Practical Transformation of Functional Programs for Efficient Execution: A Case Study*

James M. Boyle[1] *and Terence J. Harmer*[2]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA

[2] Department of Computer Science, The Queen's University of Belfast, Belfast, BT7 1NN, Northern Ireland

Abstract

Functional programming languages have, traditionally, been thought ill-suited to the specification of numerical mathematical algorithms. Conventional wisdom is that implementations of functional languages cannot provide acceptable execution performance for numerical mathematical algorithms, which are usually computationally intensive. We show how program transformation can be used to derive highly efficient implementations of a numerical mathematical algorithm specified by a functional program.

The example we discuss is a specification for a quasi-linear hyperbolic partial differential equation solver. We develop an initial specification for the one-dimensional version of this problem and show how functional programming concepts facilitate the evolution of the specification into a general, dimension-independent specification for the problem. We also discuss some of the issues that arise in transforming such specifications into programs that are efficient enough to outperform handwritten code.

1 Introduction

In this paper, we discuss an example of the use of automated program transformation to derive efficient programs from functional specifications for numerical problems. At first thought, functional programming seems not at all appropriate for numerical problems. Long-established tradition dictates expressing such algorithms in procedural languages—Fortran or perhaps C. In fact, the earliest procedural computer languages, Fortran and Algol 60, were specifically designed for expressing numeric computations.

A stronger objection to the use of functional languages for numerical computations is that they appear to be ill-suited to specifying computations that operate on data represented by arrays, a data representation that the vast majority of numerical computations use. Programs written in functional languages appear to call for copying an entire array each time the value of one of its elements changes. Such a requirement would make functional programs prohibitively expensive for most numerical computations.

If one can set aside these preconceptions, however, one can see that functional languages do have advantages for numerical computations. The general advantages of func-

* This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

tional programming—naturalness of expression (in some problem areas), ease of proof, and clarity—are well known [1, 12, 15, 9, 4]. When one actually applies functional programming to numerical computations, one finds that the "essence" of many numerical computations is more naturally expressed using recursion and *map* and *reduce* operations than by using the iteration constructs provided by procedural languages. An obvious match to recursive expression is those numerical computations expressed mathematically in terms of recursive equations. However, even computations not usually thought of as recursive—for example, LU factorization in linear algebra—turn out to be elegant in recursive form.

The simplicity and clarity of functional programming can be further enhanced by using abstract data types to specify a numerical computation. Abstract data types can be defined for the concepts and notations used in the problem domain. Such an approach to specification enhances communication between the algorithm developer and the specifier by suppressing implementation details. Communication between the two (even if they be one and the same person!) can be at the same level and in the same or similar terms, both because functional programs are declarative and because they suppress hardware-related details. Indeed, the specifier of the functional program can be, and perhaps should be, the designer of the problem solution. A programming methodology that permits the developer of an algorithm—who understands the details of the algorithm but not of programming—to specify a high-level, machine-independent representation of the algorithm is highly desirable. (As we shall discuss, the methodology also makes tuning an implementation to a particular machine largely problem-independent. Thus, the tuning can be carried out by a person other than the algorithm developer, providing a neat separation of tasks and permitting both algorithm specification and hardware tailoring to progress in parallel.)

Nevertheless, in order to take advantage of these desirable features of functional specifications, it must be possible to execute the specified computations efficiently. The very problems and algorithms for which the advantages of functional specification are most important are the ones that require prodigious amounts of computation and that cannot tolerate even relatively minor inefficiencies.

Simply put, the two main problems that must be overcome in order to execute functional programs efficiently are the (apparent) speed and storage overheads of the use of higher-order functions and the inability to update individual elements of data structures (such as arrays) without copying the entire structure. Unless these problems can be overcome, functional programming is unlikely to be used in the solution of numerical problems.

We believe that functional programming will be taken seriously only when there have been several successful demonstrations of *parity* between functional and procedural programs. A demonstration of parity is an experiment that shows that using a functional program for a significant scientific application can result in code that executes (at least) as fast and uses (at least) as little storage on a high-performance computer as does a typical handwritten procedural program for the same application on that hardware.

We discuss here a demonstration of parity between an automatically derived implementation of a functional program and a handwritten Fortran program for solving a *practical* fluid-dynamics problem on the CRAY X-MP computer. This demonstration shows that functional programming, together with automated program transformations that derive an efficient program, is a practical tool that can be used to solve today's

numerical problems simply, elegantly, and efficiently. We also discuss our approach to developing the program transformations that produce efficient implementations from functional specifications.

We have achieved parity for a numerical problem—the solution of hyperbolic partial differential equations (PDEs)—by using automated program transformations to derive an efficient, vectorizable Fortran program from a (higher-order) functional program, which serves as a specification for the computation. To derive the program, we use a sequence of basic, general-purpose transformations that implement functional specifications in Fortran, interspersed with a few transformations that perform either problem-domain-oriented or hardware-oriented optimizations. Indeed, this is one of the important advantages of using program transformations to perform the derivation: they make it easy for one to incorporate problem-domain-dependent and hardware-dependent optimizations to whatever extent is necessary to achieve the desired level of performance in the implemented program. Since achieving parity for the functional specification of the solution of hyperbolic PDEs, we have also achieved parity for an entirely different algorithm (the solution of eigenvalues and eigenvectors of a symmetric matrix) on an entirely different computer architecture (the AMT DAP SIMD array processor) [5].

Our first experiments with the functional specification for the hyperbolic PDE solver involved only six days of effort to obtain a running program. Since those early experiments we have concentrated on improving our original specification and the transformations that implement it. The improvements are aimed at preparing a functional specification for the hyperbolic PDE algorithm that is independent of the dimensionality of the problem—one from which implementations for solving one-, two-, or even three-dimensional hyperbolic PDEs can be derived. We describe the ways in which our original functional specification fell short of dimension-independence, how we minimized these dependencies in the new specification, how the new specification was transformed for execution on the CRAY X-MP, and how we plan to transform the new specification for two- and three-dimensional problems into an efficient implementation.

2 Pure Lisp with Data Abstraction as a Specification Language

For our functional specification language we use pure Lisp (essentially a form of Church's lambda calculus [10]), together with data abstraction. This specification language is similar to the pure functional subset of Scheme, as described in the *Little Lisper* [13]. The specifications that we express in this language are high level but still algorithmic. In fact, they are executable, although in many cases they would execute very slowly.

Pure Lisp is simple, even minimalist. It relies on just four basic constructs: conditional expressions, lambda abstraction (abstraction of an expression with respect to specified variables), application of lambda abstractions to arguments, and naming of lambda abstractions (to create recursive functions). Of course, higher-order functions are included. Using such a minimalist functional language has a great advantage: it is conceptually easy to transform into an implementation, because only the small number of constructs just discussed need be implemented.

We do not use the native data types of Lisp (except for Boolean and numeric types) in the upper, problem-oriented levels of our specifications. Rather, we use data abstractions

appropriate to the problem being specified. Moreover, we treat these data abstractions as if they were abstract data types, and we write our pure Lisp specifications to be strongly typed, even though pure Lisp itself does not enforce strong typing.

3 The Problem

The solution of problems in fluid flow is the subject of the functional specification and vectorizable code developed in this case study. Technically, the equations to be solved are conservation laws, or, more generally, first-order, quasi-linear, hyperbolic partial differential equations. The solution of hyperbolic PDEs arises in many practical applications that involve wave phenomena, including acoustics, elasticity, and electromagnetism.

In all problems involving hyperbolic PDEs, *characteristics* play an important role. Characteristics are curves in space-time along which information propagates from the initial data. In the case of nonlinear hyperbolic PDEs, characteristics may intersect. When characteristics intersect, the hyperbolic problem no longer has a unique solution, and a *shock* forms. The mathematical discontinuity at a shock, of course, creates difficulties for the designer of algorithms for solving hyperbolic PDEs. A new algorithm that uses a combined cellular-automaton and method-of-characteristics approach is the subject of research by Garbey and Levine [14].

For a number of reasons, this algorithm is an interesting choice for an attempt to achieve parity.

- The algorithm is numerically intensive. Such algorithms are generally regarded as being outside the area of application of functional programming.
- The algorithm itself is still being refined and changed, and each refinement or change requires the algorithm to be tested and its behavior analyzed. To permit easy evaluation of these refinements and changes, we wish to minimize the cost of recoding the executable program to incorporate these changes.
- The performance of the algorithm is being evaluated. Meaningful performance evaluation demands computing the solution of large data sets over many discrete time steps, which in turn leads to long execution times. This circumstance rules out using the functional specification as a rapid prototyping tool, because naive execution of the specification would be excruciatingly slow. We lay great importance on retaining the freedom to write functional specifications in the clearest possible form, in order to make them as simple and understandable as possible. Thus, having to *tailor* the functional specification itself to increase the economy of its direct execution is unacceptable.
- The algorithm, although conceptually simple, requires some difficult coding and optimization. These difficulties are particularly knotty in the natural extensions of the algorithm for two and three dimensions, in which the geometry of the grid (hexagonal in the two-dimensional case) requires intricate calculations.
- The algorithm is a candidate not only for vector, but also for parallel—in particular, data-parallel—implementations, because of the large grids it uses. Indeed, handwritten Fortran implementations of the algorithm were prepared both for the data-parallel Connection Machine 2 and for the vector architectures of the Alliant FX/8 and CRAY X-MP. Thus, multiple implementations of the same specification, each tailored to a

particular architecture, are ultimately required. Again, because this is an experimental algorithm, the effort expended to produce these different versions must be minimized to enable the algorithm to be evaluated on different hardware.

In short, we believe that this problem is difficult enough to represent a good test of the applicability of functional programming and program transformation to scientific computation and thus is a good example for our case study. Moreover, the requirements of ease of modification, efficient execution, and multiple executable realizations of the same specification provide opportunities for functional programming and program transformation to aid the developers of the algorithm. Finally, the complexity of the algorithm in the higher-dimensional cases should provide an opportunity to demonstrate the ability of the modularity inherent in functional programs to *factor out* complexity. If functional programming is to gain credibility, it should be demonstrably effective *on this sort of problem* and *in this type of situation.*

4 Cellular Automaton Solution of a Hyperbolic PDE

The algorithm we are specifying uses a cellular automaton (CA) approach to compute the characteristics of the solution to a hyperbolic PDE. Cellular automata have applications ranging from modeling snowflakes to computing a solution (as in this case) of a partial differential equation. In fluid dynamics computations, CA methods are typically used as a direct approximation to the molecular dynamics of the problem, because the automaton can model the behavior of a *particle* in the fluid, and the transition rules of the automaton correspond to the possible results of collisions with other particles. (Wolfram discusses many interesting applications of cellular automata [16].)

4.1 A One-Dimensional Functional Specification

A cellular automaton model consists of many identical cells, each having simple, locally determined behavior. Nevertheless, the combined behavior of the cells can be complex. The grid for a one-dimensional hyperbolic PDE is a cellular automaton consisting of a line of cells, each of which holds information about the physical state of the problem. Successively updating the state information of each cell through a sequence of discrete time steps computes the characteristics of the solution to the hyperbolic PDE. At a given time step, the state information of all cells is updated simultaneously according to the same rule.

One may specify this behavior as the application of a function **steptime** to an initial grid for a specified type and size of problem, a set of boundary values (constant over time), an initial time, and a number of time steps to be performed:

```
steptime (initgrid (problemtype, gridsize), bv, 1, maxsteps)
```

The **initgrid** function defines an initial grid representing suitable initial conditions for the specified type of hyperbolic problem.

The specification for **steptime** follows immediately from the observations that (1) if the preceding time step was the last one, the result of taking a time step is the argument **grid**, and (2) otherwise, the result is that obtained by taking another time step on an updated grid:

```
steptime (grid, bv, step, maxsteps) =
    if step > maxsteps then
        grid
    else
        steptime (updategrid (grid, bv), bv, step+1, maxsteps)
```

In a similar vein, the specification for **updategrid** follows from the observation that the grid is updated by applying a local update rule to all of its cells:

```
updategrid (grid, bv) =
    mapgrid (lambda grid, loc . updatecell (grid, loc, bv), grid)
```

The **mapgrid** operation (which is, of course, analogous to the **map** operation for other structures) applies a function (**mapgrid**'s first argument) to each cell in a grid (**mapgrid**'s second argument). For each cell in the grid, **mapgrid** applies the function to a pair of arguments—the grid and the location of the cell in the grid.

Updating a cell happens in one of two ways, depending on whether the cell is a boundary cell or an interior cell:

```
updatecell (grid, loc, bv) =
    if isonboundary (loc, grid) then
        updateboundarycell (cellat (loc, grid),
                            whichboundary (loc, grid), bv)
    else
        updateinteriorcell (cellat (loc, grid),
                            neighborsat (loc, grid))
```

This separation is convenient because the rules for updating boundary cells are significantly different from those for updating interior cells. The function for updating a boundary cell uses the boundary values provided as part of the initial input to inject additional characteristics into the model when required.

Note how the modularity inherent in functional programming can be used (by means of the function **neighborsat**) to ensure that the updating of a cell depends only on the characteristic values of that cell and its neighbors, not on the exact location of the cell in the grid. Note also that at this level of abstraction we have not yet committed to implementing a cellular automaton algorithm. The preceding functions can be used just as well in the specification for a traditional 3-point (or, in two dimensions, 5-point) difference method.

At the next level of the specification, we begin to specialize it to hyperbolic problems. In such problems, the state information of each cell in the grid contains a dependent variable that represents some aspect of the physical state of the system (velocity, density, etc.). Characteristics move across the grid at a specified speed and in a specified direction. To model the hyperbolic PDE, each cell holds four pieces of information:

- a u value, which is the value of the dependent variable in the cell;
- an x value, which is the position of the moving characteristic within the cell;
- a *state*, which denotes whether a shock has occurred in the cell (as discussed in the following paragraph); and

– a *slope*, which denotes the speed and direction of propagation of the characteristic.

In the specification for the one-dimensional hyperbolic PDE problem, the directions are west (left) and east (right). In the current specification, the direction of the slope is kept in a separate component of the cell, called the *sign*, thereby simplifying certain computations.

The CA algorithm employs a further discretization in addition to that obtained by using a grid of cells. Each cell contains a discrete number of internal points at which the characteristic can be located, as denoted by its x value. In our specification, each cell contains 100 such points. At each time step, the x value is updated by adding or subtracting (depending on *sign*) the value of *slope* to obtain a new position (time steps are normalized to 1). The number of points in the a grid cell must be chosen large enough so that after normalization all *slopes* are less than the number of points in a grid cell. This choice guarantees that when a characteristic moves out of a cell, it moves into an adjacent one without skipping a cell.

Of course, the interesting behavior of this simple CA model concerns what happens when the updated x value lies outside the current cell, possibly causing the characteristics in different cells to intersect and produce a shock. A characteristic leaves a cell when its x value exceeds the maximum number of internal points in the cell. For the one-dimensional problem, three specific cases are associated with a characteristic leaving a cell, two of which lead to the formation of a shock:

Characteristic Moves to Another Cell

A cell that does not have a characteristic may obtain one if a characteristic moves into the cell from a neighboring cell (Figure 1).

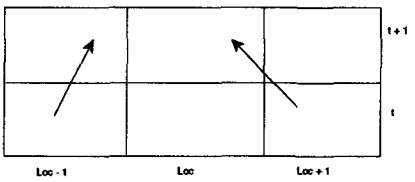

Fig. 1. A characteristic entering an empty cell at loc from its east neighbor

Shock

A shock occurs when, at some time step, one cell contains more than one characteristic. This situation obviously arises under two conditions: when a characteristic enters a cell that already contains a characteristic (and that characteristic is not leaving on the same time step) or when two characteristics enter the same cell at the same time step (Figures 2 and 3). In the CA algorithm, when shock occurs in a cell, the cell is marked "shocked" and the characteristics are removed, leaving an empty cell.

Crossing Shock

A shock (*crossing* shock) also occurs when two characteristics cross one another on a cell boundary at some time step (Figure 4). This condition is special because at

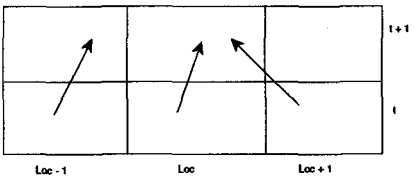

Fig. 2. Shock resulting from a characteristic entering an occupied cell at loc

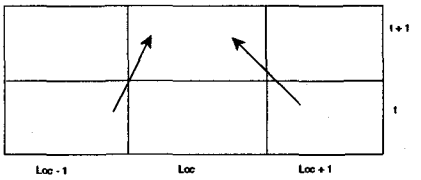

Fig. 3. Shock resulting from two characteristics entering an empty cell at loc

no time do the two characteristics actually occupy the same cell. Nevertheless, this condition is a shock because more than one characteristic would have occupied a cell if the cell boundaries of the grid had been differently positioned.

Simple Movement within a Cell

In addition to these cases, the update rule for cells also has a case for no shock—the simple movement of the characteristic within a given cell (Figure 5).

This computation of the movement of the characteristic corresponds to that of the ordinary (non-CA) method of characteristics.

From these observations follows the specification `updateinteriorcell`:

```
updateinteriorcell (cell, neighbors) =
    if isshocked (cell, neighbors) then
        emptymarkedcell (shock ())
    else if iscrossingshocked (cell, neighbors) then
        emptymarkedcell (crossingshock ())
```

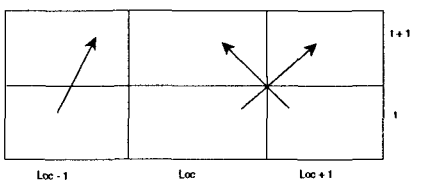

Fig. 4. Crossing shock resulting from two characteristics crossing on the east boundary of a cell at loc

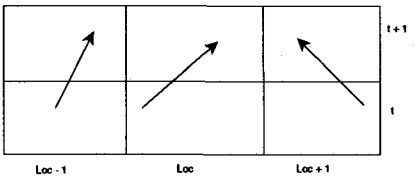

Fig. 5. Characteristic moving within a cell at loc

```
else if isenteringfrom (neighbors) then
    neighborenteredcell (neighbors)
else
    timestepedcell (cell)
```

If a shock or a crossing shock occurs, the result is an empty cell marked appropriately. If no shock occurs and a characteristic is entering the empty cell from one of its neighbors, then (since the cell was empty) the result is a cell whose state reflects that it contains a single characteristic. Otherwise, the result is a cell whose state is computed according to the method of characteristics.

Note that thus far the dimensionality of the grid has played no role in the specifications of the functions. Thus, none of the preceding functions need be altered in going to a specification for a higher-dimensional problem.

The dimensionality of the grid does enter, however, in the specification of the functions **isenteringfrom** and **neighborenteredcell**. For a one-dimensional grid, each cell has two neighbors, west (left) and east (right). A characteristic is entering a cell from one of its neighbors (in the one-dimensional case) if the characteristic is leaving the west neighbor going east or leaving the east neighbor going west:

```
isenteringfrom (neighbors) =
    isexitingeast (west (neighbors))
    | isexitingwest (east (neighbors))
```

If the characteristic is leaving the west neighbor of a cell, that characteristic is the basis for the updated state of the cell; otherwise, the characteristic from the east neighbor is the basis:

```
neighborenteredcell (neighbors) =
    if isexitingeast (west (neighbors)) then
        movedintocell (west (neighbors))
    else
        movedintocell (east (neighbors))
```

A shock occurs in a cell

- if the cell will have a characteristic on the next iteration (the cell's characteristic is not moving to a neighboring cell) and if the characteristic of one (or both) of the cell's neighbors is entering the cell; or

– if (the cell may not have a characteristic on the next iteration, but) the characteristics of both of the cell's neighbors are entering it:

```
isshocked (cell, neighbors) =
    (hasstatenextiteration (cell)
     & (isexitingeast (west (neighbors))
        | isexitingwest (east (neighbors))))
    | (isexitingeast (west (neighbors))
       & isexitingwest (east (neighbors)))
```

A crossing shock occurs in a cell if the characteristic of the cell is entering one of its neighbors at the same time as the characteristic of that neighbor is entering the cell:

```
iscrossingshocked (cell, neighbors) =
    (isexitingwest (cell) & isexitingeast (west (neighbors)))
    | (isexitingeast (cell) & isexitingwest (east (neighbors)))
```

Note that, with regard to the top level of the specification that we discuss in this section, the grid itself is still an abstract object; no implementation decisions have been made about it.

At the next level of detail of this specification, however, we do make an implementation decision to use an array to represent the grid. As one consequence, we define the `mapgrid` function in terms of a primitive function `maparraywithindex` that acts on arrays. The `maparraywithindex` function is transformed to an implementation tailored for the hardware in use, in this case the CRAY X-MP.

Specification of the remainder of the cellular automaton hyperbolic PDE solver proceeds in a similar manner until all functions (not only computational functions but also those implementing data abstractions) have been specified. (See [7] for a complete specification.)

We claim that this specification is a simple and natural one for this problem; indeed, we believe that this specification is transparently clear. Moreover, we claim that we have not knowingly biased the specification in the direction of an efficient final implementation; indeed, we have tried always to choose naturalness and clarity over efficiency.

5 Critique of the Original (One-Dimensional) Functional Specification: An n-Dimensional Specification

Solution of one-dimensional hyperbolic PDE problems is of moderate interest. However, the really interesting problems are two- and three-dimensional. Therefore, once we had shown that we could design transformations to construct an implementation for the one-dimensional problem [7], we decided to examine our functional specification to see how easily it generalized to higher dimensions.

As pointed out in the preceding section, we did succeed in keeping the dimensionality from entering the upper level of the original specification. We achieved this independence in part by using data abstractions that hide the dimensionality of the grid, cells,

and boundary values. In the original specification, it is possible to keep the dimensionality of the problem from entering until the definition of the functions **isenteringfrom**, **neighborenteredcell**, **isshocked**, and **iscrossingshocked**.

However, as one can see by examining the definitions of **neighborenteredcell** and **isshocked** given earlier, these definitions would have to be altered substantially for the two-dimensional case, and again for the three-dimensional one. For example, in the two-dimensional case, the second disjunct of the definition of **isshocked** must be expanded to account for the fact that characteristics may enter from any combination of two (or more) neighbors.

One way to develop a functional specification for higher-dimensional problems would be to *systematically* modify the original specification. Indeed, we might consider modifying it by applying program (really, specification) transformations [11]. However, the result of applying these transformations would still be very complicated definitions; and we believe that, because such transformations would not literally preserve correctness, validating them would be a problem.

Examination of the definitions in our original specification led us to conclude that the difficulty is not that the dimensionality of the problem enters the specification at too high a level, but rather that when dimensionality *does* enter, it is *not handled in a manner that enables easy generalization* to higher dimensionality.[3] The key to making dimensionality easy to generalize is to parameterize it in terms of a *set of neighbors*. The functions that would depend on dimensionality can then be specified as iterations over the set of neighbors. These functions then become dimension-independent, leaving only the (lowest-level) definitions of the neighbor sets themselves dimension-dependent.

This approach to handling dimensionality also has the advantage of catering to correctness. To the extent that one can write a functional specification that is independent of dimension, any formal proofs of correctness or testing performed validate the specification for problems of *any* dimensionality. Thus, if one has validated the one-dimensional implementation, in moving to higher dimensionality one need only concentrate on the new implementations required for the data abstractions.

Using these ideas, a natural formulation is to consider an updated grid to be the result of a *prediction* followed by a *correction*. (We defer briefly the exact definition of the overall grid-update function). The function **advancedgrid** naively predicts (generates) the new positions of the characteristics.

```
advancedgrid (grid) =
    mapgrid (lambda grid, loc . advancedcell (cellat (loc, grid)),
             grid)
```

Its value is a grid with the position of each characteristic advanced by one time step. Advancing the position of a characteristic may cause it to move out of the cell it previously occupied. The prediction generated by **advancedgrid** is naive in the sense that the positions still refer to the coordinate system of the characteristic's original cell and are still stored in the original cell.

[3] We are indebted to Brian Smith of the University of New Mexico for this observation.

The function **adjustedgrid** corrects the advanced grid by determining which cells the characteristics now occupy, moving the characteristics to these cells, and then determining whether any of these movements has caused a shock.

```
adjustedgrid (grid) =
    mapgrid
        (lambda grid, loc .
                adjustedcell (cellat (loc, grid),
                              neighborsof (loc, grid)),
         advancedgrid (grid))
```

Finally, we wish to add a feature unrelated to dimensionality to this second version of the specification for the CA algorithm. We wish to permit *interpolation* to be performed on the adjusted grid. During execution of the CA algorithm, cells become empty because shock occurs or because the cell's characteristic moves into another cell. If these characteristics are not replaced, the accuracy of the solution may be affected. To overcome this difficulty, one may wish to have a version of the algorithm that uses the mathematical technique of *interpolation* to reinsert characteristics into emptied cells. Similarly, the injection of new characteristics based on the boundary values is naturally a part of interpolation. While we do not give a specification for interpolation here (its exact mathematical formulation for the two-dimensional case is still being investigated), we do apply a function **postprocessedcell**, in which interpolation could be specified, to the adjusted grid.

```
updatedgrid (grid, boundary) =
    mapgrid (lambda adjustedgrid, loc .
                    postprocessedcell (loc, adjustedgrid,
                                       boundary, grid),
             adjustedgrid (grid))
```

(Postprocessing may depend on the value of the grid from the preceding time step as well as on the adjusted grid; therefore, both must be arguments to the **postprocessedcell** function.)

The specification for advancing cells is conceptually simpler if we consider a cell without state to have a *phantom characteristic* whose velocity is zero and whose position is the center of the cell. Then the characteristics in all cells can be advanced without testing whether a cell has state, because the phantom characteristics will not move and hence will not leave their cells and influence their neighbors. Although one might complain that the use of phantom characteristics is an implementation trick, we believe it leads to simpler, more easily understood specification. If one does not use phantom characteristics, then it is necessary to use a conditional expression testing **hasstate (cell)** in **advancedcell**. Using phantom characteristics, the value of an advanced cell is simply a new cell with the old values of state, velocity, and the dependent variable, but with an updated position:

```
advancedcell (cell) =
    newcell (stateof (cell), velocityof (cell), uof (cell),
             nextposition (positionof (cell), velocityof (cell)))
```

After advancement, the data for a characteristic are still associated with that characteristic's original cell, even though that characteristic's true position may now lie in an adjacent cell.

The adjustment phase involves moving the data for characteristics to their new cells (where required) and recognizing shock:

```
adjustedcell (cell, neighbors) =
    if isshocked (cell, neighbors) then
        emptiedcell (umarkfortypeofshock (cell, neighbors))
    else if ~haslocalcharacteristic (cell) then
        if isenteringfromone (neighbors) then
            enteredfromneighborcell (neighbors)
        else
            emptiedcell (nullu ())
    else
        cell
```

Postprocessing of the grid handles interpolation, if performed, and boundary value injection; cells requiring neither are left unchanged:

```
postprocessedcell (loc, adjustedgrid, boundary, oldgrid) =
    if isonboundary (loc, adjustedgrid) then
        postprocessedboundarycell (loc, adjustedgrid, boundary)
    else if needsinterpolation (cellat (loc, adjustedgrid)) then
        postprocessedinteriorcell (loc, adjustedgrid, oldgrid)
    else
        cellat (loc, adjustedgrid)
```

We come now to the definitions for those functions that, in the original functional version of the CA algorithm, were specific to the one-dimensional case. These are the functions that relate to the neighbors of a cell. In our original specification, these functions are particular to the one-dimensional case; our aim in the second specification is to develop a definition that is general for all dimensions.

An additional insight further simplifies specifying those predicates that will now be expressed in terms of the set of neighbors discussed earlier. This insight is to *count* the total number of characteristics in a cell to determine whether shock occurs, rather than to analyze various combinations of conditions on the characteristics entering and leaving the cell. This formulation of shock is simple in both concept and statement. After updating the positions of all the characteristics and determining in which cells they fall, the following possibilities exist:

- A cell contains no characteristic; hence the cell has no state and is empty.
- A cell contains one characteristic; hence the cell has state and is not shocked.
- A cell contains more than one characteristic; hence the cell has no state and is shocked.

A cell is shocked if it is subject to ordinary shock or to crossing shock:

```
isshocked (cell, neighbors) =
    isordinaryshocked (cell, neighbors)
    | iscrossingshocked (cell, neighbors)
```

Ordinary shock occurs if there is more than one characteristic in the cell:

```
isordinaryshocked (cell, neighbors) =
    totalcount (cell, neighbors) > 1
```

The number of characteristics in the cell after adjustment is the number remaining after advancement plus the number entering from neighbors as the result of advancement:

```
totalcount (cell, neighbors) =
    remainingcount (cell) + enteringcount (neighbors)
```

The number of characteristics remaining in a cell (after advancement) is zero or one:

```
remainingcount (cell) =
    booltoint (haslocalcharacteristic (cell))
```

A cell retains its original characteristic (after advancement) if the cell has state and the characteristic does not leave the cell:

```
haslocalcharacteristic (cell) =
    hasstate (cell) & exitdirection (cell) = none ()
```

(Note that the state component of a cell after advancement is the same as that of the cell before advancement.)

A cell has state if its state value so indicates:

```
hasstate (cell) =
    stateof (cell) = withstate ()
```

The number of characteristics entering a cell from its neighbors is the sum of the number (which must be zero or one) entering from each neighbor (*slope* is constrained so that, in a single time step, a characteristic cannot skip over a cell). The number of characteristics entering a cell from a neighbor lying in some direction from that cell is the number leaving that neighbor in the opposite direction (the direction toward the cell):

```
enteringcount (neighbors) =
    reduce
        (+, 0,
         map (lambda neighbor .
                    booltoint (exitdirection (cellof (neighbor))
                    = oppositedirection (directionof (neighbor))),
              neighbors))
```

Exactly one characteristic is entering a cell from a neighbor if the (total) entering count of the neighbors is one:

```
isenteringfromone (neighbors) =
    enteringcount (neighbors) = 1
```

Suppose that the characteristic of a cell is leaving that cell in some direction, *d*. Then, crossing shock occurs if the characteristic of the neighbor lying in direction *d* is leaving that neighbor in the direction opposite *d*. For example, if the characteristic of a cell is leaving to the east, then crossing shock occurs if the characteristic of the cell's east neighbor is leaving to the west. (Note that in this specification, if a cell's characteristic is not leaving, we give it the direction **none**; we require that the implementation chosen for **oppositedirection** guarantee that **oppositedirection (none)** $\neq$ **none**. In this specification, we designate **all** to be the direction opposite **none** in order to fulfill this requirement.)

```
iscrossingshocked (cell, neighbors) =
    exitdirection (cellof (neighborindirection (exitdirection (cell),
                                                neighbors)))
    = oppositedirection (exitdirection (cell))
```

When the characteristic of one neighbor enters an empty cell, the result for the cell is the value of entering the characteristic of that neighbor (that is, the advanced value of that neighbor's characteristic, translated to the coordinate system of the cell). The neighbor in question is that one among all the neighbors of the cell whose characteristic is leaving in the direction opposite the direction in which that neighbor lies from the cell:

```
enteredfromneighborcell (neighbors) =
    enteredfromthisneighborcell
        (elementofset
            (filter
                (lambda neighbor .
                        exitdirection (cellof (neighbor))
                        = oppositedirection (directionof (neighbor)),
                 neighbors)))
```

We restrict the use of the function **enteredfromneighborcell** to cases in which there is exactly one such neighbor cell. A safer approach, but one that would require more general simplification transformations, would be to re-express the condition that there be only one such neighbor cell in this definition. Then, when this function is applied within a context in which this condition is known to hold (as it is when **isenteringfromone(neighbors)** holds), the simplifications would remove the redundant evaluation of the condition.

The value of entering the characteristic from a particular neighbor is the value of translating the position of the characteristic from that neighbor to the cell:

```
enteredfromthisneighborcell (neighbor) =
    movedintocell (cellof (neighbor), directionof (neighbor))
```

We have now completed the specification (except for the functions relating to boundary values and for constructing the new values of cells, which we do not discuss) to the level at which choice of data representation and dimensionality must inevitably be made

explicit. By adding specifications that implement the data abstractions used in all of the preceding functions, we can produce a program that can be executed for small grids in rapid-prototyping mode using Lisp or Scheme.

6 Deriving Efficient Programs from Functional Specifications

Of course, our goal is not simply to run these specifications in rapid-prototyping mode, where execution can be excruciatingly slow. Algorithms such as these are designed to solve large problems. Slow execution cannot be tolerated in solving, or even in testing programs for solving, such problems.

Moreover, in many cases these algorithms are interesting precisely because they may permit exploiting computers having novel high-performance architectures, thereby enabling the solution of problems that have heretofore been impossible. Efficient Lisp and Scheme implementations are not likely to be available for such computers.

Naturally, the question is: How can we obtain, from the functional specification, programs that execute efficiently and exploit high-performance computers? Our answer is to use program transformations.

6.1 The Transformational Derivation for the Original Specification

From the simple, pure functional specification of Section 4.1 we automatically derive an efficient implementation for computers having vector hardware, such as the Alliant FX/8 or the CRAY X-MP. We use the TAMPR program transformation system [2, 3] to apply a sequence of sets of program transformations that derive an efficient Fortran program from the higher-order functional specification. As we indicated in the introduction, most of these transformations are basic transformations—ones that are needed to implement any functional specification in Fortran. These transformations form the framework for the derivation. Used alone, they do a highly competent job [3]. Nonetheless, because our basic transformations are applicable to a broad class of functional specification, the implementations they produce still have the characteristics of implemented functional code: extensive copying and use of fresh storage, explicit use of a stack (or heap, if functions are used as first-class objects) to implement recursion, etc.

Such an implementation performs well, but in our experience it cannot hope to equal the performance of good handwritten imperative Fortran or C code. For example, without further optimization such an implementation is prohibitively inefficient in terms of storage consumption when large arrays are used. Even if general remedies for such inefficiencies could be found, we believe that the speed of well-written code comes from taking advantage of properties of both the problem being solved and the target hardware.

It is just this type of knowledge that we can capture and codify in sets of TAMPR transformations that are problem-domain-oriented or hardware-oriented. (Fortunately, our experience shows that it is not necessary for a single set of transformations to be *both* problem-domain-oriented and hardware-oriented.) We can then apply the problem-domain-oriented transformations as part of any derivation starting from a specification in the problem area, and we can apply the hardware-oriented transformations as part of any derivation ending in a program for the target hardware, to produce high-performance,

automatically generated programs. Over time, libraries of such transformations will accumulate, enabling efficient realizations of functional specifications from various problem areas to be produced easily for many different types of hardware, including high-performance parallel machines, simply by drawing appropriate sets of transformations from the library.

Thus, to transform the specification of the hyperbolic PDE solver into the implementation that achieves parity, we intersperse into the outline formed by the basic sets of transformations a few sets of transformations that perform problem-domain-oriented or hardware-oriented optimizations. These transformations guide the derivation in the direction of producing code that will vectorize and that will, when compiled by the Cray Fortran compiler, run efficiently on the CRAY X-MP hardware.

It is these problem-domain-oriented and hardware-oriented sets of transformations that we emphasize in this section; however, we begin with a brief overview of the basic transformations to provide a framework for discussion.

6.2 Sketch of the Transformational Derivation

The derivation for the CRAY X-MP consists of about 17 major transformational steps. Each of these steps is one of three types: domain-dependent transformations that apply to *grid problems* (marked in the following list with a single bullet); hardware-specific transformations that direct the derivation toward code that is efficient on the CRAY X-MP (marked with two bullets); and general-purpose functional-to-procedural transformations, such as could be part of a compiler for functional languages (marked with plus). The major steps in the derivation are as follows:

\+ Canonicalizing the pure Lisp specification (syntactic standardization)
\+ Unfolding data abstractions and nonrecursive function definitions and simplifying the result (a form of symbolic execution)
• Converting logical connectives to *no-short-circuit-evaluation* form (supports the Cray-specific late optimization of precomputing logical expressions)
\+ Reducing the complexity of storage usage (implementing reuse of the grid array)
\+ Preparing the pure Lisp for transformation to Fortran
\+ Eliminating tail recursion
\+ Transforming the prepared pure Lisp to structured, recursive Fortran
• Introducing a loop to implement `mapgrid` and unfolding its higher-order function argument
•• Changing the grid from an array of structures to a structure of arrays
•• Hoisting boundary conditions out of the loop (partitioning the index set of the loop)
\+ Transforming recursive Fortran to nonrecursive Fortran
•• Implementing *no-short-circuit-evaluation* logical operations by Fortran logical operators
•• Flattening nested conditionals (simplifying the loop body to the point that it can be vectorized by the Cray Fortran compiler)
•• Eliminating common logical subexpressions
•• Precomputing logical expressions used in `if` tests
\+ Eliminating some unneeded type-checking from arithmetic operations

+ Cleaning up and implementing the remaining abstractions

These steps are implemented by sets of TAMPR program transformations. Each TAMPR transformation is literally a rewrite rule, having a pattern and a replacement each of which is specified in terms of the grammar of the programming language being transformed. Typically, fewer than ten of these transformation rules are required to implement each of the major steps in the derivation. However, these rules are applied many times; for the specification for the hyperbolic PDE solver, the entire derivation from pure Lisp to Cray Fortran requires about 8000 rewrites. Clearly the ability of the TAMPR transformation system to apply rules *automatically* is vital. No one could afford to apply thousands of transformations by hand, or even to provide substantial guidance for their application.

A somewhat more detailed discussion of the basic transformation steps (those that do not cater to either the grid or the Cray), including example code fragments at several stages, is given in [3] (see also an earlier version in [8]).

6.3 Performance of the Derived Program

The derivation discussed here targets the final program to a CRAY X-MP 1/8 system having the extended memory address, compressed index/gather-scatter, and vector population count hardware features.

The final version whose performance we measured is the result of many experiments with the CRAY X-MP. In each experiment, we derived a Fortran program from the specification using the then-current version of the transformations, measured that program's performance, altered or added hardware-oriented transformations to produce what we hoped would be a more efficient version of the program, and performed another experiment. This approach is necessary to understand *what* works well on the Cray; it became clear that the transformations must cater to the peculiarities of the Cray CFT77 compiler as well as to those of the raw CRAY X-MP hardware.

Despite our use of the experimental approach just described, we do not suppose that every program specifier who uses our program specification and transformation methodology will develop his own problem-domain-oriented or hardware-oriented optimizing transformations. It is not worthwhile to develop special transformations to optimize specifications for simple problems. However, when using this methodology to solve large, long-running problems it may well be worthwhile for transformation specialists to develop transformations specific to the particular problem domain or hardware. Moreover, we believe that developing such optimizing transformations and adding them to a transformational derivation carried out by the TAMPR program transformation system is *much* easier than would be developing and adding them to a conventional compiler (a step that is essentially impossible for the general user). It should be clear that such transformations can be developed and inserted into a derivation to obtain increased performance without modifying the specification itself.

Finally, problem-domain-oriented transformations tend to be independent of the target hardware, while hardware-oriented transformations tend to be independent of a particular problem. These independence properties permit problem-domain-oriented transformations to be reused in derivations for many types of hardware, and hardware-oriented

transformations to be reused in derivations for many types of problem. This reuse further reduces the cost of developing sets of transformations, by permitting the required effort to be amortized over a large number of derivations.

Performance on the CRAY X-MP. Table 1 presents the timing comparison between this version of our derived program and Garbey and Levine's program [14]. While Garbey

Table 1. Program times on CRAY X-MP 1/8 for 16,384 cells, 1000 time steps, Riemann initial conditions

Program Version	CRAY X-MP Time sec
Functional Specification Transformed to Fortran	7.283
Hand-Written Fortran	7.551

and Levine did not make a large investment in tuning their program for the CRAY X-MP, they did write the program with the architecture in mind and with the intent of getting decent vector performance on that machine.

These data show that our automatically derived program certainly achieves parity with the handwritten program. In fact, our program is about 4% faster than the handwritten program. Determining exactly why our version is faster is difficult without a detailed timing analysis of the compiler-generated assembly language code. A cursory reading of the assembly language indicates, however, that our derived program is better than the handwritten program at overlapping the use of the multiple functional units of the Cray during the precomputation of the conditions for the `if`-statements in the main loop.

Performance on the Sun 3/110C (68020). Of course, use of the experimental approach we have described invites the question of whether we were simply lucky to achieve parity, by managing to manipulate the specification into a form that takes advantage of the sophisticated optimizations performed by the Cray CFT77 compiler. Could we do as well generating code for a less sophisticated compiler? One way (at least partially) to answer this question is to compare the performance of the derived program with that of the handwritten one on a typical sequential machine. (Of course, one should not set too much store by such a comparison, because *both* programs are, in fact, tuned for the Cray vector architecture.) Table 2 gives the timing results for these runs. (Note that we have timed the programs for 100 time steps in Table 2 rather than for 1000 as in Table 1, in order to obtain reasonable running times on the Sun.)

Indeed, on the Sun our program is about 4% slower than the handwritten program. This is a result of our Cray optimizations increasing the amount of work performed by the Cray-tailored version of the program. These optimizations result in fast execution on

Table 2. Program times on Sun 3/110C (16.67 MHz) for 16,384 cells, 100 time steps, Riemann initial conditions

Program Version	Sun 3/110C Time sec
Functional Specification Transformed to Fortran	136.880
Handwritten Fortran	131.712

the Cray, but are *pessimizations* on the Sun. Omitting them from the derivation enables us to produce a program from our specification that is faster than parity on the Sun (but slower than parity on the Cray), as illustrated in Table 3.

Table 3. Program times on Sun 3/110C (16.67 MHz) and CRAY X-MP for 16,384 cells, 100 time steps, Riemann initial conditions

Program Version	Sun 3/110C Time sec	CRAY X-MP Time sec
Functional Specification Transformed to Fortran	136.880	.729
Functional Specification Transformed to Fortran without Redundant Pre-computation of Predicates	119.560	.829
Handwritten Fortran	131.712	.757

6.4 Transformational Derivation for the n-Dimensional Specification

We have developed a transformational derivation for the one-dimensional PDE problem and demonstrated that a program generated automatically by this derivation achieves parity with handwritten code for the same problem. How much of our effort in developing this derivation is applicable to the n-dimensional specification? Have we simply replaced a development style that requires the development of new procedural implementations when changes are made to the problem specification by a style that requires the development of new transformational derivations when the specification is changed?

We believe that the transformational approach will become more widely accepted only when derivations can be shared and the effort in developing transformational derivations is seen not to be wasted. Our aim is thus to develop transformations that are general—not tied exclusively to the problem under consideration. Transformations specific to a particular problem domain, for example, to algorithms for solving problems on grids, or

to cellular automata algorithms, are acceptable in fulfilling this aim. However, transformations specific to a single specification are not. Similarly, acceptable hardware-oriented transformations are applicable to a specific class of hardware architectures, for example, to machines with vector capability or to MIMD architectures. But, these transformations should, in most cases, be useful for deriving efficient implementations from many problem specifications. In a few cases, writing special-purpose transformations may be appropriate, but such applications are perhaps rare, and the number of such transformations in a typical derivation is few or none.

We believe that the majority of the transformations that we developed—perhaps we should say, all of the transformations that we *should* have developed—are applicable to the n-dimensional specification. In generalizing the specification we did discover that a few of the transformations in the derivation for the one-dimensional case were not designed with adequate generality to handle the n-dimensional case. The most significant example is that we need transformations to perform *partial evaluation* (symbolic execution) in order to obtain efficient code for *specific* dimensionalities from the n-dimensional specification. That is, to obtain a specification to solve the two-dimensional problem, we wish to augment the n-dimensional specification with a specification that the set of neighbors is can be represented by the six directions `west`, `northwest`, `northeast`, `east`, `southeast`, `southwest`. Partial evaluation should then simplify the specification to operate on these six neighbors without overhead.

We are currently developing the partial evaluation transformations to simplify the n-dimensional specification to particular dimensions. The need for these transformations has precluded our deriving executable programs for specific dimensions from the n-dimensional specification. We discuss the role of partial evaluation in obtaining an efficient program for the one-dimensional problem from the n-dimensional specification in the next section.

A One-Dimensional Specification from the n-Dimensional Specification. We can use the n-dimensional specification to obtain a specification for the one-dimensional case. The first task is to define appropriate implementations for the data abstractions. The implementations of most data abstractions follow from those used in the original one-dimensional specification.

However, it is necessary for the derivation to construct efficient implementations for the `filter`, `map`, and `reduce` functions operating on the 2-tuple holding the one-dimensional pair of neighbors. The aim is to simplify and optimize the implementations of `filter`, `map`, and `reduce` to produce intermediate code from these operations that is similar to that produced from our one-dimensional specification.

`filter`

An example of the refinement path for the `filter` operation under partial evaluation is given in Figure 6, starting from a subexpression appearing in the definition of the function `enteredfromneighborcell`. (In this example, the variable `neighbors` as argument to `filter` represents the two-element set of neighbors of a point.)

A `filter` operation constructs a set in which each element satisfies the `filter` predicate. The definition of `filter` is unfolded and restricted to the one-dimensional case. Then the set selection operation `elementofset`, which is adjacent to a set

In the specification

```
elementofset (filter (lambda neighbor .
                          exitdirection (cellof (neighbor))
                          = oppositedirection (directionof (neighbor)),
                      neighbors))
```

unfold definition of `filter` (for a 2-tuple) $\Longrightarrow$

```
elementofset(
    makeset(
        if (lambda neighbor .
                  exitdirection(cellof(neighbor))
                  = oppositedirection (directionof (neighbor))
              (first(neighbors)))
        then first(neighbors)
        else empty(),
        if (lambda neighbor .
                  exitdirection(cellof(neighbor))
                  = oppositedirection (directionof (neighbor))
              (second(neighbors)))
        then second(neighbors)
        else empty()
))
```

optimize `elementofset` on a set of one element $\Longrightarrow$

```
if (lambda neighbor .
          exitdirection(cellof(neighbor))
          = oppositedirection (directionof (neighbor))
      (first(neighbors)))
then first(neighbors)
else if (lambda neighbor .
              exitdirection(cellof(neighbor))
              = oppositedirection (directionof (neighbor))
         (second(neighbors)))
then second(neighbor)
else error()
```

Fig. 6. Refinement of a filter operation

construction operation **makeset**, is simplified into a conditional evaluation that avoids the set construction altogether. (Note that the sets produced by these definitions are singleton sets; for sets with a cardinality greater than one an alternative evaluation strategy might be used.)

map, reduce

An example of the refinement of the map and reduce operations is given in Figure 7. A map is expanded into the construction of a tuple in which each element of the tuple is the result of applying the mapped function to the corresponding tuple element. A

```
In the specification
    reduce (+, 0,
            map (lambda neighbor .
                        booltoint (exitdirection (cellof (neighbor))
                                  = oppositedirection (directionof (neighbor))),
                 neighbors))
```

unfold definition of map (for a 2-tuple) $\Longrightarrow$

```
    reduce (+, 0,
            maketuple
                (lambda neighbor .
                        booltoint (exitdirection (cellof (neighbor))
                                  = oppositedirection (directionof (neighbor)))
                    (first(neighbors)),
                 lambda neighbor .
                        booltoint (exitdirection (cellof (neighbor))
                                  = oppositedirection (directionof (neighbor)))
                    (second(neighbors))))
```

unfold definition of reduce $\Longrightarrow$

```
    (0 + lambda neighbor . booltoint (exitdirection (cellof (neighbor))
                                     = oppositedirection (directionof (neighbor)))
           (first(neighbors)))
    + lambda neighbor . booltoint (exitdirection (cellof (neighbor))
                                  = oppositedirection (directionof (neighbor)))
        (second(neighbors))
```

simplify using identity property of 0 for addition $\Longrightarrow$

```
    lambda neighbor . booltoint (exitdirection (cellof (neighbor))
                                = oppositedirection (directionof (neighbor)))
       (first(neighbors))
    + lambda neighbor . booltoint (exitdirection (cellof (neighbor))
                                  = oppositedirection (directionof (neighbor)))
        (second(neighbors))
```

Fig. 7. Refinement of reduce and map operations

reduce on this tuple then applies the reduction function (+ in this case) to combine the elements of the tuple and the initial value for the reduction (0 in this case).

A significant change in the specification from our original one-dimensional form is the approach to updating the grid of cells. In the new specification, the process is performed in two stages: the first stage updates all cells and does not consider whether characteristics are leaving or entering cells; the second stage adjusts the updated cells by moving characteristics between cells and recording shocks where necessary.

By unfolding the definition of `advancedgrid` into that of `adjustedgrid`, we obtain

```
From the specifications
    adjustedgrid (grid) =
        mapgrid (lambda grid, loc .
                        adjustedcell (cellat (loc, grid),
                                      neighborsof (loc, grid)),
                 advancedgrid (grid))

    advancedgrid (grid) =
        mapgrid (lambda grid, loc . advancedcell (cellat (loc, grid)),
                 grid)
```

obtain by unfolding $\Longrightarrow$

```
    adjustedgrid (grid) =
        lambda ngrid .
               mapgrid (lambda grid, loc .
                               adjustedcell (cellat (loc, grid),
                                             neighborsof (loc, grid)),
                        ngrid)
            (mapgrid (lambda grid, loc . advancedcell (cellat (loc, grid)),
                      grid))
```

Fig. 8. Unfolding definition of adjustedgrid

the definition shown in Figure 8. This definition appears to require the computation of the entire advanced grid before the adjustment step. However, the two **mapgrid** operations have compatible shapes (in this case, the same shape) and both involve, of course, pointwise applications of their function arguments. It is therefore possible to combine the two pointwise operations to obtain a single mapping.

Of course, the combination must take into account the need for the **neighborsof** an element of the grid. The unfolding makes clear that there are two possibilities for obtaining these neighbor values:

- Combine the mappings, and recompute and discard (or store locally) the advanced values for neighbor cells as needed.
- Retain two separate mappings, and store the intermediate grid, thereby avoiding recomputation.

Our one-dimensional specification calls explicitly for the former approach—the specification itself indicates that neighbor values are recomputed rather than being stored. This strategy is reasonable for the one-dimensional case, where there are only two neighbors, and for the Cray, where recomputation is cheap when the data are available in vector registers. Clearly we can derive an update rule that is similar to that written in our one-dimensional specification from the unfolded form of the code in Figure 8. In the two-dimensional case, on the other hand, there are six neighbors, and recomputation is

probably not economical; hence we plan to retain the implementation in Figure 8, which explicitly produces the advanced grid.

Once partial evaluation transformations are available to simplify the neighbor operations, the rest of the one-dimensional derivation may now be used without change to obtain an efficient Fortran program that will execute on the CRAY X-MP. The derived program, and hence its execution behavior, is almost identical to that outlined above for the one-dimensional form.

A Two-Dimensional Specification from the n-Dimensional Specification. We are currently working on a two-dimensional form of our n-dimensional specification. This work is still at an early stage, but it follows the same outline as that described for the one-dimensional specification. We have defined the data abstractions for the two-dimensional case, and the focus of our current work is on optimization of two-dimensional grid operations. One of the challenges in this problem is representing the computations in the hexagonal geometry of the two-dimensional grid; this geometry makes the arithmetic significantly more complicated than for the one-dimensional grid. We expect to have to develop a few new transformations, or variants of some of the ones we have already implemented, to obtain efficient performance from this specification. However, it is clear at this stage that we will be able to reuse most of the derivation described earlier, although we must generalize the problem-oriented transformations for the grid.

7 Conclusions

In the preceding sections, we have discussed how clear, simple functional specifications can be transformed into efficient implementations. Our example, a significant scientific computation, transforms into highly efficient Fortran code for the CRAY X-MP supercomputer. Compared to a handwritten Fortran program for the same application, our transformed program pays *no* price in efficiency; the performance of the program derived from our functional specification slightly outstrips that of the handwritten program.

We have explained our motivations for, and our approach to, developing an improved specification for this computation, in which the dimensionality of the problem is parameterized. This second version of the specification possesses an enormous advantage over the first—except for a very small number of functions, *the same specification solves problems of any dimensionality.* This advantage is one that cannot be achieved (while retaining efficiency) by using conventional programming languages; generality and efficiency are incompatible when the specification *is* the program.

Did we did purchase the advantages of deriving an efficient program from a functional specification at a high price in human effort? No. Preparing the original problem-domain-oriented and hardware-oriented transformations that we wrote in order to carry out this derivation required less than one man-week. (This time does not include the time required to perform timing experiments on the Cray, which would be necessary no matter what methodology were used, assuming such experiments could be done at all when using other methodologies.)

Did we waste our initial effort to develop a derivation for the one-dimensional specification? No. The same derivation, with a few additional transformations to implement

features new to the second version of the specification, was used to develop an efficient (actually almost identical) Cray Fortran implementation of a one-dimensional form of the n-dimensional specification. The effort in developing these additional transformations was approximately one man-hour!

Have we biased the outcome of these experiments, either by the way we wrote our functional specification or by the way we wrote the transformations? In regard to the specification, we believe the answer is a resounding no! The specification is available for examination in its entirety in [7]. We believe that an examination will confirm that where we faced a choice between a specification having clarity, simplicity, and generality and one having efficiency, we have chosen the former. This is particularly true for the n-dimensional example, which aims to be as general as possible.

Where do we plan to go from here? We are beginning to modify the lowest levels of the specification for solving the one-dimensional hyperbolic problem to produce a specification for solving the two-dimensional problem. This work is requiring a few new transformations that optimize new features of this specification and is also helping us to refine and generalize the transformations used in our initial one-dimensional derivation.

Modifying the specification is not the only direction for future work. In a parallel study, we have developed transformations aimed at deriving efficient programs for the AMT DAP, a SIMD machine with a 32 by 32 grid of processors, and for the Connection Machine 2, a massively parallel SIMD machine. Initial experiments using specifications for other numerical algorithms have demonstrated automatic derivations leading to parity on the DAP [5]. In conclusion, we believe that studies demonstrating parity between functional and handwritten programs on significant problems are important steps toward a goal—the goal of making functional programming useful to the wide audience of scientists and engineers badly in need of techniques to help them quickly write clear, correct, and efficient programs.

Acknowledgments

We are indebted to both David Levine and Marc Garbey for explaining their algorithm and the difficulties of implementing it, and to Hans Kaper for encouraging us to work on this problem.

References

1. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice-Hall International, New York, 1988
2. Boyle, J. M.: A transformational component for programming language grammar. Technical report ANL-7690, Argonne National Laboratory, Argonne, Ill., July 1970
3. Boyle, J. M.: Abstract programming and program transformations–An approach to reusing programs. In Software Reusability, Volume I, T. J. Biggerstaff and A. J. Perlis (eds.), ACM Press (Addison-Wesley Publishing Company), New York, 1989, 361–413
4. Boyle, J. M.: Program adaptation and program transformation. In Practice in Software Adaptation and Maintenance, R. Ebert, J. Lueger, and L. Goecke (eds.), North-Holland Publishing Co., Amsterdam, 1980, 3–20

5. Boyle, J. M., Clint, M., Fitzpatrick, S., and Harmer, T. J.: The construction of numerical mathematical software for the AMT DAP by program transformation. In Parallel Processing: CONPAR 92—VAPP V, Second Joint International Conference on Vector and Parallel Processing, Lyon, France 1-4 September 1992, Ed. L. Bougé, M. Cosnard, Y. Robert, and D. Trystan, LNCS 634, Springer-Verlag, Berlin, 1992, 761–767.
6. Boyle, J. M., Harmer, T. J.: Functional specifications for mathematical computations. In Constructing Programs from Specifications, B. Möller (ed.), North-Holland Publishing Co., Amsterdam, 1991, 205–224
7. Boyle, J. M., Harmer, T. J.: A practical functional program for the CRAY X-MP. Journal of Functional Programming, **2**(1) (Jan. 1992) 81–126
8. Boyle, J. M., Muralidharan, M. N.: Program reusability through program transformation. IEEE Transactions on Software Engineering **SE-10**(5) (Sept. 1984) 574–588
9. Burton, F. W., Kollias, J. (Yannis) G.: Functional programming with quadtrees. IEEE Software **6** (Jan. 1989) 90–97
10. Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton, N.J., 1941
11. Feather, M. S.: Constructing specifications by combining parallel elaborations. Journal of IEEE Transactions on Software Engineering **15**(2) (Feb. 1989) 198–208
12. Field, A. J., Harrison, P. G.: Functional Programming. Addison-Wesley Publishing Co., Wokingham, England, 1988
13. Friedman, D. P., Felleisen, M.: The Little LISPer. Science Research Associates, Inc., Chicago, Ill., 1986
14. Garbey, M., Levine, D.: Massively parallel computation of conservation laws. Journal of Parallel Computing **16** (1990) 293–304
15. Kelly, P.; Functional Programming for Loosely-Coupled Multiprocessors. Pitman Publishing/MIT Press, London/Cambridge, Mass., 1989
16. Wolfram, S: Theory and Applications of Cellular Automata. World Scientific, Singapore, 1986