CTADEL: A Generator of Efficient Code for PDE-based Scientific Applications

Robert van Engelen [*] [†] & *Lex Wolters* [†]
High Performance Computing Division
Department of Computer Science, Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands
{robert,llexx}@cs.LeidenUniv.nl

Gerard Cats
Royal Netherlands Meteorological Institute
P.O. Box 201, 3730 AE De Bilt, The Netherlands
cats@knmi.nl

Abstract

In this report, we present the CTADEL system, a Code-generation Tool for Applications based on Differential Equations using a very high level Language specification. The CTADEL system generates efficient and vectorizable Fortran 77 code automatically from a very high level language description of a model described by partial differential equations (PDEs). The system combines algebraic simplification and powerful global common subexpression elimination to guarantee the generation of efficient code. A prototype implementation has been developed which is currently limited to explicit finite difference methods as solution method. After an informal, but detailed description of the CTADEL system, results of this prototype implementation will be presented for the time-dependent Euler equations to simulate an inviscid, compressible flow and for the calculation of the explicit dynamical tendencies within the HIRLAM model, which is a production code for limited area numerical weather forecasting. These results show that generation of efficient code is well feasible within the presented approach.

1 Introduction

Since the early days of computing, the demand of large scale scientific applications for more powerful hardware platforms has been a driving force for the development of advanced software environments including dedicated, machine dependent libraries for scientific computing. As more and more hardware platforms emerge, the adaption of scientific applications codes to each hardware platform type requires significant programming efforts. Some of these efforts can be alleviated by employing restructuring compilers to restructure existing application codes. However, restructuring compilers need relatively 'clean code' without many

[*]Support was provided by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO) under Project No. 612-17-120.
[†]Support was provided by the Esprit Agency EC-DGIII under Grant No. APPARC 6634 BRA III.

programming tricks that are often exploited by human programmers. Otherwise, automatic restructuring of code for another hardware architecture can be an unsolved problem for those applications developed with a specific target architecture in mind; in such an implementation, part of the knowledge content of a model is lost. This knowledge may be needed for an efficient implementation of the model on another hardware platform.

While code (re)writing by hand is common practice in the field of application development, it will be evident that the (semi-)automatic generation of code directly from a very high level language description of a problem provides a fast, robust, and, therefore, attractive alternative. In this respect, an automatic code generator should translate the model into 'optimal' code with respect to the target architecture. It is of great importance for the acceptance and actual usage of an automatic code generator that the generated code should be as efficient as the 'best' hand-written code.

In this report, the CTADEL system will be presented. The CTADEL system is a Code-generation Tool for Applications based on Differential Equations using a very high level Language specification. A prototype system has been implemented. This prototype system adopts currently only explicit finite difference methods as a solution method and generates vectorizable Fortran 77 code. In addition, from the Fortran 77 code, reasonably efficient data-parallel Fortran 90 code can be obtained by using the MasPar Vast-2 parallelizing compiler [18]. Results of the prototype implementation will be presented for the time-dependent Euler equations for inviscid, compressible flow and for the explicit dynamical tendencies of the HIRLAM model, which is a production code for limited area numerical weather forecasting.

The CTADEL system includes an extensible algebraic simplifier and a powerful global common subexpression eleminator. Algebraic simplification is one of the key techniques adopted for the automatic generation of efficient code. Axiomatic laws such as laws governing linear operators are applied in the translation of a model. In fact, the role of the simplifier is twofold: firstly, to reduce the computational complexity of the problem, and secondly, to generate a discretization of a model that is close to the discretization of the model that would have been obtained by hand. Together with the global common subexpression eliminator, these techniques provide the necessary means within the CTADEL system to generate efficient code automatically from a very high level language description of the model.

Why yet another code generator? The development of the CTADEL system received a major impetus by the request of the Royal Netherlands Meteorological Institute for efficient codes to be executed on vector and parallel computer architectures to solve the HIRLAM[1] limited area numerical weather forecast model [15].

Several parallel implementations of the forecast model have been realized: a data-parallel implementation [25], a message-passing version [11], a data-transposition code [16], and others. All modifications required for these implementations were made by hand starting from the (vectorized) HIRLAM reference code. As a result, several versions of the forecast system are now available, which all differ significantly from the reference code. Clearly, this is an undesirable situation from a maintenance point of view. It also hampers the easy inclusion of new insights into the model by meteorologists, since they are not acquainted with the parallelization techniques. This was one reason to investigate the possibilities of creating a system that automatically could generate code for the HIRLAM forecast model, or at least for some parts of it.

[1]The HIRLAM system was developed by the HIRLAM-project group, a cooperative project of Denmark, Finland, Iceland, Ireland, The Netherlands, Norway, and Sweden.

An important part in the HIRLAM model consists of the calculation of the dynamics of the atmosphere. It is described by a coupled system of three-dimensional non-linear hyperbolic partial differential equations. A time-consuming routine within the dynamics computes the explicit dynamical tendency of each variable based on explicit finite difference methods. This routine can be relatively easily parallelized even for different parallel programming paradigms [7]. However, in order to obtain more efficient parallel code for the HIRLAM dynamics, extensive recoding of the dynamical tendencies is required. In general, the development of codes for large models employing finite difference methods on staggered grids is error-prone due to (de)staggering and the derivation of the bounds for the computational domains. Therefore, (semi-)automatic generation of efficient code will significantly speed up this process.

However, it was soon recognized that existing software environments in this area failed to generate efficient code for the HIRLAM dynamics for various target hardware platforms. Most existing code generating systems lack a combination of automatic discretization, algebraic manipulation and, especially, simplification, and global common subexpression elimination. These key elements for the generation of efficient code are all included in the CTADEL system.

Related work. Since the early days of computing, attempts have been made to solve scientific or physical models numerically or symbolically using computers. Today, a large collection of libraries, tools, and Problem Solving Environments (PSEs) have been developed for solving such problems. The current and future role of these environments is discussed in [10]. Well-known PSEs for solving PDEs, see [9] for an extensive overview, are ELLPACK [20] and DEQSOL [22], and in the field of parallel computing //ELLPACK [13]. The computational kernels of these PSEs consist of a large library containing routines for many numerical solution methods. These routines form the templates for the resulting code. The numerical knowledge of such systems is determined by the power of its library.

A different approach than these library-based PSEs consists of a system that generates code based on the specific problem specification without the use of a library. Here the numerical knowledge is determined by the expressiveness of the problem specification language. This implicitely means that a powerful translation mechanism from that language to the resulting code is a vital component for these kind of systems. An example is described in [6]. Also the DPML Data Parallel scientific Modelling Language [8] falls in this category. The prototype of CTADEL should be considered as a system of the last type. However, a novel approach is taken, which, in contrast to DPML, includes automatic discretization of the model and reduction of the computational complexity of the problem prior to the generation of code.

This report is organized as follows. Section 2 briefly describes finite difference methods, the methods employed by CTADEL to solve a model numerically. In Section 3, the CTADEL prototype implementation will be discussed; the high level language constructs, discretization, and code generation. Section 4 presents preliminary results of the prototype implementation for two physical models. To conclude, in Section 5, we summarize our results and conclusions and deal with some issues related to further work.

2 Finite Difference Methods and Staggered Grids

In this section, a brief introduction to finite difference methods and staggered grids will be given. For a more detailed discussion the reader is referred to a textbook, e.g. [2]. Finite difference methods combined with staggered grids are at the moment the basic methods

employed by the prototype implementation of the CTADEL system to efficiently solve a model numerically. By means of an example the numerical solution of a physical model using finite difference methods and staggered grids will be demonstrated.

Many scientific models can be written generally as a set of n partial differential equations in the form

$$\frac{\partial}{\partial t}\mathcal{L}_i(u_i) = F_i(u_1, \ldots, u_n), \qquad i = 1, \ldots, n, \tag{1}$$

together with *initial conditions* and a set of *boundary conditions* which are said to hold on the boundary $\partial\Omega$ of the domain $\Omega \subset \mathbb{R}^d$. Here, $u_i = u_i(\mathbf{x}, t)$, $i = 1, \ldots, n$, are scalar functions of the space coordinates given by $\mathbf{x} \in \Omega$, and time t, called the *dependent variables* of the problem. The space coordinates $\mathbf{x}$ and time t are called the *independent variables* of the problem. In this report, we will use the phrase variable and field interchangably. F_i is a function involving the u_i as well as their space and time derivatives and $\mathcal{L}_i$ is a space differential operator which in many cases is the identity operator, i.e. $\mathcal{L}_i(u_i) = u_i$, or the Laplacian operator, i.e. $\mathcal{L}_i = \Delta = \nabla^2$.

As an example, consider the two-dimensional advection of temperature T in an incompressible medium on the domain $(x, y) \in \mathbb{R}^2$, time t. Periodic boundary conditions are enforced. Firstly, the advection of temperature can be described by

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = 0, \tag{2}$$

where u and v are the flow velocities in the x and y-direction, respectively. Secondly, the continuity equation

$$\nabla \cdot \mathbf{w} = 0 \tag{3}$$

holds, which states that the medium is of constant mass density. In the continuity equation $\nabla = [\frac{\partial}{\partial x}, \frac{\partial}{\partial y}]^T$ and $\mathbf{w} = [u, v]^T$. By combining Equations (2) and (3) we have

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{w}\, T) = 0, \tag{4}$$

which is the advection equation in flux form. Rewriting Equation (4) in the form of Equation (1) yields

$$\frac{\partial T}{\partial t} = -\frac{\partial (u\, T)}{\partial x} - \frac{\partial (v\, T)}{\partial y}. \tag{5}$$

The problem described by Equation (5) with appropriate boundary and initial conditions, can be solved numerically using finite difference methods. Finite difference methods are examples of so-called discrete methods. In these methods the continuous domain on which the problem is defined, is replaced by a grid of discrete points, which are called *grid points*. The values of variables of interest are only calculated on these grid points by numerical approximations.

Specific to finite difference methods is the approximation of partial derivatives by finite difference schemes. These schemes are based on difference quotients, that are derived from e.g. Taylor series. In addition, values that are required in the interval between grid points are obtained using interpolation methods. See for more details e.g. [2].

An important consideration in the choice of finite difference schemes is the arrangement of variables on the grid. The use of *staggered grids* is a common technique employed to

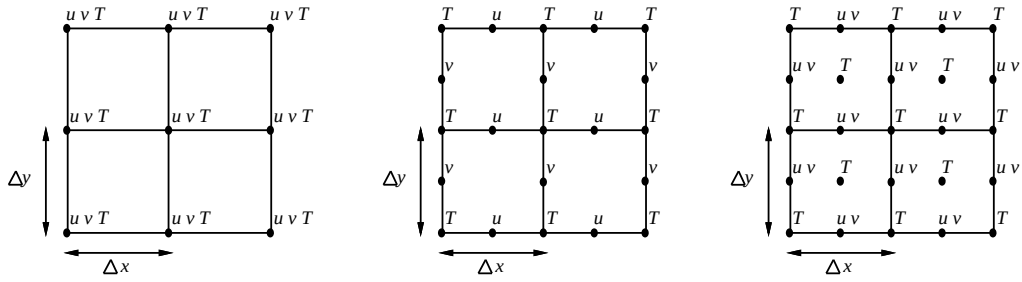


Figure 1: Arakawa A-grid (left), C-grid (middle), and E-grid (right).

avoid producing separate solutions on alternating grid points: when partial derivatives are approximated by centered difference quotients, for example

$$\frac{\partial u}{\partial x} \approx \frac{u(x_{i+1}, y_j) - u(x_{i-1}, y_j)}{x_{i+1} - x_{i-1}}, \tag{6}$$

the approximated value of the derivative of the variable u is determined by the values $u(x_{i+1}, y_j)$ and $u(x_{i-1}, y_j)$ at grid points (x_{i+1}, y_j) and (x_{i-1}, y_j) respectively, and is thereby completely decoupled from the value $u(x_i, y_j)$ at grid point (x_i, y_j). This decoupling results in two separate numerical solutions of the problem, which clearly is an undesirable result.

Various arrangements of variables in the horizontal domain for atmospheric models were classified by Arakawa [3]. Typical Arakawa grids are depicted in Figure 1. Using the C-grid, the finite difference approximation of Equation (5) using centered finite differences in space is given by

$$\frac{\partial T}{\partial t} = -\delta_x(u\,T) - \delta_y(v\,T), \tag{7}$$

where the finite difference operator δ_x is defined as

$$\delta_x\, u = \frac{u(x_i + \frac{1}{2}\Delta x, y_j) - u(x_i - \frac{1}{2}\Delta x, y_j)}{\Delta x} \tag{8}$$

with Δx the distance between two grid points in the x-direction. The operator δ_y is defined analogously with respect to y. From Equation (7) it can be observed that values of variable T are required at the same grid point positions as u and v are defined. However, with a C-grid, these values for T are not available. To obtain T at the required position, T is interpolated at a half grid point $x_i + \frac{1}{2}\Delta x$ in the x-direction. This is called *staggering* T with respect to x, denoted by $\overline{T}^x$. Analogously, $\overline{T}^y$ denotes the interpolated value of T at a half grid point $y_j + \frac{1}{2}\Delta y$ in the y-direction. In this case, since the δ_x operator is second order in Δx, linear interpolation is sufficiently accurate, that is, $\overline{T}^x$ is defined as

$$\overline{T}^x = \frac{1}{2}(T(x_i + \frac{1}{2}\Delta x, y_j) + T(x_i - \frac{1}{2}\Delta x, y_j)). \tag{9}$$

$\overline{T}^y$ is defined analogously with respect to y. With these definitions, Equation (7) becomes

$$\frac{\partial T}{\partial t} = -\delta_x(u\,\overline{T}^x) - \delta_y(v\,\overline{T}^y). \tag{10}$$

This equation shows one of the possible space discretizations of Equation (5). For the discretization in time, several finite difference schemes can be employed depending on the trade-off between required accuracy, stability, and computational complexity of the scheme. For example, *leap-frog* time differencing of the advection of temperature problem yields

$$T^{k+1} = T^{k-1} - 2\,\Delta t\,(\delta_x(u\,\overline{T^k}^x) + \delta_y(v\,\overline{T^k}^y)), \tag{11}$$

where T^k is the temperature at time k and Δt the time step. By applying finite difference methods combined with staggered grids, an approximation method has been chosen to derive discrete formula(s) from the original problem which gives for each time step a numerical solution of the problem.

3 CTADEL Implementation

In this section an outline of the CTADEL system will be presented. The CTADEL system takes a very high level language description of a model comprising a set of coupled partial differential equations, discretizes these equations using finite difference methods, and generates efficient Fortran 77 code for solving the model numerically. An overview of the CTADEL system is depicted in Figure 2. The translation of a model into code by the CTADEL system consist of three main phases.

Phase 1: In the first phase, the model specified in CTADEL's very high level language is discretized by replacing the partial derivatives by finite difference operators and by (de)staggering variables if necessary, as will be discussed in Section 3.3. The discrete equations are simplified using axiomatic laws from basic algebra and trigonometry, as will be discussed in Section 3.4. The model and the resulting simplified discretized equations of this first phase are also saved in LaTeX form for user verification and automatic report generation.

Phase 2: In the second phase, an intermediate internal representation of the discretized model is created prior to the generation of code in the third phase. Firstly, the discrete operators are expanded, e.g. finite difference operators are replaced by their corresponding finite difference quotients. Secondly, the resulting equations are simplified and optimized by employing global common subexpression elimination, as will be described in Section 3.5. The result of this phase is an intermediate form of code for the model, consisting of a set of assignments to (temporary) array variables. In addition, the array boundaries of each variable are derived symbolically through global inspection. These bounds are expressions containing symbolic and numerical constants and `max` and `min` functions. The obtained set of assignments is again saved in LaTeX form for report generation.

Phase 3: Finally, in the third phase, vectorizable Fortran 77 code is generated for the model, as will be discussed in Section 3.6. For the intermediate representation of the discretized model code is generated in the form of a Fortran 77 procedure. This procedure computes the numerical approximation of the problem for one time step. A scheme for the time integration should then be chosen by the user to complete the program. At the moment, the user has to code the time integration scheme by hand. The arguments of the procedure comprise the fundamental variables and derived variables of the model.

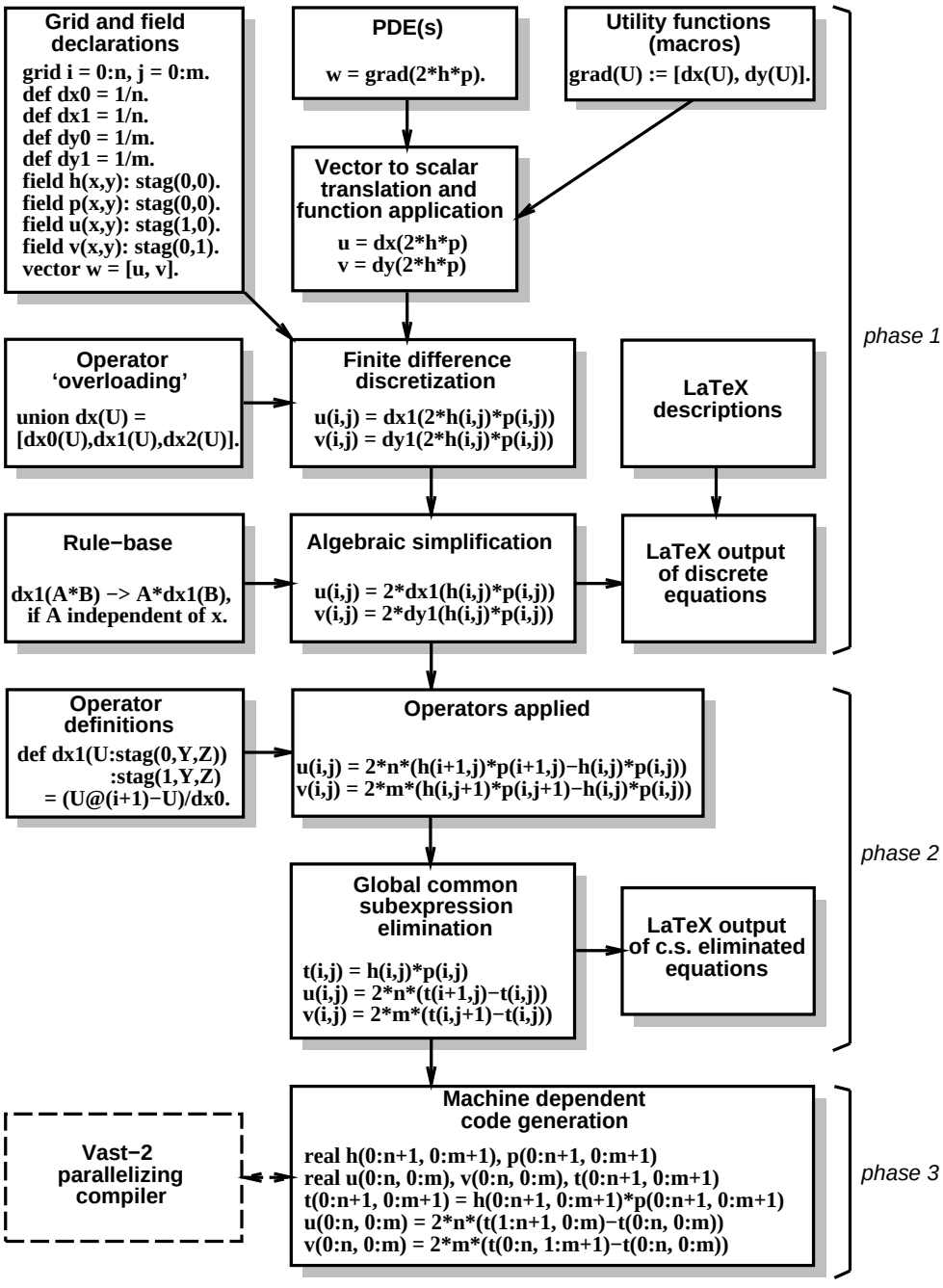


Figure 2: CTADEL system overview.

Array declarations for the fundamental, derived and temporary variables are generated with appropriately derived array bounds. The generated code consists of `do`-loops and `if-then-else` statements for the assignments to the array variables.

The CTADEL system supports the automatic generation of reports. As was mentioned above, the model, the set of discretized equations, and the set of assignments that result after elimination of common subexpressions are saved as LaTeX files. These files can be easily included in a report. Furthermore, the LaTeX output may serve as a feedback to the user to verify the result of each phase. Clearly, the LaTeX output for automatic report generation is more convenient for the user to verify than textual output.

The prototype CTADEL system is implemented using the Prolog programming language with the public domain SWI-Prolog [23] package.

3.1 The Very High Level Language of CTADEL

The very high level language of the CTADEL system provides a level of abstraction to hide the numerical solution techniques from the description of a problem. In this way, a model can be described in a natural way, close to the mathematical formulation of the problem. With the very high level language, the problem can be described for one time step.

In the following informal description of the high level language *expr* denotes an arithmetic expression involving fields, vectors, arithmetic operators and logarithmic and trigonometric functions. An *index-expr* involves indices and symbolic or integer constants only. A *logical-expr* is an expression which, in addition to *expr*, involves the logical connectives `and`, `or`, and `not`, the relational operators `<`, `=<`, `>`, `>=`, `=`, `\=`, and the constants `true` and `false`.

A vector in an expression is written `[`*expr*`,`*expr*`,...]` and a matrix in an expression is written `[[`*expr*`,`*expr*`,...],...]`, like a 'vector of vectors' where the innermost vectors constitute the rows of the matrix.

The following keywords can be used in the very high level language description of a model. Each keyword specification ends with a period (`.`). Parts enclosed in brackets *[* and *]* are optional.

`grid` *rangelist*. The description of a model should start with the declaration of a one, two, or three-dimensional grid. *rangelist* is a comma-separated list of (symbolic) ranges for each dimension of the form *index* `=` *lb* `:` *ub*. The *lb* and *ub* denote expressions consisting of (symbolic) integer constants only. This defines the indices and lower and upper boundaries of the discrete domain. For example,

```
grid i = 1:n, j = 1:n^2+1.
```

defines a two-dimensional $n \times n^2 + 1$ grid. The constant `n` is assumed to be an integer parameter declared explicitly by the user in the final Fortran 77 code.

`field` *fieldname*`(`*coordinatelist*`)`*[*`:stag(`*fieldstaglist*`)`*]*. Declares a field *fieldname* where *coordinatelist* is a comma-separated list of `x`, `y`, and/or `z` in this order which specifies the spatial coordinates of which the field is dependent. Optionally, the type of staggering of the field can be specified by appending the declaration with `:stag(`*fieldstaglist*`)`, where *fieldstaglist* is a comma-separated list of `0` (no staggering) and/or `1` (staggering on half grid point), each corresponding to the `x`, `y`, and `z` coordinates. The declaration of a field may appear only once. For example

```
field u(x,y): stag(1,0).
field v(z).
```

declares two fields u and v. The first declaration specifies that u is placed on a half grid-point for the x-dimension and on a whole grid point for the y-dimension. Field v is placed on whole grid points in the z-dimension implicitly.

`vector` *vectorname* `=` `[`*componentlist*`]`. Declares a vector *vectorname*, where *componentlist* is a comma-separated list of expressions for the components of the vector. The declaration of a vector may appear only once. For example

```
vector v = [sin(u), cos(u), 2].
```

declares a vector `v` with components `sin(u)`, `cos(u)`, and `2`.

lhs `=` *rhs*. Specifies an equation where *lhs* and *rhs* are expressions.

`infix` *op*. `prefix` *op*. `postfix` *op*. Declares an infix, prefix, or postfix function or operator *op*, respectively. This declaration should precede the definition of *op* and the use of *op* in an expression. To assign a left- or right-associative precedence to an infix function/operator, the `infix` *op* declaration may be optionally followed by `:leftassoc` or `:rightassoc`. The precedence of a user defined infix function/operator is stronger than the precedence of '`+`' and '`-`', but weaker than the precedence of '`*`' and '`/`'. The precedence of a user defined prefix or postfix function/operator is stronger than the precedence of any other operator. Use brackets when in doubt, or write *op* with its arguments in brackets, i.e. *op*`(`*arg*`,...)`. Only one of the three declarations `infix`, `prefix`, or `postfix` may be given for a given function or operator. As an example the statements

```
postfix !.
def N! extern fac(N).
```

declare the postfix factorial function which is defined externally as a Fortran 77 function in the final code by the user (see `def extern`).

funcname`[(`*arg*`,`*arg*`,...)]` `:=` *expr*. Defines a function *funcname* with optional formal arguments `(`*arg*`,`*arg*`,...)`. In this report, we will call these functions *utility functions* to distinguish these type of functions from Fortran 77 functions. Utility functions can be used to define e.g., gradient, divergence, and Laplacian operators. They are applied symbolically in the first phase of the translation. Each formal argument *arg* is either a name starting with an upper-case letter or is a vector of names starting with upper-case letters, e.g. `[A,B]`. Different utility functions may share the same function name as long as they have a different number of arguments and/or different kinds of arguments. For example, the divergence in one-, two-, and three-dimensional cartesian geometry can be defined as

```
prefix div.
div [A,B]   := dx A + dy B.
div [A,B,C] := dx A + dy B + dz C.
div A       := dx A.
```

utility function	meaning
`apply(`*op*`,[`*arg*`,...])`	apply *op* on arguments *arg*,..., i.e. *op*`(`*arg*`,...)`
`protect(`*expr*`)`	protect *expr*, see Section 3.4
`reduce(`*op*`,`*vector-expr*`)`	reduce a vector using binary operator *op*
`rhs(`*fieldname*`)`	return right hand side expression of equation for *fieldname*
vector-expr `.*` *vector-expr*	inner product
vector-expr `#` *vector-expr*	outer product
matrix-expr `:*` *vector-expr*	matrix-vector product

Table 1: Built-in utility functions.

When a utility function with multiple declarations is encountered in an expression, the system searches from the first to the last declaration for a definition of the function with matching arguments. The first matching declaration of the function will be selected and applied. Therefore, utility functions with more specific arguments, i.e., vectors, should always precede utility functions with more general arguments if they share the same function name. Utility functions without arguments can be considered as macros, i.e., their bodies will be substituted in the expression. Table 1 lists the predefined utility functions. For example, `reduce(+,[2,a,sin(u)])` sums the elements of the vector `[2,a,sin(u)]` to give `2+a+sin(u)`.

`def` *op*`[(`*arg*`[:stag(`*defstaglist*`)],`*arg*`[:stag(`*defstaglist*`)],...)][:stag(`*defstaglist*`)] =` *expr*.
Defines a discrete operator *op* with optional formal arguments `(`*arg*`,`*arg*`,...)`. Each formal argument *arg* should start with an upper-case letter and can be optionally qualified by a three-dimensional staggering specification `:stag(`*defstaglist*`)` where *defstaglist* is a comma-separated list of three elements. Each element is either `0`, `1`, `_` (underscore) or a name starting with an upper-case letter, see Section 3.2 for more details on the role of *defstaglist*. In addition, *op* can have an optional specification for a three-dimensional staggering for the value to be returned. Since the body of a discrete operator is expanded after the equations of a model are discretized, a body of an operator should comprise a discrete expression: fields should be indexed appropriately and the use of unions and utility functions is allowed in the body of the operator. Different operators can share the same operator if and only if their number of arguments differ. See Sections 3.2 and 3.3 for more details. Table 2 lists the predefined discrete operators. Their definitions and usage will be discussed in Section 3.3. The `@` operator can be used to index an expression and the arguments in the body of an operator. For example, the definition of the built-in central difference operator `dx2` is

```
def dx2(U:stag(0,Y,Z)):stag(0,Y,Z) = (U@(i+1,j,k)-U@(i-1,j,k))/(2*dx0).
```

A detailed description of this definition will be given in Section 3.3. The *index* of the reduction operators `any`, `all`, `count`, `sum`, `prod`, `min`, and `max` should be an index declared by the `grid` declaration. For example, in the model description

```
grid i=1:n, j=1:m.
field u(x,y).
field v(x,y).
v=sum(i=i:n,u).
```

operator	meaning
`any(`*index* `=` *index-expr* `:` *index-expr*, *logical-expr*`)`	global or
`all(`*index* `=` *index-expr* `:` *index-expr*, *logical-expr*`)`	global and
`count(`*index* `=` *index-expr* `:` *index-expr*, *logical-expr*`)`	count true elements
`sum(`*index* `=` *index-expr* `:` *index-expr*, *expr*`)`	sum of elements
`prod(`*index* `=` *index-expr* `:` *index-expr*, *expr*`)`	product of elements
`max(`*index* `=` *index-expr* `:` *index-expr*, *expr*`)`	maximum of elements
`min(`*index* `=` *index-expr* `:` *index-expr*, *expr*`)`	minimum of elements
`if(`*logical-expr*, *expr*, *expr*`)`	if-then-else
expr `@ (`*index-expr*[, *index-expr*]*`)`	value at discrete index
`sx(`*expr*`)` `sy(`*expr*`)` `sz(`*expr*`)`	(de)staggering with respect to x, y, or z
`sx0(`*expr*`)` `sy0(`*expr*`)` `sz0(`*expr*`)`	(See Sections 3.2 and 3.3)
`sx1(`*expr*`)` `sy1(`*expr*`)` `sz1(`*expr*`)`	,,
`dx(`*expr*`)` `dy(`*expr*`)` `dz(`*expr*`)`	centered finite differences w.r.t. x, y, or z
`dx0(`*expr*`)` `dy0(`*expr*`)` `dz0(`*expr*`)`	(See Sections 3.2 and 3.3)
`dx1(`*expr*`)` `dy1(`*expr*`)` `dz1(`*expr*`)`	,,
`dx2(`*expr*`)` `dy2(`*expr*`)` `dz2(`*expr*`)`	,,
`qx(`*lo*`,`*hi*`,`*expr*`)` `qy(`*lo*`,`*hi*`,`*expr*`)` `qz(`*lo*`,`*hi*`,`*expr*`)`	numerical quadrature *lo*,...,*hi* w.r.t. x, y, or z
`qx0(`*lo*`,`*hi*`,`*expr*`)` `qy0(`*lo*`,`*hi*`,`*expr*`)` `qz0(`*lo*`,`*hi*`,`*expr*`)`	(See Sections 3.2 and 3.3)
`qx1(`*lo*`,`*hi*`,`*expr*`)` `qy1(`*lo*`,`*hi*`,`*expr*`)` `qz1(`*lo*`,`*hi*`,`*expr*`)`	,,
`qx2(`*lo*`,`*hi*`,`*expr*`)` `qy2(`*lo*`,`*hi*`,`*expr*`)` `qz2(`*lo*`,`*hi*`,`*expr*`)`	,,
`nhx(`*expr*`)` `nhy(`*expr*`)` `nhz(`*expr*`)`	value of *expr* at next half grid point
`nhx0(`*expr*`)` `nhy0(`*expr*`)` `nhz0(`*expr*`)`	w.r.t. x, y, or z (See Sections 3.2 and 3.3)
`nhx1(`*expr*`)` `nhy1(`*expr*`)` `nhz1(`*expr*`)`	,,
`phx(`*expr*`)` `phy(`*expr*`)` `phz(`*expr*`)`	value of *expr* at previous half grid point
`phx0(`*expr*`)` `phy0(`*expr*`)` `phz0(`*expr*`)`	w.r.t. x, y, or z (See Sections 3.2 and 3.3)
`phx1(`*expr*`)` `phy1(`*expr*`)` `phz1(`*expr*`)`	,,
`nosx(`*expr*`)` `nosy(`*expr*`)` `nosz(`*expr*`)`	prevent the implicit use of `sx`, `sy`, or `sz`
	to convert *expr* at a half grid point to a
	whole grid point and v.v., w.r.t. x, y, or z

Table 2: Built-in operators.

field v will contain all partial sums of field u from n to 1, that is, $v_{n,j} = u_{n,j}$, $v_{n-1,j} = u_{n-1,j} + u_{n,j}$, ..., $v_{1,j} = u_{1,j} + \cdots + u_{n,j}$.

`def` *op*`[(`*arg*`[:stag(`*defstaglist*`)],`*arg*`[:stag(`*defstaglist*`)],...)][:stag(`*defstaglist*`)]` `extern` *funcname*`[(`*arg*`,`*arg*`,...)]`. Similar to `def` above but the body of the declaration is replaced by a Fortran 77 function call: it declares an external Fortran 77 function *funcname* to be called where *op* occurs in an expression. The arguments *arg* should be names starting with an upper-case letter. The actual arguments of *op* are copied to the corresponding call to *funcname*. Example

```
postfix !.
def N! extern fac(N).
```

`union` *unionfieldname* `=` `[`*fieldlist*`]`. Declares a union *unionfieldname* of the set *fieldlist*. *fieldlist* is a comma-separated list of differently staggered fields and non-staggered fields. Each occurrence of the union *unionfieldname* in an equation will be replaced by one of the fields in *fieldlist*. The first field in *fieldlist* with the 'closest' matching type of staggering will be selected and substituted for `unionfieldname`. See Sections 3.2 and 3.3 for more details. For example,

```
field u00(x,y): stag(0,0).
field u10(x,y): stag(1,0).
field v(x,y): stag(0,0).
union u = [u00, u10].
v = u+sin(u).
```

declares that `u` is a union of `u00` and `u10`. In this example `u00` will be substituted for `u` in the equation for `v`.

`union` *unionop*`[(`*arg*`,`*arg*`,...)]` **`=`** `[`*oplist*`]`. Declares a union *unionop* of a set of operators *oplist*. The optional arguments *arg* of *unionop* are names starting with an upper-case letter and *oplist* is a comma-separated list of *op*`(`*arg*`,`*arg*`,...)`, where *arg* should be expressions containing the arguments of the union *unionop*`(`*arg*`,`*arg*`,...)`. Each occurrence of *unionop*`(`*arg*`,`*arg*`,...)` in an equation will be replaced by one of the operators in *oplist*. The first operator in *oplist* with the 'closest' matching type of staggering will be substituted. See Sections 3.2 and 3.3 for more details.

`latex` *name* **`=`** `[`*latexlist*`]`. **`latex`** *op*`(`*arg*`,`*arg*`,...)` **`=`** `[`*latexlist*`]`. The first form specifies a LaTeX description for the name of a field, vector, function or operator. The second form is more specific for functions and operators as it allows the LaTeX description for *op* and its arguments. The formal arguments *arg* are names starting with an upper-case letter. *latexlist* is a comma-separated list of either strings enclosed in double quotes (`"`) or arguments *arg* of the function/operator *op*. The default LaTeX output of the system prints fields in italic style with names of Greek symbols printed as Greek symbols, boldface style for names of vectors, and roman style for the names of functions. Furthermore, default LaTeX descriptions are given for the built-in utility functions and operators. These descriptions can be overwritten. See the examples in Section 4. Vectors and matrices will be printed using brackets. These specifications are only necessary to overwrite the default LaTeX representation. It is mandatory to include a LaTeX description for names that contain characters and/or character sequences that are interpreted by TeX or LaTeX. For example, if an operator `&` is defined, a latex description should be given, e.g.

```
latex & = ["\&"].
```

Some other examples are

```
latex t = ["T"].
latex div Vec = ["\nabla\cdot",Vec].
```

will result in the printing of T for `t` and $\nabla \cdot \begin{bmatrix} u \\ v \end{bmatrix}$ for `div [u,v]`.

`%` *comment* Comments can be placed anywhere. A comment begins with a `%` and lasts to the end of a line.

It is important to note that names for fields, vectors, functions, and operators should all start with a lower-case letter. Except for the grid declaration and the **`prefix`**, **`postfix`** and **`infix`** keywords, the keywords described above can be given in any order.

The current prototype implementation of the system adopts explicit finite difference methods as a solution method only. In addition, the left-hand side of an equation should comprise a field or a vector of fields only, that is, in Equation (1) $\mathcal{L}_i$, for all $i = 1, \ldots, n$, is the identity

operator. Hence, elliptic problems cannot be formulated yet. In the future, however, the system will be extended with (semi-) implicit finite difference methods and elliptic problems can be specified as well.

3.2 Operator Overloading and Type Inference

During the first phase of the translation of a model the default and user defined utility functions, which define for example inner product, gradient, and Laplacian operators, are applied symbolically. In this way, the system of partial differential equations of a model is translated to a system of equations containing logarithmic and trigonometric functions and discrete operators on scalars. Then, if necessary, each resulting equation, which may still be formulated in vector form, is translated into a vector of equations, each in scalar form, by the usual deterministic mathematical rewriting. For example, `[u,v]=[1,-1]-sin(2*[p,q])^2` is translated into `[u=1-sin(2*p)^2,v=-1-sin(2*q)^2]`. After converting the model equations in scalar form, type inference is employed in order to determine the type of staggering of fields and discrete operators in the model equations. When this type information is available, the equations can be discretized. In the equations of the model, appropriate discrete operators are selected depending on the type of arguments of the operator. Furthermore, subexpressions in the equations are (de)staggered to convert between several types of staggering.

Operator overloading provides a mechanism for defining discrete operators that operate differently on differently staggered expressions. For example, the definition of the central difference of a field depends on the type of staggering of this field, as will be discussed in Section 3.3. To this end, the definition of the union for a discrete operator provides a means for overloading the discrete operator. That is, depending on the type of the context and arguments of the operator, an appropriate operator is selected from the list specified for this overloaded operator.

In general, using overloaded operators results in sets of feasible types for an expression, see e.g. [1]. Therefore, the type inference mechanism follows a repeated forward/backward scheme. Repetition is necessary due to the implementation of parameterized types with possible variable parameters. In fact, for the implemented parameterized `stag` type, constant, variable and wildcard parameters are to be distinguished. Constant parameters are either `0` (no staggering) or `1` (staggering on half grid point). The variable parameters of `stag` are specified as names starting with an upper-case letter. These variable parameters take either a value of `0` or `1`. For example, a variable parameter of the type of a formal argument of an operator takes a value of the type of the actual argument of the operator. A wildcard parameter, denoted by `_` (underscore), discards any value for the type parameter.

Using variable type parameters, the type of staggering of the actual arguments of an operator can be copied to the result type of the operator and vice versa. For example, consider the following declarations:

```
field u(x,y,z): stag(0,1,0).
field v(x,y,z): stag(0,0,0).
def op(A: stag(0,Y,_)): stag(1,Y,1) = expressionfor(A).
field w(x,y,z): stag(1,1,1).
w = op(u) + op(v).
```

Type inference of the specified equation reveals that the type of `op(u)` is `stag(1,1,1)`. The variable type parameter `Y` in the declaration of `op` takes the value `1` when the type of the expression `op(u)` is being inferred. Furthermore, `_` discards the second `0` type parameter of

`stag(0,1,0)`. Similarly, the type of `op(v)` is `stag(1,0,1)`. In this way, the type system is said to have sub-typing semantics, e.g. `stag(0,1,_)` is a subtype of `stag(X,1,_)` while `stag(0,0,0)` is not. That is, an operator with `stag(X,1,_)` as argument type will accept an actual argument of type `stag(0,1,_)` and not an actual argument of type `stag(0,0,0)`. Clearly, the main advantage of the implemented type inference scheme employing variable type parameters is that operators do not have to be defined for every possible multi-dimensional type of staggering.

As was mentioned above, the definition of a union for a set of operators provides a mechanism for overloading an operator. When a union is declared for some set of operators and this union is encountered in an equation, the first operator of the specified list is selected which has the 'closest' matching type of staggering for replacing the union. By selecting the 'closest' matching type, a minimum of type conversions by (de)staggering are needed for converting the type of the actual arguments and result of the operator. The 'closest' matching type is computed using the Hamming distance between the types of the formal and actual arguments in the space spanned by the `0`–`1` type parameter values. The first operator in the specified list for a union is selected with the smallest sum of Hamming distances of the result and arguments of the operator. Similarly, the first field is selected from the list with the smallest Hamming distance between the actual and required type of staggering. After this selection, CTADEL inserts `sx`, `sy`, and/or `sz` operators to convert the arguments and resulting value of a field or operator that (de)stagger with respect to x, y, and/or z, respectively. For example, consider the following declaration for three staggered fields p, q, and r and an overloaded operator *op*:

```
field p0(x,y,z): stag(0,1,0).
field p1(x,y,z): stag(1,1,0).
union p = [p0, p1].
field q(x,y,z): stag(0,0,0).
def op0(A: stag(1,Y,_)): stag(0,Y,1) = expressionfor(A).
def op1(A: stag(0,Y,_)): stag(0,Y,1) = expressionfor(A).
union op(U) = [op0(U), op1(U)].
field r(x,y,z): stag(0,1,1).
r = op(p) + op(q).
```

Note that field p is staggered, via $p1$, as well as non-staggered, via $p0$, with respect to x, which is specified using the union declaration for p. Now, `op(p)` in the equation for r will be replaced by `op0(p1)` since the type of the context in which `op(p)` occurs is `stag(0,1,1)`, the type of field r. Similarly, `op(q)` will be replaced by `sy1(op1(q))`, where `sy1` converts the type of `op1(q)` to `stag(0,1,1)` by staggering with respect to y.

Clearly, the use of unions in expressions possibly with variable parameter types complicates type inference considerably and, therefore, the selection and replacement of overloaded operators.

In the next section, we will discuss the finite difference discretization using the default discrete operators and illustrate this automatic process of discretization by means of an example.

3.3 Discretization

As discussed in the previous section, during the first translation phase of a model, the model equations are discretized after type inference has taken place. In this section, we will discuss

the default second order finite difference discretization and set guidelines to obtain other user-defined finite difference discretizations.

For the default discretization of a model, the following operations are defined:

- (De)staggering by linear interpolation of a field u

$$\overline{u}^x = \begin{cases} \overline{u}^{x-} = \frac{1}{2}(u_{i-1} + u_i) & \text{if } u \text{ is staggered in the } x\text{-direction} \\ \overline{u}^{x+} = \frac{1}{2}(u_i + u_{i+1}) & \text{otherwise.} \end{cases}$$

- Central finite differences of a field u

$$\delta_x\, u = \begin{cases} \delta_x^-\, u = \frac{u_i - u_{i-1}}{\Delta x} & \text{if } u \text{ is staggered in the } x\text{-direction} \\ \delta_x^+\, u = \frac{u_{i+1} - u_i}{\Delta x} & \text{otherwise,} \end{cases}$$

$$\delta_{2x}\, u = \frac{u_{i+1} - u_{i-1}}{2\,\Delta x}.$$

- Numerical quadrature over a field u using the midpoint and trapezoidal formula

$$\mathrm{midp}_x(a,b,u) = \begin{cases} \mathrm{midp}_x^-(a,b,u) = \displaystyle\sum_{i=a}^{b-1} u_i\,\Delta x & \text{if } u \text{ is staggered in the } x\text{-dir.} \\ \mathrm{midp}_x^+(a,b,u) = \displaystyle\sum_{i=a+1}^{b} u_i\,\Delta x & \text{otherwise,} \end{cases}$$

$$\mathrm{trap}_x(a,b,u) = \left(\sum_{i=a+1}^{b-1} u_i\,\Delta x\right) + \frac{1}{2}\left(u_a\,\Delta x + u_b\,\Delta x\right).$$

In the definitions above, Δx denotes the grid-point distance in the x-direction. Furthermore, u_i denotes to the discretized version of a field u, that is, if u is non-staggered with respect to x, we have that $u_i = u(x_i)$ on the grid points x_i, $i = 1, \ldots, n$ and if u is staggered with respect to x, we have that $u_i = u(x_{i+\frac{1}{2}})$ on the half grid $x_{i+\frac{1}{2}}$, $i = 1, \ldots, n$.

The corresponding definitions for the built-in operators formulated in the very high level language are:

```
def sx0(U:stag(1,Y,Z)):stag(0,Y,Z) = (U@(i-1,j,k) + U)/2.
def sx1(U:stag(0,Y,Z)):stag(1,Y,Z) = (U + U@(i+1,j,k))/2.

def dx0(U:stag(1,Y,Z)):stag(0,Y,Z) = (U - U@(i-1,j,k))/dx1.
def dx1(U:stag(0,Y,Z)):stag(1,Y,Z) = (U@(i+1,j,k) - U)/dx0.
def dx2(U:stag(0,Y,Z)):stag(0,Y,Z) = (U@(i+1,j,k) - U@(i-1,j,k))/(2*dx0).

def qx0(A,B,U:stag(1,Y,Z)):stag(0,Y,Z) = sum(i=A:B-1, dx1*U).
def qx1(A,B,U:stag(0,Y,Z)):stag(1,Y,Z) = sum(i=A+1:B, dx0*U).
def qx2(A,B,U:stag(0,Y,Z)):stag(0,Y,Z) = sum(i=A+1:B-1, dx0*U)
                                       + ((dx0*U)@(A,j,k) + (dx0*U)@(B,j,k))/2.

union sx(U) = [sx0(U), sx1(U)].
union dx(U) = [dx0(U), dx1(U), dx2(U)].
union qx(Lo,Hi,U) = [qx0(Lo,Hi,U), qx1(Lo,Hi,U), qx2(Lo,Hi,U)].
```

and the corresponding default LaTeX specifications for the above operators are:

```
latex sx(U) = ["{\overline{", U, "}^x}"].
latex sx0(U) = ["{\overline{", U, "}^{x_-}}"].
latex sx1(U) = ["{\overline{", U, "}^{x_+}}"].
latex dx(U) = ["{\frac{\partial}{\partial x}\,}", U].
latex dx0(U) = ["{\delta_x^-\,}", U].
latex dx1(U) = ["{\delta_x^+\,}", U].
latex dx2(U) = ["{\delta_{2x\,}", U].
latex qx(A,B,U) = ["\int_{x_{", A, "}}^{x_{", B, "}}", U, "\,dx"].
latex qx0(A,B,U) = ["midp_x^-\left(", A, ",", B, ",", U, "\right)"].
latex qx1(A,B,U) = ["midp_x^+\left(", A, ",", B, ",", U, "\right)"].
latex qx2(A,B,U) = ["trap_x\left(", A, ",", B, ",", U, "\right)"].
```

The definitions of the built-in operators with respect to y and z and their LaTeX specifications are similar. Note that the built-in operators assume that the grid is defined by the indices `i`, `j`, and `k`. All of these operator definitions can be redefined by the user.

To specify a uniform staggered grid with grid point distance h, the user can specify the following declarations

```
grid i = 1:n, ...
def x0: stag(0,_,_) = i*h.
def x1: stag(1,_,_) = (i+0.5)*h.
union x = [x0, x1].
def dx0 = h.
def dx1 = h.
```

In case a tensor product grid is required where the consecutive grid points are stored in $x0(i)$, $i = 1, \ldots, n$, and the consecutive half grid points are stored in $x1(i)$, $i = 1, \ldots, n$, the following declarations can be given:

```
grid i = 1:n, ...
field x0(x): stag(0).
field x1(x): stag(1).
union x = [x0, x1].
def dx0: stag(0,_,_) = x0(i+1)-x0(i).
def dx1: stag(1,_,_) = x1(i+1)-x1(i).
```

The definitions with respect to y and z are similar. It may seem odd to include a definition for `x` in the specification of a grid as it is an independent variable. However, this definition is very convenient because `x` simply denotes the value of the independent variable x at each (half) grid point in an equation where `x` occurs. For example, in the following high level language specification

```
field hx(x): stag(1).
hx = cos(x).
```

the cosine of x at each half grid point is assigned to `hx`, that is, `hx(i) = cos((i+0.5)*h)` in case of a uniform grid and `hx(i) = cos(x1(i))` in case of a tensor product grid.

As is described in Section 3.1, the infix `@` operator can be used to index an expression. This operator only takes integer index expressions. Hence, the value of an expression at a neighboring half grid point cannot be indexed using the `@` operator. To this end, the prefix `nh` and `ph` operators index an expression at the next and previous half grid point with respect to `x`, `y`, or `z`. With respect to `x`, the definitions of these operators are:

```
% A-grid advection of temp
grid i = 1:n, j = 1:m.
def dx0 = 1/(n-1).
def dy0 = 1/(m-1).
field u(x,y).
field v(x,y).
vector w = [u, v].
field t(x,y).
field dTdt(x,y).
prefix div.
div [A,B] := dx A + dy B.
dTdt = -div(w*t).
```

```
% C-grid advection of temp
grid i = 1:n, j = 1:m.
def dx0 = 1/(n-1).
def dx1 = 1/(n-1).
def dy0 = 1/(m-1).
def dy1 = 1/(m-1).
field u(x,y): stag(1,0).
field v(x,y): stag(0,1).
vector w = [u, v].
field t(x,y): stag(0,0).
field dTdt(x,y): stag(0,0).
prefix div.
div [A,B] := dx A + dy B.
dTdt = -div(w*t).
```

```
% E-grid advection of temp
grid i = 1:n, j = 1:m.
def dx0 = 1/(n-1).
def dx1 = 1/(n-1).
def dy0 = 1/(m-1).
def dy1 = 1/(m-1).
field u01(x,y): stag(0,1).
field u10(x,y): stag(1,0).
union u = [u01, u10].
field v01(x,y): stag(0,1).
field v10(x,y): stag(1,0).
union v = [v01, v10].
vector w = [u, v].
field t00(x,y): stag(0,0).
field t11(x,y): stag(1,1).
union t = [t00, t11].
field dTdt(x,y): stag(0,0).
prefix div.
div [A,B] := dx A + dy B.
dTdt = -div(w*t).
```

Figure 3: Very high level language model description for one time step of the two-dimensional advection of temperature in an incompressible medium discretized on a uniform Arakawa A-grid (left), C-grid (middle), and E-grid (right).

```
def nhx0(U:stag(1,Y,Z)):stag(0,Y,Z) = U.
def nhx1(U:stag(0,Y,Z)):stag(1,Y,Z) = U@(i+1,j,k).

def phx0(U:stag(1,Y,Z)):stag(0,Y,Z) = U@(i-1,j,k).
def phx1(U:stag(0,Y,Z)):stag(1,Y,Z) = U.

union nhx(U) = [nhx0(U), nhx1(U)].
union phx(U) = [phx0(U), phx1(U)].
```

Now, for example, the equation

$$v_{i,j,k} = u_{i,j,k+1\frac{1}{2}}$$

can be written

```
v = nhz u @ (i,j,k+1).
```

The insertion of the `sx`, `sy`, and/or `sz` conversion operators can be prevented by using `nosx`, `nosy`, and/or `nosz`, respectively. These operators are defined as

```
def nosx(U:stag(_,Y,Z)):stag(_,Y,Z) = U.
def nosy(U:stag(X,_,Z)):stag(X,_,Z) = U.
def nosz(U:stag(X,Y,_)):stag(X,Y,_) = U.
```

To conclude this section, we will discuss the discretization of the advection of temperature problem from Section 2 on an A-, C-, and E-grid. Figure 3 depicts the model descriptions for an A-, C-, and E-grid, respectively.

In Figure 3, `grid i=1:n, j=1:m.` declares an $n \times m$ grid indexed by `i` and `j`. The grid is uniform in the x- and y-directions as is declared by the `def`'s for `dx0`, `dy0` (grid distances

between whole grid points), and `dx1`, `dy1` (grid distances between half grid points). The type of a grid is determined by using `:stag(`*fieldstaglist*`)` qualifications for fields and by declaring unions for sets of fields. Note that Equation (4) can be specified in the very high level language for all the three types of grids without any changes.

The discretization by the CTADEL system of the problem formulated for the A-grid results in

```
dTdt(i,j) = -dx2(u(i,j)*t(i,j))-dy2(v(i,j)*t(i,j))
```

which is presented to the user using the LaTeX package as

$$\frac{\partial T}{\partial t}_{\mathrm{i,j}} = -\delta_{2x}(u_{\mathrm{i,j}}\, T_{\mathrm{i,j}}) - \delta_{2y}(v_{\mathrm{i,j}}\, T_{\mathrm{i,j}})$$

The discretization of the problem formulated for the C-grid results in

```
dTdt(i,j) = -dx0(u(i,j)*sx1(t(i,j)))-dy0(v(i,j)*sy1(t(i,j)))
```

with corresponding LaTeX output

$$\frac{\partial T}{\partial t}_{\mathrm{i,j}} = -\delta_x^-(u_{\mathrm{i,j}}\, \overline{T_{\mathrm{i,j}}}^{x_+}) - \delta_y^-(v_{\mathrm{i,j}}\, \overline{T_{\mathrm{i,j}}}^{y_+}),$$

Finally, the discretization of the problem formulated for the E-grid results in

```
dTdt(i,j) = -dx0(u10(i,j)*sx1(t00(i,j)))-dy0(v01(i,j)*sy1(t00(i,j)))
```

which is presented to the user as

$$\frac{\partial T}{\partial t}_{\mathrm{i,j}} = -\delta_x^-(u10_{\mathrm{i,j}}\, \overline{T00_{\mathrm{i,j}}}^{x_+}) - \delta_y^-(v01_{\mathrm{i,j}}\, \overline{T00_{\mathrm{i,j}}}^{y_+}).$$

Note that for the E-grid `dTdt` is unstaggered, and thus the model only provides tendencies for the variable `t` on the unstaggered gridpoints `t00`.

This example illustrated the basic idea of the employment of operator overloading techniques for the automatic discretization of equations.

3.4 Algebraic Simplification

Algebraic simplification of the discretized model equations is extensively used in the first phase of the translation. While a problem can be posed conveniently using the very high level language constructs, the underlying computational complexity of the problem can be vast. Much of this complexity can be reduced by application of algebraic simplification. To this end, the decision was made to build a dedicated algebraic simplifier within the CTADEL system. Soon it was realized that existing symbolic algebra packages such as Maple [4] and Mathematica™ [24], were not very suitable for this task. Although these packages are powerful tools with respect to, for example, symbolic differentiation, they lack a suitable algebraic simplifier that can be easily extended. Furthermore, most existing algebraic simplifiers produce expressions in *canonical form* which are easy to read but not very economical with respect to arithmetic complexity. Basically, the implemented simplifier adopts a *hill climbing* strategy by applying as many simplification rules as possible to an expression. For the simplification of an expression, an 'expand-contract' strategy has been adopted. Firstly, the expression is expanded by using the laws governing linear operators and the distributive law. Secondly, the

expression is contracted by using the laws governing linear operators, the distributive law, and by using Horner's form for polynomials. Furthermore, the CTADEL simplifier is able to extensively simplify expressions involving the built-in operators as well as trigonometric and logarithmic functions. To this end, the simplifier consults an extensible rule base each time simplification takes place. The rewriting rules in the rule-base are of the form *tag*; *lhs* `==>` *rhs[*, *condition*, *condition*,...*]*. where the list of *condition*s is an optional conjunction of logical conditions. For example:

```
lin;  dx0(X) + dx0(Y) ==> dx0(X+Y).
alg;  dx0(X)          ==> 0, indepx(X).
alg;  sum(I=A:B,X)    ==> if(B>=A,(B-A+1)*X,0), indep(I,X).
trig; sin(X) * cos(X) ==> sin(2*X)/2.
log;  log(X^Y)        ==> Y*log(X).
```

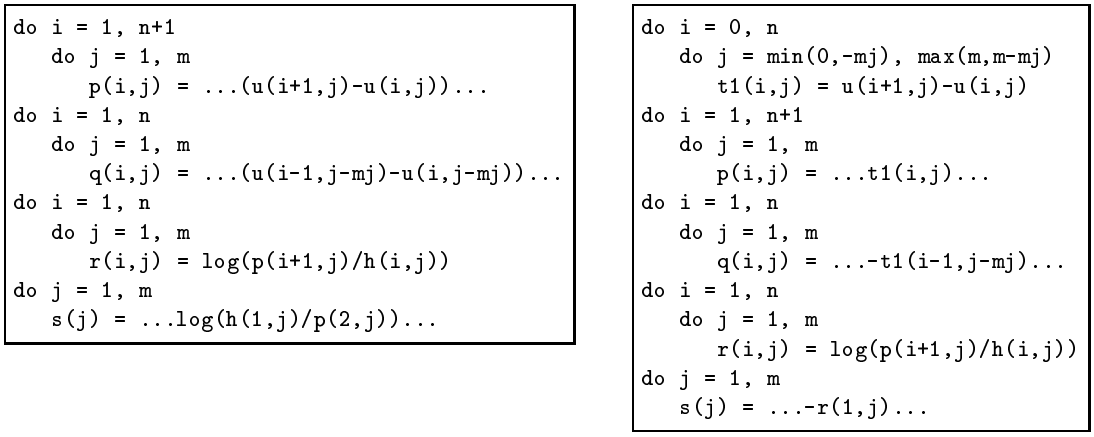
The tag `lin` denotes that this rule is derived from the laws governing linear operators. `alg` denote that these rules are always applicable. Here, `indepx(A)` checks that `A` is independent of the x-dimension by checking for the occurrence of the index variable corresponding with x. Similarly, `indep(I,X)` checks for the occurrence of the index `I` in `X`. The checking is performed through interprocedural analysis of the body of operators contained in the expression. `trig` and `log` denote rules that are to be applied for simplification of expressions involving trigonometric and logarithmic functions.

The employment of operator overloading techniques for automatic discretization falls short in case of the discretization involving basic arithmetic binary operations between expressions which are defined on differently staggered grid points. Consider an expression of the form `u*v*w` where fields `u` and `w` are staggered with respect to x and `v` is not. Then, if the `*` operator is subject to overloading, i.e. the following declarations are given:

```
def *00(X: stag(0,Y,Z), Y: stag(0,Y,Z)): stag(0,Y,Z) = X * Y.
def *01(X: stag(0,Y,Z), Y: stag(1,Y,Z)): stag(0,Y,Z) = X * sx0(Y).
def *10(X: stag(1,Y,Z), Y: stag(0,Y,Z)): stag(0,Y,Z) = sx0(X) * Y.
...
union *ol(A,B) = [*00(A,B), *01(A,b), *10(A,B), ...].
```

we will obtain the discrete form `sx0(u)*v*sx0(w)`. In contrast to this, if the expression had been formulated as `u*w*v`, we would have obtained the discrete form `sx0(u*w)*v`. Both forms are not identical numerically. In this respect, it should be mentioned that the product of two (de)staggered expressions should be rewritten into the (de)staggered product of expressions. The only solution to this problem is to prohibit the overloading of the multiplication operator and to use the CTADEL algebraic simplifier supplied with a set of appropriate rewriting rules concerning the stagger, finite difference, and quadrature operators. In this way, the discrete expression `sx0(u)*v*sx0(w)` is rewritten by the CTADEL simplifier into `sx0(u*w)*v` using the commutative and associative laws of the `*` operator. Hence, the simplifier rewrites discrete expressions into more economical expressions that would have been obtained by hand. For example, the expression `sy0(u(i,j))*dx0(sy0(a*v(i,j))+sy0(v(i,j)))` is rewritten into `(a+1)*sy0(u(i,j)*dx0(v(i,j)))`. The amount of arithmetic involved in the latter expression is less than that of the former.

Expressions involving the `dx`, `dy`, `dz`, `qx`, `qy`, and `qz` operators are extensively simplified. For example, the expression `qz0(a,b,dx1(u(i,j)*v(i,j,k))` is rewritten into the equivalent expression `dx1(u(i,j)*qz0(a,b,v(i,j,k)))` which is less complex considering the fact that `qz0` performs a summation in `k`. In addition, if-then-else expressions will be 'migrated' as far



```
do i = 1, n+1
   do j = 1, m
      p(i,j) = ...(u(i+1,j)-u(i,j))...
do i = 1, n
   do j = 1, m
      q(i,j) = ...(u(i-1,j-mj)-u(i,j-mj))...
do i = 1, n
   do j = 1, m
      r(i,j) = log(p(i+1,j)/h(i,j))
do j = 1, m
   s(j) = ...log(h(1,j)/p(2,j))...
```

```
do i = 0, n
   do j = min(0,-mj), max(m,m-mj)
      t1(i,j) = u(i+1,j)-u(i,j)
do i = 1, n+1
   do j = 1, m
      p(i,j) = ...t1(i,j)...
do i = 1, n
   do j = 1, m
      q(i,j) = ...-t1(i-1,j-mj)...
do i = 1, n
   do j = 1, m
      r(i,j) = log(p(i+1,j)/h(i,j))
do j = 1, m
   s(j) = ...-r(1,j)...
```

Figure 4: Example global common subexpression elimination, before (left) and after (right).

as possible to the front of an equation. During this process, nested if-then-else expressions can be simplified. For example, `if(k=1,if(k=0,a,b),c)` is rewritten into `if(k=1,b,c)`.

The CTADEL simplifier tries to combine as many terms as possible in order to reduce the underlying arithmetic complexity. In some rare cases, however, it is necessary to protect a subexpression from combining with other subexpressions. To this end, the built-in `protect` operator can be used. Each occurrence of `protect(`*expr*`)` protects *expr* from combining with subexpressions 'outside'. For example, `protect(u)*v` where `u` and `v` are staggered fields in the x-direction, results in $\overline{u}^{x-}\,\overline{v}^{x-}$ instead of $\overline{u\,v}^{x-}$ if the value of their product is required on a non-staggered grid point. This introduces a mechanism to keep additional constraints satisfied. An example of such a constraint could be that the discretisation scheme does not break conservation of energy. In [21] the reader finds how such constraints may determine in what order operators like differentiation and staggering have to be applied.

3.5 Global Common Subexpression Elimination

In the second translation phase of the CTADEL system, the elimination of common subexpressions is performed on the equations that result from the discretization of the model equations.

The elimination of common subexpressions on a global scale may greatly reduce the number of arithmetic operations at a moderate cost of introducing temporary variables. Especially the application of finite difference and (de)staggering operators inevitably leads to arithmetic expressions involving many subterms of the form $u_{i\pm c_1,j\pm c_2,k\pm c_3}$, where u is a three-dimensional discretized field and c_1, c_2, $c_3 \in \mathbb{N}$. In general, the employment of finite difference methods complicates global common subexpression elimination since the index of a common subexpression may differ by a constant offset. The CTADEL global common subexpression eliminator is capable of finding subexpressions of common value for which the computational domains are shifted by a constant distance with respect to each other. That is, the subexpressions are identical, but the indices of the variables comprising the expression are offset by a (symbolic) constant. In addition, the eliminator will find common subexpressions on rectangular subspaces of the computational domain such as common expressions for boundary conditions. In these expressions, (symbolic) constants are assigned to one or more of the

indices of the variables that occur in the expression. See for example the assignment of `s` in Figure 4. The example shown in Figure 4 is coded in a Fortran-like notation while an internal representation is used by the CTADEL system. From this example it can also be seen that in the final code the array bounds for `t1` and the loop bounds for the assignment to `t1` have to be derived symbolically from the loop bounds for `p`, `q`, `r`, and `s`.

New names for temporaries are chosen such that they do not clash with names of fields. The computational complexity of the implemented eliminator is $\mathcal{O}(n^2)$, where n is the number of basic arithmetic operators in the model.

3.6 Code Generation

The third and final translation phase of the CTADEL system comprises the generation of vectorizable Fortran 77 code. The structure of the code allows restructuring Fortran 77 compilers to apply many optimizations. In addition, reasonably efficient data-parallel Fortran 90 code can be automatically obtained by using a parallelizing compiler.

Prior to the generation of code, the bounds for the rectangular computational domain for each (temporary) variable are derived symbolically to guarantee the assignment of correct values to the variables. In addition, the derivation of the bounds helps the user to manually extend the generated code such that periodic boundary conditions can be enforced: values of fields at the boundaries should be copied in a wrap-around fashion using `do`-loops. In the future, this will be done automatically by the system if the model is formulated in the high level language with directives for periodic boundary conditions.

In the final code, the (partial) sums, products, etc. are implemented using `do`-loops and conditional expressions are implemented using `if-then-else` statements.

4 Results

In this section, preliminary results of the CTADEL code generation for two physical models are presented. The first model consists of the time-dependent Euler equations for inviscid, compressible flow. The second model is described by one of the primitive equations of the HIRLAM weather forecast system [15].

4.1 Time-Dependent Euler Equations

This problem is used to compare different strategies for the global common subexpression eliminator in the generation of code. The time-dependent Euler equations for an inviscid, compressible flow in a two-dimensional geometry can be written in conservation law form in an (x, y) cartesian coordinate system as

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{w})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{w})}{\partial y} = 0, \tag{12}$$

where

$$\mathbf{w} = \begin{bmatrix} \rho \\ \rho\, u \\ \rho\, v \\ \rho\, E \end{bmatrix}, \quad \mathbf{F}(\mathbf{w}) = \begin{bmatrix} \rho\, u \\ \rho\, u^2 \\ \rho\, u\, v \\ u(\rho\, E + p) \end{bmatrix}, \quad \mathbf{G}(\mathbf{w}) = \begin{bmatrix} \rho\, v \\ \rho\, u\, v \\ \rho\, v^2 \\ v(\rho\, E + p) \end{bmatrix},$$

ρ is the mass density, p the pressure, E the total energy, and (u, v) are the (x, y) components of the flow velocity, respectively. The above system of equations is closed via the equation of state

$$p = \rho(\gamma - 1)(E - \frac{1}{2}(u^2 + v^2)), \tag{13}$$

where γ is the ratio of specific heats of the medium.

By application of the product law, $\frac{\partial uv}{\partial t} = u\frac{\partial v}{\partial t} + v\frac{\partial u}{\partial t}$, Equation (12) can be written in explicit form

$$\begin{aligned} \frac{\partial \rho}{\partial t} &= -\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} \\ \mathbf{d}(\mathbf{w}') &= -\rho^{-1}\left(\frac{\partial \mathbf{f}(\mathbf{w}')}{\partial x} + \frac{\partial \mathbf{g}(\mathbf{w}')}{\partial y}\right) - \frac{\partial \rho}{\partial t}\,\mathbf{w}', \end{aligned} \tag{14}$$

where

$$\mathbf{w}' = \begin{bmatrix} u \\ v \\ E \end{bmatrix}, \quad \mathbf{d}(\mathbf{w}') = \begin{bmatrix} \frac{\partial u}{\partial t} \\ \frac{\partial v}{\partial t} \\ \frac{\partial E}{\partial t} \end{bmatrix}, \quad \mathbf{f}(\mathbf{w}') = \begin{bmatrix} \rho\, u^2 + p \\ \rho\, u\, v \\ u\,(\rho E + p) \end{bmatrix}, \quad \mathbf{g}(\mathbf{w}') = \begin{bmatrix} \rho\, u\, v \\ \rho\, v^2 + p \\ v\,(\rho E + p) \end{bmatrix}.$$

Since the vectors $\mathbf{f}(\mathbf{w}')$ and $\mathbf{g}(\mathbf{w}')$ have already subexpressions in common, this problem serves as a good test for the global common subexpression eliminator.

A complete description of the CTADEL automatic code generation for this example can be found in Appendix A.1. In Table 3, different strategies for common subexpression elimination are compared by structural inspection of the codes. The total number of arithmetic operators and the total number of assignments to (temporary) variables are shown. Each operator and assignment is evaluated on the complete two-dimensional domain. In the second strategy, common subexpression elimination is employed after expanding the discrete operators (exact post-matching). In the third strategy, common subexpression elimination is employed on the discretized equations before the discrete operators are expanded (exact pre-matching). In our experience, this is the strategy often exploited by human programmers. The last strategy in the list corresponds with the CTADEL's eliminator. In addition to the previous strategies, this eliminator matches also expressions containing fields with indices having constant offsets, as is described in Section 3.5.

From Table 3 it can be concluded that the implemented strategy for global common subexpression elimination is the best for this example with respect to the number of arithmetic operations. Furthermore, the strategy requires a relatively small number of assignments.

c.s.e. strategy	#unary mins	#add and sub	#mul and div	total #op's	#assign
no c.s.e.	3	49	39	91	5
c.s.e. with exact post-matching	3	43	34	80	17
c.s.e. with exact pre-matching	3	43	29	75	31
c.s.e. with index-offset matching	3	38	22	63	20

Table 3: A static comparison of the generated codes for the Euler equations with different strategies for global common subexpression elimination (c.s.e.).

4.2 HIRLAM Primitive Equations

This problem is used to compare the performance of the hand-written HIRLAM reference code and the code generated by the CTADEL system on various types of hardware platforms.

The HIRLAM system is a production code used at several European meteorological institutes to produce weather forecasts up to 36 hours. The CTADEL code generator is able to generate efficient code for the HIRLAM model equations of the dynamic tendencies, the so-called Primitive Equations [12]. In this report, we will restrict ourselves to present results of the code generation for only one of these primitive equations, namely the surface pressure tendency.

In the primitive equations of the HIRLAM dynamics, the equation of the surface pressure tendency in vertical η coordinate is written as

$$\frac{\partial p_s}{\partial t} = -\int_0^1 \nabla \cdot \left(\mathbf{w} \frac{\partial p}{\partial \eta} \right) d\eta, \tag{15}$$

where $p = A + B\,p_s$ is the pressure formulated in the hybrid vertical coordinate η, p_s is the surface pressure, $\mathbf{w} = [u, v]^T$ is the horizontal wind vector, $\nabla \cdot \left(\mathbf{w} \frac{\partial p}{\partial \eta} \right)$ denotes the divergence of the wind. The divergence in spherical coordinates is defined as

$$\nabla \cdot [F, G]^T = \frac{1}{a\,h_x\,h_y} \left(\frac{\partial}{\partial x}(F\,h_y) + \frac{\partial}{\partial y}(G\,h_x) \right).$$

Here, a is the radius of the earth and h_x and h_y are metric coefficients. The HIRLAM primitive equations are discretized using central differences on an Arakawa C-grid. The discretization of Equation (15) by hand, see [15], using midpoint quadrature with a vertical z coordinate, results in

$$\frac{\partial p_s}{\partial t} = -\frac{1}{a\,h_x\,h_y} \left(\delta_x(h_y \sum_{j=0}^{\mathrm{nlev}-1} (u_j\, \overline{\Delta_z p}^x)) + \delta_y(h_x \sum_{j=0}^{\mathrm{nlev}-1} (v_j\, \overline{\Delta_z p}^y)) \right),$$

where $\Delta_z p = p_{k+1} - p_k$.

The code generation process for Equation (15) by the CTADEL system is discussed in Appendix A.2. In this appendix it is shown that the CTADEL system produces exactly the same discretization. Table 4 shows the performance of the hand-written Fortran 77 and CTADEL generated Fortran 77 code for the numerical solution of Equation (15) in a $30 \times 30 \times 16$ grid. The performance of the Fortran 77 codes was measured on a SGI Indy (`f77 -O3 -ddopt`), an HP 9000/720 system (`f77 +O3`), a Convex C4 system (`fc -O2`, one CPU), and a CRAY C90 system (`cf77 -Ovector3`, one CPU). The performance of the data-parallel Fortran 90

	#floating point oper.			total elapsed time (ms)					
	add & sub	mul	total	SGI Indy	HP 720	MasPar MP-1	CRAY T3D	Convex C4	CRAY C90
hand-written code	75,750	52,290	128,040	10.5	9.1	3.17	1.12	1.44	0.39
generated code	86,430	62,175	148,605	9.7	12.4	3.48	1.38	0.57	0.44
gen., k-loops fused	86,430	62,175	148,605	8.7	7.5	3.17	1.36	0.58	0.45

Table 4: Performance of the hand-written and generated codes for the HIRLAM surface pressure tendency with a $30 \times 30 \times 16$ grid.

codes was measured on a MasPar MP-1 system (`mpfortran -Omax`, 1024 PEs, SIMD architecture) [17] and on a CRAY T3D (`cf77 -Oscalar3 -X64`, 64PEs, MIMD architecture) [5]. The Fortran 90 codes were obtained with the MasPar Vast-2 compiler [18] which translates Fortran 77 to Fortran 90 code.

We will briefly discuss the preliminary performance results shown in Table 4. A detailed performance analysis of the results falls outside the scope of this report. From Table 4 it can be concluded that for most platforms the generated code with fused `k`-loops is more efficient than the hand-written code, which is especially true for the Fortran 77 codes executed on the workstations. The hand-written code is more efficient for the CRAY C90 architecture. The reason is that this code was optimized for CRAY C90 architectures. The CRAY T3D Fortran 90 compiler is still immature. Therefore, small differences between the codes can have significant effects on performance. The performance of the codes on the Convex C4 and CRAY C90 demontrate that the same code can differ significantly in efficiency on different vector architectures. Here again, subtle differences between the codes can have significant effects on performance. Loop fusion was performed by hand, a simple optimization which most Fortran compilers did not perform. Therefore, the next future extension of CTADEL will consist of an additional phase for loop fusion.

5 Conclusions and Further Work

In this report, we have presented the CTADEL system, a prototype translation system for the automatic generation of efficient Fortran 77 code from a very high level language description of a scientific model. The system employs finite difference methods and staggered grids to obtain a numerical solution. Preliminary results show that the generation of efficient code is well feasible within the presented approach. The efficiency of the generated code by the system is a result of incorporating algebraic simplification and extensive global common subexpression elimination.

By developing a prototype system, we were able to share preliminary experiences of (potential) users. Firstly, these experiences where invaluable with respect to the design of the very high level language for the description of models. The high level language provides a means for the specification of a problem in a natural way; its power of expressiveness is close to the declarative mathematical formulation of the model. The language also provides an abstraction level that hides the discretization and implementation of a model. Secondly, the knowledge of experts on the discretization of models is incorporated into the system. The automatic discretization of continuous model equations by the system is as close as possible to the discretization by hand. Ideally, the discretization should be the same. However, in general, this is not possible for every case due to the fact that complex domain knowledge, e.g. conservation of energy, cannot be taken into account. However, the current default discretization performed by the CTADEL system is deterministic, efficient, and easy to verify by the user. In addition, the system allows the user to insert appropriate discretization operators into the model equations in order to override the insertion of operators by the system for the default discretization. Finally, the efficiency of the code generated by the system for some example models was checked against existing hand-written code. The results are most encouraging and showed that for most hardware platforms the efficiency of the generated code was at least equal to the hand-written code.

In a future implementation of the CTADEL system, new rules may be needed that assist

the simplifier in simplifying expressions involving new operators added to the system by the user. However, many problems may occur if the user is allowed to extend the rule base by his own new set of rules. In the worst case, inconsistencies and non-termination of the simplifier may occur. To avoid problems and still allow new rules to be added to the system, user defined operators may be qualified with keywords `linear` and `dependent(x)` to automatically generate and add rules for a new linear, x-dependent operator.

We believe that the techniques implemented in the global common subexpression eliminator are quite powerful and adequate for most situations. However, other techniques may be employed as well in a future implementation. For example, the current eliminator will not recognize that $a^2\,b$ and $a\,b\,c$ have the expression $a\,b$ in common. Techniques that can handle such expressions are developed for the SCOPE package of REDUCE [14]. Although the capabilities of SCOPE are impressive, it optimizes only inline code. Furthermore, the SCOPE eliminator does not recognize common subexpressions containing arrays with indices that differ by a constant offset.

Although arithmetic complexity of the code to solve a model remains fundamental and should be as low as possible by employing the techniques presented in this report, extra computations may not incur the penalty that they would have on parallel and vector architectures [19]. Therefore, future research will be aimed at extending the system with algorithms for alignment of grids and data on a processor mesh, data replication, domain decomposition, and message coalescing, in order to generate various types of efficient parallel codes using the data-parallel and message-passing parallel programming paradigms. Furthermore, for vector architectures the two- and three- dimensional generated loops can be collapsed to one-dimensional loops to obtain efficiently vectorizable code while keeping numerical results the same.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading MA, 1985.

[2] W.F. Ames, *Numerical Methods for Partial Differential Equations, second edition*, Academic Press, New York, 1977.

[3] A. Arakawa and V.R. Lamb, *A Potential Enstrophy and Energy Conserving Scheme for the Shallow Water Equations*, Monthly Weather Review **109**, pp. 18–36, 1981.

[4] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt, *MAPLE Reference Manual*, Waterloo, 1988.

[5] Cray Research Inc., *CRAY T3D System Architecture Overview*, March 1993. Available by anonymous ftp from `ftp.cray.com`.

[6] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua, *A MATLAB Compiler and Restructurer for the Development of Scientific Libraries and Applications*, in proceedings of the 8$^{\text{th}}$ International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, New York, 1995 (to appear).

[7] R. van Engelen and L. Wolters, *A Comparison of Parallel Programming Paradigms and Data Distributions for a Limited Area Numerical Weather Forecast Routine*, in

proceedings of the 9th ACM International Conference on Supercomputing, pp. 357–364, ACM Press, New York, 1995.

[8] R.S. Francis, I.D. Mathieson, P.G. Whiting, M.R. Dix, H.L. Davies, and L.D. Rotstyan, *A Data Parallel Scientific Modelling Language*, Journal of Parallel and Distributed Computing **21**, pp. 46–60, 1994.

[9] E. Gallopoulos, E. Houstis, and J.R. Rice, *Future Research Directions in Problem Solving Environments for Computational Science: Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science*, Tech. Report 1259, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1992.

[10] E. Gallopoulos, E. Houstis, and J.R. Rice, *Problem-Solving Environments for Computational Science*, IEEE Computational Science & Engineering **1**, pp. 11–23, Summer 1994.

[11] N. Gustafsson and D. Salmond, *A Parallel Spectral HIRLAM with the Data Parallel Programming Model and with Message Passing – A Comparison*, in G.-R. Hoffmann and N. Kreitz (eds.), *Coming of Age*, proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, pp. 32–48, World Scientific Publ., Singapore, 1995.

[12] G.J. Haltiner and R.T. Williams, *Numerical Prediction and Dynamic Meteorology, second edition*, John Wiley & Sons, New York, 1980.

[13] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C. Karathanasis, P.N. Papachiou, M.K. Samartzis, E.A. Vavalis, Ko-Yang Wang, and S. Weerawarana, *//ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines*, in proceedings of the 4th ACM International Conference on Supercomputing, pp. 96–107, ACM Press, New York, 1990.

[14] J.A. van Hulzen, B.J.A. Hulshof, B. L. Gates, and M.C. van Heerwaarden, *A Code Optimization package for REDUCE*, in proceedings ISSAC '89, pp. 163–170, ACM Press, New York, 1989.

[15] P. Kållberg (editor), *Documentation Manual of the Hirlam Level 1 Analysis-Forecast System*, June 1990. Available from SMHI, S–60176 Norrköping, Sweden.

[16] T. Kauranne, J. Oinonen, S. Saarinen, O. Serimaa, and J. Hietaniemi, *The Operational HIRLAM 2 Model on Parallel Computers*, in G.-R. Hoffmann and N. Kreitz (eds.), *Coming of Age*, proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, pp. 63–74, World Scientific Publ., Singapore, 1995.

[17] MasPar Computer Corporation, *MasPar MP-1 Hardware Manuals*, Sunnyvale, CA, July 1992.

[18] MasPar Computer Corporation, *MasPar Vast-2 User Guide*, Sunnyvale, CA, July 1992.

[19] J.M. Ortega and R.G. Voigt, *Solution of Partial Differential Equations on Vector and Parallel Computers*, SIAM Review **27**, no. 2, pp. 149–240, June 1985.

[20] J.R. Rice and R.F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1985.

[21] A. Simmons and D.M. Burridge, *An Energy and Angular-Momentum Conserving Vertical Finite-Difference Scheme and Hybrid Vertical Coordinates*, Monthly Weather Review **109**, pp. 758–766, 1981.

[22] Y. Umetani, M. Tsuji, K. Iwasawa, and H. Hirayama, *DEQSOL: A Numerical Simulation Language for Vector/Parallel Processors*, in B. Ford and F. Chatelin (eds.), *Problem Solving Environments for Scientific Computing, Proceedings*, North-Holland, Amsterdam, 1987.

[23] J. Wielemaker, *SWI-Prolog Reference Manual*, University of Amsterdam, 1995. Available by anonymous ftp from `swi.psy.uva.nl`.

[24] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Boston, 1988.

[25] L. Wolters, G. Cats, and N. Gustafsson, *Data-Parallel Numerical Weather Forecasting*, Scientific Programming **4**, pp. 141–153, 1995.

A CTADEL Code Generation for the Examples

The translation of the examples in Section 4, time-dependent Euler equations and HIRLAM surface pressure tendency, will be described in A.1 and A.2, respectively.

A.1 Code Generation for the Time-Dependent Euler Equations

The following model description describes the time-dependent Euler Equations (14). An E-grid is chosen for the finite difference discretization.

```
% Euler equations for inviscid, compressible flow in two-dimensional geometry
grid i=1:n, j=1:m.                % two-dimensional cartesian grid
def dx0 = 1.                      % ...with unit gridpoint distances
def dx1 = 1.
def dy0 = 1.
def dy1 = 1.
def gamma = 1.4.                  % ratio of specific heats
field rho(x,y): stag(0,0).        % mass density of the fluid
field u(x,y):   stag(0,0).        % fluid flow velocity in x-direction
field v(x,y):   stag(0,0).        % fluid flow velocity in y-direction
field e(x,y):   stag(0,0).        % energy
latex e = ["E"].
field p(x,y): stag(0,0).
p = rho*(gamma-1)*(e-(u^2+v^2)/2). % pressure equation
field drho(x,y): stag(1,1).       % tendency of mass density
field du(x,y):   stag(1,1).       % tendency of velocity in x-direction
field dv(x,y):   stag(1,1).       % tendency of velocity in y-direction
field de(x,y):   stag(1,1).       % tendency of energy
latex drho = ["\frac{\partial\rho}{\partial t}"].
latex du   = ["\frac{\partial u}{\partial t}"].
latex dv   = ["\frac{\partial v}{\partial t}"].
latex de   = ["\frac{\partial E}{\partial t}"].
```

```
vector w  = [ u,
              v,
              e ].
vector dw = [ du,
              dv,
              de ].
vector fw = [ rho*u^2+p,
              rho*u*v,
              u*(rho*e+p) ].
vector gw = [ rho*u*v,
              rho*v^2+p,
              v*(rho*e+p) ].
drho = -dx(rho*u)-dy(rho*v).       % tendency equations
dw   = -rho^(-1)*(dx fw + dy gw) - drho*w.
```

The discretization by CTADEL results in the following system of discretized equations

$$
\begin{aligned}
p_{\mathrm{i,j}} &= (\gamma - 1)\,\left(E_{\mathrm{i,j}} - 0.5\,\left({u_{\mathrm{i,j}}}^2 + {v_{\mathrm{i,j}}}^2\right)\right)\,\rho_{\mathrm{i,j}} \\
\frac{\partial \rho}{\partial t}_{\mathrm{i,j}} &= -\overline{\delta_y^+\,(\rho_{\mathrm{i,j}}\,v_{\mathrm{i,j}})}^{x_+} - \overline{\delta_x^+\,(\rho_{\mathrm{i,j}}\,u_{\mathrm{i,j}})}^{y_+} \\
\frac{\partial u}{\partial t}_{\mathrm{i,j}} &= -\frac{\overline{\delta_y^+\,(\rho_{\mathrm{i,j}}\,u_{\mathrm{i,j}}\,v_{\mathrm{i,j}})}^{x_+} + \overline{\delta_x^+\,({u_{\mathrm{i,j}}}^2\,\rho_{\mathrm{i,j}} + p_{\mathrm{i,j}})}^{y_+}}{\overline{\overline{\rho_{\mathrm{i,j}}}^{y_+}}^{x_+}} - \frac{\partial \rho}{\partial t}_{\mathrm{i,j}}\,\overline{\overline{u_{\mathrm{i,j}}}^{y_+}}^{x_+} \\
\frac{\partial v}{\partial t}_{\mathrm{i,j}} &= -\frac{\overline{\delta_y^+\,({v_{\mathrm{i,j}}}^2\,\rho_{\mathrm{i,j}} + p_{\mathrm{i,j}})}^{x_+} + \overline{\delta_x^+\,(\rho_{\mathrm{i,j}}\,u_{\mathrm{i,j}}\,v_{\mathrm{i,j}})}^{y_+}}{\overline{\overline{\rho_{\mathrm{i,j}}}^{y_+}}^{x_+}} - \frac{\partial \rho}{\partial t}_{\mathrm{i,j}}\,\overline{\overline{v_{\mathrm{i,j}}}^{y_+}}^{x_+} \\
\frac{\partial E}{\partial t}_{\mathrm{i,j}} &= -\frac{\overline{\delta_y^+\,((E_{\mathrm{i,j}}\,\rho_{\mathrm{i,j}} + p_{\mathrm{i,j}})\,v_{\mathrm{i,j}})}^{x_+} + \overline{\delta_x^+\,((E_{\mathrm{i,j}}\,\rho_{\mathrm{i,j}} + p_{\mathrm{i,j}})\,u_{\mathrm{i,j}})}^{y_+}}{\overline{\overline{\rho_{\mathrm{i,j}}}^{y_+}}^{x_+}} - \frac{\partial \rho}{\partial t}_{\mathrm{i,j}}\,\overline{\overline{E_{\mathrm{i,j}}}^{y_+}}^{x_+}
\end{aligned}
$$

Then, CTADEL expands the stagger and finite difference operators, simplifies the result and finally eliminates common subexpressions. The result is the following set of assignments

$$
\begin{aligned}
p_{\mathrm{i,j}} &= 0.4\,\left(E_{\mathrm{i,j}} - 0.5\,\left({u_{\mathrm{i,j}}}^2 + {v_{\mathrm{i,j}}}^2\right)\right)\,\rho_{\mathrm{i,j}} \\
\frac{\partial \rho}{\partial t}_{\mathrm{i,j}} &= -0.5\,(\mathtt{t8}_{\mathrm{i,j}} + \mathtt{t10}_{\mathrm{i,j}} - \mathtt{t8}_{\mathrm{i+1,j-1}} - \mathtt{t10}_{\mathrm{i-1,j-1}}) \\
\frac{\partial u}{\partial t}_{\mathrm{i,j}} &= -\left(\mathtt{t22}_{\mathrm{i,j}}\,(u_{\mathrm{i,j}} + u_{\mathrm{i,j+1}} + u_{\mathrm{i+1,j}} + u_{\mathrm{i+1,j+1}})\right) \\
&\quad -\mathtt{t29}_{\mathrm{i,j}}\,(\mathtt{t27}_{\mathrm{i,j}} + \mathtt{t26}_{\mathrm{i,j}} + \mathtt{t15}_{\mathrm{i,j}} + \mathtt{t16}_{\mathrm{i,j}} - \mathtt{t15}_{\mathrm{i+1,j-1}} - \mathtt{t16}_{\mathrm{i-1,j-1}}) \\
\frac{\partial v}{\partial t}_{\mathrm{i,j}} &= -\left(\mathtt{t22}_{\mathrm{i,j}}\,(v_{\mathrm{i,j}} + v_{\mathrm{i,j+1}} + v_{\mathrm{i+1,j}} + v_{\mathrm{i+1,j+1}})\right) \\
&\quad -\mathtt{t29}_{\mathrm{i,j}}\,(\mathtt{t26}_{\mathrm{i,j}} + \mathtt{t24}_{\mathrm{i,j}} + \mathtt{t25}_{\mathrm{i,j}} - \mathtt{t27}_{\mathrm{i,j}} - \mathtt{t24}_{\mathrm{i+1,j-1}} - \mathtt{t25}_{\mathrm{i-1,j-1}}) \\
\frac{\partial E}{\partial t}_{\mathrm{i,j}} &= -\left(\mathtt{t22}_{\mathrm{i,j}}\,(E_{\mathrm{i,j}} + E_{\mathrm{i,j+1}} + E_{\mathrm{i+1,j}} + E_{\mathrm{i+1,j+1}})\right) \\
&\quad -\mathtt{t29}_{\mathrm{i,j}}\,(\mathtt{t36}_{\mathrm{i,j}} + \mathtt{t37}_{\mathrm{i,j}} - \mathtt{t36}_{\mathrm{i+1,j-1}} - \mathtt{t37}_{\mathrm{i-1,j-1}}) \\
\mathtt{t7}_{\mathrm{i,j}} &= v_{\mathrm{i,j+1}} - u_{\mathrm{i,j+1}} \\
\mathtt{t8}_{\mathrm{i,j}} &= \mathtt{t7}_{\mathrm{i,j}}\,\rho_{\mathrm{i,j+1}} \\
\mathtt{t9}_{\mathrm{i,j}} &= u_{\mathrm{i+1,j+1}} + v_{\mathrm{i+1,j+1}} \\
\mathtt{t10}_{\mathrm{i,j}} &= \mathtt{t9}_{\mathrm{i,j}}\,\rho_{\mathrm{i+1,j+1}} \\
\mathtt{t15}_{\mathrm{i,j}} &= \mathtt{t8}_{\mathrm{i,j}}\,u_{\mathrm{i,j+1}}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{t16}_{\mathrm{i,j}} &= \mathtt{t10}_{\mathrm{i,j}}\, u_{\mathrm{i+1,j+1}} \\
\mathtt{t22}_{\mathrm{i,j}} &= 0.25\, \frac{\partial \rho}{\partial t}_{\mathrm{i,j}} \\
\mathtt{t24}_{\mathrm{i,j}} &= \mathtt{t8}_{\mathrm{i,j}}\, v_{\mathrm{i,j+1}} \\
\mathtt{t25}_{\mathrm{i,j}} &= \mathtt{t10}_{\mathrm{i,j}}\, v_{\mathrm{i+1,j+1}} \\
\mathtt{t26}_{\mathrm{i,j}} &= p_{\mathrm{i+1,j+1}} - p_{\mathrm{i,j}} \\
\mathtt{t27}_{\mathrm{i,j}} &= p_{\mathrm{i+1,j}} - p_{\mathrm{i,j+1}} \\
\mathtt{t29}_{\mathrm{i,j}} &= \frac{2}{\rho_{\mathrm{i,j}} + \rho_{\mathrm{i,j+1}} + \rho_{\mathrm{i+1,j}} + \rho_{\mathrm{i+1,j+1}}} \\
\mathtt{t35}_{\mathrm{i,j}} &= E_{\mathrm{i,j+1}}\, \rho_{\mathrm{i,j+1}} + p_{\mathrm{i,j+1}} \\
\mathtt{t36}_{\mathrm{i,j}} &= \mathtt{t35}_{\mathrm{i,j}}\, \mathtt{t7}_{\mathrm{i,j}} \\
\mathtt{t37}_{\mathrm{i,j}} &= \mathtt{t35}_{\mathrm{i+1,j}}\, \mathtt{t9}_{\mathrm{i,j}}
\end{aligned}
$$

The t's are temporaries introduced by the common subexpression eliminator. The irregular numbering of these temporaries is a consequence of the internal representation of the model by the system. The code generated by the system is

```
subroutine tstep(n, m, e, rho, u, v, p, drho, du, dv, de)
integer n, m
real e(1 : n + 1, 1 : m + 1)
real rho(1 : n + 1, 1 : m + 1)
real u(1 : n + 1, 1 : m + 1)
real v(1 : n + 1, 1 : m + 1)
real p(1 : n + 1, 1 : m + 1)
real drho(1 : n, 1 : m)
real du(1 : n, 1 : m)
real dv(1 : n, 1 : m)
real de(1 : n, 1 : m)
real t7(1 : n + 1, 0 : m)
real t8(1 : n + 1, 0 : m)
real t9(0 : n, 0 : m)
real t10(0 : n, 0 : m)
real t15(1 : n + 1, 0 : m)
real t16(0 : n, 0 : m)
real t22(1 : n, 1 : m)
real t24(1 : n + 1, 0 : m)
real t25(0 : n, 0 : m)
real t26(1 : n, 1 : m)
real t27(1 : n, 1 : m)
real t29(1 : n, 1 : m)
real t35(1 : n + 1, 0 : m)
real t36(1 : n + 1, 0 : m)
real t37(0 : n, 0 : m)
do i=1,n
do j=1,m
t29(i, j) = 2 / (rho(i, j) + rho(i, j + 1) + rho(i + 1, j) +
. rho(i + 1, j + 1))
enddo
enddo
do i=1,n + 1
do j=1,m + 1
p(i, j) = 0.4 * (e(i, j) - 0.5 * (u(i, j) ** 2 + v(i, j) ** 2)) *
. rho(i, j)
```

```
enddo
enddo
do i=1,n + 1
do j=0,m
t35(i, j) = e(i, j + 1) * rho(i, j + 1) + p(i, j + 1)
enddo
enddo
do i=1,n + 1
do j=0,m
t7(i, j) = v(i, j + 1) - u(i, j + 1)
enddo
enddo
do i=1,n + 1
do j=0,m
t8(i, j) = t7(i, j) * rho(i, j + 1)
enddo
enddo
do i=0,n
do j=0,m
t9(i, j) = u(i + 1, j + 1) + v(i + 1, j + 1)
enddo
enddo
do i=0,n
do j=0,m
t10(i, j) = t9(i, j) * rho(i + 1, j + 1)
enddo
enddo
do i=1,n
do j=1,m
drho(i, j) = - 0.5 * (t8(i, j) + t10(i, j) - t8(i + 1, j - 1) -
. t10(i - 1, j - 1))
enddo
enddo
do i=1,n + 1
do j=0,m
t15(i, j) = t8(i, j) * u(i, j + 1)
enddo
enddo
do i=0,n
do j=0,m
t16(i, j) = t10(i, j) * u(i + 1, j + 1)
enddo
enddo
do i=1,n
do j=1,m
t22(i, j) = 0.25 * drho(i, j)
enddo
enddo
do i=1,n + 1
do j=0,m
t24(i, j) = t8(i, j) * v(i, j + 1)
enddo
enddo
do i=0,n
do j=0,m
t25(i, j) = t10(i, j) * v(i + 1, j + 1)
enddo
```

```
enddo
do i=1,n
do j=1,m
t26(i, j) = p(i + 1, j + 1) - p(i, j)
enddo
enddo
do i=1,n
do j=1,m
t27(i, j) = p(i + 1, j) - p(i, j + 1)
enddo
enddo
do i=1,n
do j=1,m
dv(i, j) = - t22(i, j) * (v(i, j) + v(i, j + 1) + v(i + 1, j) + v(
. i + 1, j + 1)) - t29(i, j) * (t26(i, j) + t24(i, j) + t25(i, j) -
. t27(i, j) - t24(i + 1, j - 1) - t25(i - 1, j - 1))
enddo
enddo
do i=1,n
do j=1,m
du(i, j) = - t22(i, j) * (u(i, j) + u(i, j + 1) + u(i + 1, j) + u(
. i + 1, j + 1)) - t29(i, j) * (t27(i, j) + t26(i, j) + t15(i, j) +
. t16(i, j) - t15(i + 1, j - 1) - t16(i - 1, j - 1))
enddo
enddo
do i=1,n + 1
do j=0,m
t36(i, j) = t35(i, j) * t7(i, j)
enddo
enddo
do i=0,n
do j=0,m
t37(i, j) = t35(i + 1, j) * t9(i, j)
enddo
enddo
do i=1,n
do j=1,m
de(i, j) = - t22(i, j) * (e(i, j) + e(i, j + 1) + e(i + 1, j) + e(
. i + 1, j + 1)) - t29(i, j) * (t36(i, j) + t37(i, j) - t36(i + 1,
. j - 1) - t37(i - 1, j - 1))
enddo
enddo
end
```

Note that in the final code no attempt has been made to reuse temporary variables. This issue will be resolved in a future implementation of the CTADEL system.

A.2 Code Generation for the HIRLAM Surface Pressure Tendency

The surface pressure tendency of the HIRLAM model, Equation (15), is described by the following model description

```
% HIRLAM dynamics surface pressure tendency
grid i=1:nlon, j=1:nlat, k=0:nlev. % three-dimensional grid
def dx0 = 1/rdlam.                 % gridpoint distance in x-direction
def dx1 = 1/rdlam.                 % ditto for half grid
def dy0 = 1/rdth.                  % gridpoint distance in y-direction
```

```
def dy1 = 1/rdth.                    % ditto for half grid
def dz0 = 1.                         % unit gridpoint distance in z-direction
def dz1 = 1.                         % ...for formulation in hybrid vertcal coordinates
def ra = 1/6.371e6.                  % 1/earth radius
field ps(x,y): stag(0,0).            % surface pressure
field u(x,y,z): stag(1,0,1).         % wind velocity in x-direction
field v(x,y,z): stag(0,1,1).         % wind velocity in y-direction
vector w = [u,v].                    % wind vector
field a(z): stag(0).                 % weight functions between vertical levels
field b(z): stag(0).
field hxv(x,y): stag(0,1).           % metric coefficient along x-coordinate
field hyu(x,y): stag(1,0).           % metric coefficient along y-coordinate
field rhxu(x,y): stag(0,0).          % 1/metric coefficient along x-coordinate
field rhyv(x,y): stag(0,0).          % 1/metric coefficient along y-coordinate
field dps(x,y): stag(0,0).
prefix div.                          % divergence in spherical coordinates
div [U,V] := 1/ahxhy*(dx(U*hyu)+dy(V*hxv)).
ahxhy := 1/(ra*rhxu*rhyv).           % utility function
p := a + b*ps.                       % pressure as a function of ps and hybrid coordinates
latex ps = ["{p_s}"].                % LaTeX descriptions for report generation
latex a = ["A"].
latex b = ["B"].
latex dz U  = ["\Delta_z",U].
latex dz0 U = ["\Delta_z^-",U].
latex dz1 U = ["\Delta_z^+",U].
latex div X = ["\nabla\cdot",X].
latex dps = ["\frac{\partial p_s}{\partial t}"].
dps = -qz(0, nlev, div(w*dz p))).  % equation for surface pressure tendency
```

The discretization by CTADEL results in

$$\begin{aligned}\frac{\partial p_s}{\partial t}_{\mathrm{i,j}} &= -\mathrm{ra}\left(\delta_x^-\left(hyu_{\mathrm{i,j}}\,\mathrm{midp}_z^-\left(0,\mathrm{nlev},\left(\Delta_z^+\left(B_{\mathrm{k}}\,\overline{p_s}_{\mathrm{i,j}}^{x+}+A_{\mathrm{k}}\right)\,u_{\mathrm{i,j,k}}\right)\right)\right)\right.\\ &\quad\left.+\delta_y^-\left(hxv_{\mathrm{i,j}}\,\mathrm{midp}_z^-\left(0,\mathrm{nlev},\left(\Delta_z^+\left(B_{\mathrm{k}}\,\overline{p_s}_{\mathrm{i,j}}^{y+}+A_{\mathrm{k}}\right)\,v_{\mathrm{i,j,k}}\right)\right)\right)\right)\,rhxu_{\mathrm{i,j}}\,rhyv_{\mathrm{i,j}}\end{aligned}$$

with midpoint quadrature $\mathrm{midp}_z^-(0,\mathrm{nlev},u)=\sum_{k=0}^{\mathrm{nlev}-1}u_k$. After eliminating common subexpressions, the system produces the following set of assignments which is quite similar to the hand-written form

$$\begin{aligned}\frac{\partial p_s}{\partial t}_{\mathrm{i,j}} &= -1.5696\mathrm{E}-7\;(rdlam\;(\mathtt{t7}_{\mathrm{i,j}}-\mathtt{t7}_{\mathrm{i-1,j}})+rdth\;(\mathtt{t17}_{\mathrm{i,j}}-\mathtt{t17}_{\mathrm{i,j-1}}))\,rhxu_{\mathrm{i,j}}\,rhyv_{\mathrm{i,j}}\\ \mathtt{t7}_{\mathrm{i,j}} &= hyu_{\mathrm{i,j}}\,\mathtt{t6}_{\mathrm{i,j}}\\ \mathtt{t17}_{\mathrm{i,j}} &= hxv_{\mathrm{i,j}}\,\mathtt{t16}_{\mathrm{i,j}}\\ \mathtt{t6}_{\mathrm{i,j}} &= \sum_{\mathrm{k}=0}^{(\mathrm{nlev}-1)}\left(\left(\mathtt{t13}_{\mathrm{k}}+\mathtt{t11}_{\mathrm{k}}\left(p_{s\,\mathrm{i,j}}+p_{s\,\mathrm{i+1,j}}\right)\right)\,u_{\mathrm{i,j,k}}\right)\\ \mathtt{t16}_{\mathrm{i,j}} &= \sum_{\mathrm{k}=0}^{(\mathrm{nlev}-1)}\left(\left(\mathtt{t13}_{\mathrm{k}}+\mathtt{t11}_{\mathrm{k}}\left(p_{s\,\mathrm{i,j}}+p_{s\,\mathrm{i,j+1}}\right)\right)\,v_{\mathrm{i,j,k}}\right)\\ \mathtt{t11}_{\mathrm{k}} &= 0.5\;(B_{\mathrm{k+1}}-B_{\mathrm{k}})\\ \mathtt{t13}_{\mathrm{k}} &= A_{\mathrm{k+1}}-A_{\mathrm{k}}\end{aligned}$$

The code generated by the system is

```
subroutine tstep(nlon, nlat, nlev, rdlam, rdth, a, b, ps, u, v,
                 hxv, hyu, rhxu, rhyv, dps)
integer nlon, nlat, nlev
real rdlam
real rdth
real a(0 : nlev)
real b(0 : nlev)
real ps(0 : nlon + 1, 0 : nlat + 1)
real u(0 : nlon, 1 : nlat, 0 : nlev - 1)
real v(1 : nlon, 0 : nlat, 0 : nlev - 1)
real hxv(1 : nlon, 0 : nlat)
real hyu(0 : nlon, 1 : nlat)
real rhxu(1 : nlon, 1 : nlat)
real rhyv(1 : nlon, 1 : nlat)
real dps(1 : nlon, 1 : nlat)
real t6(0 : nlon, 1 : nlat)
real t7(0 : nlon, 1 : nlat)
real t11(0 : nlev - 1)
real t13(0 : nlev - 1)
real t16(1 : nlon, 0 : nlat)
real t17(1 : nlon, 0 : nlat)
do k=0,nlev - 1
t11(k) = 0.5 * (b(k + 1) - b(k))
enddo
do k=0,nlev - 1
t13(k) = a(k + 1) - a(k)
enddo
do i=1,nlon
do j=0,nlat
t16(i, j) = 0.0
enddo
enddo
do k=0,nlev - 1
do i=1,nlon
do j=0,nlat
t16(i, j) = t16(i, j) + (t13(k) + t11(k) * (ps(i, j)
. + ps(i, j + 1))) * v(i, j, k)
enddo
enddo
enddo
do i=0,nlon
do j=1,nlat
t6(i, j) = 0.0
enddo
enddo
do k=0,nlev - 1
do i=0,nlon
do j=1,nlat
t6(i, j) = t6(i, j) + (t13(k) + t11(k) * (ps(i, j)
. + ps(i + 1, j))) * u(i, j, k)
enddo
enddo
enddo
do i=0,nlon
do j=1,nlat
t7(i, j) = hyu(i, j) * t6(i, j)
enddo
```

```
 enddo
 do i=1,nlon
 do j=0,nlat
 t17(i, j) = hxv(i, j) * t16(i, j)
 enddo
 enddo
 do i=1,nlon
 do j=1,nlat
 dps(i, j) = - 156961.230576 * (rdlam * (t7(i, j) - t7(i - 1, j))
. + rdth * (t17(i, j) - t17(i, j - 1))) * rhxu(i, j) * rhyv(i, j)
 enddo
 enddo
 end
```