Software Reuse for Scientific Computing Through Program Generation

MARTIN ERWIG and ZHE FU
Oregon State University

We present a program-generation approach to address a software-reuse challenge in the area of scientific computing. More specifically, we describe the design of a program generator for the specification of subroutines that can be generic in the dimensions of arrays, parameter lists, and called subroutines. We describe the application of 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. 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.

Categories and Subject Descriptors: D.2.13 [**Software Engineering**]: Reusable Software; D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms: Languages

Additional Key Words and Phrases: Software Reuse, Program Generator, Ocean Science, Inverse Modeling, Fortran, Haskell, Type System

1. INTRODUCTION

This paper describes the successful application of a program-generation approach to enable the reuse of software in an area of scientific computing. The motivating example for the reported research is a software project in ocean science, the Inverse Ocean Modeling (IOM) system [Chua and Bennett 2001; IOM], which requires the adaptation of inversion programs to different ocean models. Inversion programs are needed to obtain important feedback about the quality of, in particular, ocean models, and more generally, models from other scientific disciplines. One inversion program typically consists of several tools, which are implemented by subroutines and which work together with code implementing the ocean model. Different inversion programs can be obtained by selecting different sets of inversion tools.

A problem in this particular software project is that the same inversion algorithms and tools have to be (re)implemented for different ocean models because different models use different data structures to represent the ocean state and the inversion tools have to work on these data structures and have to interact with a code that

Authors' email addresses: erwig@cs.orst.edu and fuzh@cs.orst.edu.
This work was supported by the National Science Foundation under the ITR/AP grant OCE-0121542.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1049-331X/20YY/0700-0001 $5.00

already exists for the ocean models. Scientific models to predict the behavior of ecological systems are routinely transformed by scientists from a mathematical description into simulation programs. Since these simulation programs have to deal with huge data sets (up to terabytes of data), they are often implemented in a way that exploits the given computing resources as efficiently as possible. In addition to implementation strategies that try to make best use of parallel computer architectures, the representation of the data in the simulation programs is highly specialized for each model. Alas, this high degree of specialization causes significant software engineering problems that impact the advance of scientists in evaluating and comparing their models. One particular problem is that inversion programs currently have to be rewritten for each individual forecasting model, although the process of inversion is principally the same for all models (at least for one chosen inversion algorithm).

In particular, the software project poses the following challenges.

(1) Tools should be implemented or specified only once.
(2) Tool specifications should be readily usable, that is, scientists should not be required to modify the inversion tools.
(3) Existing model code should be reusable without needing any changes.
(4) The inversion programs should also work with future models.

Especially the third requirement rules out an approach to develop a software infrastructure that offers components with well-defined interfaces to implement composable and reusable components. This approach is pursued by the Earth System Modeling Framework (ESMF) collaboration [ESMF]. One drawback of that approach is that scientists have to implement their models against these newly defined interfaces, which is unfortunate because scientists are forced to reimplement (large parts of) their models. Refactoring a collection of Fortran programs (in many cases consisting of hundreds of files and ten thousands of lines of code) is a time-consuming and error-prone task that many scientists are just not willing to perform. Instead, they rather seem to prefer to reimplement inversion tools specifically targeted for their model.

The approach we have taken is driven by the goal to leave existing model code 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 inversion tools to that particular model. We have developed a language called Forge (an acronym for "**For**tran **ge**nerator"), 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 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.

We have employed this language for specifying inverse ocean modeling tools. Moreover, we have developed a graphical user interface that allows scientists to create complete inversions. Our approach can be understood as a program generator that facilitates program reuse on different levels. First of all, existing model code

can be reused in different kinds of inversion programs. Second, inversion tools that are defined in Forge can be reused for inversions of different ocean models, which is achieved by compiling them into Fortran code depending on a set of parameters that describe the particular ocean model. Finally, the graphical user interface allows users to select different inversion outputs, which triggers the generation of the required inversion tools and, in particular, the creation of a Fortran inversion program that computes the requested inversion outputs.

Although the program generator was specifically designed to be used with inversions for ocean modeling, the scope and applicability of the results reported in this paper 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. Inversion differs in objectives from, but is mathematically identical to, the engineering activity of optimal control of dynamical systems.

We believe that the emphasis that we have placed on software reuse in our approach was essential for its success. In fact, we have combined what has been called *generative reuse* with *compositional reuse* [Mili et al. 2002] in the following way. First of all, the code of each ocean model is completely reused. No adaptation is needed. The ocean model code is composed with inversion tools to create inversion programs, which is therefore an example of compositional reuse. Second, the reuse of inversion tools is realized through the program generator, that is, the tools are described only once and will be translated into Fortran programs according to the parameters that are defined for the current ocean model, which is therefore an example of generative reuse. Third, the inversion programs that are individually generated for particular selected inversion output are also examples for generative reuse.

In the remainder of this Introduction, we briefly describe the background of the application area and the program-generator approach. In Section 2 we present how to generate Fortran code for IOM tools through two examples. We will describe how the type system will prevent type errors in the generated Fortran programs in Section 3. In Section 4 we discuss related work, and conclusions given in Section 5 complete this paper. In the appendices we provide the complete syntax of Forge, the formal rules of the type system, and a complete example of a generated subroutine.

1.1 Tools for Inverse Ocean Modeling

Ocean scientists are using ocean models to simulate and predict the state of oceans. Ocean models are conventionally formulated as equations of motion. The equations are solved by numerical approximation. Different ocean models use different numerical approximations. Models usually use arrays over time and space to store the state values of oceans, such as velocity or temperature. Some models also consider data at fixed locations and use arrays only over time, or average over time and use arrays only over space.

The Inverse Ocean Modeling (IOM) system [Chua and Bennett 2001; IOM] is a data assimilation system, which enables the developers of ocean models to combine their models with real observations of the ocean. The output of the IOM is a weighted least-squared best-fit to the equations of motion and to the data.

The IOM consists of tools that are used for solving the equations of best fit. The information obtained by inversion programs gives important information about the quality of data produced by ocean and weather forecasting models. The accuracy of forecasting models is important for the successful planning of flight or ship routes, navy operations, and many other applications. In addition, inversion can also reveal important information about the efficiency of the observation process.

Since every ocean model uses its own data structure to describe the ocean, it is very difficult for the developers of the IOM to write a system that can work for different ocean models, although the algorithms for inverse ocean modeling are the same for all models. The problem is that the genericity that is inherent in the problem cannot be expressed by Fortran. For example, we cannot declare an array variable to have a varying number of dimensions. Moreover, we cannot add a varying number of loops over a block of statements or express array indexing for a variable number of dimensions. Similarly, we cannot define subroutines with a variable number of parameters, or pass subroutines as parameters to other Fortran subroutines. Although some of these language features are available in other programming languages, for example, function pointers in C or higher-order functions in Haskell [Peyton Jones 2003], or dimension-independent array functions in APL [Iverson 1984] or J [Iverson 1995], no single language provides all the required features. Moreover, since most of the existing ocean-model code is written in Fortran, we have to work with Fortran as a language, at least on the model side, anyway.

1.2 A Program Generator for Inverse Ocean Modeling Tools

Our solution is the design and implementation of a specification language Forge that can be used to generate the tools provided by the IOM. Tool descriptions 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. Individual tools are combined into larger inversion programs. This process is controlled by a graphical user interface that presents ocean modelers with a variety of inversion options, such as the selection of inversion outputs and the choice of different inversion algorithm options. The graphical user interface also controls system configurations, such as Makefile and Fortran compiler options, the definition of model parameters, and different execution options which are particularly important for efficient parallel execution on supercomputers. Currently, the GUI can support three different execution options: The IOM module and the model module can be 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 can be run as two separate executables that communicate via shared data files. We plan to support more fine-grained parallelism in future versions.

The graphical user interface is implemented in Java and runs together with the Forge and Fortran compilers on Windows, Mac OS, and Solaris. A snapshot of the current system is shown in Figure 1.

The ability to provide model parameters, to select a particular combination of inversion outputs, and to customize the inversion algorithm offers to ocean modelers

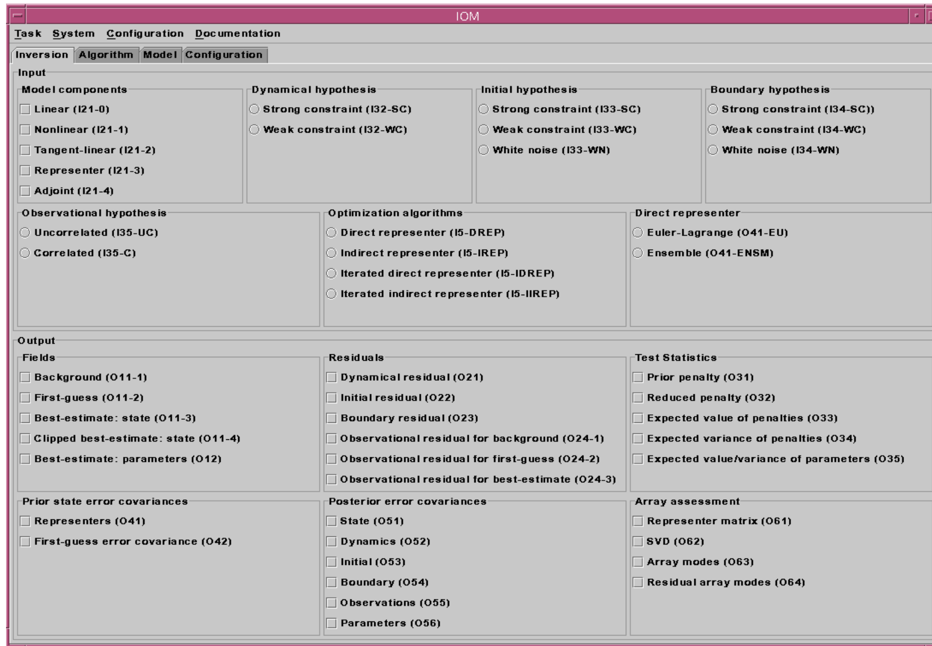

Fig. 1. Graphical User Interface.

a flexible customization of inversion programs along three dimensions. In this paper, we do not discuss the combination of tools into inversion programs, but rather focus on the issues concerning the specification of individual tools and the design and implementation of the program generator. Figure 2 illustrates the architecture of the system.

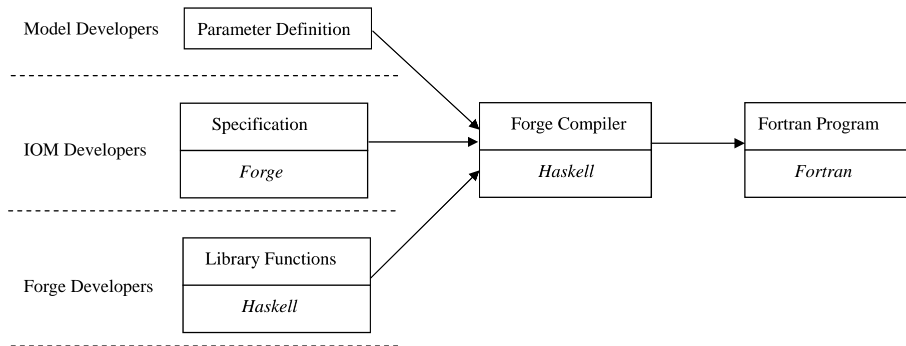

Fig. 2. System architecture.

The developers of the IOM system (this is the group of people developing in-

version 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 2.3. The library functions are written in Haskell [Peyton Jones 2003] 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 inputs and produces a Fortran subroutine. These generated subroutines, which correspond to the inversion tools, will be part of the inversion programs that are composed by the graphical user interface.

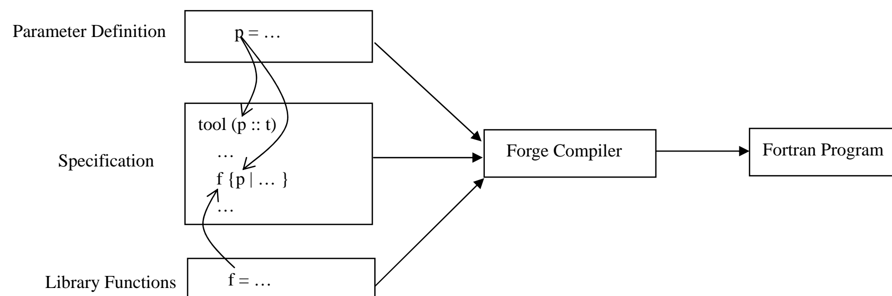

Fig. 3. Using model parameters in tools and library function calls

Figure 3 shows a schematic instance of Figure 2 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`, which 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.

1.3 The Forge Compiler

For the implementation of the Forge front end we have used the Haskell scanner generator Alex [Dornan 1997] and the parser generator Happy [Marlow and Gill 2000]. The main part of the Forge compiler is written in Haskell [Peyton Jones

2003]. There are several advantages of using a language like 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 [Sheard 2001]. This syntax-correctness guarantee is not tied to the use of Haskell, but can be achieved by using a corresponding representation in any strongly typed language. For example, in Java syntactic categories could be represented as abstract classes and different grammar rules could be represented as concrete classes that extend the abstract classes and that contain corresponding constructors.

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++.[1] Multi-language support has not been an explicit goal of this project, but the abstract syntax representation supports it well in case this issue should become important in the future.

Third, the core part of the Forge compiler consists of functions that translate equations into Fortran. Special attention has been paid to library functions that are implemented in Haskell. The translation functions need access to these library functions, which are referenced through the specification to be compiled.

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 that are part of the IOM system and that are used in practice. Convolution tools are extremely important in the area of data simulation. 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 a 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

[1] As anecdotal evidence, after presenting the current IOM system during the 2003 IOM Annual Conference, one participant from the Navy Research Lab Stennis asked about the possibility to use the whole system for their model NCOM, which is implemented in C++. After the presentation we were able to customize the IOM to produce C++ code in about 30 minutes.

different convolutions, a language like 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. There are dozens of different types of tools in the IOM system, for example, tools to check the consistency of models or tools to perform transformations from the data space to the IOM space. Each must be rebuilt for compatibility with each user's ocean model, according to the regular or irregular numerical grid, and this task can be automated by specifying the tools with Forge.

Furthermore, 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. Therefore, Forge can be also used by scientists working in other disciplines. However, Forge is not a general-purpose program generator. An essential ingredient of the covered tool descriptions are discrete equations. Therefore, tools that cannot be represented by discrete equations cannot be generated by Forge.

We provide some data on the use of Forge in Table I.

Application	N	ℓ	Σ
Markovian convolution in time	10	13	360
Bell-shaped convolution in space	32	20	816
Combination functional	25	44	1440
Low-pass filter for image processing	20	16	320
Low-pass filter for data simulation	10	16	160
Array slicing	55	8	1154

Table I. Size of generated Forge code

The main benefit of using Forge is not so much to generate one big program from one small specification, but rather to generate many different programs for different models from the same specification. In the table we show in the column labeled N the number of different parameter combinations, or models, for that particular application. Note that N is not just the number of dimensions of involved arrays. For example, in the first example application, the Markovian convolution in time, we consider only up to four-dimensional arrays. The remaining number of models result from the different possibilities to place the time dimension within the multi-dimensional array. Column ℓ gives the number of lines of the specification, and the last column shows the number of lines of code for all generated models. Markovian convolution in time and Bell-shaped convolution in space will be explained in detail in Sections 2.1 and 2.5, respectively. Combination functional is an IOM tool that maps a vector into an array whose shape is derived from two model parameters. Low-pass filters can be used in both image processing and data simulation for smoothing data. Array slicing can project an $(n-1)$-dimensional array out of an n-dimensional array by selecting a dimension to project and a value within that dimension.

2.1 An Example Tool: Markovian Convolution in Time

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

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

$F(x, x')$ 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, a is the weighted error, and b is the deweighted error. A similar kind of convolutions is also used in computer science. For example, in image processing, such convolutions are often used to reduce noises in an image [Castleman 1996].

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 ocean science and mathematics background can be found in [Chua and Bennett 2001]. We omit the derivation here since it is not important for the translation of the discrete equations into Fortran.

The *Markovian convolution in time* is one particular convolution tool defined by the following formula. The weighting function $F(t, t')$ is $\exp(-|t - t'|/\tau)$, and the variables t and t' range over time between 0 and T, the upper time boundary. The coefficient τ is the correlation time scale which is input by the modelers. The smaller τ is, the more quickly the values of the old points will be forgotten.

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

The corresponding discrete equations of Markovian convolution are as follows. 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.

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

In the above equations, h is a temporary array, and L (U) is the lower (upper) boundary of the arrays. The arrays are over time and space where different models may use different array dimensions for space. Moreover, different models represent the time dimension in different positions in array dimensions for efficiency reasons. Hence, in general, models differ in the dimensions of the manipulated arrays as well as in the interpretation of the stored data. Therefore, the IOM has to be able

to create different subroutines for all the possible data structures used in all the models. Moreover, the IOM should also be able to provide tools for any new model that uses data structures in a completely new way.

2.2 Expressing Discrete Equations in Forge

Each 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 `dim` to represent the number of space dimensions of the underlying ocean model. The corresponding representation in Forge is as follows.

```
timeConv(integer::dim)
```

The body of a specification consists of an interface, which is discussed in Section 2.3, 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 `timeConv` we use a definition that is similar to the following.

```
genLoops{dim | h[L] = 0.0}
```

Here `genLoops` is a *library function* that generates `dim` nested loops containing a Fortran 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? Because the Fortran statement depends on the value of `dim`. For example, if `dim` is `0`, the Fortran statement derived from `h[L] = 0.0` will be simply the following.

```
h(L) = 0.0
```

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

```
h(L,dim1i,dim2i) = 0.0
```

where `dim1i` and `dim2i` are integer variables that are generated to be used as loop variables. In this case, the application of `genLoops` creates the following Fortran code.

```
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 `for` loops are parameters that are created for the subroutine `timeConv` from the dependent-index declaration `index space = 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 `genLoops` with the model parameter `dim` as follows.

```
genLoops{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 `dim` = 2, the Fortran program shown in Figure 4 will be generated.

```
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
end do
```

Fig. 4. Fortran translation of discrete equations.

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 `n` ranges over all values in the 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.

The parameter declaration of a tool definition provides the information that is required to distinguish equations that are translated into simple assignments from those that are translated into loops. For example, since `L` and `U` are declared as parameters, their concrete values will be available when the subroutine is called, unlike `n` which has to be bound explicitly through a loop. Moreover, since `L` and `U` are used in the definition of the arrays, we know that they are constant with respect to the size of the arrays. However, since `n` is not used to define the arrays, `n` is a variable with respect to the size of every array.

2.3 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 declarations of local variables. In our example, `dt`, `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 [Xi and Pfenning 1999] 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.

```
index space = X:Y[1: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, `X` and `Y` must be yet unused variables and the two range values must be integer expressions. Note that the difference between the first and second range value might be negative, for example, if `dim` is `0`, in which case no variable declarations will be generated.

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 might 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`.

```
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 variable `timePos`, in the list of variables represented by a dependent-index expression.

```
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 `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 dependent indices. For example, the library functions `cons` or `insertAt` allow a fixed index part to be combined with dependent indices in array declarations. In Section 2.2 we have seen the library function `genLoops` that generated nested loops and appropriate array indexing. Library functions like `genLoops` therefore combine several tasks in guiding the translation of discrete equations by model-dependent parameters.

The dependent part of an array index depends on model parameters of type `integer`, which are used to calculate the number of the variable dimensions. By using dependent types, we can use a generic form to represent array types used in different models, which have different dimensions. The dependent part of an array is never referenced in the discrete equations, but it is necessary for generating Fortran programs.

2.4 Summary: The Complete Tool Specification for the Markovian Convolution

The Forge specification of `timeConv` is shown in Figure 5. The line numbers are shown to simplify the discussion of type checking that will be presented in Section 3.

The specification refers to one model parameter `dim`. The dependent-index declaration of `space` depends on `dim`, which causes the array types of `a`, `b`, and `h` to be of dimension `dim+1` since their index type is defined to depend on `space`, which is of dimension `dim`, and adds to `space` exactly one dimension, `L:U`. The formulas of the body correspond exactly to the discrete equations that we have seen in Section 2. 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`

```
timeConv(integer::dim)
  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;
  genLoops{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)
  }
```

Fig. 5. The Forge specification for Markovian convolution in time.

nested loops for the equations defining `h[L]` and `b[U]` (since the index is just a boundary variable) and `dim+1` nested loops for the other two equations. Since the arrays are indexed by a variable that is not an array boundary, an additional loop over the time dimension is created.

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. This design supports the idea of "gentle slope" [Myers et al. 2000] and makes the specification language easier to learn and to apply. The complete syntax of Forge is shown in Appendix A.

We can compile the above specification into a Fortran program by calling the Forge compiler `forge`. In addition to the specification, we have to provide the name of the model for which we want to generate code. For example, we can generate the Fortran subroutine for the tool Markovian Convolution and the model `PEZ` by the following command. The model name is used to locate the model specification, which will be introduced in Section 2.6.

```
forge timeConv PEZ
```

The generated Fortran program is written to a file `timeConvPEZ.f90` whose content is shown for illustration in Appendix C. Users of the IOM system will not invoke the compiler directly, because they interact with the program generator through the graphical user interface shown in Section 1.2.

2.5 Bell-Shaped Convolution in Space

We consider another IOM tool to demonstrate a further important feature of Forge, namely the possibility of parameterizing specifications by model-dependent procedures. Similar to the Markovian convolution, the *Bell-shaped convolution in space* is also a tool for computing deweighted errors. The convolution is formally defined by the following formula.

$$b(x) = \int_0^X \exp(-(x - x')^2/L^2)a(x')\mathrm{d}x'.$$

In the above continuous formula, a is again the weighted error, and b is the deweighted error. 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 and is input by the modelers. The smaller L is, the more quickly the values of far neighbors will be omitted. The Bell-shaped convolution can be discretized to the following equations.

$$\begin{aligned} \theta_i^0 &= a_i \\ \theta_i^{n+1} &= \theta_i^n + \Delta s \cdot RHS_i^n \qquad \text{for } 0 \le n \le N-1 \\ b_i &= \theta_i^N \end{aligned}$$

θ is an intermediate array for solving the final result b. More specifically, θ_i^n represents the ith element of the array θ in the nth iteration. The final value of the output array is calculated in the Nth iteration. Δs is the pseudo-time step used for computing the next θ from the current θ. N is the number of iterations for computing the final result. N is given by $(L^2/4)/\Delta s$. RHS_i^n is the abbreviation for

$$\frac{\theta_{i+1}^n - 2\theta_i^n + \theta_{i-1}^n}{(\Delta x)^2}$$

Since different ocean models use different representations for space, RHS_i^n is computed differently in different models. Therefore, Forge must be able to specify tools that are generic in the way to compute RHS_i^n. This dependency on a computation provided by each model is expressed by a `subroutine` parameter in the specification header.

It is not really important to understand why the 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 inversion tools.

The specification for the tool bell-shaped convolution is shown in Figure 6.

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 type correctness for generated Fortran programs that were created 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

```
1  spaceConv (integer::dim;  subroutine::computeRHS;)
2    index  space =  X:Y[1:dim];
3    param  real    :: ds;
4           integer :: N;
5           integer :: L, U;
6           real, dimension (cons{L:U,space}) :: a, b;
7    local  real, dimension (cons{L:U,space}) :: rhs, theta;
8           integer :: i, n;
9    genLoops{dim |
10      theta[i] = a[i]
11   };
12   do n = 1, N
13      call computeRHS(space,theta,rhs);
14      genLoops {dim |
15         theta[i] = theta[i] + ds * rhs[i]
16      }
17   end do
18   genLoops{dim |
19     b[i] = theta[i]
20   }
```

Fig. 6. The specification for the bell-shaped convolution in space.

errors will be contained in the subroutines provided by the user who is responsible for delivering correct subroutines.

2.6 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*.*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 [Pacanowski and Griffies 1999]. Two model parameters are defined in the model specification. The first, `dim`, is a global model parameter that has the value `3`. The second, `computeRHS`, is a model parameter that can be only used by the tool `spaceConv`, and refers to a Fortran subroutine provided by the modelers who want to generate Fortran code for `spaceConv`. The code of `computeRHS` is contained in the Fortran source file `"laplace.f90"`.

```
model PEZ;
dim=3;
spaceConv.computeRHS = "laplace.f90"
```

Currently, model specifications allow only the definition of parameters through a

collection of name/value pairs. Future work will extend the model specifications to allow for specifications of model-specific optimizations for parallel computations.

3. THE FORGE TYPE SYSTEM

The generation of syntax- and type-correct Fortran programs 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 must be avoided at all cost because it could quickly lead to users losing trust in the system and eventually not using 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 mistake 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 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. We will demonstrate how the type system can prevent type errors in the generated Fortran programs through several examples. The reader interested in further details can consult the formal typing rules of the type system shown in Appendix B.

3.1 Declaration Typing

In the following, we refer to the specification for the Markovian convolution in Figure 5. 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 Δ contains the type information for all model parameters, including external Fortran subroutines, and another typing environment Γ 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 Δ. For example, the following model parameter declaration (line 1) adds (`dim`, `integer`) into Δ.

```
integer :: dim
```

Local variable declarations and parameter declarations are used to declare Fortran variables. For example, the following declaration (line 3) adds (`dt`, `real`) and (`tau`, `real`) into the environment Γ.

```
real :: dt,tau
```

In the typing environment Γ 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 example, the following declaration (line 5) adds (`a`, (`cons`, (`L`, `U`), `space`, `real`)) and (`b`, (`cons`, (`L`, `U`), `space`, `real`)) into Γ.

```
real, dimension (cons{L:U,space}) :: a, b
```

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

```
space = X:Y[1:dim];
```

The type checker adds (`space`, `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 `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 12. Array types can be constructed by calling library functions, where the validity of the library function calls is checked by the rule $\text{ATY}_{\vdash}$. Consider again the declaration in line 5. Since `L` and `U` have type `integer`, `L:U` is a valid fixed index. Because the name `space` has the type `depix`, it is a valid dependent index. In the typing environment Γ, we store the type of the library function `cons`, which is `range->depix->range`, where `range` represents fixed index types. The type environment Γ 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 `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.

3.2 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 14 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 $n \geq k$ (see rule $\text{ARRAY}_{\vdash}$).

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 `real`, so the expression `-0.5*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 `b` and `h` are declared by the same statement in line 5 and 6 in Figure 5, they have the same type. Therefore, the expression `b[n+1] - dt*(h[n] + b[n+1]/tau)` in line 12 is type correct. In contrast, the expression `a + b[n]` would cause a type error since `a` and `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.

3.3 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 5, the left-hand side of the assignment has the type of array of real numbers; the right-hand side has the type of `real`. Since a `real` array is upward compatible with `real`, this assignment is valid. Similar reasoning shows that all the assignments in Figure 5 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 6, 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 16 in Appendix B 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 5, 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 16 in Appendix B in the rule $\text{LIB}_{\vdash}$.

4. RELATED WORK

The Earth System Modeling Framework (ESMF) defines an architecture that allows the composition of applications using a component-based approach [Dickenson et al. 2002]. The focus of the ESMF is to define standardized programming interfaces and to collect and provide data structures and utilities for developing model components.

Forge is a program generator, which is a special kind of metaprogramming tool [Sheard 2001]. The approach for implementing Forge can be considered as single-stage programming since Forge programs are compiled to Fortran programs directly, and program generation only happens at compile time. Program generators also exist as *multi-stage* programming languages, such as MetaML [Taha and Sheard 2000] or MetaOCaml [Calcagno et al. 2003], where code can be generated at run-

time. Program generators implemented in multi-stage programming languages are all embedded languages [Elliott et al. 2000], which means the new language is both syntactically and semantically a subset of an existing language. Forge is implemented in Haskell, but it is not a language embedded in Haskell since the syntax and semantics of Forge are not a subset Haskell.

Scientific computing is a popular domain for the application of program generation/program synthesis. Several program synthesis systems are being used in different scientific areas. SciNapse [Akers et al. 1997] is a system for solving partial differential equations (PDEs). It generates Fortran or C programs from a PDE specification. SciNapse is built on top of a general-purpose knowledge-based system that contains transformation rules. The generated programs are constructed by instantiating predefined selected algorithm templates. SciFinance [Akers et al. 2001] applies a similar approach to financial modeling—it generates C programs from financial model specifications written in ASPEN (Algorithm SPEcification Notation). CTADEL [van Engelen et al. 1997] is a software environment for generating multi-platform high-performance Fortran code for PDE-based problems from abstract problem specifications. A CTADEL specification is first transformed to intermediate code. Optimizations will be applied to this intermediate code, and the refined intermediate code will be translated to Fortran. Planware [Blaine et al. 1998] generates Lisp programs for high-performance scheduler software. Planware first refines an abstract problem specification to a particular scheduling problem, then generates Lisp code by applying a predefined code-generation tactic. The Amphion system [Lowry and Baalen 1995] guides users in developing a formal specification of a problem and then implements this specification as a Fortran program consisting of calls to subroutines from a library. The program generation is based on intuitionistic propositional logic. AutoBayes [Fischer et al. 2000] is a program-synthesis system implemented in SWI-Prolog for statistical data analysis. It generates optimized C/C++ code from the description of a data analysis problem in the form of a statistical model. The generated programs are documented; assumptions and proof obligations that have not been discharged during the program generation process are presented in the documentations or are converted into rum-time assertions. A system for generating numerical programs for simulation of rigid mechanical systems in physics-based animation is introduced in [Ellman et al. 2003]. It generates a numerical C++ simulation program that drives a real-time animation of a specified scenario. The program generation is implemented by instantiating a generic parameterized program scheme.

All these systems have a similar goal to Forge, which is to generate programs automatically for a specific scientific application from abstract specifications. However, there are three main differences between these applications and our system. First, the described systems work on more abstract specifications, for example, partial differential equations or scheduling problem descriptions, while Forge works with discrete equations. Generating programs from a more abstract specification may save users work, for example, deriving discrete equations from a continuous equation. However, this approach also limits the applicability of the systems since they often have more constraints on the problems they are solving. Since discrete equations can be derived from almost all scientific formulas, Forge can be used

more broadly. Second, most of the above systems focus on generating efficient code for a specific problem instead of reusing existing user code. In contrast, we are aiming at generating model-dependent programs, which requires the existing user code to be reused as much as possible. Finally, in our system, guaranteeing the type correctness of the generated program is a major goal. However, the described systems do not guarantee the type correctness of the generated programs.

In metaprogramming systems like MetaML [Taha and Sheard 2000] or Template Haskell [Sheard and Peyton Jones 2002] the metaprogramming language is an extension of the object language. In our system, metaprogramming happens in three different languages: (1) the specifications are defined in Forge, (2) the library functions are written in Haskell, and (3) the generated programs are Fortran. Existing Fortran metaprogramming tools include Foresys [Simulog, SA 1996], whose focus is on the refactoring of existing Fortran code, for example, transforming a Fortran77 program into Fortran90 program. Sage++ [Bodin et al. 1994] is a tool for building Fortran/C metaprogramming tools. However, to express applications as the ones shown here a user has to write metaprograms in Sage++ for transforming Fortran, which is quite difficult and error prone and probably beyond the capabilities of scientists who otherwise just use Fortran. In contrast, Forge allows users to work mostly in Fortran and express generic parts by parameters; most of the metaprogramming issues are hidden inside the compiler and parameter definitions. Macrofort [Gomez and Capolsini 1996] can generate Fortran code from Maple programs, but does not provide the mechanism to deal with generic, model-dependent code.

Recently, we have investigated an alternative approach to Fortran program generation through a language called *Parametric Fortran* [Erwig and Fu 2004] that allows the definition of Fortran extensions by parameter structures that can be referred to in Fortran programs to specify the dependency of program parts on these parameters. By providing parameter values, a parameterized Fortran program can be translated into a regular Fortran program. These parameters are very different from the model parameters of Forge. The behavior of these parameters has to be specified explicitly through Haskell definitions for each parameter type, whereas this behavior is essentially predefined in Forge and can be adjusted through library functions. With Parametric Fortran we can create complete Fortran programs, whereas Forge is limited to the specification and generation of subroutines that will be part of larger simulation programs. Since Parametric Fortran is a more general approach than Forge, it lacks most of the safety guards provided by the Forge type system.

To summarize, our approach can be considered as the combination of generative programming [Czarnecki and Eisenecker 2000] and template metaprogramming [Veldhuizen 1995; Sheard and Peyton Jones 2002]. Forge is a generative programming language because the Forge programs are used to generate Fortran programs. Forge also supports template definitions, for example, the array types parameterized by model parameters.

5. CONCLUSIONS

Using generic descriptions for inverse ocean modeling tools enables the developers of the IOM system to specify tools once and let a program generator create Fortran implementations automatically for different ocean models. The mix of a declarative language for a mathematical specification of tools by discrete equations and the availability of library functions to incorporate model-specific aspects into the translation process makes the language Forge a high-level language that also provides a well-defined, flexible path for extensions. A sophisticated type system facilitates the consistency checking of specifications independently of particular models, that is, tools can be checked before they are released and used by scientists. Preventing the generation of erroneous Fortran programs increases the reliability and helps to promote the acceptance of the approach.

The design of Forge supports the collaboration of ocean scientists, software developers, and computer scientists and provides at the same time a large degree of independence that facilitates the further development of tools and the specification language.

The IOM tools generated by Forge are currently intensively used with the ocean model PEZ at Oregon State University and the National Center for Atmospheric Research, and with the model KDV at Arizona State University. Other ocean modeling groups are currently in the process of adapting the IOM system, so that the graphical user interface and the Forge compiler will soon be also used for models, such as ROMS (developed at Rutgers and University of Colorado), ADCIRC (developed at Arizona State University and University of North Carolina), and SEOM (also developed at Rutgers).

One main goal for future versions of the Forge is to improve the support for parallel computing. To cope with huge data sets, modelers typically prefer to run the IOM tools in parallel. To support this goal, Forge needs to be able to generate parallel code for different parallel infrastructures, for example, data decomposition for distributed memory supercomputers, the class of shared-memory vector supercomputers, or NUMA scalar supercomputers, depending on what infrastructure the modelers use. To express parallelism within tools requires the extension of Forge to support parameterizing tool specifications with different parallel infrastructures and generating parallel code.

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*), see Figure 7.

The binary operators `*` and `/` in Figure 7 can be used on two values or a value and an array, but not on two arrays. The syntax of interfaces is summarized in Figure 8, which builds on the syntax for dependent-index declarations, which are shown in Figure 9.

Dependent index expressions are a special form of dependent types [Xi and Pfenning 1999].

The syntax of types is shown in Figure 10. The type `fortran` is used in the types of library functions that are used as program transformations. Function types are

```
Spec     ::= TName(MPList) Interface Stmt
MPList   ::= Type::PName;...;Type::PName
Stmt     ::= FName{PNames|Stmt} | Assg | SubCall | DoStmt | Stmt;Stmt
PNames   ::= PName,...,PName
SubCall  ::= call SubName(FParam,...,FParam)
DoStmt   ::= 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      ::= + | - | * | /
```

Fig. 7. Syntax of specifications.

```
Interface ::= DepIx;Param;Local;
DepIx     ::= index IxDecl;...;IxDecl
Param     ::= param VarDecl;...;VarDecl
Local     ::= local VarDecl;...;VarDecl
VarDecl   ::= Type::VName,...,VName
```

Fig. 8. Syntax of interfaces.

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

Fig. 9. Syntax of dependent-index declarations.

used to represent the types of library functions and external Fortran subroutines.

```
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
```

Fig. 10. Forge types.

Finally, the syntax of model specifications is defined in Figure 11.

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

Fig. 11. Syntax of parameter definitions.

Appendix B: 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*, respectively. We use t to range over the Forge types. For convenient use in the typing rules, we represent array types by quadruples $(f, \mathit{FixInd}, \mathit{DepInd}, \mathit{Bty})$. The judgment $\mathit{Aty} \Downarrow t$ used in the rule $\textsc{avs}_{\vdash}$ expresses that t is the quadruple corresponding to the array type *Aty*. The judgment $\mathit{Aty} \Downarrow t$ is defined in Figure 13 by inference rules.

This style of defining typing judgments by rules is explained in detail, for example, in Benjamin Pierce's book [Pierce 2002]. 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. All the rules in Figure 13 are facts, but the rules in Figure 12 contain premises. Premises and conclusions are expressed in form of judgments, which are essentially relations between the participating objects. For example, the judgment $\mathit{Aty} \Downarrow t$ expresses the relationship between an array type *Aty* and its quadruple representation.

The rules in Figure 12 define how to construct two typing environments Δ and Γ 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 Δ contains the type information for all model parameters, including external Fortran subroutines to be called, whereas Γ contains the type information for constants, variables and library functions. Γ_0 is the initial typing environment, which contains the type information of constants and library functions. The judgment $\Delta; \Gamma \vdash d \succ \Delta'; \Gamma'$ expresses that the declaration d changes the current typing environments from Δ and Γ to Δ' and Γ'. 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 Γ and Δ, 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 $\textsc{mp}_{\vdash}$ and $\textsc{mps}_{\vdash}$ show how Δ 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 Δ. $\textsc{dixe}_{\vdash}$ 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 Δ and Γ_0, which do not contain type information for Fortran variables, to infer the type of the two expressions in the rule $\textsc{dixe}_{\vdash}$. The presence of Fortran subroutine names in Δ causes no problem, since their result type is `subroutine`, which means that they can never successfully contribute to a well typed integer expression. The rule $\textsc{dix}_{\vdash}$ shows that if a dependent-index expression is valid and is bound to a name, then the name has the type `depix` in the typing environment Γ. We do not have to distinguish different dependent-index types in the type system, because the type `depix` is only used to check if a library func-

$$\text{MP}_\vdash\ \frac{p \notin dom(\Delta)}{\Delta;\Gamma \vdash t\texttt{::}p \succ \Delta \cup \{(p,t)\};\Gamma} \qquad \text{MPS}_\vdash\ \frac{\Delta_i;\Gamma \vdash t_i\texttt{::}p_i \succ \Delta_{i+1};\Gamma \quad 1 \le i \le n}{\Delta_1;\Gamma \vdash t_1\texttt{::}p_1\texttt{;}\ldots\texttt{;}t_n\texttt{::}p_n \succ \Delta_{n+1};\Gamma}$$

$$\text{DIXE}_\vdash\ \frac{\Delta;\Gamma_0 \vdash e_1 : \texttt{integer} \quad \Delta;\Gamma_0 \vdash e_2 : \texttt{integer} \quad l_i, h_i \notin \Gamma \quad i \ge 1}{\Delta;\Gamma \vdash l\texttt{:}h\texttt{[}e_1\texttt{:}e_2\texttt{]} : \texttt{depix}}$$

$$\text{DIX}_\vdash\ \frac{IName \notin dom(\Gamma) \quad \Delta;\Gamma \vdash DepExpr : \texttt{depix}}{\Delta;\Gamma \vdash IName\texttt{=}DepExpr \succ \Delta;\Gamma \cup \{(IName, \texttt{depix})\}}$$

$$\text{DIXS}_\vdash\ \frac{\Delta;\Gamma_i \vdash IxDecl_i \succ \Delta;\Gamma_{i+1} \quad 1 \le i \le n}{\Delta;\Gamma_1 \vdash \texttt{index}\ IxDecl_1\texttt{;}IxDecl_2\texttt{;}\ldots\texttt{;}IxDecl_n \succ \Delta;\Gamma_{n+1}}$$

$$\text{BVS}_\vdash\ \frac{v_i \notin dom(\Gamma) \quad 1 \le i \le n \quad i \ne j \Rightarrow v_i \ne v_j}{\Delta;\Gamma \vdash Bty\texttt{::}v_1\texttt{,}v_2\texttt{,}\ldots\texttt{,}v_n \succ \Delta;\Gamma \cup \{(v_1,Bty),(v_2,Bty),\ldots,(v_n,Bty)\}}$$

$$\text{RANGE}_\vdash\ \frac{\emptyset;\Gamma \vdash e_1 : \texttt{integer} \quad \emptyset;\Gamma \vdash e_2 : \texttt{integer}}{\Delta;\Gamma \vdash e_1\texttt{:}e_2 : \texttt{range}}$$

$$\text{FIX}_\vdash\ \frac{\emptyset;\Gamma \vdash r : \texttt{range} \quad \emptyset;\Gamma \vdash FixInd : \texttt{range}}{\Delta;\Gamma \vdash r\texttt{,}FixInd : \texttt{range}}$$

$$\text{ATY}_\vdash\ \frac{\begin{array}{c}\Delta;\Gamma \vdash FixInd : \texttt{range} \quad \Delta;\Gamma \vdash DepInd : \texttt{depix} \quad \Delta;\Gamma_0 \vdash p_i : t_i \\ \emptyset;\Gamma_0 \vdash f : t_1\texttt{->}\ldots\texttt{->}t_k\texttt{->range->depix->range}\end{array}}{\Delta;\Gamma \vdash Bty\texttt{, dimension(}f\texttt{\{}p_1\texttt{,}\ldots\texttt{,}p_k\texttt{|}FixInd\texttt{,}DepInd\texttt{\})}}$$

$$\text{FATY}_\vdash\ \frac{\Delta;\Gamma \vdash FixInd : \texttt{range}}{\Delta;\Gamma \vdash Bty\texttt{, dimension(}FixInd\texttt{)}} \qquad \text{DATY}_\vdash\ \frac{\Delta;\Gamma \vdash DepInd : \texttt{depix}}{\Delta;\Gamma \vdash Bty\texttt{, dimension(}DepInd\texttt{)}}$$

$$\text{AVS}_\vdash\ \frac{v_i \notin dom(\Gamma) \quad 1 \le i \le n \quad \Delta;\Gamma \vdash Aty \quad Aty \Downarrow t \quad i \ne j \Rightarrow v_i \ne v_j}{\Delta;\Gamma \vdash Aty\texttt{::}v_1\texttt{,}v_2\texttt{,}\ldots\texttt{,}v_n \succ \Delta;\Gamma \cup \{(v_1,t),(v_2,t),\ldots,(v_n,t)\}}$$

$$\text{PS}_\vdash\ \frac{\Delta;\Gamma_i \vdash vd_i \succ \Delta;\Gamma_{i+1} \quad 1 \le i \le n}{\Delta;\Gamma_1 \vdash \texttt{param}\ vd_1\texttt{;}vd_2\texttt{;}\ldots\texttt{;}vd_n \succ \Delta;\Gamma_{n+1}}$$

$$\text{LS}_\vdash\ \frac{\Delta;\Gamma_i \vdash vd_i \succ \Delta;\Gamma_{i+1} \quad 1 \le i \le n}{\Delta;\Gamma_1 \vdash \texttt{local}\ vd_1\texttt{;}vd_2\texttt{;}\ldots\texttt{;}vd_n \succ \Delta;\Gamma_{n+1}}$$

$$\text{ENV}_\vdash\ \frac{\begin{array}{c}\emptyset;\Gamma_0 \vdash MPList \succ \Delta;\Gamma_0 \quad \Delta;\Gamma_0 \vdash DepIx \succ \Delta;\Gamma \\ \Delta;\Gamma \vdash Param \succ \Delta;\Gamma' \quad \Delta;\Gamma' \vdash Local \succ \Delta;\Gamma''\end{array}}{\emptyset;\emptyset \vdash TName\texttt{(}MPList\texttt{)}\,DepIx\texttt{;}Param\texttt{;}Local\texttt{;} \succ \Delta;\Gamma''}$$

Fig. 12. Rules for environment judgments.

tion call for constructing a dependent array type is type correct by the rule ATY$_\vdash$ and DATY$_\vdash$. 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 `depix`.

The rules BVS$_\vdash$ and AVS$_\vdash$ check variable declarations. If a variable has a base type, the rule BVS$_\vdash$ adds the pair (v, t) into Γ. If a variable has an array type, the rule AVS$_\vdash$ first checks if the array type is valid, in which case the name and the quadruple representing the array type is added into Γ. The rule RANGE$_\vdash$ 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 Δ in the premises of the rules $\textsc{range}_\vdash$ and $\textsc{fix}_\vdash$. The rules $\textsc{faty}_\vdash$ and $\textsc{daty}_\vdash$ check if an array type is valid when there is no library function involved. $\textsc{faty}_\vdash$ says that if an array type only contains fixed indices, the fixed indices must have the type `range`. $\textsc{daty}_\vdash$ 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 $\textsc{aty}_\vdash$ needs to check if the library function call is type correct. Every library function is a Haskell function whose type signature is contained in Γ_0. A library function used for merging array indices must have the type t_1`->`...`->`t_k`->range->depix->range`. The type of every parameter of the library function has to match the type signature.

The rules $\textsc{ps}_\vdash$ and $\textsc{ls}_\vdash$ add the typing information of all the parameter variables and local variables into the typing environment Γ. The rule $\textsc{env}_\vdash$ shows how the type environments are constructed from the interface part of a specification. We start from the initial typing environments $\emptyset$ and Γ_0. The result typing environment Δ 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 Γ''.

$$\textsc{mix}_\Downarrow \frac{}{\mathit{Bty}\texttt{,dimension(}f\{\mathit{PNames}\,|\,\mathit{FixInd}\texttt{,}\mathit{DepInd}\}\texttt{)} \Downarrow (f, \mathit{FixInd}, \mathit{DepInd}, \mathit{Bty})}$$

$$\textsc{fix}_\Downarrow \frac{}{\mathit{Bty}\texttt{, dimension(}\mathit{FixInd}\texttt{)} \Downarrow (\emptyset, \mathit{FixInd}, \emptyset, \mathit{Bty})}$$

$$\textsc{dep}_\Downarrow \frac{}{\mathit{Bty}\texttt{, dimension(}\mathit{DepInd}\texttt{)} \Downarrow (\emptyset, \emptyset, \mathit{DepInd}, \mathit{Bty})}$$

Fig. 13. Array representations.

The rules in Figure 14 define the typing rules of Forge expressions. The judgment $\Delta; \Gamma \vdash e : t$ means the expression e has the type t under the typing assumptions Δ and Γ.

$$\textsc{con}_\vdash \frac{c \text{ is a constant of } \mathit{Bty}}{\Delta; \Gamma \vdash c : \mathit{Bty}} \qquad \textsc{var}_\vdash \frac{(v, t) \in \Gamma}{\Delta; \Gamma \vdash v : t} \qquad \textsc{par}_\vdash \frac{(v, t) \in \Delta}{\Delta; \Gamma \vdash v : t}$$

$$\textsc{array}_\vdash \frac{\Delta; \Gamma \vdash e : (f, (r_1, r_2, \ldots, r_n), \mathit{DepInd}, \mathit{Bty}) \quad \Delta; \Gamma \vdash e_i : \texttt{integer} \quad 1 \le i \le k \le n}{\Delta; \Gamma \vdash e\texttt{[}e_1\texttt{,}e_2, \ldots, e_k\texttt{]} : (f, (r_{k+1}, r_{k+2}, \ldots, r_n), \mathit{DepInd}, \mathit{Bty})}$$

$$\textsc{uop}_\vdash \frac{\Delta; \Gamma \vdash e : t}{\Delta; \Gamma \vdash \mathit{UOp}\ e : t} \qquad \textsc{bop}_\vdash \frac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta; \Gamma \vdash e_2 : t_2 \quad t_1 \sim t_2}{\Delta; \Gamma \vdash e_1\ \mathit{BOp}\ e_2 : \mathit{max}(t_1, t_2)}$$

Fig. 14. Typing rules for expressions.

The rule $\textsc{array}_\vdash$ 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 UOP$_\vdash$ is trivial since we currently have negation as the only unary operator in our abstract syntax; see Figure 7. 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 15. The rule BA$_\prec$ essentially allows the initialization of an array with a constant or to add a constant to every element of an array, cf. rule ASSGN$_\vdash$ in Figure 16. This rule overloads constants of base types with array types.

$$\text{BASE}_\prec \; \frac{}{\texttt{integer} \prec \texttt{real}} \qquad \text{REFL}_\prec \; \frac{}{t \prec t} \qquad \text{BA}_\prec \; \frac{Bty_1 \prec Bty_2}{Bty_1 \prec (f, FixInd, DepInd, Bty_2)}$$

$$\text{ARRAY}_\prec \; \frac{FixInd_1 = FixInd_2 \quad DepInd_1 = DepInd_2 \quad Bty_1 \prec Bty_2}{(f, FixInd_1, DepInd_1, Bty_1) \prec (f, FixInd_2, DepInd_2, Bty_2)}$$

Fig. 15. Partial order on types.

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

The typing rules in Figure 16 define the typing of Forge statements. The judgment $\Delta; \Gamma \vdash s$ means that the statement s is valid under the typing environments Δ and Γ. The rule ASSGN$_\vdash$ 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 Γ. The rule SEQU$_\vdash$ expresses that a sequence of two statements is valid only if both statements in the sequence are valid. In the rule SUB$_\vdash$ we can only check whether the called Fortran subroutine *SubName* has been declared as a model parameter. We cannot check the type consistency of the arguments because we have not access to the code of the called subroutine at this time. The rule LIB$_\vdash$ checks a library function call in a similar way as the rule ATY$_\vdash$ in Figure 12. The typing rule SPEC$_\vdash$ 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.

$$\text{ASSGN}_{\vdash}\ \frac{\emptyset;\Gamma \vdash v : t_1 \quad \emptyset;\Gamma \vdash e : t_2 \quad t_2 \prec t_1}{\Delta;\Gamma \vdash v\texttt{=}e}$$

$$\text{SEQU}_{\vdash}\ \frac{\Delta;\Gamma \vdash stmt_1 \quad \Delta;\Gamma \vdash stmt_2}{\Delta;\Gamma \vdash stmt_1\texttt{;}stmt_2}$$

$$\text{DO}_{\vdash}\ \frac{\emptyset;\Gamma \vdash v : \texttt{integer} \quad \emptyset;\Gamma \vdash e_1 : \texttt{integer} \quad \emptyset;\Gamma \vdash e_2 : \texttt{integer} \quad \Delta;\Gamma \vdash s}{\Delta;\Gamma \vdash \texttt{do}\ v\texttt{=}e_1\texttt{,}e_2\ s}$$

$$\text{SUB}_{\vdash}\ \frac{\emptyset;\Gamma \vdash e_i : t_i \quad 1 \leq i \leq k \quad \Delta;\emptyset \vdash SubName : \texttt{subroutine}}{\Delta;\Gamma \vdash \texttt{call}\ SubName\texttt{(}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{,}e_k\texttt{)}}$$

$$\text{LIB}_{\vdash}\ \frac{\begin{array}{c}\emptyset;\Gamma_0 \vdash f : t_1\texttt{->}\ldots\texttt{->}t_k\texttt{->fortran->fortran} \\ \Delta;\emptyset \vdash p_i : t_i \quad 1 \leq i \leq k \quad \Delta;\Gamma \vdash s\end{array}}{\Delta;\Gamma \vdash f\{p_1\texttt{,}\ldots\texttt{,}p_k\texttt{|}s\}}$$

$$\text{TOOL}_{\vdash}\ \frac{\emptyset;\emptyset \vdash TName\texttt{(}MPList\texttt{)}Interface \succ \Delta;\Gamma \quad \Delta;\Gamma \vdash s}{\emptyset;\emptyset \vdash TName\texttt{(}MPList\texttt{)}Interface\ s}$$

Fig. 16. Type rules for Forge statements.

Appendix C: The Result of Compiling timeConv 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
  real :: tau
  integer :: L
  integer :: U
  real , dimension (L:U, X1:Y1, X2:Y2, X3:Y3) :: a
  real , dimension (L:U, X1:Y1, X2:Y2, X3:Y3) :: 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
```

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for the comments and many detailed suggestions that helped to improve this paper.

REFERENCES

Akers, R. L., Bica, I., Kant, E., Randall, C., and Young, R. L. 2001. SciFinance: A Program Synthesis Tool for Financial Modeling. *AI Magazine 22,* 2, 27–42.

Akers, R. L., Kant, E., Randall, C. J., Steinberg, S., and Young, R. L. 1997. SciNapse: A Problem-Solving Environment for Partial Differential Equations. *IEEE Computational Science & Engineering 4,* 3, 32–42.

Blaine, L., Gilham, L., Liu, J., Smith, D. R., and Westfold, S. J. 1998. Planware - Domain-

Specific Synthesis of High-Performance Schedulers. In *Automated Software Engineering*. 270–280.

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

Calcagno, C., Taha, W., Huang, L., and Leroy, X. 2003. Implementing Multi-Stage Languages using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering (GPCE)*. LNCS 2830. 57–76.

Castleman, K. R. 1996. *Digital Image Processing*. Prentice-Hall International, Englewood Cliffs, NJ.

Chua, B. and Bennett, A. F. 2001. An Inverse Ocean Modeling System. *Ocean Modelling 3*, 137–165.

Czarnecki, K. and Eisenecker, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA.

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

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

Elliott, C., Finne, S., and de Moor, O. 2000. Compiling Embedded Languages. In *Semantics, Applications, and Implementation of Program Generation (SAIG)*. 9–27.

Ellman, T., Deak, R., and Fotinatos, J. 2003. Automated Synthesis of Numerical Programs for Simulation of Rigid Mechanical Systems in Physics-Based Animation. *Automated Software Engineering 10*, 367–398.

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

ESMF. `http://www.esmf.ucar.edu/`.

Fischer, B., Schumann, J., and Pressburger, T. 2000. Generating Data Analysis Programs from Statistical Models. In *Semantics, Applications, and Implementation of Program Generation (SAIG)*. 212–229.

Gomez, C. and Capolsini, P. 1996. Macroc and Macrofort, C and Fortran Code Generation Within Maple. *Maple Technical Newsletter 3,* 1.

IOM. `http://iom.asu.edu/`.

Iverson, K. E. 1984. *Introduction to APL*. APL Press.

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

Lowry, M. and Baalen, J. V. 1995. Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems. In *10th Conference on Knowledge-Based Software Engineering*. 2–10.

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

Mili, H., Mili, A., Yacoub, S., and Addy, E. 2002. *Reuse-Based Software Engineering*. John Wiley & Sons, New York, NY.

Myers, B., Hudson, S., and Pausch, R. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction 7,* 1, 3–28.

Pacanowski, R. C. and Griffies, S. M. 1999. *MOM 3.0 Manual*. NOAA/GFDL.

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

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

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

Sheard, T. and Peyton Jones, S. L. 2002. Template Metaprogramming for Haskell. In *Haskell Workshop*.

Simulog, SA 1996. *FORESYS, FORtran Engineering SYStem, Reference Manual v1.5*. Simulog, SA, Guyancourt, France. http://www.cs.vassar.edu/~ellman/ause-689-03-revised.pdf.

Taha, W. and Sheard, T. 2000. MetaML and Multi-Stage Programming with Explicit Annotations. *Theor. Comput. Sci. 248,* 1–2, 211–242.

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

Veldhuizen, T. L. 1995. Template metaprograms. *C++ Report 7,* 4, 36–43.

Xi, H. and Pfenning, F. 1999. Dependent Types in Practical Programming. In *26th ACM Symp. on Principles of Programming Languages*. 214–227.