n,dim1i,dim2i,dim3i) = &
            b(n+1,dim1i,dim2i,dim3i)-dt*(h(n,dim1i,dim2i,dim3i) &
              +b(n+1,dim1i,dim2i,dim3i)/tau)
        end do
      end do
    end do
  end do
end subroutine timeConv
Appendix C: The Haskell Implementation of the Parameter Type

Diff

-- Diff.hs
-- This file implements the parameter type Diff
--

module Diff where
import Data.Generics
import Param
import FortranP

-- Diff parameter
--
data Diff = TL [VarName]
  | AD [VarName]
  | NoDiff -- dummy value that represents no parameterization
        deriving (Show, Data, Typeable, Read)

-- Programs are not affected by Diff parameters
--
instance Param Diff Program where

-- Diff parameters can affect the variable name in parameter list
-- e.g. x -> tl_x or ad_x if x is an active variable
--
instance Param Diff ArgName where
  gen (TL vs) (ArgName a) = if elem (VarName a) vs
    then ASeq (G Void (ArgName a))
        (G Void (ArgName ("tl_"++a)))
    else ArgName a
  gen (AD vs) (ArgName a) = if elem (VarName a) vs
    then ArgName ("ad_"++a)
    else ArgName a
  gen _ a = a

-- Diff parameters can affect variable declarations
For every active variable, generate a new declaration for its tangent-linear variable or adjoint variable.

nullE is a constant representing a null expression.

instance Param Diff Decl where
  gen (TL ts) d@(Decl vs t) = Decl zs (noDiff t)
    where ds = repMVs "tl_" ts (map fst vs)
        zs = zip ds (replicate (length ds) nullE)
  gen (AD ts) d@(Decl vs t) = Decl zs (noDiff t)
    where ds = repMVs "ad_" ts (map fst vs)
        zs = zip ds (replicate (length ds) nullE)
  gen _ d = d

Blocks are not affected

instance Param Diff Block where

Fortran types are not affected

instance Param Diff Type where

Diff parameters can affect Fortran statements

instance Param Diff Stmt where
  -- if the s does not change an active variable, remove all the Diff parameters in s; otherwise, leave s unchanged, the expressions in s will then be differentiated by the gen function
  -- for Expr
  gen (TL vs) s@(Assg e e') = if isActive e vs then Assg e e' else noDiff s
  -- if both s1 and s2 change active variables, reverse their order, otherwise, leave s unchanged.
  gen (AD _) s@(FSeq s1 s2) = if active s1 && active s2 then FSeq s2 s1 else s
-- if the loop body changes active variables, reverse the loop,
-- otherwise, leave s unchanged.
--
gen (AD _) s@(For v l u step body)
  = if active body
     then For v u l (negate s) body
     else s
     where negate = Void (Unary UMinus)
-- if the left-hand side of s is an active variable, create the
-- Jacobian matrix for the assignment, and generate statements
-- from the transposed Jacobian Matrix. The Giering’s algorithm
-- for assignments is implemented in the functions adje and adjs.
-- otherwise, leave s unchanged.
--
gen (AD vs) s@(Assg e e')
  = if isActive e vs
     then adjs (genF e) (adje vs (genF e'))
     else s
  gen _ s = s

-- Diff parameters can affect Fortran expressions
--
instance Param Diff Expr where
-- if the variable is active, change its name,
-- otherwise, return 0
--
gen (TL vs) e@(Var (v,es)) = if elem v vs
    then Var (vc "tl_" v,(noDiff es))
    else Con "0"
-- for other expressions, apply the chain rule to the expression
-- the chain rule is implemented by the function chainRule
--
gen (TL vs) e = chain_Rule e

-- if the variable is active, change its name,
-- otherwise, return 0
--
\[
\text{gen (AD vs) e@}(\text{Var (v,es)}) = \begin{cases} 
\text{if elem v vs} \\
\text{then Var (vc "ad_" v,(noDiff es))} \\
\text{else e}
\end{cases}
\]

-- for adjoint generation, the Jacobian matrix is obtained from
-- the tangent-linear code, so chainRule is also applied here.

\[
\text{gen (AD vs) e} = \text{chainRule e}
\]
\[
\text{gen _ e} = e
\]

-- utilities

-- the function chainRule applies the chain rule on the
-- input expression.

--

chainRule :: Expr -> Expr
-- c' = 0

chainRule (Con s) = Con "0"
-- (u*v)' = u'v + v'u

chainRule (Bin Mul u v) = Bin Plus (E Void (Bin Mul u (noDiff v)))
(E Void (Bin Mul (noDiff u) v))
-- (u/v)' = (u'v-uv')/v^2

chainRule (Bin Div u v) = Bin Div t0 t3
\text{where} t0 = E Void (Bin Minus t1 t2)
\quad t1 = E Void (Bin Mul (noDiff v) u)
\quad t2 = E Void (Bin Mul (noDiff u) v)
\quad t3 = E Void (Bin Mul (noDiff v) (noDiff v))
-- (u^-n)' = n*(u^-n-1)u'

chainRule (Bin Power u n) = Bin Mul (noDiff n) t3
\text{where} t1 = E Void (Con "1")
\quad t2 = E Void (Bin Minus (noDiff n) t1)
\quad t3 = E Void (Bin Power (noDiff u) t2)
-- (\sin(u))' = u'\cos(u)
chainRule (Sin u) = Bin Mul u t1
    where t1 = E Void (Cos (noDiff u))
-- (\cos(u)\') = -u'sin(u)
--
chainRule (Cos u) = Bin Mul u t2
    where t1 = E Void (Sin (noDiff u))
    t2 = E Void (Unary UMinus t1)
-- (e^-u)' = u'*e^-u
chainRule e@(Exp u) = Bin Mul u (E Void (noDiff e))
chainRule e = e

-- adje constructs the Jacobian matrix represented by
-- the type [(Expr,ExprP)]
--
adje :: [VarName] -> ExprP -> [(Expr,ExprP)]
adje vs (E p (Var (v,es)))
    = if elem v (map (vc "ad_") mvs)
        then [(Var (v,(noDiff es)),(E Void (Con "1")))]
        else []
adje vs (E p (Con s)) = []
adje vs (E p (Bin Mul e e'))
    = map (t e') (adje mvs e) ++ map (t e) (adje mvs e')
        where t f (v,f') = (v, E Void (Bin Mul f f'))
adje vs (E p (Bin Div e e'))
    = map (d e') (adje mvs e) ++ map (d e) (adje mvs e')
        where d f (v,f') = (v, E Void (Bin Div f' f))
adje vs (E p (Bin Power e e')) = []
adje vs (E p (Bin Plus e e')) = adje mvs e ++ adje mvs e'
adje vs (E p (Bin Minus e e'))
    = adje mvs e ++ adje mvs (E Void (Unary UMinus e'))
adje vs (E p (Unary UMinus e)) = map m (adje mvs e)
    where m (v,f) = (v, E Void (Unary UMinus f))
adje_ e = []

-- adjs creates adjoint code fragments from the transposed Jacobian matrix
adjs :: ExprP -> [(Expr,ExprP)] -> Fortran
adjs e [] = FSeq (F Void (Assg tmp e))
  (F Void (Assg e (E Void (Con "0")))))
  where tmp = toADtmp e'
adjs e ((v,e'):es)
  = FSeq (F Void (adjs e' es))
  (F Void (Assg (E Void v)
    (E Void (Bin Plus (E Void v)
      (E Void (Bin Mul e tmp)))))))
  where tmp = toADtmp e'

-- to decide if an expression changes is an active variable
--
isActive :: ExprP -> [VarName] -> Bool
isActive (E _ (Var ((v,):_))) vs = elem v vs

-- Replace all the parameters of the type Diff to NoDiff
--
noDiff :: (Data a) => a -> a
noDiff = everywhere (id `extT` removeDiff)
removeDiff :: Diff -> Diff
removeDiff p = NoDiff

repMVs :: String -> [VarName] -> [ExprP] -> [ExprP]
repMVs pre ts = concatMap (\(E p (Var (v,es))) ->
  if elem v ts
  then [E Void (Var (vc pre v,es))]
  else [E Void (Var (v,es))])

-- generate a temporary variable from an active variable
-- required by Giering’s algorithm
--
toADtmp :: ExprP -> ExprP
toADtmp (E p (Var [(VarName v,es)]))
  = E p (Var [(VarName (v++"_tmp"),[])])
toADtmp e = e
--- string concatenation for VarName
---
vc :: String -> VarName -> VarName
vc s (VarName v) = VarName (s++v)

nullE :: ExprP
nullE = E Void NullExpr

Appendix D: The Installation of Parametric Fortran

The distribution of Parametric Fortran can be downloaded from

http://web.engr.oregonstate.edu/~erwig/pf/PF1.0.tar.gz.

After unzipping the file, the source code of the Parametric Fortran compiler is in the directory src. The examples are in the directory examples.

For compiling the source code of the Parametric Fortran compiler, users must have ghc 6.2 or higher version on your computer. First, users need to set the value for the environment variable PFHOME, which is the directory in which to unzip the Parametric Fortran distribution. For example, if using the C Shell, the file PF1.0.tar.gz is unzipped in the directory /usr/local/PF1.0. Then the following line has to be placed in the file .cshrc.

```
set PFHOME = /usr/local/PF1.0
```

For Windows 2000/XP users, the source code can be compiled by running

```
pf.bat
```
in the directory `src`.

For Unix users, the source code can be compiled by running

```
pf
```

in the directory `src`.

The binary file of the Parametric Fortran compiler will be in the directory `bin`. We also provide binary file of the Parametric Fortran compiler for Windows 2000/XP users and Sun-Solaris users. The binary file for Windows can be downloaded from

```
http://web.engr.oregonstate.edu/~erwig/pf/pfc.exe.
```

The binary file for Sun-Solaris can be downloaded from

```
http://web.engr.oregonstate.edu/~erwig/pf/pfc.
```