ADDING MULTIDIMENSIONALITY TO PROCEDURAL PROGRAMMING LANGUAGES*

P. RONDOGIANNIS

Department of Computer Science, University of Ioannina, P.O. Box 1186, GR-45110 Ioannina, Greece
E-mail: prondo@cs.uoi.gr

One of the most serious shortcomings of multidimensional languages is their inability to collaborate with conventional programming languages and systems. Multidimensional languages are used in order to define (potentially infinite) streams, grids, cubes, and so on, concepts which resemble in nature to the familiar imperative arrays. The main difference is that the former entities are lazy while the latter are generally eager. This paper proposes the embedding of multidimensional languages into conventional ones as a form of definitional lazy arrays. The paper describes the details of an implementation of the proposed idea as well as the expressibility and the performance of the resulting system. The main advantage of the new approach is that multidimensional languages can now benefit from the advanced features that have been developed for traditional languages. Moreover, multidimensionality adds to conventional languages the idea of lazy arrays, which in many cases offer significant advantages compared to the classical imperative arrays.

1 Introduction

Multidimensional programming\(^1,2\) is a recent programming paradigm which is especially appropriate for expressing problems which involve entities that vary along different dimensions. Many such problems exist in various areas such as scientific computing, signal and image processing, data processing, and so on \(^3,4,2\). One of the most serious obstacles in the wider use of multidimensional languages is the fact that they interface poorly with existing procedural programming languages and systems. As a result, multidimensional languages can not take advantage of advanced features that have been developed for traditional languages. On the other hand, conventional languages lack the problem expressing capabilities of multidimensional languages.

This paper proposes the embedding of multidimensional programs into conventional programming languages by viewing such programs as defining lazy multidimensional arrays. A lazy array is one whose entries are filled on demand (and not eagerly as in conventional languages). The proposed ap-

---

\*An extended version of this paper has been accepted for publication in a scientific journal.
proach allows multidimensional languages to take advantage of existing features available in conventional programming systems. Moreover, when viewed as lazy arrays, multidimensional programs may offer significant benefits compared to conventional (imperative) arrays. This is especially true in cases where only a small fragment of the elements of an array are needed in order to compute a desired result. Such cases are difficult to handle with imperative arrays, especially if the number of dimensions involved is significant. Moreover, in the lazy multidimensional array approach one can trade time for space, something which is not easy in traditional imperative arrays.

The amalgamation of multidimensional and procedural languages proposed in this paper is simple and it only requires small changes to the syntax and semantics of both formalisms. The paper describes the details of an implementation of the proposed idea as well as the expressibility and the performance of the resulting system.

The rest of the paper is organized as follows: Section 2 describes the technique for embedding multidimensional languages into conventional ones and Section 3 presents the details of an implementation of the proposed approach. Sections 4 and 5 explain why the technique introduced in the paper compares favorably in many cases to the use of imperative arrays and imperative procedures respectively. Section 6 concludes the paper with discussion of future extensions.

2 Multidimensional Programs as Lazy Arrays

The basic idea behind the proposed approach is that the data definition section of a conventional program can be extended to include a multidimensional lazy array definition part. The definitions of this part are in fact definitions of a multidimensional program. Entities that are defined in this way can be used in the remaining conventional part of the program as ordinary multidimensional arrays. The only exceptions to the use of lazy arrays, are the following:

- The value of a specific entry of a lazy array can not be altered by the procedural program. So, for example, a reference to a lazy array element can not appear in the left hand side of an assignment statement.

- Lazy arrays can not (in the present implementation) be passed (as a whole) as parameters nor returned as results of conventional imperative procedures. However, individual elements of such arrays can be passed as parameters to procedures.

All other uses are legitimate. For example, one can compare lazy array elements, use them in expressions, assign them to variables of the same type,
and so on. An important characteristic of this type of arrays is that they are referentially transparent: references to the same element of such an array in different parts of the program always correspond to the same value.

The basic advantage of the proposed approach is simplicity and semantic clarity. The two formalisms (namely multidimensional definitions and conventional program statements) do not mix in an arbitrary way, but through a specific, well-defined interface. So, for example, the multidimensional part of the program can not contain any imperative statements nor can it use any imperative variables. On the other hand, the imperative part of the program can only refer to specific elements of entities that are defined in the multidimensional part; it can not handle multidimensional entities in arbitrary ways.

To illustrate the above ideas, consider the following problem taken from the area of dynamic programming\(^5\). Suppose two teams, A and B, are playing a match to see who is the first to win \(n\) games for some particular \(n\). Let \(p(x, y)\) be the probability that if A needs \(x\) games to win, and B needs \(y\) games, that A will eventually win the match. It can be easily shown that \(p\) is given by the following recurrence relationship:

\[
p(x, y) = \begin{cases} 
  1 & \text{if } x = 0 \text{ and } y \geq 0 \\
  0 & \text{if } x > 0 \text{ and } y = 0 \\
  \frac{(p(x - 1, y) + p(x, y - 1))}{2} & \text{otherwise}
\end{cases}
\]

The above relation can be coded in the multidimensional language GLU\(^1,2\) as follows\(^a\):

```java
dimensions x, y;
p = 1 fby_x (0 fby_y (next_y(p)+next_x(p))/2);
```

Multidimensional programs such as the above will be embedded into conventional ones. The “host language” we will be considering in this paper is the object-oriented (and also imperative) language Java. The main reason that Java has been chosen is the fact that it supports a large number of useful classes which can be used in combination with multidimensional programs. For example, using Java’s Abstract Windowing Toolkit package (AWT), one can visualize the execution of multidimensional programs; using Java’s threads one can experiment with the parallel execution of multidimensional programs, and so on.

The syntax of the mixed language is demonstrated by the following extended Java program:

\(^a\)The syntax we use is slightly different than the actual GLU syntax, mainly for simplicity reasons. More on the syntax of the subset we adopt will be given in subsequent sections.
public class multi {
    /* The multidimensional array definition section */
    [[
        dimensions x, y;
        p = 1 fby_x (0 fby_y (next_y(p)+next_x(p))/2);
    ]]

    public static void main(String [] argv){
        System.out.println(p[100][100]);
    }
}

Notice that in the conventional part of the program, the variable p is used as an ordinary array. In particular, the above program will print to the standard output the value of the two-dimensional entity p at x coordinate 100 and at y coordinate 100.

As it will be demonstrated in the next section, lazy arrays can be used in cases where the use of conventional ones appears to be problematic (e.g. whenever space allocation for imperative arrays becomes a problem due to the large number of dimensions of a specific application). Moreover, in many cases, lazy arrays appear to be much more expressible and problem-oriented than their imperative counterparts.

3 Implementation

In this section we present the details of an implementation of the lazy arrays idea. More specifically, we outline the basic ideas behind a compiler we have developed which translates the extended language of the previous section into pure Java code. The compiler has been written almost exclusively in Prolog (in fact, a large part of the code has been borrowed from a compiler we had developed for a higher-order functional language 6).

The multidimensional language adopted is a simple subset of GLU. More specifically, programs of this subset consist of an initial declaration of the dimensions that will be used, followed by a set of individual variable definitions. In other words, the user-defined functions and the nested where clauses of GLU are not used. The language has been kept as simple as possible because it is not intended to compete with the more general-purpose “host language”. The simplicity of the language also leads to a relatively easy compilation procedure. The only GLU operators adopted are first, next, fby, and prev (working in all user-defined dimensions). The language also supports many usual operators (e.g. +, >, if-then-else, etc.). Roughly speaking, the above
subset of GLU corresponds to the syntax of a multidimensional extension of the logic programming language Chronolog \textsuperscript{7,8}.

On the other hand, the host language has also been restricted. Only one multidimensional array definition section is allowed in every class definition, and it appears as the first thing in the class definition. Lazy arrays are assumed to be private in the class where they are defined (i.e., they can not be accessed from outside the class). These restrictions are not serious and have been imposed for syntactic simplicity reasons.

The basic idea behind the implementation of the proposed lazy arrays approach is that every individual definition of the multidimensional program is compiled into a conventional Java method. Each such method has as formal parameters all the dimensions that are used by the corresponding multidimensional program. Moreover, the various operators are compiled according to their semantic definitions. As an example, consider the "two teams" example of the previous section. A (simple-minded) implementation of the example would give:

```java
public class multi{
    static double eval_p(int x, int y) {
        return ((x==0)?(1):
            ((y==0)?(0):(eval_p(x-1,y)+eval_p(x,y-1))/2));
    }
    public static void main(String argv[]){
        System.out.println(eval_p(100,100));
    }
}
```

Notice that the individual variable definition of p has been transformed into a Java method eval_p(int x, int y), where x and y are formal parameters that model the corresponding dimensions of the initial program. Notice also that the reference p[100][100] in the conventional part of the initial program has been transformed into the method invocation eval_p(100,100).

The above compilation is a naive one, in the sense that it results to extremely inefficient output code. The resulting program is a highly recursive one, and calls to eval_p(int x, int y) fail to terminate even for very small values of x and y.

The actual output of the compiler is the following (much more efficient) program:

```java
public class multi{
    static Warehouse w = new Warehouse();
    static double eval_p(int x, int y) {
        double temp;
    }
```
Key k = new Key(1, x, y);
Value v = w.get(k);
if (v != null) return v.result;
temp = (((x==0)?(1):
   (((y==0)?(0):(eval_p(x-1,y)+eval_p(x,y-1))/2));
w.put(k, new Value(temp));
return temp;
}
public static void main(String argv[]){
System.out.println(eval_p(100,100));
}

Certain explanations are in order. The main difference between the above program and the previous one is that the warehouse idea is now used. Notice that the above program presupposes the existence of a Warehouse class, which will be described shortly. The basic idea of the above code is that whenever a call eval_p(x,y) takes place, a search of the warehouse is first performed with the hope that the same call has occurred before and the corresponding result is immediately available. If the search is unsuccessful, the result value is computed as usual. After the computation has taken place, the value together with the corresponding context is added to the warehouse.

The Warehouse class implements an open hash table. For a two-dimensional program the warehouse has the structure shown in Figure 1. More generally, each entry of the table consists of a Key (with attributes the identifier name encoded by a small integer, and the coordinates under which the identifier has been demanded), the Value part which holds the value that corresponds to the particular entry, and the age of the particular entry (which is further discussed below). The Warehouse class, apart from its constructor method, also supports a void put(Key key, Value value) for storing values to the warehouse, a Value get(Key key) method for retrieving values from the warehouse, the int h(Key key) which implements a hashing function, and the void garbage_collect() method which performs a regular garbage collection of the warehouse.

The garbage collection scheme used is a slight variation of the one used in early Lucid implementations. More specifically, entries are stored in the warehouse together with an item which characterizes their age. In the variation we use, there exists a global age variable which is increased every time a new entry is added to the warehouse. The age field of the new entry

\[\text{In the usual approach for implementing Lucid-like languages, the age of an entry is...}\]
is then set to the new value of \texttt{global\_age}. In this way, the oldest entries are the ones having the smallest values in their age fields. Every time a garbage collection takes place, all entries whose age is below a certain limit are discarded; the age of all remaining entries is decreased accordingly. The overall behavior of the warehouse depends upon three parameters: the \textit{size} $S$ which is the number of buckets of the warehouse table (usually a prime number), the \textit{garbage collection limit} $G$ which is the number of entries in the warehouse after which the garbage collection algorithm starts executing, and the \textit{rate} $R$ which characterizes the percentage of the elements in the warehouse that will be deleted during garbage collection. For example, $S = 6997$, $G = 5000$ and $R = 0.5$ means that the warehouse table has 6997 buckets, garbage collection starts whenever 5000 entries exist in the warehouse and that 2500 entries will be deleted in every garbage collection.

The present implementation does not support dimensionality analysis \cite{11}, counted by the number of garbage collections that the entry has survived, and it is usually a small integer.
a technique that has been developed for identifying for each variable in a multidimensional program the minimum set of dimensions that the variable depends upon. Dimensionality analysis is especially useful when the number of dimensions that a program uses is significant. Knowing the dimensions on which a variable depends is important because one can then store (and subsequently search) in the warehouse only the coordinates that are relevant regarding a specific variable.

The next section discusses the performance of the above implementation and compares it with the performance of imperative arrays.

4 Comparison with Imperative Arrays

Conventional arrays is the standard means for implementing all kinds of applications that require storing and subsequent fast access to intermediate computation values. One problem however with imperative arrays is that often, in order to compute the value of a specific element, one has to compute the values of other elements on which the desired value depends. In general, this is not an easy task. For example, suppose one uses conventional arrays in order to compute the element \( p[10][10] \) (where \( p \) is the multidimensional entity modeling the “two teams” problem). The value of the particular element depends on the values of \( p[9][9] \) and \( p[10][9] \). These elements themselves depend on other elements of \( p \), and so on. Writing a program that would calculate the value of \( p[10][10] \) using imperative arrays is not straightforward, because the programmer has to think how to structure the nested for loops that will fill the array elements leading to the calculation of the desired value. The following Java program implements such a solution for the “two teams” problem:

```java
public class Comp {
    public static double p[][] = new double[1001][1001];

    public static double odds(int i, int j) {
        p[0][0] = 1;
        for (int s = 1; s <= i+j; s++) {
            p[0][s] = 1.0;
            p[s][0] = 0.0;
            for (int k = 1; k <= s-1; k++) {
                p[k][s-k] = (p[k-1][s-k]+p[k][s-k-1])/2.0;
            }
        }
        return p[i][j];
    }
}
```


public static void main(String argv[]){
    System.out.println(odds(500, 500));
}

The above program actually computes the elements of the array in diagonals. The resulting code is certainly not very straightforward to think and write. In the lazy arrays approach one simply has to think about the problem in hand and forget about the low level implementation details which are taken care by the eduction mechanism. During execution, when a specific element of a lazy array is demanded, the eduction mechanism uses the corresponding definition in order to calculate the desired output. This results in demands for other elements to be generated, and so on. In other words, the eduction engine itself reveals and follows the existing dependencies.

On the other hand, it should be noted here that imperative arrays are significantly faster than their lazy counterparts (because they use direct access instead of hashing). A comparison of the performance of the “two teams” program in the above imperative and the compiled lazy version of Section 3, is shown in Table 1 and Figure 2. Both programs were set to compute and print the value of $p[i][i]$ ten times (in order to get non-zero timing results for small values of $i$). The input value $i$ ranged from 100 to 650. For the lazy version, the parameters used in the warehouse were $S=6997$ (a prime number), $R = 0.5$ and $G = 5000$. As expected, lazy arrays are slower (about 7 times)\(^c\). However, using standard curve fitting techniques, one can see that both curves express quadratic complexity and they differ only by a constant factor. Notice also that the imperative solution fails for large input values possibly due to space problems (an unexpected NullPointerException is generated), while the lazy array solution continues to give results even for quite large values of the input. Moreover, the lazy program has an overhead due to the recursive method calls introduced by the compilation strategy of Section 3. Another overhead is due to the garbage collection of the heap which is performed by the Java runtime (and which is higher for the lazy program due to the objects used as entries by the warehouse).

Another shortcoming of imperative arrays is that in many applications, in order to compute a desired element, one has to compute all the values in the array. However, not all values will generally be needed and possibly only a very

\(^c\) All the performance results were obtained on a Sun UltraSparc 1, with 192 MB RAM, and the corresponing timings were recorded with the UNIX time utility. The programs were compiled and run using JDK 1.1.4.
Table 1. Time Comparison Between Lazy and Imperative Arrays

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Lazy Arrays</th>
<th>Imperative Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>150</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>250</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>300</td>
<td>52</td>
<td>7</td>
</tr>
<tr>
<td>350</td>
<td>65</td>
<td>9</td>
</tr>
<tr>
<td>400</td>
<td>90</td>
<td>12</td>
</tr>
<tr>
<td>450</td>
<td>112</td>
<td>15</td>
</tr>
<tr>
<td>500</td>
<td>129</td>
<td>19</td>
</tr>
<tr>
<td>550</td>
<td>173</td>
<td>23</td>
</tr>
<tr>
<td>600</td>
<td>202</td>
<td>-</td>
</tr>
<tr>
<td>650</td>
<td>239</td>
<td>-</td>
</tr>
</tbody>
</table>

small fragment of them will ever become necessary. This discussion is closely related to the old debate between lazy and eager functional programming languages. There are cases where an eager functional program is preferable and others where laziness is of paramount importance. It should be noted at this point that although this debate has a long history, it seems that the trends in modern functional languages are more in favor of lazy evaluation. This possibly suggests that lazy arrays may prove a useful complement to the traditional array concept.

A case where multidimensional lazy arrays appear preferable to conventional ones is for applications that use a significant number of dimensions. The space required for storing conventional arrays that use a large number of dimensions is by no means negligible. For example, a three dimensional array with 100 elements in each dimension has a total of one million entries. As the dimensions increase, the corresponding arrays tend to get unmanageable. On the other hand, lazy arrays do not need to be stored as a whole. When a value of a variable is demanded, the definition of the variable is used to compute the desired result. The result can then be saved in the warehouse so as that it can be used by subsequent computations. However, the result need not stay in the warehouse for ever. If space becomes a problem, the warehouse can be garbage collected. It can even be completely emptied if this is required. The values that are thrown away can always be recomputed by the corresponding
multidimensional definitions. In other words, in the lazy multidimensional array approach one can trade time for space, something which is not easy in traditional imperative arrays. Table 2 and Figure 3 show the space consumption comparison for the “two teams” problem. In the lazy version, the same parameters as before were used in the warehouse. The space for the lazy program is computed using a worst-case scenario: the space taken-up by the warehouse just before a garbage collection will take place. This space is constant because the garbage collection takes place whenever $G=5000$ entries have been added in the warehouse, and each such entry has a specific size.

Notice also that the warehouse size can be regulated according to the space limitations that an application may have. Smaller size of the warehouse does not necessarily mean worse performance. This is illustrated in Table 3 and Figure 4 which present the execution times for the calculation of $p[200][200]$ (ten times) with varying warehouse size. More specifically, $S = \text{varying (and usually a prime)}$, $G = 2/3 \times S$, and $R = 0.5$. Notice that performance is affected only when $S$ gets very low (in which case many useful entries are
Table 2. Space Comparison Between Lazy and Imperative Arrays

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Lazy Arrays</th>
<th>Imperative Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>36997</td>
<td>10201</td>
</tr>
<tr>
<td>150</td>
<td>36997</td>
<td>22801</td>
</tr>
<tr>
<td>200</td>
<td>36997</td>
<td>40401</td>
</tr>
<tr>
<td>250</td>
<td>36997</td>
<td>63001</td>
</tr>
<tr>
<td>300</td>
<td>36997</td>
<td>90601</td>
</tr>
<tr>
<td>350</td>
<td>36997</td>
<td>123201</td>
</tr>
<tr>
<td>400</td>
<td>36997</td>
<td>160801</td>
</tr>
<tr>
<td>450</td>
<td>36997</td>
<td>203401</td>
</tr>
<tr>
<td>500</td>
<td>36997</td>
<td>251001</td>
</tr>
</tbody>
</table>

Figure 3. Imperative Arrays vs Lazy Arrays: Space Comparison

thrown away during garbage collection and have to be recomputed).
Table 3. Execution Time as a Function of Warehouse Size

<table>
<thead>
<tr>
<th>Warehouse Size (S)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>443</td>
<td>355</td>
</tr>
<tr>
<td>461</td>
<td>259</td>
</tr>
<tr>
<td>503</td>
<td>291</td>
</tr>
<tr>
<td>547</td>
<td>283</td>
</tr>
<tr>
<td>599</td>
<td>364</td>
</tr>
<tr>
<td>601</td>
<td>19</td>
</tr>
<tr>
<td>607</td>
<td>18</td>
</tr>
<tr>
<td>919</td>
<td>20</td>
</tr>
<tr>
<td>1847</td>
<td>19</td>
</tr>
<tr>
<td>3697</td>
<td>20</td>
</tr>
<tr>
<td>7393</td>
<td>23</td>
</tr>
<tr>
<td>14783</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure 4. Execution Time as a Function of Warehouse Size
Of course, there are cases where multidimensional arrays are not appropriate. For example, there exist problems in which arrays are not used in order to model some multidimensional problem but simply to keep values that are used over and over again by the program. As an example, consider traditional sorting of an array of values. Sorting of numbers can also quite easily be performed in a multidimensional language \(^2\), but it does not appear as natural as the usual sorting algorithms that use imperative arrays. In general, we believe that the two different forms of arrays are complementary in nature, each one having its own successful application domains.

5 Comparison with Imperative Procedures

As discussed in Section 3, the definitions of the multidimensional lazy arrays can be compiled to a set of conventional procedures (in our case a set of Java methods). This brings us to the question of why not just use procedures instead of extending a traditional language with lazy arrays.

An important drawback of the use of procedures to describe multidimensional entities, is the fact that procedures can not in general be memoized. This is due to the fact that imperative programming languages are not in general referentially transparent. Two syntactically identical procedure calls in a program may correspond to two different returned values. Therefore, any attempt to memoize conventional procedure calls seems fruitless. On the other hand, memoization (through the use of the warehouse) plays a vital role in the performance of the lazy arrays approach. Table 4 and Figure 5 illustrate the performance of the procedure implementation of the “two teams” example discussed in previous sections. Notice that even for small values of the input, the time required to solve the problem is significant. Actually the complexity of computing \(p(i,j)\) for the specific problem in a recursive way without memoization is \(O(2^{i+j})\).

One possible solution to the above problem would be to appropriately mark the procedures used for describing multidimensional concepts. Then, the compiler would know that these are referentially transparent ones and could perform some type of memoization. However, this type of marking also means that the syntax of the source language has to be altered (slightly) and the memoization may require changes to the runtime of the language. The advantage of such an approach would be that the programmer would not have to learn yet another formalism (i.e. multidimensional languages) and would be able to program multidimensional applications using conventional procedures.

However, there exists another serious obstacle to the use of imperative
Table 4. Performance of Procedures without Memoization

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>84</td>
</tr>
<tr>
<td>13</td>
<td>328</td>
</tr>
<tr>
<td>14</td>
<td>1225</td>
</tr>
</tbody>
</table>

Figure 5. Performance of Procedures without Memoization

procedures for encoding multidimensional entities. Such an encoding is far less natural than a direct encoding in a multidimensional language. Using procedures, one has to carry around all the possible dimensions that are used, in contrast to the GLU definitions in which the contexts do not appear explicitly. Consider for example the following fragment of a GLU program:

dimensions x, y, z, t;
\[ a = b + c; \]
\[ b = \text{if } (d+e > 2*e) \text{ then } d+c \text{ else } 2*e; \]

Coding the above program using procedures, would result in something of the following form:

\[
\text{static double } a(\text{int } x, \text{ int } y, \text{ int } z, \text{ int } t) \{ \\
\text{ return } b(x, y, z, t) + c(x, y, z, t); \\
\}
\]
\[
\text{static double } b(\text{int } x, \text{ int } y, \text{ int } z, \text{ int } t) \{ \\
\text{ if } (d(x, y, z, t)+e(x, y, z, t) > 2*e(x, y, z, t)) \\
\text{ return } d(x, y, z, t)+c(x, y, z, t); \\
\text{ else } \text{return } 2*e(x, y, z, t); \\
\}
\]

Obviously, the use of procedures to describe multidimensional entities is generally cumbersome and inelegant. Things become even more difficult when one allows the multidimensional language to support user-defined functions. In such a case, the description using procedures becomes much more difficult. These issues are further discussed in Section 6 that follows.

6 Discussion

In this paper, we have presented a technique for amalgamating multidimensional and procedural programming languages in a controlled way. More specifically, we have demonstrated that multidimensional programs can be embedded into procedural programs as multidimensional lazy arrays. Such an approach offers benefits to both worlds: multidimensional languages can benefit from the extensive capabilities of traditional programming and procedural languages can use a different form of arrays which in many applications offers advantages over imperative ones. In particular, lazy arrays can be used in applications where not all elements of an array are needed in order to compute the desired output. Moreover, they are especially appropriate for problems that involve a large number of dimensions, and in which imperative arrays would face significant space problems.

There are certain aspects of the work presented in this paper that require further investigation and development. First of all, the subset of GLU considered is somewhat restrictive, in the sense that it only uses zero-order (i.e. nullary) definitions. It would be natural to ask whether functions can easily
be added to this subset. The answer is that first-order functions as well as a significant class of higher-order ones can be added to the language without any semantic complications. However, the implementation of the resulting multidimensional language would not be as straightforward as the language considered in this paper (it would require the translation into a branching time language\textsuperscript{13,14,15}).

Another useful extension would be the addition of \textit{multidimensional pattern matching}, which is similar in spirit to the pattern matching idea of modern functional languages\textsuperscript{12}. This would allow the more natural coding of applications. For example, the “two teams” problem of Section 2 would then be coded as follows:
\begin{verbatim}
  first_x next_y p = 1;
  next_x first_y p = 0;
  next_x next_y p = next_y p + next_x p;
\end{verbatim}
This syntax corresponds more closely to the recursive definition of $p$ given in Section 2, avoids the use of the \texttt{fby} operator and is in our opinion much more natural.

Finally, we believe that further experimentation is required in order to identify the nature of applications that require lazy instead of imperative arrays and vice-versa. One area which is rich in multidimensional problems that can form the basis of such an investigation, is the area of scientific computing. Recent work\textsuperscript{16} has shown that multidimensional languages are especially appropriate for describing scientific computing problems. It is our opinion however that lazy and imperative arrays are not competing notions but complementary ones, and a language for scientific computing based on both ideas would offer significant benefits.

Acknowledgments

The author would like to thank J. Plaice for many useful comments on an earlier version of the paper.

References


