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
Tel.: +30-651-97312
Fax.: +30-651-48131

Abstract

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 conventional 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.

Keywords: Multidimensional Programming, Imperative Arrays, Procedures.

Introduction

Multidimensional programming [1, 2] is a relatively 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 conventional 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 approach 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 conventional 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 following two sections of the paper give an introduction to multidimensional programming and to the basic technique (namely *eduction*) for evaluating multidimensional programs. The remaining sections describe how multidimensional languages can be embedded into conventional ones. Moreover, lazy multidimensional arrays are shown to compare favorably in many cases to the use of imperative arrays, imperative procedures and monolithic functional arrays. The paper concludes with possible future extensions of the proposed approach.

Multidimensional Programming

The development of multidimensional programming languages [1, 2, 5, 6] followed the development of temporal languages such as Lucid [7]. Lucid is built on the notion of time: the basic data value in a Lucid program is the *stream*, an infinite sequence of ordinary data values. One can think of such streams as entities that vary with time. In the following we give a brief presentation of Lucid, necessary for introducing the more demanding paradigm of multidimensional programming.

To illustrate the ideas behind Lucid, consider the following statement defining the (infinite) sequence of natural numbers:

```
nat  =  0 fby (nat+1);
```

Notice the use of the operator `fby` (read *followed by*) in the above statement. The semantics of the operator will be given shortly. The declarative reading of the statement is as follows:

> The first value of the natural number sequence is 0. Each next value in the sequence can be produced by adding 1 to the previous value of the sequence.

The above statement defines the natural number sequence in a formal way, which however differs from the usual mathematical definition which involves the use of a *time index*:

$$\begin{aligned} nat_0 &= 0 \\ nat_{t+1} &= nat_t + 1 \end{aligned}$$

The Lucid statement avoids the explicit use of the time index. The sequence of natural numbers is defined using the temporal operator `fby` rather than using subscripts (i.e. indices).

The statements of a Lucid program are equations defining individual and function variables required to be true at every context (time point). Ordinary data operations (such as `+`, `*` and `if-then-else`) are defined in a pointwise way. This means, for example, that the value of `a + b` at time-point t is the (ordinary) sum of the values of `a` and `b` at the same time-point t. The basic Lucid operators are `first`, `next`, and `fby`. The operator `first` switches us to time point 0, `next` takes us from t to $t+1$. The operator `fby` takes us back from $t+1$ to t (giving us the value of its second operand at that point) or from 0 to 0 (giving us the value of its first operand at time point 0).

Let a and b be Lucid sequences. Then, the above ideas are formalized by the following semantic equations:

$$\begin{aligned} (a+b)_t &= a_t + b_t \\ first(a)_t &= a_0 \\ next(a)_t &= a_{t+1} \\ (a\ fby\ b)_t &= \begin{cases} a_0 & \text{if } t = 0 \\ b_{t-1} & \text{if } t > 0 \end{cases} \end{aligned}$$

The `fby` operator allows us to express many iterative algorithms concisely; the first operand of `fby` gives the initial value, and the second specifies the way in which each next value is determined by the current value. For example, the following program computes the stream $\langle 1, 1, 2, 3, 5, \ldots \rangle$ of all Fibonacci numbers:

```
fib  =  1 fby (fib+g)
g    =  0 fby fib
```

The Lucid language also supports user-defined functions that operate on sequences and return sequences as results. However, functions are not considered in the main part of the paper (see the discussion on future extensions in the concluding section).

Lucid has recently been extended to support more than one-dimensional entities [1, 2]. This extended language was named GLU and its first implementation was developed in SRI. GLU allows the user to declare dimensions and to define multidimensional entities that vary across these dimensions. So, a two-dimensional entity can be thought as an infinite table, a three-dimensional one as a cube extending infinitely across the three dimensions, and so on. One can perform various operations with arguments such higher-dimensional entities or even define functions that take such entities as parameters and return new ones as results. Moreover, the language supports generalized operators that work along each dimension.

As an illustration of the expressive power of GLU, we consider a simple program which first appeared in [8] and which models heat transfer in solids. More specifically, consider a long thin metal rod which initially has temperature 0 and which touches a heat source of temperature 100. The problem is to determine the temperature at a point of the rod after a number of time points has passed (we assume that the rod has been divided into a large number of individual pieces each of which has a different temperature). The temperature $T_{t,s}$ at time t and at space (i.e. piece) s is given by the following recurrence relation:

$$\begin{array}{lcl} T_{0,s+1} & = & 0 \\ T_{t,0} & = & 100 \\ T_{t+1,s+1} & = & k * T_{t,s} - (1 - 2 * k) * T_{t,s+1} + k * T_{t,s+2} \end{array}$$

where k is a constant that depends upon the material of the rod. The above equations can easily be coded in GLU as follows[1]:

```
dimensions t, s;
T = 100 fby_s (0 fby_t (k*T-(1-2*k)*(next_s T)+k*(next_s next_s T)));
k = 0.3;
```

The first line in the above example declares that two dimensions are used, namely `t` and `s`. Notice the new operators that appear in the definition of `T`: `fby_t`, `next_s` and `fby_s`. These operators are straightforward generalizations of the corresponding Lucid operators. For example, if a and b are entities varying across the `s` and `t` dimensions, the semantics of `fby_s` is:

$$(a \ fby\_s \ b)_{t,s} = \begin{cases} a_{t,0} & \text{if } s = 0 \\ b_{t,s-1} & \text{if } s > 0 \end{cases}$$

and the semantics of `next_s` is:

$$next\_s(a)_{t,s} = a_{t,s+1}$$

The semantic equations for the other multidimensional operators are similar in nature.

It should be noted here that the example described above is expressed very compactly and naturally in the GLU language. The solution of such a problem in a conventional imperative language would most probably require the use of a three-dimensional array together with the use of nested `for` loops in order to fill the entries of the array.

The next section describes the main technique that is used in order to evaluate multidimensional programs.

[1] 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.

The Evaluation of Multidimensional Programs

The basic technique that has been extensively used for computing the output of temporal and more generally multidimensional programs is known as *eduction* [7]. The main idea behind eduction is that the evaluator of a program *demands* the value of a given variable at given coordinates; each such demand generates corresponding demands for other variables of the program in possibly different coordinates. In general, eduction follows the semantic equations of the operators in order to compute the final output of a given program. The demand-driven nature of eduction makes it a lazy implementation technique: it does not compute things that are not needed in the calculation of the desired result.

The above ideas are illustrated in the example that follows. We first consider the simpler case of temporal programs. Assume we want to compute the output of the `fib` definition at time point 2. By $EVAL$ we denote the evaluator (interpreter) of the temporal language.

$$
\begin{array}{ll}
 & EVAL(\texttt{fib}, 2) \\
= & EVAL(\texttt{(1 fby (fib+g))}, 2) \\
= & EVAL(\texttt{(fib+g)}, 1) \\
= & EVAL(\texttt{fib}, 1) + EVAL(\texttt{g}, 1) \\
= & EVAL(\texttt{(1 fby (fib+g))}, 1) + EVAL(\texttt{(0 fby fib)}, 1) \\
= & EVAL(\texttt{(fib+g)}, 0) + EVAL(\texttt{fib}, 0) \\
= & EVAL(\texttt{fib}, 0) + EVAL(\texttt{g}, 0) + EVAL(\texttt{(1 fby (fib+g))}, 0) \\
= & EVAL(\texttt{(1 fby (fib+g))}, 0) + EVAL(\texttt{(0 fby fib)}, 0) + 1 \\
= & 1 + 0 + 1 \\
= & 2
\end{array}
$$

A careful examination of the above steps reveals that there exist computations that take place in more than one occasions. For example, the value of `fib` at time 0 is demanded twice. This suggests that an efficient implementation of a temporal language should store values of variables that have been computed under specific contexts, so as that these results will be available if demanded later during evaluation. The process of storing intermediate results is known as *warehousing* and the data structure that is used for this purpose (usually a hash-table) is known as the *warehouse*. Maintaining the warehouse during execution is not always an easy task: a garbage collection is often required as the table tends to get full with old entries (which may be useless for future calculations). Many techniques have been devised for managing the warehouse component of the implementation [7, 9, 10]. A variation of the technique used in [9] has been used in the implementation of the lazy arrays approach proposed in this paper (more details are given in a later section).

When one considers multidimensional programs, things become more complicated in terms of implementation. The evaluator has to compute the values of variables under more than one dimensions. Moreover, the warehouse has to be more general as it now stores the values of variables under more complicated contexts. To illustrate these ideas, consider the following example which computes the value of the multidimensional entity `T` introduced in the previous section at time point 1 and at space coordinate 1:

$$
\begin{array}{ll}
 & EVAL(\texttt{T}, 1, 1) \\
= & EVAL(\texttt{100 fby\_s (0 fby\_t (k*T-(1-2*k)*(next\_s T)+k*(next\_s next\_s T)))}, 1, 1) \\
= & EVAL(\texttt{k*T-(1-2*k)*(next\_s T)+k*(next\_s next\_s T)}, 0, 0) \\
= & 0.3 * EVAL(\texttt{T}, 0, 0) - (1 - 2 * 0.3) * EVAL(\texttt{T}, 0, 1) + 0.3 * EVAL(\texttt{T}, 0, 2) \\
= & 30
\end{array}
$$

As it can be easily verified, the eductive implementation of multidimensional programs involves extensive recalculations (for an illustration of this, consider the calculation of the value of `T` at time 3 and space 3). The warehouse idea is now even more necessary than in the single dimensional (temporal) case. The warehouse is usually implemented as an open hash table. More details on its structure and implementation will be given in subsequent sections.

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. We assume that the calling mechanism in such cases is call-by-value. A call-by-reference mechanism is not allowed (passing as a parameter the location of a lazy array element has no meaning, as the particular element is not supposed to change).

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* [11]. 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:

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

The above recurrence can be coded in GLU as follows:

```
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:

```
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.

It should be emphasized here that the array indices in the conventional part of the program, correspond to the dimensions of the multidimensional program in the order that these are declared in the beginning of the multidimensional array definition section. So, for example, an expression `p[5][10]` in the program would mean that the value of the multidimensional entity `p` is demanded at `x` coordinate 5 and `y` coordinate 10. Therefore, the meaning of an array expression in the conventional part of the program only depends on the order of the declared dimensions (and not on how the particular multidimensional entity is defined).

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.

On the other hand, the embedding described in this section offers immediate practical benefits for multidimensional languages. Figure 1 illustrates the visualization of the demand patterns during the eduction procedure of the temperature program (and more specifically for the computation of the value `T[8][6]`). Such simple visualization applications are easy to code once given the implementation of the mixed language that was introduced in this section.

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 ex-

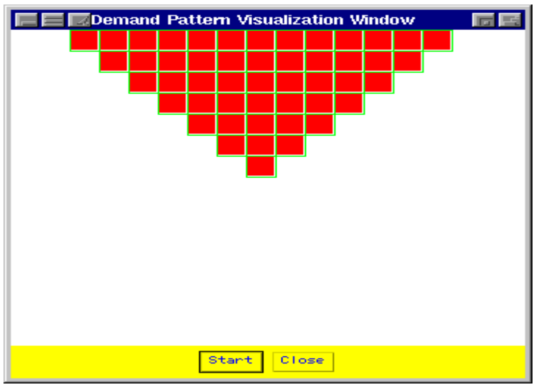

Figure 1: Demand Pattern in the Execution of a Multidimensional Program

clusively 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 [12]).

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. At present, the only type supported by the multidimensional language is `double`, although other types can easily be added. 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 [13, 14].

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. Each such section starts with the `dimensions` declaration and all the variables defined in the section are assumed to vary along these dimensions (see also the discussion on *dimensionality analysis* later in the paper). 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:

```
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)`. Again, we note that the indices in the array reference as well as the parameters of the resulting method invocation, correspond to the dimensions of the multidimensional program as these are declared in the beginning of the multidimensional array definition section.

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:

```
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 [15]. For a two-dimensional program the warehouse has the structure shown in Figure 2. 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). It should be noted here that the constructor that corresponds to the `Key` part of the entry, has as parameters *all* the dimensions that have been declared in the multidimensional array definition section. In other words, only one constructor for `Key` is generated, because all entries in the warehouse are assumed to vary under all the declared dimensions. The current implementation first checks the declared dimensions and then generates the constructor code that corresponds

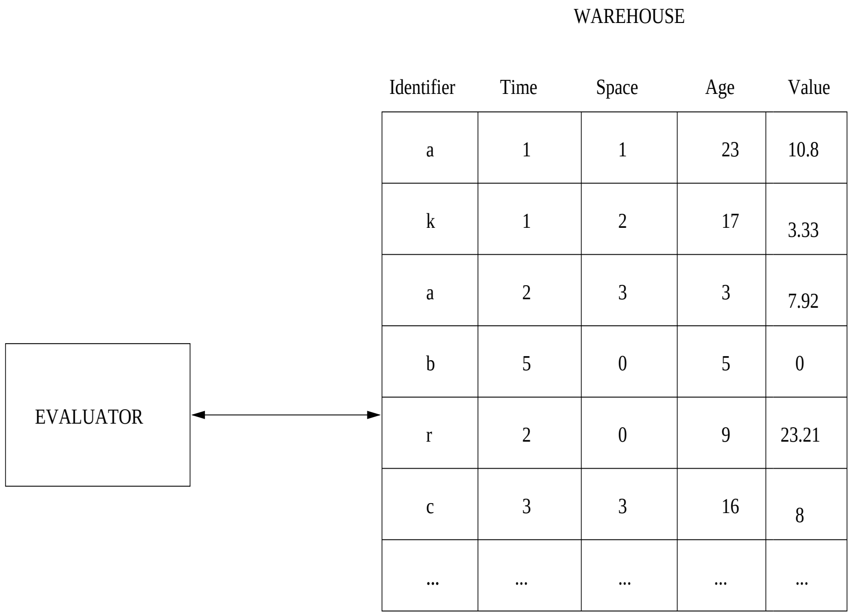

Figure 2: The Structure of the Warehouse

to these dimensions. An implementation that would use *dimensionality analysis* (see end of this section) would have to use a different scheme for implementing the `Key` concept. Also, we should mention here that the `result` field of the `Value` class is of type double (as mentioned before, in the present implementation this is the only type used for elements of lazy arrays).

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 proposed in [9]. 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[2]. The `age` field of the new entry is then set to the new value of `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 *size* S which is the number of buckets of the warehouse table (usually a prime number), the *garbage collection limit* G which is the number of entries in the warehouse after which the garbage collection algorithm starts executing, and the *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 any form of dimensionality analysis [16], 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

[2] In [9], the age of an entry is counted by the number of garbage collections that the entry has survived, and it is usually a small integer.

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. In the present implementation, all variables are assumed to vary across all dimensions that are specified by the `dimensions` declaration in the beginning of the multidimensional array definition section.

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

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 (given in [11]) for the "two teams" problem:

```
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

	Time (in sec)	
Input Size	Lazy Arrays	Imperative Arrays
100	5	1
150	11	2
200	20	3
250	31	5
300	52	7
350	65	9
400	90	12
450	112	15
500	129	19
550	173	23
600	202	-
650	239	-

Table 1: Time Comparison Between Lazy and Imperative Arrays

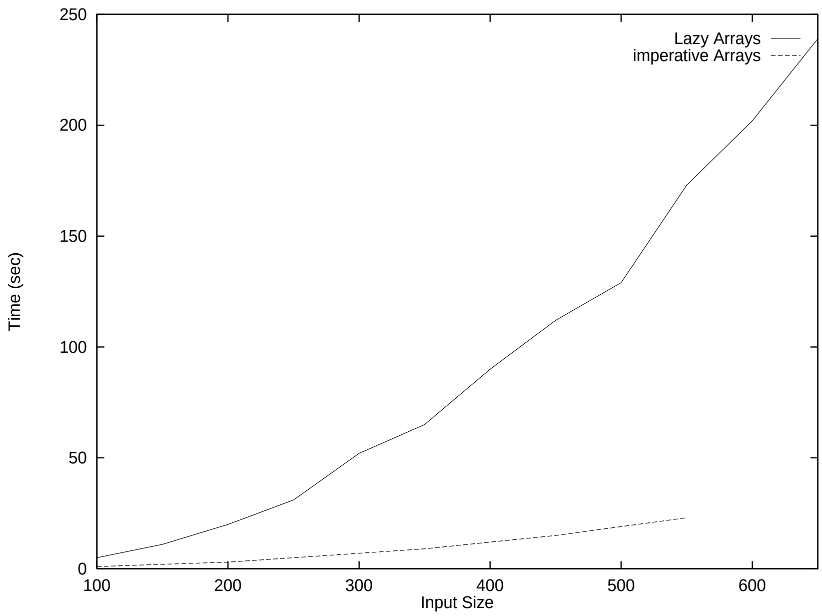

Figure 3: Imperative Arrays vs Lazy Arrays: Time Comparison

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 (given in the previous section), is shown in Table 1 and Figure 3. 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)[3]. 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 described in the previous section. 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 small fragment of them will ever become necessary (see for example the dependencies for the calculation of `T[8][6]` in Figure 1). This discussion is closely related to the old debate between lazy and eager functional programming languages [17]. 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 conventional 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 conventional imperative arrays. Table 2 and Figure 4 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 5 which present the execution times for the calculation of `p[200][200]` (ten times) with varying warehouse size. More specifically, S = varying (and usually a prime), $G = (2/3) * S$, and $R = 0.5$. Notice that performance is affected only when S gets very low (in which case many useful entries are thrown away during garbage collection and have to be recomputed).

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

[3]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.

	Space (in data items)	
Input Size	Lazy Arrays	Imperative Arrays
100	36997	10201
150	36997	22801
200	36997	40401
250	36997	63001
300	36997	90601
350	36997	123201
400	36997	160801
450	36997	203401
500	36997	251001

Table 2: Space Comparison Between Lazy and Imperative Arrays

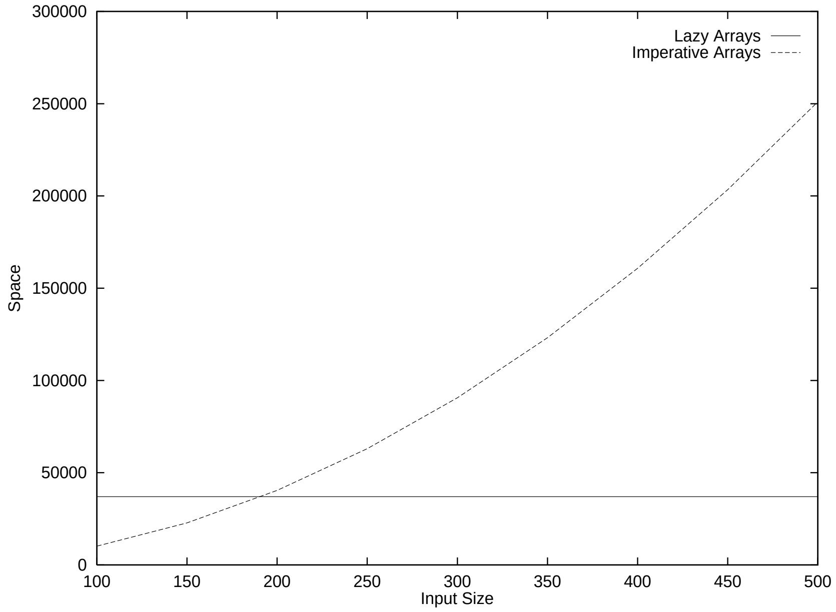

Figure 4: Imperative Arrays vs Lazy Arrays: Space Comparison

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][page20], 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.

Comparison with Imperative Procedures

As discussed in previous sections, 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 conventional language with

Warehouse Size (S)	*Time (sec)*
443	355
461	259
503	291
547	283
599	364
601	19
607	18
919	20
1847	19
3697	20
7393	23
14783	23

Table 3: Execution Time as a Function of Warehouse Size

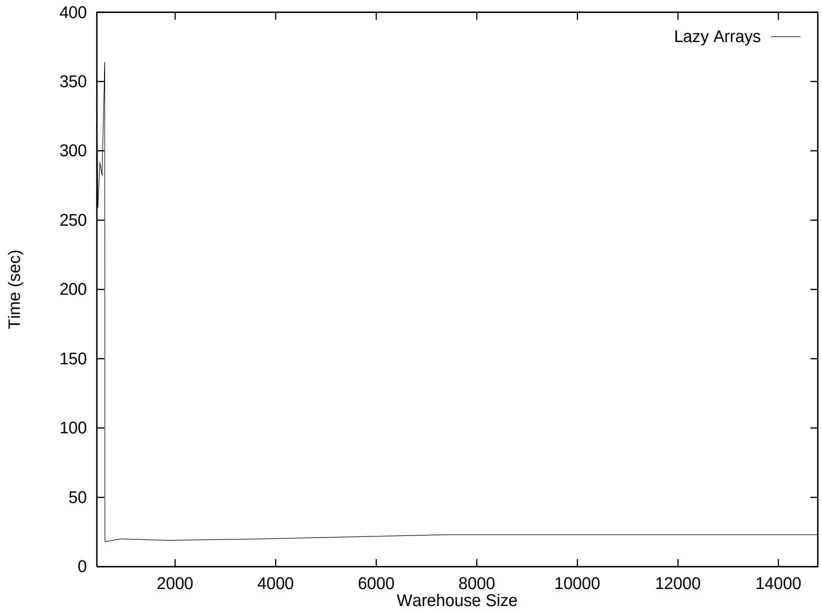

Figure 5: Execution Time as a Function of Warehouse Size

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 6 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

Input Size	*Time (sec)*
8	0
9	1
10	5
11	20
12	84
13	328
14	1225

Table 4: Performance of Procedures without Memoization

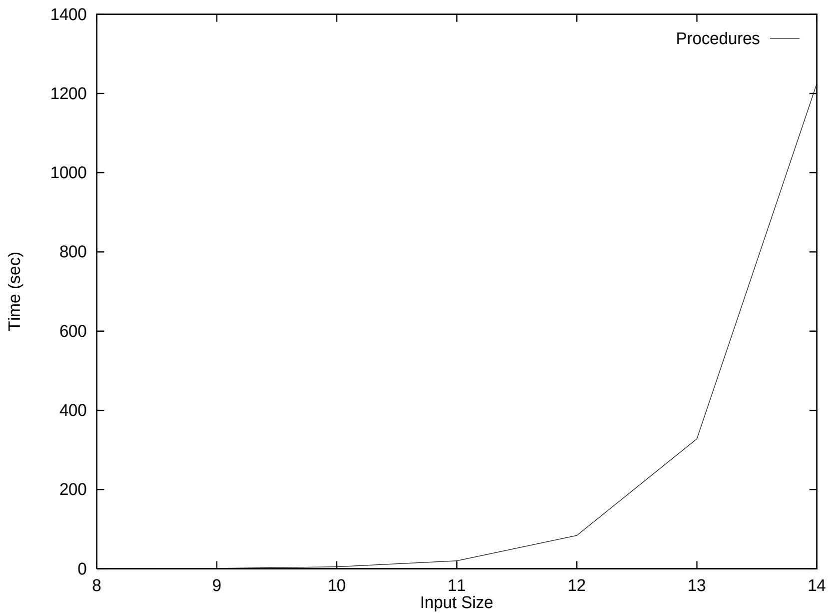

Figure 6: Performance of Procedures without Memoization

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 only advantage of such an approach would be that the programmer would not have to learn yet another formalism (i.e. multidimenional languages) and would be able to program multidimensional applications using conventional procedures.

However, there exists another serious obstacle to the use of imperative 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 = if (d+e > 2*e) then d+c else 2*e;
c = 5;
          ...
```

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

```
                            ...
static double  a(int x, int y, int z, int t) {
      return b(x, y, z, t) + c(x, y, z, t);
}
static double  b(int x, int y, int z, int t) {
      if (d(x, y, z, t)+e(x, y, z, t) > 2*e(x, y, z, t))
             return d(x, y, z, t)+c(x, y, z, t);
      else   return 2*e(x, y, z, t);
}
static double  c(int x, int y, int z, int t) {
      return 5;
}
                            ...
```

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 the concluding section.

In general, we believe that conventional procedures are inappropriate for describing multidimensional concepts in a concise way. However, they can be used, as discussed in previous sections, to provide a viable implementation of the lazy multidimensional arrays idea.

Comparison with Functional Arrays

The approach proposed in this paper embeds a simple multidimensional language (a subset of GLU) in a procedural programming language. One could possibly argue that other formalisms, which are conceptually similar to multidimensional programs, could also be successfully embedded in conventional languages. One such possible formalism could be a more mainstream functional language that supports some form of functional arrays. Many modern functional languages (e.g. Haskell [18], Id [19], Sisal [20]) support what is usually called *monolithic* arrays, a form of arrays that are constructed once and remain unchanged during the execution of the program.

Monolithic arrays are conceptually similar to the multidimensional GLU programs adopted for the purposes of this paper. Both formalisms are functional in nature and one could suggest that it is possible and potentially useful to embed functional arrays in procedural programming languages. There are however important differences between monolithic functional arrays and multidimensional programs (see for example [21][pages 23-30] for a detailed discussion on this topic). The main such difference is that monolithic arrays are generally static entities that do not appear to differ in philosophy from ordinary recursive functions, while GLU programs are inherently dynamic and model more successfully many multidimensional concepts. Consider for example the monolithic array definition of the usual Fibonacci program in Haskell [22] (the infix operator `!` is the array indexing operator):

```
fibs n = a where a = array (0,n) ([(0, 0), (1,1)] ++
                                  [(i,a!(i-2)+a!(i-1)) | i <- [2..n]])
```

The corresponding program in GLU is much more natural and does not use any form of indexing:

```
dimensions x;
fibs = 0 fby_x (1 fby_x (fibs + next_x fibs))
```

Monolithic arrays become even more cumbersome when one considers the definition of entities that vary across more than one dimensions. Consider for example the so-called *wavefront* matrix, in which elements of the first row and first column have the value 1 and each other element has a value that is the sum of the elements west, northwest and north. The solution using functional arrays is the following [22]:

```
a = array ((1,1),(n,n))
      ([((1,j), 1) | j <- [1..n]] ++
       [((i,1), 1) | i <- [2..n]] ++
       [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                    | i <- [2..n], j <- [2..n]])
```

The corresponding GLU program for modeling the same problem, is:

```
dimensions x, y;
a = 1 fby_x (1 fby_y (next_x(a) + a + next_y(a)));
```

As the above example illustrates, it is generally conceptually simpler to code a multidimensional application in a language that directly supports multidimensionality, rather than in a language that simply supports arrays as an extra feature. In other words, functional arrays could potentially be embedded in a conventional procedural language, but they would offer restricted expressive capabilities for describing multidimensional concepts.

Monolithic arrays is the most common form of arrays supported by modern functional languages. Another approach to functional arrays is the so-called *incremental* approach. Incremental arrays actually simulate the destructive assignment operation which is possible in imperative arrays. Logically, one can imagine that in incremental arrays, every time that a particular element changes value, the whole array is replicated with the particular value changed. Many sophisticated schemes have been devised for avoiding excessive copying (see [23] for a detailed presentation of the corresponding literature). The GLU programming language does not support updating of values (although this could possibly be added along the same lines as with mainstream functional languages). It is questionable however whether support for updates (which is mainly a characteristic of imperative arrays) would add to the expressive power of multidimensional languages. This is certainly an interesting research question which however requires further (and extensive) experimentation with multidimensional languages and their expressive capabilities.

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 conventional programming and procedural languages can use a different form of arrays which in many applications offer 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 (see for example [24, 25]).

Another useful extension would be the addition of *multidimensional pattern matching*, which is similar in spirit to the pattern matching idea of modern functional languages [17]. This would allow the more natural coding of applications. For example, the "two teams" problem would then be coded as follows:

```
first_x next_y p = 1;
next_x first_y p = 0;
next_x next_y  p = next_y p + next_x p;
```

This syntax corresponds more closely to the corresponding recursive definition of `p`, avoids the use of the `fby` operator and is in our opinion much more natural. Of course, the above three parts of the definition of `p` would at compile time be transformed into a single one; for this purpose, similar techniques as those employed in conventional functional languages could be used.

As one of the reviewers remarked, in the proposed approach one can only define lazy arrays that are static (i.e. global to all objects created from a class). We believe that it should not be difficult (at least from a semantic point of view) to support non-static lazy arrays. This could be accomplished by allowing the GLU definitions to contain free variables (i.e. variables that receive values externally from the GLU program). The free variables would take a constant value during object creation, in the constructor of the class. The idea of non-static lazy arrays is certainly a genuine extension of the approach proposed in this paper. However, the implementation details are certainly not straightforward. For example, in the present approach, we use a warehouse structure for each class. This would probably not be sufficient if each object was allowed to have its own version of a lazy array. However, extensions like this one, appear to be semantically clear and useful and will further be considered in future extensions of the system.

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 [26] has shown that multidimensional languages are especially appropriate for descibing scientific computing problems. It is our opinion however that lazy and imperative arrays are not competing notions but complementaty ones, and a language for scientific computing based on both ideas would offer significant benefits.

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

References

[1] E. A. Ashcroft, A. A. Faustini, and R. Jagannathan. An intensional language for parallel applications programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

[2] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional programming.* Oxford University Press, 1995.

[3] Iskender Agi. GLU for Multidimensional Signal Processing. In M. Orgun and E. Ashcroft, editors, *Intensional Programming I*, pages 135–148. World Scientific, 1996.

[4] P. Rao and R. Jagannathan. Developing Scientific Applications in GLU. In *Proceedings of the Seventh International Symposium on Lucid and Intensional Programming*, pages 45–52, 1994.

[5] W. Du and W. W. Wadge. A 3D Spreadsheet Based on Intensional Logic. *IEEE Software*, pages 78–89, July 1990.

[6] W. Du and W. W. Wadge. The Eductive Implementation of a Three-dimensional Spreadsheet. *Software-Practice and Experience*, 20(11):1097–1114, November 1990.

[7] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[8] A. A. Faustini and W. W. Wadge. Intensional Programming. Technical Report DCS-55-IR, Department of Computer Science, University of Victoria, 1986.

[9] A. Faustini and W. W. Wadge. An Eductive Interpreter for the Language pLucid. In *Proceedings of the SIGPLAN 87 Conference on Interpreters and Interpretive Techniques (SIGPLAN Notices 22(7))*, pages 86–91, 1987.

[10] R. Bagai. Compilation of the Dataflow Language Lucid. Master's thesis, Department of Computer Science, University of Victoria, 1986.

[11] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987.

[12] P. Rondogiannis and W. W. Wadge. Higher-Order Dataflow and its Implementation on Stock Hardware. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, 1994.

[13] W. W. Wadge. Tense Logic Programming: a Respectable Alternative. In *Proceedings of the First International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.

[14] M. A. Orgun and W. Du. Multi-dimensional logic programming: theoretical foundations. *Theoretical Computer Science*, 158(2):319–345, 1997.

[15] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[16] C. Dodd. Rank Analysis in the GLU Compiler. In M. Orgun and E. Ashcroft, editors, *Intensional Programming I*, pages 76–82. World Scientific, 1996.

[17] A. Field and P. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[18] P. Hudak, S. Peyton Jones, P. Wadler, Arvind, B. Bontel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughs, T. Johnson, R. Kieburtz, W. Partain, and J. Peterson. Report on the Functional Programming Language Haskell (version 1.2). *ACM SIGPLAN Notices*, 27(5), July 1992.

[19] K. Ekanadham. A Perspective on Id. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 197–253. ACM Press, 1991.

[20] S. Skedzielewski. Sisal. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. ACM Press, 1991.

[21] A. A. Faustini E. A. Ashcroft and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

[22] P. Hudak, J. Peterson, and J. Fasel. A Gentle Introduction to Haskell. *Journal of Functional Programming*, 1997. (A previous version of this report appeared in ACM SIGPLAN Notices, 27(5), 1992).

[23] M. O'Neill and W. Burton. A New Method for Functional Arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997.

[24] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.

[25] P. Rondogiannis and W. W. Wadge. Higher-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 1999. (to appear).

[26] J. Paquet. *Intensional Scientific Programming*. PhD thesis, Département d'informatique, Université Laval, Québec, Canada, 1999. (in preparation).