Abstract. We describe the design of a domain-specific language (DSL) for the specification of generic ocean modeling tools, and we describe the implementation of its compiler. The goal of the DSL is to allow the specification of widely usable tools for ocean modeling once, and to allow its translation into different Fortran programs for individual ocean models depending on particular model parameters. A particular challenge is to balance the need for high flexibility in the tool specifications against the declarative nature and high-level expression of the specification language. This has been achieved through the design of a domain-specific embedded language (DSEL) where the tool description DSL is embedded into an environment that allows the use of Haskell functions in tool specifications. This setting facilitates the smooth evolution of tool specifications as well as extensibility of the available functions.

Keywords: Domain-Specific (Embedded) Language, Ocean Science, Inverse Method, Fortran, Program Generator, Haskell

1 Introduction

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. Every model uses an array over time and space to store the state values of oceans. The dimension of time is always one, but there may be different dimensions for space, depending on the algorithms and theory underlying the particular ocean model.

The Inverse Ocean Modeling (IOM) system [2] 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

* This work was supported by the National Science Foundation under the grant ITR/AP-0121542.
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. Currently, the IOM is written in Fortran, a programming language which is routinely employed for scientific computing.

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 deal with different ocean models, although the equations that all the ocean models are using are very similar.

Our solution is to implement a Domain Specific Language (DSL) for defining 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 can then generate the Fortran programs implementing the tools for any specific ocean model. The program generator is a compiler whose source language is the DSL and whose destination language is Fortran.

The system architecture is shown in Figure 1. The developers of the IOM system define the tool specifications to describe the tools that the IOM provides. The tool specifications can use library functions, which are basic Fortran program transformers, for example, for generating loops or importing Fortran code directly. A Fortran subroutine is generated for each tool specification. A model specification contains the information that are specific to a particular ocean model, such as the dimension of the array that is used to store the state values of the ocean. This information may be different for different models and will affect the generated programs. The DSL compiler is a program generator. It takes tool specifications, model specifications and library function definitions as input and outputs Fortran subroutines. These generated subroutines, which correspond to the tools, comprise the IOM system.

The DSL compiler is written in Haskell [15]. The Fortran programs are internally represented by a Haskell data type. One nice aspect of this design is that the type system of Haskell guarantees that the generated Fortran programs are syntactically correct. Library functions are implemented by Haskell functions, which work on the same Haskell data type representation of abstract Fortran syntax, which again supports the syntactical correctness of the resulting Fortran programs. Furthermore, since library functions generally require many low-level operations such as string manipulation or arithmetic operations, it is generally not a good idea to design a new language from scratch to express their definitions. A more economical approach is to reuse the implementation of an already existing general-purpose language. This general-purpose language serves as a host language into which the DSL is embedded, which is the reason why such an approach to realizing a DSL is also called Domain Specific Embedded Language (DSEL). In our case, Haskell is the host language into which the DSL for describing equations and tools is embedded.
Fig. 1. System architecture

2 Related Work

A domain specific language (DSL) is a programming language that is tailored for a particular application domain. It is usually a small declarative language which can offer expressive power for the domain. Hundreds of DSLs are being used today. There are some DSLs that are very widely used: SQL [10] is a DSL for data querying, LaTeX [6] is a DSL for document type setting, and Yacc [9] is a DSL for defining and generating parsers. An overview over DSL research is given in [22].

A domain specific embedded language (DSEL) is a DSL that is built on top of a host language, usually a functional language. Since the DSEL can inherit the infrastructure of the host language, the developers of the DSEL can concentrate on semantic issues of the main abstractions to be offered by the DSL. A comparison between DSLs and DSELS can be found in [12]. Paul Hudak gives some ideas about how to build DSELS whose host language is Haskell [7].

A DSL can be also regarded more generally as a program transformation tool; program transformation includes program compilation, optimization, synthesis, generation, refactoring, migration, normalization and improvement [23]. In our case, the library functions that can be used as Fortran program transformations are actually an example of meta programming. Since library functions generally do not analyze the Fortran code they receive as arguments, they are more precisely characterized as program generators. Tim Sheard describes the current status of research and remaining problems in this field in [17]. DSLs and DSELS are often used for program generation, which means programs written in the DSLs are actually specifications for programs in other languages. A program generator for producing an executable specification of a formatter for a
programming language from its context-free grammar is described in [21]. Sam Kamin gives some examples of program-generating DSLs in [12]. Other examples of program-generating DSLs are: GAL is a DSL for generating video device drivers [19], Verischemelog [11] introduces a DSEL embedded in Scheme for generating hardware descriptions, and in [16] the design and implementation of a DSL for dynamically building HTML documents is described. Some meta environments for defining domain-specific languages, generation of program analysis and transformation tools have also been developed, such as the ASF+SDF Meta-Environment [20]. There are also some DSLs for generating Fortran programs. Foresys [18] is a product distributed by Simulog for the manipulation of Fortran code, such as the migration from Fortran77 to Fortran90 or optimization of current Fortran programs. Claude Gomez describes the Macrofort package which permits complete Fotran77 code generation in Maple [5]. In [3] an example of index array flattening through program transformation for Fortran program is given.

3 An Example of an IOM Tool: Markovian Convolution in Time

The IOM first computes weighted, smallest misfits to the equations of motion. The former are then deweighted by the IOM to find smallest misfits. These having been found, the IOM finally solves the equations of motion, yielding the ocean state that best fits the equations of motion. The scientifically crucial deweighting step is performed by convolution with covariances in time or in space. The Markovian Convolution in Time is a mathematical method to do just this. Consequently, Markovian convolution is also a tool of the IOM system. The convolution is formally defined by the following formula.

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

In the formula, \(a\) is the weighted misfit, \(b\) is the deweighted misfit. 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 [2]. We omit the derivation here since it is not important for the translation of the discrete equations into Fortran.

\[
\begin{align*}
    h_L & = 0 \\
    \frac{h_n - h_{n-1}}{\Delta t} + \tau^{-1} h_{n-1} & = -2\tau^{-1} a_n \\
    b_H & = -(\tau/2) h_H \\
    \frac{b_{n+1} - b_n}{\Delta t} + \tau^{-1} b_{n+1} & = h_n
\end{align*}
\]
These discrete equations have to be translated into Fortran programs. Usually, one tool results in one Fortran subroutine.

In the above equations, \( h \) is a temporary array; \( L \) is the lower boundary of the arrays, it is supposed to be 0 for the above example; \( H \) is the upper boundary of the arrays. Normally, the arrays are over time and space. The array dimension of time is always one. However, 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 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.

All these requirements ask for a design of a language that is (1) \emph{declarative} in its expression of equations describing the IOM tools on a mathematical level so that the ocean scientists who develop the IOM can comfortably specify all the tools they want to provide and (2) \emph{flexible} to allow for customizations that can take into account parameters for particular ocean models.

## 4 The Design of the DSL for Defining Tools

The goal of our DSL is to allow IOM developers to specify any IOM tool only once so that can be translated into different Fortran programs for different ocean models. The DSL formally consists of three sublanguages, one for defining tools, one for defining model-specific information, and one for defining library functions (cf. Figure 1). The DSL for defining tools is used by the IOM developers to describe the tools they are providing. Such a tool descriptions typically consists of discrete equations enriched by calls to library functions. The model developers define model-specific information to tell the IOM all the model-specific information that may be used by IOM tools, such as the dimension of the arrays in the Markovian convolution. Currently, the “language” to define model parameters allows simply the definition of a set of name/value pairs. However, in future we expect to extend these facilities, in particular, when we consider the incorporation of model-specific optimizations for parallel computations. To facilitate the flexible translation of tool descriptions into Fortran, tool specifications can make use of library functions. A particular task of library functions is to guide the translation of discrete equations by model-dependent parameters. In a sense, library functions connect these two parts for the purpose of translation. A library function is essentially a Fortran program transformer that refines the translation of discrete equations in a predetermined way possibly controlled by model parameters. Library functions are Haskell functions that are developed by computer scientists in cooperation with the IOM developers. Using Haskell for the definition of library functions supports their easy adaptation and extension.

The syntax of tool specifications is as shown in Figure 2. It builds on a lexical syntax where \textit{VName} represents the syntactic category of Fortran vari-
able names, \(TName\) represents tool names, \(PName\) represents model parameter names, and \(FName\) represents library function names.

\[
\text{Tool} ::= TName(PName:Type;\ldots;PName:Type)\text{ : Decls Def}
\]
\[
\text{Decls} ::= [Bounds;][Lvars;]
\]
\[
\text{Bounds} ::= \text{boundary Bound,\ldots,Bound}
\]
\[
\text{Bound} ::= \text{Name} | \text{Const}
\]
\[
\text{Lvars} ::= \text{local VName,\ldots,VName}
\]
\[
\text{Def} ::= FEqu;\ldots;FEqu
\]
\[
\text{FEqu} ::= FName(PName,\ldots,PName|FEqu;\ldots;FEqu) | \text{Equ}
\]
\[
\text{Equ} ::= LValue=Expr
\]
\[
\text{LValue} ::= \text{VName} | \text{Array}
\]
\[
\text{Array} ::= \text{VName}[IExpr,\ldots,IExpr]
\]
\[
\text{IExpr} ::= \text{VName} | \text{IConst} | IExpr+IExpr | \ldots
\]
\[
\text{Expr} ::= \text{VName} | \text{Array} | \text{Const} | \text{Expr+Expr} | \ldots
\]
\[
\text{Type} ::= \text{Int} | \text{Real} | \text{Fortran} | \ldots
\]

Fig. 2. Syntax of tool specifications.

Every tool has a name and possibly some parameters. These parameters are called \textit{model parameters} since they refer to model-specific information that is used to guide the translation of the tool specification. The model parameters are given in the form \(PName:Type\) through the tool’s parameter list. The definition for model parameters are taken from model specifications, which we will discuss later in this section.

Every tool specification contains formulas, corresponding to the discrete equations that define the tool. The tool developers can declare boundaries and local variables to be used in the formulas. Boundaries are constants or variables that are used as array boundaries. The program generator does not generate a loop for an equation which is simply an assignment for an array-boundary element. Local variables are all variables that do not need to be input from outside of the subroutine. The program generator will not add local variables into the parameter list of the generated subroutine. All other variables used in the formulas are implicitly assumed to be input variables that will be added into the parameter list of the generated subroutine.

Local and input variables are names for object variables that will actually appear in the generated Fortran programs whereas model parameters are variables of the DSL, whose values will be just used in the translation process.

A library function can be applied to one equation or a block of equations, that is, the library function will be applied to the Fortran code that is generated for the equation or equations. The \(|\) separates input parameters of the
library function from equations. These input parameters must be declared in the parameter list of the tool specification.

Here is an example of a tool specification for Markovian convolution in time. The name of this tool is `timeConv`.

```fortran
library function from equations. These input parameters must be declared in the parameter list of the tool specification.

Here is an example of a tool specification for Markovian convolution in time. The name of this tool is `timeConv`.

```fortran
timeConv(dim: Int):
  bound L, H;
  local h, n;
  genLoops{dim |
    h[L] = 0.0;
    h[n] = h[n-1] - dt*(h[n-1]/tau + 2.0*a[n]/tau);
    b[H] = -0.5*h[H]/tau;
    b[n] = b[n+1] - dt*(h[n] + b[n+1]/tau)
  }
```

We observe that `dim` is a model parameter that has to be defined in every model specification for which the tool `timeConv` is to be generated. `L` and `H` are array boundary variable names that will appear as parameters in the generated Fortran subroutine, which also means that a caller of the subroutine has to provide the values for these. In contrast, `h` and `n` are declared as local variables and will appear as locally declared variables in the generated Fortran subroutine. The formulas of the specification correspond exactly to the discrete equations we have seen in Section 3. The library function `genLoops` takes an integer as input parameter to generate a corresponding number of nested loops.

The syntax of the DSL for defining model-specific information is shown in Figure 3. A model specification is a model name followed by some model parameter definitions. Each definition has the form of `PName=PExpr` or `TName.PName=PExpr`. The former is for global model-parameter definitions, which means the model parameter can be used by any tool 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. Tool-specific model parameters cannot be used by other tools. `PExpr` is either a constant or a Fortran code fragment. Below we give a simple example of a model specification.

```fortran
model PEZ;
  timeConv.dim=3
```

The name of the model specification is `PEZ`, which is an actual ocean model used by the ocean scientists. The PEZ (Primitive Equation Z-coordinate) Model is
a variant of Bryan-Cox-Semtner class model [14]. Only one model parameter is defined in it, $\text{dim}$, which is specific to the tool $\text{timeConv}$.

Having defined a tool and a model specification, we can now generate a Fortran program file $\text{timeConv.f90}$ which contains the Fortran subroutine corresponding to the tool $\text{timeConv}$ for the model PEZ by calling the DSL compiler $\text{iom2f}$.

\[
\text{iom2f timeConv PEZ}
\]

The default name of the generated Fortran file is the same as the name of the tool being compiled plus the suffix ".f90". The content of the generated file $\text{timeConv.f90}$ is shown in Appendix 1.

After we have developed the tool specification for time convolution, the ocean scientists asked us about the following generalization. For efficiency reasons, developers of different models may want to use different positions for the index of the time dimension in the array describing the state of the ocean. So this additional parameterization has to be taken into account in the tool specification and the model specification(s). This generalization could be fairly easily accomplished. First, we introduced a new parameter $\text{timepos}$ into the tool specification of $\text{timeConv}$. This parameter had to be used by the library function $\text{genLoops}$ to generate the loop for the time index at the appropriate dimension. Since $\text{genLoops}$ was not designed to handle this case, we were forced to generalize the definition. However, this turned out to be very easy because library functions are written in Haskell, and we could use the power of a fully-featured programming language to achieve the required generalization. As a result we obtained a new library function $\text{genTimeLoops}$ which takes both $\text{dim}$ and $\text{timepos}$ as input parameters. We will discuss how to define library functions in Section 5. This experience with a smooth evolution of a simple tool demonstrates the advantage of using a DESL approach for the DSL.

One question regarding the DSL design is: Why do we require explicit loop specifications (through library function)? There are two reasons for this. First, the IOM developers (sometimes) require explicit control over the loop generation process to achieve a high degree of model dependency (recall the variable positioning of the time index). Second, they want the possibility of using array syntax in the generated Fortran programs, which means that the use of array variables in equations cannot unambiguously determine the required loops.

\section{The DSL Compiler}

For the implementation of the DSL front end we have used the Haskell scanner generator Alex [4] and the parser generator Happy [13].

The DSL compiler is basically implemented by a couple of functions that translate equations into Fortran loops. Special attention has to be paid to library functions that are implemented in Haskell. The translation functions need access to these library functions, which are referenced through the tool specification to be compiled. Moreover, the translating functions are state based since they
have to create new Fortran variables and therefore have to keep track of the variables already generated. Therefore, we have chosen an implementation of the translation functions that is based on state monads. More precisely, we use a combined state/exception monad \texttt{Gen} that is based on the one described in [1, Chapter 10]. In addition to the used Fortran variables, we also keep the tool and model specifications in this state, which simplifies most function interfaces. Finally, we also store the list of available library functions in the state. The state of the \texttt{Gen} monad is therefore defined as follows.

\begin{verbatim}
data DSLState = DSLState (Model, Tool, [LibFun], Var)
\end{verbatim}

The type \texttt{Var} is just a synonym for \texttt{Int}. This state parameter is used as a counter to keep track of generated variables. The data types for model and tool specifications are defined according to the definition in the grammar from Figure 2. Library functions are represented in the state by their name and the Haskell function that performs the actual transformation.

\begin{verbatim}
data LibFun = LibFun FName TransFun

type TransFun = [PName] -> Fortran -> Gen Fortran
\end{verbatim}

A transformation function, whose type is \texttt{TransFun}, is defined for every library function. Every transformation function takes a list of names of the model parameters that are the input parameters of that library function.

For example, the library function \texttt{genLoops} is represented as follows.

\begin{verbatim}
genLoopsF :: TransFun
genLoopsF ps f = ...

genLoops :: LibFun
genLoops = LibFun "genLoops" genLoopsF
\end{verbatim}

Library functions are stored in a list and are accessed by their names. The selected library function’s transformation function, the Fortran program transformer, is actually not directly applied to its argument, the block of equations, but rather to the Fortran code that results from the translation of these equations.

Equations are represented by the Haskell data type \texttt{FEqu}:

\begin{verbatim}
data FEqu = LibCall FName [PName] [FEqu] | Equ Equ
\end{verbatim}

The first line represents the case when a library function is applied to a list of model parameters and a block of equations. The second line represents the case of a single equation where an equation is just a pair consisting of a variable and an expression.

Tool specifications are translated into Fortran code, which can be regarded in this context simply as a list of elementary Fortran statements.
We also have to define a function that converts the abstract syntax of Fortran statements into concrete syntax, to eventually generate Fortran program files. This is done in Haskell by defining the data type `FStmt` as an instance of the `Show` class. We give examples of three kinds of Fortran statements here, namely, loop statements, assignment statements, and variable declaration, respectively.

```
instance Show FStmt where
    show (FAssg lv e) = line [show lv," = ",show e]
    show (FFor v (b1,b2) fs step) =
        line ["do ",v," = ",tuple [b1,b2,step]] ++
        showAll fs ++
        line ["end do"]
    show (FDecl v t) = line [show t," :: ",v]
    ...
```

We have simplified the shown definitions a little bit; the definitions used in the DSL compiler have to take care of indentation and implicit generation of line breaks for lines that would otherwise become too long for the Fortran compiler. We have not employed any sophisticated pretty-printing library (see, for example, [8]) since the generated Fortran code is not really intended to be read by humans, but is rather embedded in other programs and fed into a Fortran compiler.

This approach of defining the abstract syntax of Fortran statements as a Haskell data type and then developing a `show` function for it gives us the flexibility to easily generate programs written in other high level programming languages, such as C or C++. If the abstract syntax of those languages are similar to Fortran, we mainly have to re-implement the `show` functions, so that most of the DSL compiler can be reused.

The main function `iom2f` reads tool and model specifications and runs the DSL compiler on an initial DSL state. The compiler function `compile`, like most
other functions that are concerned with the actual translation, is a monadic function for the monad \texttt{Gen} with result type \texttt{Fortran}. It retrieves the tool specification from the state and initiates the translation of the defining equations by calling a corresponding function \texttt{transFEqus}.

\begin{verbatim}
ion2f :: TName \rightarrow MName \rightarrow IO ()
ion2f tn mn = do let t = readTool tn
                   let m = readModel mn
                   let f = run compile (DSLState (m,t,libFuns,0))
                   writeFile (tn++".f90") (showAll f)

compile :: Gen Fortran
compile = do (Tool tn tp td fequs) <- getTool
              transFEqus fequs
\end{verbatim}

For generating a complete Fortran subroutine, the translator also has to generate declaration statements for all the variables that are used in the subroutine. Since we do not have explicit type declaration (except for model parameters), the program generator has to infer the type of these generated Fortran variables. This process is complicated by the fact that library functions can generate an arbitrary number of variables, which is generally not statically known because it may depend on model parameters that are provided at runtime.

Currently, we have implemented a simple, pragmatic approach to just determine all variables that are undeclared in the generated Fortran code. In other words, we are performing a kind of three-phase program generation: in the first phase, Fortran code is generated. In the second phase, variables that need a declaration are determined by a static analysis of the Fortran code generated in the first phase. Finally, in the third phase, the code is extended by declarations for the variables inferred in the second phase.

6 Conclusions and Future Work

Using a DSL for specifying ocean modeling tools enables the developers of the IOM system to specify tools once and let a compiler generate 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 DSL a high-level, application-specific language that at the same time provides a well-defined, flexible path for extensions of the DSL itself via library functions.

In future work we will develop a type system for the DSL that can yield guarantees about type correctness of the generated Fortran programs. We also consider the extension of the DSL to allow for optimization-oriented specifications. In particular, control over code generation for parallel computing is required.

The tool generator will be just one part of a bigger system framework to support ocean modeling. In this framework, users can select among many tools
and options to create a customized simulation program. The DSL compiler will be triggered on demand to compile tools that are needed for a particular simulation. Creating this framework and integrating the DSL compiler into it will also be part of our future work.

References


Appendix 1: The Result of Compiling *timeConv* for PEZ

```fortran
subroutine timeConv(dim1s, dim1e, dim2s, dim2e, dim3s, dim3e, L, H, dt, tau, a, b)
  integer :: dim3i
  integer :: dim3s
  integer :: dim3e
  integer :: dim2i
  integer :: dim2s
  integer :: dim2e
  integer :: dim1i
  integer :: dim1s
  integer :: dim1e
  integer :: L
  integer :: H
  real , dimension (dim1s:dim1e, dim2s:dim2e, dim3s:dim3e, L:H) :: h
  integer :: n
  real :: dt
  real :: tau
  real , dimension (dim1s:dim1e, dim2s:dim2e, dim3s:dim3e, L:H) :: a
  real , dimension (dim1s:dim1e, dim2s:dim2e, dim3s:dim3e, L:H) :: b
  do dim3i = dim3s, dim3e, 1
    do dim2i = dim2s, dim2e, 1
      do dim1i = dim1s, dim1e, 1
        h(dim1i,dim2i,dim3i,L) = 0.0
        do n = L+1, H-0, 1
          h(dim1i,dim2i,dim3i,n) = h(dim1i,dim2i,dim3i,n-1) &
            -dt*(h(dim1i,dim2i,dim3i,n-1)/tau+2.0*a(dim1i,dim2i,dim3i,n)/tau)
        end do
        a(dim1i,dim2i,dim3i,H) = -0.5*h(dim1i,dim2i,dim3i,H)/tau
        do n = L-1, H+0, -1
          a(dim1i,dim2i,dim3i,n) = &
            b(dim1i,dim2i,dim3i,n+1)-dt*(h(dim1i,dim2i,dim3i,n) &
            +b(dim1i,dim2i,dim3i,n+1))/tau
        end do
      end do
    end do
  end do
end subroutine timeConv
```