# Dimensional Analysis of Data Flow Programs

by

**Abdulmonem Ibrahim Shennat**

Bachelor of Electronics and Communication Engineering, from the Faculty of Industrial Technology, Misrata, Libya, 2000
Master's Degree of Information & Communication Technology (ICT), from University Utara Malaysia, Sintok, Kedah, Malaysia, 2008

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In the Department of Computer Science

# Dimensional Analysis of Data Flow Programs

by

**Abdulmonem Ibrahim Shennat**

Bachelor of Electronics and Communication Engineering, from the Faculty of Industrial Technology, Misrata, Libya, 2000
Master's Degree of Information & Communication Technology (ICT), from University Utara Malaysia, Sintok, Kedah, Malaysia, 2008

**Supervisory Committee**

Dr. William Wadge, Department of Computer Science
**Supervisor**

Dr. Alex Kuo, Health Information Science Science
**Co-Supervisor**

Dr. Alex Thomo, Department of Computer Science
**Departmental Member**

Dr. Manolis Gergatsoulis, Department of Archives, Library Sciences and Museology, Ionian University
**External Examiner**

# Abstract

Our main objective is to design Dimensional Analysis (DA) algorithms for the multidimensional dialect PyLucid of Lucid, the equational data flow language.

The significance is that the DA is indispensable for an efficient implementation of multidimensional Lucid and should aid the implementation of other data flow systems, such as Google's TensorFlow.

Data flow is a form of computation in which components of multidimensional datasets (MDDs) travel on communication lines in a network of processing stations. Each processing station incrementally transforms its input MDDs to its output, another (possibly very different) MDD.

MDDs are very common in Health Information Systems and data science in general. An important concept is that of relevant dimension. A dimension is relevant if the coordinate of that dimension is required to extract a value. It is very important that in calculating with MDDs we avoid non-relevant dimensions, otherwise we duplicate entries (say, in a cache) and waste time and space.

Suppose, for example, that we are measuring rainfall in a region. Each individual measurement (say, of an hour's worth of rain) is determined by location (one dimension), day, (a second dimension) and time of day (a third dimension). All three dimensions are a *priori* relevant.

Now suppose we want the total rainfall for each day. In this MDD (call it N) the relevant dimensions are location and day, but time of day is no longer relevant and must be removed. Normally this is done manually. However, can this process be automated?

We answer this question affirmatively by devising and testing algorithms that produce useful and reliable approximations (specifically, upper bounds) for the dimensionalities of the variables in a program. By dimensionality we mean the set of relevant dimensions. For example, if M is the MDD of raw rain measurements, its dimensionality is {location, day, hour}, and that of N is {location, day}. Note

that the dimensionality is more than just the *rank*, which is simply the number of dimensions.

Previously, there's extensive research on dataflow itself, which we summarize. However, an exhaustive literature search uncovered no relevant previous DA work other than that of the GLU (Granular Lucid) project in the 90s. Unfortunately the GLU project was funded privately and remains proprietary – not even the author has access to it.

Our methodology is that we proceeded incrementally, solving increasingly difficult instances of DA corresponding to increasingly sophisticated language features. We solved the case of one dimension (time), two dimensions (time and space), and multiple dimensions.

We also solved the difficult problem (which the GLU team never solved) of determining the dimensionality of programs that include user defined functions, including recursively defined functions. We do this by adapting the PyLucid interpreter (to produce the DAM interpreter) to evaluating the entire program over the (finite) domain of dimensionalities.

As a result, the experimentally validated algorithms in our dissertation can produce useful upper bounds for the dimensionalities of the variables in multidimensional PyLucid programs. That also includes those with user-defined functions.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgements

First of all, I would like to gratefully and sincerely thank my beloved parents for their unlimited giving to make me a good person who is always enthusiastic about acquiring more knowledge.

I would also like to express my deep gratitude and thanks to Dr. William Wadge, my research supervisor, for his patient guidance, enthusiastic encouragement, and beneficial critiques of this research work. Even though he faced health issues, he did not stop helping me throughout this research endeavor in so many ways. Bill has also created a supportive and encouraging space for scientific inquiry that has enabled me to identify and pursue my research goals.

My grateful thank is also extended to Dr. Alex Kuo, my research co-supervisor, for his advice and assistance for different things I needed for my courses and research, as well as to Dr. Alex Thomo for his support and encouragement throughout my study.

My special gratitude also goes to Mr. Howard Breen and Mr. Jonathan Lapthorne. On several occasions, they have generously volunteered their time, attentive listening, and insightful reflections on the most fundamental research questions, including the academic writing skills that have helped me clarify the purpose, scope, and goal of my study.

Finally, I am grateful to Dr. Sue Whitesides, Dr. Sudhakar Ganti, Dr. Lisa Lix, Dr. Rich Little, Dr. Omar Alaqeeli, and the professors and grad students at the University of Victoria; and who are involved in the VADA program from the University of Manitoba. They have imparted their knowledge to me in the forms of lectures, discussions, and or different ways for problem-solving.

# Chapter 1

# Introductions

This introductory chapter presents an overview of the Dimensional Analysis of Data-Flow Programs (DADFP) and the main objective of DADFP, including a significant example. It also explains the basics of the Data Flow Programming Language Lucid, the PyLucid interpreter, and other topics that are important for understanding the principles of DADFP. Lastly, this chapter illustrates our system, DAM, along with a brief overview of formulas and definitions for each algorithm.

## 1.1 Introduction

A multidimensional data set is one in which the individual values are indexed by one, two, or many dimensions/parameters. For example, the measurement of an hour's rainfall can depend on the location (two coordinates), the date (three more dimensions) such as year, month, and day, and the time, which is the sixth dimension [29]. Also, multidimensional data is very common in healthcare systems. There are numerous Multidimensional DataBase (MDB) systems and approaches in different areas in the industry. For example, Online Analytical Processing (OLAP) works also based on a multidimensional model [10] [19].

In some cases, certain dimensions may not be needed in a dataset. For example,

suppose we accumulate hourly rainfall into daily rainfall. Then the time (of day) is not needed to specify a value. In that case, we say that time of day is now an irrelevant dimension. Thus, we need to know which dimensions are irrelevant because retaining them causes duplicate entries and is very inefficient[14].
Normally a database manager is responsible for identifying and removing irrelevant dimensions while retaining the relevant ones. Our research aims to automate this process, at least for data sets produced by data flow programs.[7]

## 1.2 The Main Objective

Our main objective is to create Dimensional Analysis algorithms for the equational data flow language Lucid that discover which coordinates are required for the evaluation of each particular variable in a data flow program and are relevant or irrelevant [1].
To illustrate the phenomenon of irrelevance, we consider a small data set of blood sugar measurements. Three parameters can determine these measurements: patient, physician, and finger. Figure 1.1 illustrates these measurements which give three possible values.

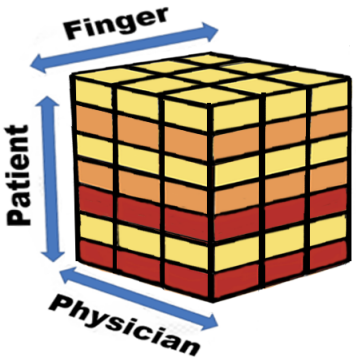


Figure 1.1: Blood-sugar Measurements.

The yellow color represents high blood sugar, orange, for average blood sugar, red, which shows low blood sugar. Also, the dimensions patient, physician and finger form three coordinates. The physician and finger dimensions have three possible values each but the patient dimension has (in our example) seven possible ordinates (possible coordinate values).

For example, Smith a possible coordinate the physician dimension; Baker and James are for the patient dimension; and the thumb is for the finger dimension. The patient dimension should be the only relevant dimension for this data set [36]. Consequently, every multidimensional data set has relevant and (possibly) irrelevant dimensions.

In this dissertation, we create the model, DAM, which has four algorithms or stages, to figure out the relevant dimensions in a data flow program.

## 1.3 Background

The first appearance of the data flow programming language Lucid was in 1976. It was a functional language in which every data object was a stream, a sequence of

values. In 1985 Bill Wadge and Ed Ashcroft presented Lucid as a Data Flow Programming Language. All Lucid operations mapped streams into streams. Therefore, a program in Lucid is simply an expression combined with definitions of the transformation and data referred to in the expression. Lucid is a data flow language in which the variables denote the sequences of data values passing between the actors, which correspond to the functions and operators of the language. The output of that program is simply the data that is denoted by the program as an expression, the value of the program. [35].

### 1.3.1 Statements in Lucid

Lucid program statements are simply equations. They have more in common with high school algebra than with the C or JAVA programming languages. Usually, programmers in an imperative language are concerned primarily with what a machine is required to perform, rather than the data that the machine is required to produce.

Equations in Lucid are different. For example, some of the equations appear conventional, such as:

```
x = a + b
```

While others appear unconventional, for example:

```
d = x - next x
```

Other equations in Lucid use mysterious "temporal" operators not found in conventional languages, e.g.:

```
i = 1 fby i + 1
```

Traditional equations that look like assignments, such as:

```
k = 2*k + 1
```

are allowed but produce completely unexpected results because they usually have no solutions except those which specify nonterminating computations. The definition in Lucid is the only kind of statement; there are no read or write statements and no control statements of any kind. These are excluded because they would block a dataflow implementation. Programmers can define their own functions in Lucid. However, Lucid can successfully extend the data flow methodology. In other words, Lucid can free data flow from its dependence on imperative languages like the UNIX shell language itself, which allows commands too. [13] [5].

### 1.3.2 Lucid Programs

A simple Lucid program consists of definitions of variables representing streams. These definitions may be recursive; for example, the series of Fibonacci numbers below:

```
fib = 1 fby (1 fby (fib + next fib))
```

Here, `next` takes a stream and discards the first element; while `fby`, written as an infix operator, takes two streams and produces a resulting stream that consists of the first element of the first stream followed by the whole second stream. It is easy to see that the first two elements of `fib` are 1; also, it can be seen that element $n + 2$ is equal to the sum of elements $n$ and $n + 1$. Moreover, in Lucid the programmer can think of some variables as denoting stored values that are repeatedly modified in the course of the computation.

For example, the equations have `fby`, which is the node that primes the pump, here with the number 1:

```
i = 1 fby i + 1
fac = 1 fby fac * i
```

These can be understood as specifying a data flow network, but in this particular case, the data flow view is somewhat unnatural. Equations like the above are

much easily interpreted in terms of an iterative activity, that initializes both `i` and `fac` to 1, then repeatedly updates these values. Lucid, in particular, has one great advantage over other data flow languages in that the programmer can think in terms of other operational concepts and can understand some statements as specifying an iterative algorithm, as a result. [35].

### 1.3.3 The basic principles of data flow

The concept of Data Flow in terms of the semantic model allows the concurrent execution of non-interfering program parts, at the same time, and the language for representing computational procedures based on that idea. Also, a simple data flow model could be developed as a program graph that should be reviewed and analyzed as a form of parallel program schema [21].

A data flow network is a diagram that contains communication channels (arcs) down which data tokens travel; the nodes are the processing stations. Figure 1.2 illustrates a simple data flow network that produces the sequence $1, 2, 3, 5, 8, ...$ of Fibonacci numbers. These nets are continuously operating devices. This network illustrates the use of several important nodes.

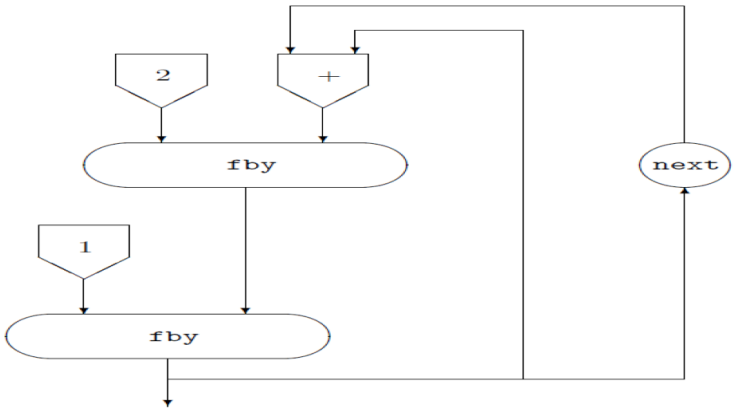


Figure 1.2: A Simple Data Flow Network.

The simplest are those like the one labelled '+' that correspond to ordinary

operations on data items. The '+' node continuously waits for the arrival of tokens on its input lines; the moment that there are tokens on both lines, they are consumed and a token representing the sum of the two tokens is then sent down the output line. As with the other nodes, input on the different lines need not arrive simultaneously or even at the same rate, and tokens awaiting processing queue on the arcs. A special case of this kind is nodes that are constant, nodes with no input lines. The node labelled '2' simply produces an endless stream of tokens representing the number 2.

The remaining nodes do not process tokens but just move them around. The '`next`' node rejects the first token which arrives but passes the rest on. The '`fby`' (followed by) node waits for the first token to arrive on its left input, passes it on as its first output, but after that passes on whatever appears on its second input line. After, on the first input line, any tokens that might arrive will be discarded, but no input from the second line is rejected. The '`fby`' node permits arcs to be initialized, though possibly with values computed by other parts of the net. Both nodes act as if they have a two-state internal memory [3] [34].

### 1.3.4 Lucid 2d

The best understanding of Lucid is as functional programming with an added time dimension. Also, in terms of the expressions and equations, various features can be realized in the program, as well as, the static structures, like strings and lists, are not problematic, and it is possible to add finite arrays to Lucid. However, one problem caused by using arrays is that they rquire an algebra of finite multidimensional arrays as in A Programming Language (APL); in contrast, the use of such an array algebra will often produce results that are complex and difficult to parse.

A better approach is that Lucid uses a much simpler algebra of infinite arrays that can be thought of as frozen streams. These could be realized by introducing,

in the simplest case, a space parameter $s$ that works like the time parameter $t$. For example, Lucid variables would denote functions of $s$ and $t$, not just $t$. Therefore, the result would appear as time-varying infinite array [17].

### 1.3.5 Granular Lucid (GLU)

In diverse platforms, which are derived from different workstation clusters, some colleagues in California designed a high-level system called Granular Lucid (GLU) for creating and implementing high quality applications to share memory and multiprocessors and achieve parallelism. One high-performance application in GLU composes sequential functions expressed in C or JAVA. Code fragments from existing sequential applications are reused in GLU with little new code to be written. Also, applications expressed in GLU were implicitly parallel and inherently adaptive, and GLU applications were highly portable across multiple platforms.

Even though GLU is a hybrid of Lucid and C, Lucid was used as a language for composing applications from C functions; thus, a GLU program is a Lucid program. Basic Lucid uses C functions and data types as uninterpreted entities, and C functions are called by value, in which the user defines those functions where the values can be any valid C object. For example, Lucid can be structured to point to $chars$, $ints$, $floats$, $doubles$, and $fixed-length-arrays$ [23].

### 1.3.6 Dimensionality

Our treatment extends to the case of an arbitrary number of fixed dimensions, not just two. We are given a set of $\triangle$ dimensions which we will assume infinite. A context is a map that assigns ordinates to dimensions. We assume that ordinates are natural numbers. We let $K$ be the set of all contexts.
Definition: the set $K$ of contexts is $\triangle \rightarrow N$.

An intension, multidimensional extension of the notions of a stream, is a map from the set of contexts $K$ to the set of values $V$. If $I$ is an intension and $k$ a context, we say that $I(k)$ is the value of $I$ in context $K$.
Definition. The set $H$ of intensions is $K \to V$.
We can now define the notion of dimensionality. If $D$ is a set of dimensions, it may be that in general $I(k)$ is determined if we know only the ordinates of dimensions in $D$. In this case, we say that $I$ has dimensionality $D$.
Definition: for any intension $I$ and any subset $D$ of $\triangle$, $I$ has dimensionality $D$ if for any contexts $k$ and $k'$.

if $k(d) = k'(d)$ for all $d$ in $D$, then $I(k) = I(k')$.

It is clear that if $I$ has dimensionality $D$ and $D \subseteq D'$, then $I$ has dimensionality $D'$. Not quite so obvious is the fact that the intersection of dimensionalities is also a dimensionality.

Proposition: for any intension $I$ and any family $F$ of subsets of $\triangle$: $I$ has dimensionality $D$ for every $D$ in $F$, then $I$ has dimensionality $\cap F$. This implies every $I$ has a least dimensionality, which we call $dim(I)$.
Definition: for any intension $I$, $dim(I) = \cap \{ D \subseteq \triangle : I \text{ has dimensionality } D\}$.

For Lucid 2d (PyLucid) the dimensions are $s$ and $t$ and so whatever $I$ is, $dim(I)$ is either $\{\ \}$ (the empty set), $\{s\}$ , $\{t\}$ , or $\{s, t\}$. However $dim$ is not computable; there is no general algorithm that will take a Lucid 2D program and a variable $V$ and return $dim(V)$.

The value of $V$ is for infinitely many values of $s$ and $t$; however, it is usually enough to have an approximation to $dim(V)$. There is a simple abstract interpretation algorithm that returns a dimensionality which is almost always accurate but is, in every case, an upper bound. In other words, the algorithm returns a

dimensionality $D$ with the guarantee that $dim(I) \subseteq D$. The optimizations that we will perform using $D$ will preserve correctness when $D$ is such an upper bound. The DA algorithm; we use is bottom-up and eager, calculating successive approximations until a fixed point is reached.

This algorithm is an abstract interpretation over the partial order of dimensionalities ordered by set inclusion, and the Lucid program is a set of equations. [17] [20]. We begin by assigning every program variable the value { }. Then we repeatedly reevaluate every variable using relat interpretations of the basic operations over the domain of dimensionalities.

For example, if the definition of `A` is

```
A = X + Y
```

and the current values of `X` and `Y` are $\delta_1$ and $\delta_2$ respectively, then the new value of `A` is $\delta_1 \cup \delta_2$. This means, `A` is 'sensitive' to any dimension to which either `X` or `Y` is sensitive to.

The same is true of any purely data operation: the new value of the dimensionalityis, the union of the values of the operands.
If there are no operands, *i.e*, then `A` is defined as a constant, as following:

```
A = fixed 45.7
```

Then the new value is { } . This leaves the temporal and spatial operators.
if:

```
A = X fby Y
```

then the new value of `A` is $\delta_1 \cup \delta_2 \cup \{t\}$. In other words, it is not only inherits the sensitivities of `X` and `Y`, we must also assume that `A` itself is time sensitive.
Also, if:

```
A = first X
```

then the result is $\delta_1$ - $\{t\}$. `A` inherits the sensitivities of `A` except that `A` is not time sensitive.

If:

```
A = next X
```

then the new value of `A` is simply $\delta_1$. `A` inherits the sensitivities of `X`. Notice that even though the operation `next` is 'temporal', it does not introduce time sensitivity.

If:

```
A = X asa Y
```

then the new value of `A` is $\delta_1 \cup \delta_2 - \{t\}$, so that it inherits the sensitivities of `X and Y` but is not time sensitive.

Also, if:

```
A = X whenever Y
```

Then the new value of `A` is $\delta_1 \cup \delta_2 \cup \{t\}$.

Furthermore, We must consider the space-time operations `all` and `elements`. Therefore, if:

```
A = all X
```

then the new value of `A` is $\delta_1 \cup \{s\} - \{t\}$ if $t$ is in $\delta_1$ , otherwise $\delta_1$.

Similarly, if:

```
A = elements X
```

Then the new value of `A` is $\delta_1 \cup \{t\} - \{s\}$ if $s$ is in $\delta_1$, otherwise $\delta_1$.

### 1.3.7 The PyLucid Compiler

We incrementally transform the input program, a set of equations, into another, simpler (though larger) program in which each equation is atomic. By atomic, we mean that the equations can not be simplified: each consists of a variable on the left and a single operator, and its operands on the right which are all variables. Thus, the primes program given above:

```
N = 2 sby N+1
S = N fby S wherever S % init S ne 0
```

We applied the first equation on a program; however, the conversion involves introducing more temporary variables, such as {`V00`, `V01`, `V02`, ... etc}. Then, the second equation should generate the output of that program.

The atomic form is as shown below:

```
N = V00 fby V01;
V00 = int2;
V01 = N + V02;
S = B sby V03:
V03 = S wherever V04;
V04 = V05 ne V06;
V05 = int SX + V06;
V06 = S ne V07;
V07 = int 0
```

This assumption and the output are in a clear example in chapter 3.
It should be emphasized that these two sets of equations are equivalent in the following sense: any solution to the first set can be expanded by adding values for the V's into a solution for the second; and from any solution to the second we can extract, by discarding the values of the V's, a solution to the first set [35].

In particular, the least fixed points (domain-theoretic minimal solutions) correspond in this way. Accordingly, from the point of view of the Lucid denotational

semantics that they have the same meaning.

## 1.4 The Dimensional Analysis Model (DAM)

We proceed incrementally by the following stages, from simplest to more difficult as shown below in figure 1.3. In the first four stages, as following:

*First*, the constants: identifying the variables constant in time, as follows:

```
Z
where
 Z = first Y
end
```

*Second*, the space $s$ as well as the time $t$ dependencies, where there are four possibilities: $s$, $t$ if the variable depends on both dimensions $s$ and $t$; $s$ if it depends on $s$ only; $t$ is $t$ only, and for constants.
*Third*, dimensionality analysis, when there are many dimensions.
*Fourth*, the analysis in the presence of user defined functions is finally solved.

Completing these stages will make possible the efficient implementation of full multidimensional Lucid.

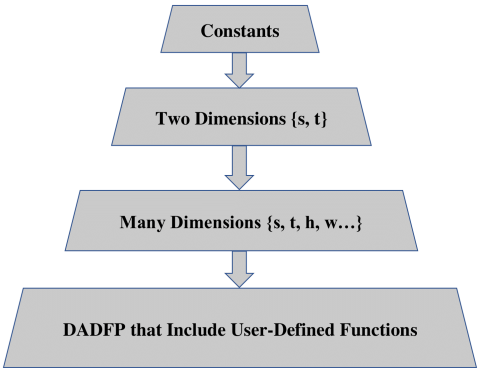


Figure 1.3: The DAM Stages.

### 1.4.1 Constants

It is about identifying the variables constant in time, as following:

```
X
where
 X = first Y
end
```

However, for space and time dimensions, it's for identifying time or space constant variables; then identifying those variables that do not vary in time or space. Figure 1.4 shows an example if the program is:

```
a = int 5
b = first j
j = 1 fby j+1
c = a+b
```

Figure 1.4: DADFP Program.

Then `a,b`, and `c` are time and space constant but `j` is not.

### 1.4.2 Space and Time

This technique illustrates identifying variables that are constant in both dimensions space $s$ and time $t$, then those that are constant in time but not space, and then those that are constant in space but not time, and the last one the variables that vary in space and time.

### 1.4.3 Many Dimensions

This algorithm explains the dimensionality analysis when there is a large but fixed set of dimensions in data flow programs.

### 1.4.4 DADFP that Include User-Defined Functions

It is about the dimensionality of data flow programs in the presence of user defined functions that are translated into atomic equations. [1].

## 1.5 Conclusions

Our DA algorithms provide good details, step by step, for implementing functional software for the dimensionality analysis of data flow programming.

Also, Our Dimensional Analysis Model (DAM) will enable the efficient implementation of multidimensional Lucid. It allows the DA users to look at the data across multiple dimensions, so that they have a deeper understanding of user interfaces for end-user systems, such as SwiftUI and healthcare system-related challenges. These new advances in multidimensional data modeling will ultimately improve real-time data analysis, increase the efficiency and accuracy of such analysis, and further advance data analyst tracking, forecasting, and maintenance of the multidimensional data flow model.

# Chapter 2

# Data Flow Programs, Hardware and Applications

Data flow programming is not well known among many programmers and mathematicians. Thus, this chapter describes different data flow programming languages and applications, starting with data flow history. We briefly present the syntax and semantics of a few dataflow systems. Furthermore, we briefly describe the hardware of modern data flow systems, such as TensorFlow, and Hadoop, and the recently developed Functional Reactive Programming (Data Flow Programming) system produced by Apple, and named Combine or SwiftUI.

## 2.1 The History of data flow programming

For several decades, computer scientists have considered how to deal with big data. Data flow programs emerged from these considerations. A data flow program is a network, where nodes function as operations and edges function as data pathways. Data flow is a distributed model of computation: there is no single place of control. It is also asynchronous, the firing of a node in general begins when the matching data is ready at nodes input ports. In the original data flow models, data tokens are consumed when the node fires (executes). Some models were extended with

sticky tokens, tokens that set a constant input and corresponded with tokens arriving on other inputs. Then, nodes can have varying levels of detail present in a set of data granularity from instructions to functions.

In 1962, Petri invented Petri nets, and then in 1963, Estrin and Turn suggested an early data flow model. Later, in 1966, Karp and Miller proposed schemas without branches or merges where the control information is associated with statements. In this model, data flow control depends on a statement that can be started when its input data is available. In 1969, Rodriguez extended and formalized Estrin's model, and Chamberlain suggested a single assignment language for data flow in 1971.

A simple parallel processing language with queues on the edges was designed by Khan in 1974, and then Dennis developed a data flow architecture with one token buffers for each edge. Later, in 1977 Arvind and Gostelow, and separately Gurd and Watson, proposed a tagged token data flow model in which the edges work like bags [21].

In Karp and Miller's schemes, the memory, consisting of queues, is used instead of one location. Each queue is assigned to a certain pair of statements (distributed memory). One statement writes the data into the queue and another reads it out. The current state of the queue can be empty. Data written to the queue is added to the tail, and data is read from its head. Queues allow pipelining when the data flows successively through a set of concurrently executed statements.

Generally, the queue's capacity is unbounded, so the scheme memory is potentially unbounded. In these cases, the expressive capabilities of data flow schemes, in particular the maximum attainable parallelism, become superior to those of bounded memory schemes with locations that are potentially supplemented by arrays to make the situation more balanced. Karp & Miller's schemes were one of the first models that easily demonstrated that data flow control and distributed memory eliminate shared location conflicts, and, consequently, make data flow control completely decentralized. [8].

## 2.2 Data Flow Analysis

To test a simple sequence of assignment statements written in a high level language, let us propose a simple python program in six steps, as shown in figure 2.1, below.

```
1- P = X+Y
2- Q = P/Y
3- R = X*P
4- S = R-Q
5- T = R*P
6- Result = S/T
```

Figure 2.1: Simple Code in Python.

To analyze this program, it is clear that some instructions can be understood concurrently as soon as certain constraints are met. These can be like a graph with nodes that represent those instructions and an arrow between one instruction and the next. That means the second instruction may not be executed until the first has been accomplished. Therefore, the permissible computation sequences consist of others as follows: (1,3,5,2,4,6), (1,2,3,5,4,6), and (1, [2 and 3 simultaneously], [4 and 5 simultaneously], 6).

These steps are confirmed at run and compile-time in the mathematics processing units and optimizing compilers, where the data flow analysis yields improved use of temporary memory locations. However, that program can be compiled by using the registers instead of the main memory once it can be determined. GOTOs make this determination very difficult. As a result, writing optimizing compilers for languages like Python is difficult [3].

A multiprocessor system should allocate a processor for each instruction, with suitable instructions to impose the sequencing constraints, but the execution would be hopelessly inefficient since the parallelism of this example is far too "fine

grained" for a multiprocessor. The overhead in the process scheduling and the wait and signal instructions would be many times greater than the execution time of the mathematics operations. On the other hand, a data flow computer is designed to execute algorithms with a fine grain of parallelism efficiently. In these machines, parallelism is exploited at the level of individual instructions, as in the above example, and at all higher levels as well. Normally, in most programs, there are many sections that are often far removed from each other, at which computations may proceed at the same time [12].

Using parallelism at all levels demands that the instruction sequencing constraints must be taken from the program itself. Therefore, the machine-level program which uses a data flow computer is essentially represented as a graph with pointers between nodes, and these pointers represent both the flow of data and the sequencing constraints. The status of each instruction is kept in a special memory that is capable of firing the instruction when all of the necessary data values have arrived, and subsequently using the result to update the statuses of the destination instructions [9].

Therefore, the programming language for a data flow computer must satisfy two criteria. First, it must be possible to deduce the data dependencies of the program operations. Second, the sequencing constraints must be exactly the same as the data dependencies, so that the instruction firing rule can be based simply on the availability of data.

## 2.3 Related Languages and Applications

### 2.3.1 The Lucid Programming Language

Lucid first appeared in 1976. It is a functional language, and every program variable in Lucid denotes a stream, a sequence of values. In 1985, Bill Wadge and Ed Ashcroft presented Lucid as a "Data Flow Programming Language" where its operations transform streams. However, Lucid code is similar to a code in

other programming languages. Therefore, a Lucid program is simply functions and operators that transform data values as they pass from one node to the next in a network.

#### 2.3.1.1 The Syntax of Lucid

The expressions in Lucid are truly mathematical; however, in math, the value a function returns is determined only by a value given to that function as an argument. However, that function has no memory recording whether such a value was computed before [35]. Nevertheless, in Lucid, there are streams, which are data transformed by a filter that is a series of data objects rather than just one. Therefore, the contradiction between the static nature of conventional mathematics and the dynamic nature of programming is resolved.
The semantics of Lucid is fundamentally different from those for a language like C because defining those filters that act on time varying data streams. Thus, Lucid syntax is purposely designed to be remarkable and different to prevent programmers from using procedural-programming notions. Basic Lucid supported a small set of data types, such as integer, real, and symbols.

The basic algebraic syntax in Lucid permits iterative programming with no resorting to tail-recursive functions. Thus, pseudo-functions `first` and `next` were provided to allow equational definitions of loops that were not on the surface consistent. For instance, these equations define a loop with two loop variables `I` and `J`:

```
first(I) = 0
next(I) = I+1
first(J) = 0
next(J) = J+2*I+1
```

**I** starts a counter 0, 1, 2, 3, . . . ., but **J** enumerates the squares 0, 1, 4, 9, . . . .

In this example, the semantics of `first` and `next` are not clear because their domains are not well-defined. If there is no formal semantics, it is not possible to settle paradoxes like the following:

```
next(0) = 0
next(1) = 1
next(2) = 2
```

In other words; even though, the next iteration 3 will be 3 as well, but this should not mean that:

`first(I)= I`, for all the `I` stream.

Also, the equation:

```
next(I) = I+1
```

is reduced to:

```
I = I+1
```

In this case, the domain consists of infinite sequences, histories of data values. However, `first` and `next` have simple formal definitions, like:

```
first(<a,b,c,d,. . . >) = <a,a,a,a,. . . >
next(<a,b,c,d,. . . >) = <b,c,d,e,. . . >
```

As a result, everything becomes clear. The equations for `I` and `J` given above have a unique solution, namely:

```
I = <0,1,2,3, . . . >
J = <0,2,4,9, . . . >
```

This can be explained better in this the equation below:

```
next J = J+2*I+1
```

"The left hand side is clearly <1,4,9,25,...>. The right hand side is the sum of series `J, 2*I`, and 1. `J`, we know that it's <0,2,4,9,...>. What about `2*I`? Obviously, the `kth` value of `2*I` is simply twice the `kth` value of `I`. So, `2*I` is <0,2,4,6,...>. As for "1", the `kth` value of "1" is 1, so that the numeral "1" denotes <1,1,1,1,...>".

To make sure that the equation is usable, we need to know if the sequence (<,1,4,9,25,...>) is equal to the sum of the sequences. For example, how to add these sequences:

```
<0, 1, 4, 9, . . . >
<0, 2, 4, 6, . . . >
<1, 1, 1, 1, . . . >
```

For more details [1], let's take it step by step. If we have the value of `L+M+N` at 'time' `k` that means the value of `L` at time `k` plus the value of `M` at time `K` plus the value of `N` at time `k`. Therefore, it is clear that the numbers work out. Back to the three sequences above, in the third column 4+4+1 = 9, as it should be in the sequence on the left hand side. Indeed, the semantics not only confirmed laws like:

```
next(L+M) = next(L)+next(M)
```

but they also cleared up the contradiction that `next` appeared to be the identity function.

The fact that shows:

```
next(0)=0
```

`next(1)=1`, ... *etc.*

Just reflects the fact that any constant sequence <a,a,a,a,...> is a fixed point of `next`. Nevertheless, because `I` may not be constant there is no contradiction, it's not possible to assume that:

```
next(I) = I
```

Afterward, a temporal programming language became one based on a temporal logic, with temporal operators such as:

```
first and next
```

Then, in addition to those two operators, immediately, two convenient temporal operators were created as well.

```
asa
```

`As Soon As`, was created to extract individual values from a loop. In general, if we have:

```
X asa P
```

is the value of `X` that corresponds to the first true value of `P`. For instance, the value of

```
J asa I
```

as defined previously equal to 8, is 64. In fact, it's <64,64,64,. . . >.
The other operator is:

```
fby
```

Followed-By, which allows us to combine the `first` and `next` equations into one, as is shown in this form below:

```
<a,b,c,d,. . . > fby <p,q,r,s,. . . > = <a,p,q,r,s,. . . >
```

Consequently, the four equations for `I` and `J` in the beginning, can be combined to:

```
I = 0 fby I+1
J = 0 fby J+2*I+1
```

This can be the real significance of the `fby` operator, but we need more clarification to understand its use in the data flow interpretation.

### 2.3.2 Streams and Iteration in the Single Assignment Language SISAL

SISAL is a data flow programming language that was defined in 1983 by James McGraw. It is a general-purpose, single assignment functional programming language with strict semantics, implicit parallelism, and efficient array handling. SISAL originated in the data flow community as the language VAL, Value-oriented Algorithmic Language, that has as a target an architecture-independent data flow graph intermediate form. Also, it was used to program the Manchester Data Flow Machine.

The language has since evolved into a complete functional language; for example, it has higher-order functions. Implementations exist for a variety of uniprocessors, shared-memory multiprocessors, and other machines. It's been the main object of the SISAL project to demonstrate sequential and parallel execution performance competitive with programs that are written in conventional languages, and impressive results have since been achieved [12].

#### 2.3.2.1 Statements in SISAL

SISAL has powerful features for manipulating arrays, including vector subscripts to select and manipulate sub-arrays and non-strict stream types, which are produced in order by one expression evaluation and consumed in the same order by one or more other expressions evaluations. As an example of a non-strict operation on streams, consider figure 2.2 from a Sieve of Eratosthenes program:

```
function Sieve(S: stream[integer];
M: integer returns stream[integer])
    for I in S returns
        stream of I unless mod(I, M) = 0
    end for
end function
```

Figure 2.2: Sieve of Eratosthenes Program.

The above function accepts a stream of integers and produces another stream. The result could be used before the stream is completely computed. Also, the production and consumption of streams might be pipelined. Streams are usually generated by **for** expressions, as shown above. There are two forms of these **for** expressions.

In the first form, the values are distributed to the multiple instances of the body of the **for** expression. Each body instance contributes a value to the overall result. The result might be an array or stream, and a reduction operator might also be applied.

The **Sieve** function above has this type of **for** construction. In the second form, an iteration, dependencies are expressed between values defined in one body instance and values defined in the preceding body instance. Then, each body instance returns a value that contributes to the result [8].

### 2.3.3 The LUSTRE Data Flow Programming Language

The first formal presentation of the LUSTRE language was in 1991, however, it began as a research project in the early 1980s. It is a formally defined, declarative, and synchronous dataflow programming language for programming reactive systems. It progressed to particular industrial uses in commercial products and could be the core language of the industrial environment.

#### 2.3.3.1 LUSTRE's Streams

LUSTRE is a streaming language that owes its inspiration to Lucid. Even though there are important differences between the approach which is used in this language and the Lucid approach, there is also a sense in which LUSTRE is much closer to what is usually meant by data flow. However, there are important distinctions, the main one being that queuing of values on arcs does not occur.

This language forms a family of tools for the design of reactive systems, including real-time systems and control automata. Any variable or expression in LUSTRE can denote a flow containing a possible infinite sequence of values of a given type, and a clock that represents a sequence of times. A flow takes the nth value of its sequence of values at the nth time of its clock. Any program or part of the program has a cyclic behavior, and that cycle defines a sequence of times called the basic clock of the program. A flow whose clock is the basic clock then takes its nth value at the nth execution cycle of the program [12].

Lucid can define a stream in which future values depend on past values or vice versa, as long as there is some definition for each element; however, LUSTRE streams can be thought of as evolving in time. This means that for each stream, the future elements depend only upon past elements of the same and other streams, because operators that are not point to point are always causal. Moreover, each stream variable has associated with it a clock, representing, in an abstract sense, the rate at which thee values are produced. [35].

### 2.3.4 The Translucid Programming Language (Cartesian Programming)

The TransLucid programming language was created and designed to be adequate for such a low-level intensional language as Lucid, and to be the language of the object code for translating the common programming paradigms into it. With that said, TransLucid is still fully declarative. Objects manipulated by TransLucid, called *hyperdatons*, are arbitrary-dimensional infinite arrays, and are indexed by multidimensional tuples of arbitrary types.

In 2005, Bill Wadge and John Plaice started TransLucid with the idea that partial results would be incremental because of the requirements to the warehouse caching. In other words, when requesting an identifier/tag pair, the execution engine would begin with an empty tag, and the warehouse would either return a result or a request for information about further dimensions. This means a new request would need to be made with an extended tag. Then, when enough information was provided, the warehouse could provide the answer or launch a calculation, if it is required [26].

This eduction makes an initial set of rules for a language with developed contexts or values. It is a real programming language that is either used directly or as a target language for compiling in different models or frameworks. Figure 2.3 above shows an example which illustrates the Ackermann function which is defined as two dimensional `ack` that varies in dimensions 0 and 1 :

The evaluation of `ack` is partial because of the function mapping respect to a context # *dimensions* and *ordinates*.
The expression #.1 represents the application of the context # to dimension 1, so as to return the current 1 ordinate. "The context of evaluation can be changed using the `@` operator by specifying the new ordinates for some dimensions" [26].

```
ack = if #.1 == 0
then #.0 + 1
elsif #.0 == 1
then ack @ [1 ← #.1 - 1, 1 ← 1]
else ack @ [1 ← #.1 - 1, 0 ← ack @  [0 ← #.0-1]] fi
```

Figure 2.3: The Ackermann Function.

"The use of [...] allows the specification of a new context relative to the current context, replacing the values for an arbitrary set of dimensions. As in GLU , new types and operators can be added to the language"[26] [23]. There are two types of data structure for TransLucid. The first kind is the explicit tuple that in TransLucid is used as a context, the explicit tuple corresponds precisely to a multidimensional coordinate into the implicit infinite multidimensional array hyperdaton, which is the second kind of the data structure. Therefore, Plaice coined a new term for programming in TransLucid, "Cartesian programming", with a clear reference to the Cartesian coordinate system.

Problems in data structures can be better described by increasing the dimensions for the datasets. It is possible in TransLucid to translate other programming paradigms, to have a single intermediate language. On the other hand, there is also the potential to add some of the multidimensional features of TransLucid to other languages. The development of TransLucid is ongoing, with many experiments in implementation, which are highly relevant to the original discussion in chapter 1. This demonstrates the importance of architecture algorithmic design to the development of multidimensional data flow programming language.

Acccording to this idea, the values associated with the dimensions in a context are only calculated if needed in TransLucid. Also, applying this method led to not only faster-running programs, but also to more powerful programming model that develop TransLucid to deploy infinite contexts.

### 2.3.5 Data Flow Hardware

Data flow architecture itself is a software architecture; however, it needs common standard hardware architecture for its reactive solutions that are built on top of hardware dataflow. This hardware dataflow-oriented architecture tries to introduce software means to provide reactive support.

Logically, a dataflow architecture is parallelized once multiple computing units re-evaluate formulas to keep the constraints satisfaction problem updated. Hardware dataflow-oriented architectures, dataflow processors, are data-driven for an instruction on the classical control-driven, Central Processing Unit (CPU) to be enabled and then executed only when all of its operands are available.

Also, dataflow processors work to avoid data dependencies and control flow problems by following instructions in the pipeline stemming from different contexts. Clearly, the dataflow architecture has many advantages over other software based ones, including faster execution and the removal of dependency checks inherent in common software constraint satisfaction algorithms; however, this approach requires new dedicated hardware that is likely to be expensive [31] [8].

#### 2.3.5.1 Programming Reactive Systems and Data Flow Synchrone

These systems interact continuously with their external environment and have to meet strong temporal constraints. For this reason, some programming languages are dedicated to the design of reactive systems. Computer-based implementations relied on low-level programming languages such as assembly, C, or Ada and are available for general-purpose programming. Also, those languages are used for providing a strong and precise control over the resources. However, reactive systems often run with very limited memory and their reaction time needs to be statically known; thus, better-adapted languages were needed.

Consequently, the concurrent programming model was the first that is targeted to get closer to the reactive system specification.

Certainly, reactive systems are fundamentally concurrent because every system and its environment are evolving in parallel and concurrently. This is the natural method to compose systems from more elementary ones. Therefore, Dataflow Synchrone, Synchronous Lucid, works as a dataflow programming language which is dedicated to designing reactive systems. It is based on the synchronous model of the LUSTRE programming language that extends features usually found in functional languages, such as higher-order or constructed data types.

Even though this language is fundamentally and extensively evolved, only the main functional features explained in terms of dataflow software and hardware are important to know for the next steps in our research. "The functionality of the system can be described by making an hypothesis of instantaneous computations and communications, as long as it can be verified afterward that the hardware is fast enough for the constraints imposed by the environment" [6].

The high-level programming languages have to compile efficient chronological code in order support parallel constructions. The normalization between logical time and physical time is verified by computing the worst-case reaction time of the software [4].

### 2.3.6 Data Processing on Large Clusters ( Hadoop Software)

Hadoop is an open-source software framework that uses Data flow programming languages Hiev and very recently, Lucid Hiev, for very large data sets on computer clusters. These programming languages work to process structured data and perform all kinds of data manipulation operations. They are also in Hadoop for expressing Map/Reduce-programs and analyzing large Hadoop Distributed Files System (HDFS) and distributed data sets that lead into full-dimensional experience, as well as, to support custom specific, User Defined Functions (UDFs).

The first study was for Google File System in 2003, then the first project that used Hadoop was a top-level project at Apache Hiev, in 2008. One of the im-

portant properties of Hadoop is the partitioning of data and computation across many thousands of hosts, and it executes application computations in parallel close to their data. The HDFS name-space is a hierarchy of files and directories that are represented on the Name-Node, which records attributes such as permissions, modifications and access times, name-space, and disk-space quotas. The file content is split into large blocks, typically 128 megabytes are selected file by file, and each block is independently replicated at multiple Data Nodes. Therefore, Data-Node identifies the block replicas in its possession to the Name-Node to confirm that the Data-Node is operating and the block replicas it hosts are available. Hadoop clusters by simply adding commodity servers, which can scale computation capacity, storage capacity, and IO bandwidth. Furthermore, Hadoop broke a world record by becoming the first system to sort a terabyte of data in 209 seconds. [18][28].

### 2.3.7 TensorFlow

#### 2.3.7.1 TensorFlow Basics

As a data flow system, TensorFlow can separate the specification of application logic from the execution of that logic. TensorFlow is a cloud computing framework and includes a large library of nodes for mathematical operations where different machine learning applications can be run. Also, it possesses a simple run-time that executes the application with low overhead across a diverse set of hardware systems including Central Processing Units (CPUs) and Graphics Processing Units (GPUs). These processors or features are supported by data flow and can be applied to computations beyond machine learning. These features are probably the minimal design that a processor should not be able to distinguish between data items across concurrent requests. However,TensorFlow, which uses data flow programs, should not need much reprogramming in order to deploy different processors to implement computing applications. Additionally, no global

memory is needed since the data does flow in the computing model in real-time. Heterogeneous implementation is one example where hardware units like CPU and GPU are employed using TensorFlow based on data flow. Heterogeneous computing proceeds from the embedded homogeneity of conventional computing by allowing software to easily integrate hardware with various architectures. In multidimensional data systems, heterogeneous computing is a relatively new concept, and it is beneficial because multiple types of hardware can confidentaly use algorithms, though others may find it unfavorable [30] [37]. Furthermore, the algorithms that exclusively iterate multiple calculations are intrinsically parallel and non-sequentially dependent are different from that iterate sequentially; they are also based on the hardware that is used for a dataset. For example, a GPU capable of a highly-parallel processing system may perform best with an algorithm that iterates non-sequentially; on the other hand, an algorithm that exclusively iterates sequentially may perform better with a CPU with greater single-threaded performance. Therefore, using a combination of numerous and parallel processing features or functions with a defined sequential calculation should be the best method, as is shown in Figure 2.7 [24][2].

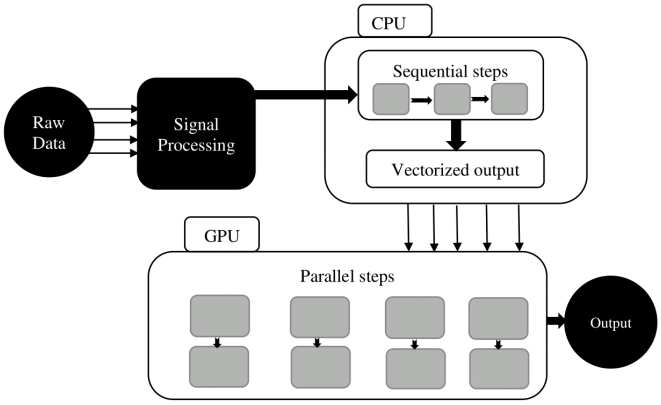


Figure 2.4: Application of Heterogeneous Architecture.

This application uses heterogeneous architecture, and even intrinsic to the hardware itself. There is no restriction on interacting with homogeneous cores or having identical cores on the same chip.

### 2.3.8 The Combine Framework (SwiftUI)

The Combine framework is Apple's reactive framework that provides Swift Application Programming Interface (API) for processing values over time where these values can be symbols for many types of asynchronous events, such as data, procedures, measures or actions. Combine also declares users to be publishers who publish values that can change over time, and/or subscribers who receive those values from the publishers [15][16].

SwiftUI has a very state-driven model of the UI. Thus, if a value changes, the UI should reflect that because it needs to render again. Swift can also be combined with TensorFlow to create Swift for TensorFlow (S4TF).This is used as a first language to learn programming and even machine learning because S4TF has many powerful features and users will find it very useful to run programs on

hardware accelerators, such as CPUs and GPUs. UI also represents the data and state, and the event drives a method that has code to update the current data [33].

# Chapter 3

# One-Dimensional DA Algorithm - Time Sensitivity

The iteration time in Lucid changes in discrete increments, but the time variable in calculus varies continuously. Nevertheless, the interpretation of Lucid is of a dynamically changing system, which is similar to how the interpretation of a set of differential equations might work for that system [25]. "Lucid statements are just equations, but the resemblance between some of these statements and assignment statements is not just superficial and syntactic. The variables in a clause really can be thought of as having values which change with time"[36]. In this chapter, we explain how to create a data flow program and interpreter program computations based on one dimension, time, for the first algorithm of our model DAM.

## 3.1 Introduction

If we only have the time dimension, the DA algorithm is especially simple. We iteratively accumulate the set of variables that we estimate may vary in time. Variables that are never added to this set are guaranteed to be constant in time. We begin with the empty set, then repeatedly add variables that we deduce may

```
A asa abs(A*A - X) <eps
where
     X = first input;
     eps = 0.0001;
     A = 1 fby (X+A/X)/2;
end
```

Figure 3.1: Code for Newton's Method.

vary in time. When no new variables are added to the set, the algorithm terminates. Dimensionality is a simple idea but surprisingly subtle. Let us begin with the simplest case, time sensitivity in 'classical' time-only Lucid calculating which variables need to know the value of the time parameter, `t` [11] [36]. As an example, Figure 3.1 above, illustrates a simple program to compute the square root of first value of the input (a floating point number) using Newton's method:

## 3.2 The Algorithm Structure

The first step is to translate the data flow program to a set of *atomic equations* by using the PyLucid compiler/interpreter. Each variable is defined as a single operator applied to operands, which are all variables. The conversion involves introducing more 'temporary' variables, such as {`V00`, `V01`, `V02`, ... etc}.

```
output = A asa V00;
```

```
V00 = V01 < eps;

V01 = abs(V02);

V02 = V03 - X:

V03 = A * A;

X = first input;

eps = real 0.0001;

A = 1 fby V04;

V04 = V05 / 2

V05 = X + V06;

V06 = A / X;
```

The DA algorithm is iterative [1]. It begins by assuming that none of the variables other than `input` are time-sensitive and works through the equations, accumulating variables that can't be assumed to be constant.

In the first pass we notice that `A` is defined by `fby`. Also, such a definition must be assumed to be time sensitive. It may not be but this is a worst case analysis.

On the other hand, on this first pass no other variables are added to our list. For example, `eps` is defined as a constant and `V02` is the difference of two variables that at this stage are still assumed to be constant.

On the next pass the time sensitivity of `A` is passed on to `V03` and `V06`. To be on the safe side, we also have to assume that an arithmetic operation applied to variables at least one of which is time sensitive is itself time sensitive.

At this stage our list of possibly time sensitive variables is `A`, `V03`, `V06`. The

next pass adds `V02` and `V05` to the list and the pass after that adds `V04` and `V01`. Another pass adds `V00` but the next pass makes no changes.

We're finished, and at this point we can be sure that the variables not in the list – `output`, `X`, and `eps` – are time constant. If they were not we would have found out by now.

This information tells us that the output of the program is a single constant. There is no need for the program to produce successive values of `output` because they will all be the same.

The general rules for the evaluation steps are very simple. Sensitivity is propagated (or not) depending on the operator. In particular, if:

```
V = first X
```

then V remains insensitive.

However, if:

```
V = next X
```

Then `V` becomes sensitive, if `X` is sensitive.

Also, if

```
V = X fby Y
```

Then `V` becomes sensitive whether or not Y is

.

Moreover, if:

```
V = X asa P
```

then V remains insensitive.

Lastly, if:

```
V = X + Y
```

and either X or Y is sensitive, then V becomes sensitive

That is the same as the multiplication, division, and any other data operations including `if-then-else-fi`. The reevaluation proceeds until it settles down, until no new sensitive variables are found.

Once we conclude that the variable `output` is a constant, we can simplify the output and save space when caching. The values of variables like A are cached and in so doing each value is tagged with the value of the `t` coordinate. That is because different values correspond to different time-points. However, this is not necessary with a variable like X which is known to be constant. If we don't take this information into account we will store many copies of the same value.

Dimensionality analysis is simple enough when we work on only the time dimension. However, adding even one extra dimension, such as the space dimension `s`, it seriously complicates the analysis, as we will see later.

### 3.2.1 Algorithm Analysis

As mentioned [36], we use atomic equations, equations with either a constant or an expression with one operation and only variables on the right-hand side. The form of the equation determines whether or not the variable defined is added to the set being accumulated. The forms below are analyzed based on this study as follows:

1- If the variable is defined as a constant, for example:

```
V = 5
```

Then, `V` is not added

2- If the variable is defined in terms of a data operation, such as:

```
V = X + Y
```

Then, `V` is added if either `X` or `Y` are already in the set

3- If the equation is of the form:

```
V = first X
```

Then, `V` is not added

4- If the equation is of the form:

```
V = next X
```

Then, `V` is added if `X` is already in the set

5- If the equation is of the form:

```
V = X fby Y
```

Then, V is added

These equations above complete the **time-only** algorithm.

## 3.3 Conclusions

In Section 3.2, we can understand how the Lucid interpreter matches either one or two dimensions, time&space. Also, in the algorithm structure section, we know that the idea of this algorithm depends on a study that considers only time-sensitive variables working through the equations accumulating streams that can not be constants.

Section 3.2.1 illustrates more details about the first algorithm, the time dimension algorithm. It shows different equations where the streams can be either constant, not added, or added to the set being accumulated. However, this depends on the operators that are used in these equations, which may take into account all variables in that equation or a part of them, and this is according to our study of the first algorithm in DAM.

# Chapter 4

# The Two-Dimensional DA Algorithm

In the previous chapter, we explained how to approximate the dimensionality of variables in a classic time-only Lucid program, how they can be constants that do not change with time. These results prove their vital importance for the output and an efficient caching. In this chapter, we consider the second algorithm in the DAM model. It explains the case of two-dimensional Lucid, using the PyLucid compiler/interpreter, in which the variables may vary in two independent dimensions, time, `t`, and space `s`.

## 4.1 Introduction

In this algorithm, the situation is more complex because a variable can be constant, sensitive to time only, sensitive to space only, or sensitive to both time and space. If we define the dimensionality of a variable as the set of all dimensions to which it is sensitive, the four possibilities are empty set {}, time {`t`}, space {`s`}, and time and space {`t,s`}.

In terms of the PyLucid interpreter, we can consider the atomic equations form of the program as the analog of assembly language.

The evaluations of this assembly language are just as straightforward. The evaluator requests values of specific variables for particular space and time coordinates. These coordinates are stored in registers as an alternative to being passed around as parameters. For instance, if we demand `X` as follows:

```
X = Y + Z
```

we evaluate `Y` and `Z` without changing registers and return the summation of these values. However, if:

```
X = next Y
```

We bump the time register by one, then evaluate `Y`, and then decrement the time register to restore its initial value. Also, if:

```
X = Y sby Z
```

we examine the space register. If it's zero we evaluate `Y`, otherwise we decrement it, then evaluate `Z`, then increment it.

## 4.2 The Algorithm Interpreter

Implementing the PyLucid compiler/interpreter requires the first stage, parsing by using a single function that is parametrized with precedence level instead of functions for every level of a mixture of recursive and operator precedence. Then, a function `expect` is parametrized with a lexical token for error checking. Thus, an error message is produced along with the first five following tokens, rather than only a token.

### 4.2.1 Operators in the Interpreter

The operators or pointwise functions in the Lucid interpreter are without side effects, unlike those of other programming languages. The program below repeatedly reads in an integer and outputs the corresponding factorial:

```
fac(m)
where:
 fac(m) = if m eq 0 then 1
 else m * fac(m-1) fi;
end
```

The function `fac` is defined in the body `where` the clause can be thought of as a machine or black box which continuously consumes values of m and produces values of `fac(m)`. That means, it is a filter in the UNIX sense; however, it filters streams of integers instead of characters. Figure 4.1 shows a sequence of snapshots for a specific sequence of inputs [35].

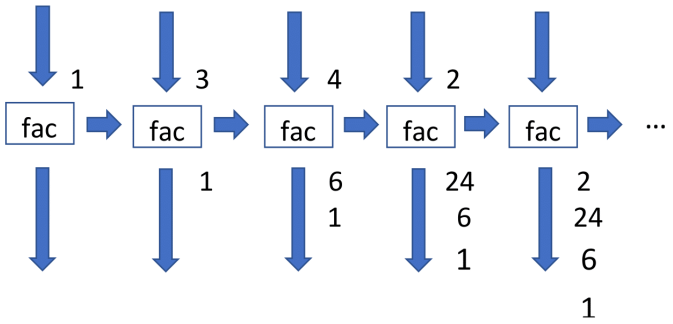


Figure 4.1: A Sequence of Inputs.

The filter outputs each of the factorials `fac(m)` 1, 6, 24, 2 as `m` takes each of the values 1, 3, 4, 2.

### 4.2.2 The PyLucid Compiler/Interpreter

PyLucid is written in Python in which we preserved the syntax and domain of data objects, based as they were on the syntax and data domain support data types of POP2, including integers, reals, Booleans, words, character strings, and lists. PyLucid also supports numbers and words (alphanumeric symbols), and its lexical rules are the same as the input/output conventions.
Furthermore, Plyucid is an expression evaluator. For instance, this simple expression `x + y` is a complete pLyucid program, and involves ongoing interaction between the user and the evaluator. It should proceed as follows:

`x(0)`: 2 the first demand for a value of `x`

`y(0)`: 3 the first demand for a value of `y`

`output(0)`: 5 the first value of the `sum`

`x(1)`: 1 the second demand for a value of `x`

`y(1)`: -8 the second demand for a value of `y`

`output(1)`: -7 the second value of the `sum`

`x(2)`: 2.73 the third demand . . .etc.

`y(2)`: 1, then the `output(2)`: 3.73. Next, `x(3)`: a fourth demand for `x`, for which the user has not yet supplied a value. However, this program is a list of interactions between the user and the evaluator which might look as follows in the table 4.1.

If the input is not a stream of length at least one, the resulting output is the error value?. These examples can be understood as filters. The first example is an addition filter because it takes two streams of inputs and produces the stream of their sums, but the second is a sequence of snapshots of the data flow computation. A filter can produce output values that depend upon values already processed. An

| Input List | The Evaluation List |
|---|---|
| L(0): [42 7 5] | output(0): 7 |
| L(1): [3 [2.4 8] 9] | output(1): [2.4 8] |
| L(2): [2] | output(2): ? (the PyLucid error object) |
| L(3): [5 1 2] | output(3): 1 |

Table 4.1: The Interaction between the User and the Evaluator

example of a filter that seems to have memory is one that takes as input a sequence of numbers, one at a time, and produces as output the smallest and the largest of the numbers read so far. Also, the addition of a space parameter `s` is the most significant feature in PyLucid because it allows 'streams' that vary in both space and time. These two-dimensional intensions can be thought of as infinite streams of infinite arrays, or as arrays of streams, and the two dimensions are treated symmetrically. Furthermore, an extra dimension should not be a problem for demand-driven data flow, which is the basis of the PyLucid implementation. The space analogs of the operators, `first`, `next`, and `fby`, are:

```
init
succ
sby
```

More information about the PyLucid compiler/interpreter is presented in chapter 6 and the appendix.

## 4.3 The Algorithm Structure

Our dimensionality analysis assigns one of the four possible dimensionalities to every program variable, including `output`. Note that these assignments are upper bounds. For example, we may assign {`t, s`} to `X`; even though, when we run the program, it turns out that the space parameter does not affect the results. Upper

```
N2 = 2 sby N2+1;
S = N2 fby (S wherever (S mod init S != 0));
output = init S;
```

Figure 4.2: A Two Dimensional Program.

bounds are good enough to avoid caching duplicate values.

Figure 4.2 shows a two dimensional program that produces the stream of prime numbers:

The first step, as before, is to convert it into a set of atomic equations:

```
N2 = 2 sby V00;
V00 = N2+1;
S = N2 fby V01;
V01 = S wherever V02;
V02 = S mod V03;
V03 = init S;
V04 = V02 != 0;
output = V03;
```

The next step is to set everything to dimensionality {} (constant). On the second iteration, we set `N2` to {`s`} and `S` to {`t`}.

On the third iteration, `V00` gets {`s`}, `S` gets {`t,s`}, `V01` gets {`t`}, `V02` gets {`t`}, `V03` gets {`t`}, `V04` gets {`t`}, `output` gets {`t`}.

Finally, on the fourth iteration, `N2` gets {`s`}, `V00` gets {`s`}, `S` gets {`t,s`}, `V01` gets {`t,s`}, `V02` gets {`t,s`}, `V03` gets {`t`}, `V04` gets {`t,s`}, and `output` gets {`t`}.

The next stage gives the same result so the iteration is complete. We conclude in particular that `N2` has dimensionality {`s`} (a pure vector), `S` has dimensionality {`t,s`} (a time varying array), and output has dimensionality {`t`} (a pure stream).

When N2 is cached one space tag is enough, caching S requires a time tag and a space tag, and the output is a stream of scalars.

At each stage the new approximate dimensionality of each variable is calculated from the previous approximate dimensionalities according to rules for the operators. These rules are similar to the rules for time sensitivity but more elaborate. Here are some of them, if:

```
V = X sby Y
```

Then, V is space sensitive, and also time sensitive, if either X or Y is.
Also, if:

```
V = init X
```

then V is space insensitive, but time sensitive if X is.
Furthermore, if:

```
V = X wherever P
```

then V is space sensitive, and time sensitive if either X or P is.
Lastly, if:

```
V = X + Y
```

then V is space sensitive, if either X or Y is and time sensitive if X or Y is.

Each rule with operator o can be understood as applying a corresponding operator o* to the dimensionalities of the operands. For example {} sby* {t} is {t,s} and +* is the union (of dimensionalities). These *- operators are all monotonic in the subset ordering of dimensionalities so that if $d_i$ is the approximate dimensionality on the ith iteration, $d_i \subseteq d_i$. This guarantees that the iterations eventually settle down [1].

## 4.4 Algorithm Examples and Results

In this algorithm instead of accumulating a single set, we accumulate a table that assigns to each variable a subset of {`s, t`}. The subset is the estimated dimensionality of the variable, i.e., the set of dimensions relevant to the value of the variable.

We begin by assigning the empty set {} to each variable. That means, for each variable, we estimate that it is constant in time and space. Then we repeatedly update our estimates until there are no further changes.

*If the variable is defined as a constant, the estimate remains* `{}`.

*If the variable is defined in terms of a data operation, e.g.*

```
V = X + Y
```

*Then,* `V` *is assigned the* union *of the sets assigned to* `X` *and* `Y`.

*If the equation is of the form:*

```
V = first X
```

*then* `V` *is assigned the set for* `X` *minus the dimension* `t` *(if present).*

*If the equation is of the form:*

```
V = next X
```

*then* `V` *is assigned the set for* `X`.

*If the equation is of the form:*

```
V = X fby Y
```

*then* `V` *is assigned the union of the sets for* `X` *and* `Y` *plus the dimension* `t`.

*Also,*

*If the equation is of the form:*

```
V = init X
```

*then* `V` *is assigned the set for* `X` *minus the dimension* `s` *(if present).*

*If the equation is of the form:*

```
V = succ X
```

*then* `V` *is assigned the set for* `X`*.*

*If the equation is of the form:*

```
V = X sby Y
```

*then* `V` *is assigned the union of the sets for* `X` *and* `Y` *plus the dimension* `s`*.*

As mentioned, the algorithm terminates when no further changes are made in the set of relevant dimensions. Once this happens, we can guarantee that the sets assigned are upper bounds on the dimensionalities of the variables. In particular, if `V` ends up being assigned `{s}`, then it is time constant; but if assigned `{t}` then it is space constant. However, if it is assigned `{}`, it is constant in space and time, i.e., an absolute constant. Figure 4.3 shows a program that is a group of lucid equations.

```
I = 6
J = A fby B
C = first J
A = C + J
D = next A
```

Figure 4.3: DA Accumulation Program.

Then, respectively, there are five stages in accumulating the set of time-sensitive variables, as a result, as shown below [36]:

```
{ }
{J}
{J, A}
{J, A, D}
{J, A, D}
```

Consequently, in terms of the Two-dimension DA, time `t` and space `s` are considered. For example, we need to know the dimensinalities of these variables which appear in their data flow program, as shown in figure 4.4.

```
P = 8
Q = X sby T
T = P fby Q
V = T + Q
W = init V
X = first Q
```

Figure 4.4: A Two-dimensional DA Program.

| **Variables** | **Set of Relevant Dimensions** | | | |
|---|---|---|---|---|
| P | {} | {} | {} | {} |
| Q | {} | { s } | { s,t } | { s,t } |
| T | {} | { t } | { s,t } | { s,t } |
| V | {} | {} | { s,t } | { s,t } |
| W | {} | {} | { t } | { t } |
| X | {} | {} | { s } | { s } |

Table 4.2: Four Stages of the Two-Dimensional DA Algorithm

Table 4.2 shows the four stages of their dimensionalities based on the relevant dimension of each variable in those equations.

The first column illustrates the data flow variables which are represented in the data flow program in different equations. However, the second column is for the dimensionalities in terms of relevant dimensions of each variable, in which they are space and time `{s, t}`, space `{s}`, time `{t}`, or `{}` empty-set. From the table 4.2, we can briefly clarify the dimensionalities of the variables `P` and `Q`. Neither time nor space is relevant to the variable `P`; whereas, `Q` has relevant dimensions as space and time `{s, t}`, or only time `{t}` [36].

## 4.5 Conclusions

A comprehensive foundation regarding the second algorithm in DAM is provided in section 4.3, in which we clarify two main steps in terms of specific dimensionalities that are `{}`, `{t}`, `{s}`, and `{t, s}` for the program. That is also done based on the iterations of the data flow program. Then, we verified our algorithm in section 4.4 by assuming a sequence of different equations. Those equations are evaluated by several iterations, as well based on the information mentioned in section 4.3, we figured out the dimensions relevant to the variables in each equation in terms of the time and space dimensions.

# Chapter 5

# The Multiple-Dimensional (MD)-DA Algorithm

In this chapter, we consider the third algorithm in our model, DAM. We present multidimensional data flow programs with the aim of understanding how to evaluate the equations in a program, then calculate the dimensionality of its variables.

## 5.1 Introduction

The algorithm for multiple dimensions is more complex than that for two dimensions [25]. In this algorithm, there are unlimited dimensions. . The calculation of dimensionalities can range from only a constant, which has as dimensionality the empty set {}, to many dimensions which can be relevant to that variable.

First of all, from our study of the first and second algorithms of one and two dimensions, for example, if:

```
a = 6 sby x
```

the `a` is sensitive to space

For a simple version of the dimensionality problem, suppose: `X` is of dimensionality {`a`, `b`, `c`} which means, `X` can be sensitive to dimensions `a`, `b`, and `c`, but all other dimensions are irrelevant to `X`. Also, suppose that `Y` is of dimensionality {`c`, `d`}. Similarity, `Y` can be sensitive to dimensions `c` and `d`, but any other dimension is irrelevant.

## 5.2 The Algorithm Structure

What if we have more than two dimensions, such as vertical `v`, horizontal `h`, and forward and backward `z`,. Naturally, we do not invent dozens of new operators like:

```
fby
sby
forward
up
down
left
```

Instead, we parameterize them with a dimension's name attached with a period. For example, in the vertical direction:

`fby.h` is `fby` in the `h` dimension.

And, in the forward/backward dimension:

`first.z` is `first` in the `z` dimension.

Ordinary `fby` becomes `fby.t`.

Here is a GLU version of the prime program, but with the primes enumerated in the `z` direction. The last three lines convert the `t`-enumeration to a `z`-enumeration [23][25].

```
N2 = 2 fby.s N2+1;
S = N2 fby.t (S whenever.t (S mod first.s S != 0));
primest = first.s S;
primestz = primest fby.z next.t primestz
output = first.t primestz
```

Dimensional analysis assigns {s} to N2, {s,t} to S, {t} to primest, {t,z} to primestz, and {z} to output.

The algorithm proceeds as above, except that the dimensionalities of the values are subsets of the set {t,s,v,h,z, ..., etc.} of all dimensions. We can assume that since any particular program is finite, this set itself is finite too. The general rules are obvious generalizations of those given above for {s,t}:

if:

```
V = next.d X
```

Then, V is assigned the current value of X.

if:

```
V = first.d X
```

Then, V is assigned the current value of X, minus d if it is in this set.

if:

```
V = A fby.d Y
```

Then, V is assigned the union of the values of A and Y, plus d.

## 5.3 The Basic Idea of the Algorithm

To apply this algorithm of dimensional analysis GLU-type programs the first stage is to compile the program into a set of atomic equations where each equation defines a single variable as the result of applying a single operation to the operands, which are variables [17][23].We calculate the dimensionalities of the variables as a series of approximations. Thus, the first approximation assumes no variable depends on any dimension, so the approximate dimensionalities are all the empty set. Then we repeatedly re-evaluate the dimensionalities following the rules of the GLU program and the algorithm that we summarized above. For example, if:

```
Z = X + Y
```

the new approximation to `Z`'s dimensionality is the union of the approximations of `X` dimensionality which is sensitive to only the dimensions `a`, `b`, and `c`; and `Y` is sensitive to only the dimensions `c` and `d`.
Also, if:

```
Z = X fby.d Y
```

then the next approximation for `Z` is the union of those of `X` and `Y` plus `d`. Thus, `Z` is sensitive to only dimensions `a`, `b`, `c`, and `d`.

### 5.3.1 The Algorithm Flow Chart

After the first , we can continue the re-evaluation until there are no further changes in these dimensions. Figure 5.2 shows a flow chart that illustrates the analysis steps for the Multiple-Dimensions (MD) algorithm.

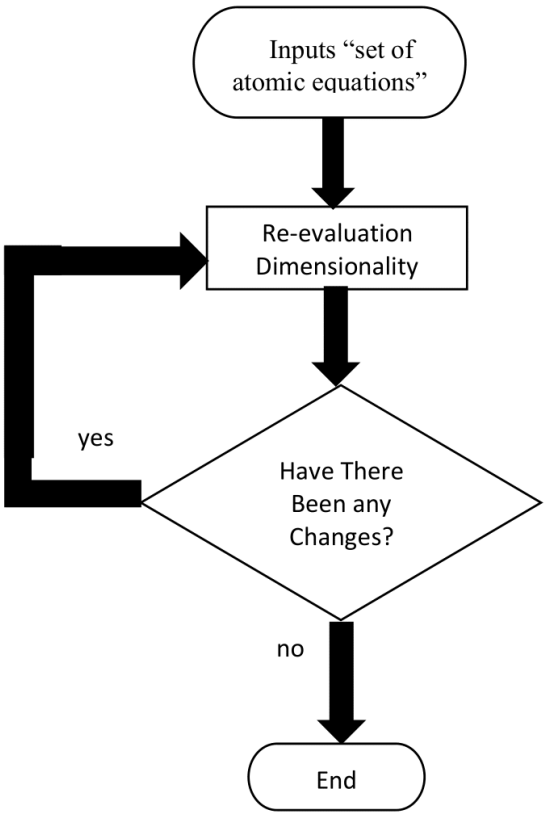


Figure 5.1: Flow Chart of (MD) DA.

First, by default, every variable has the empty set { } of the dimensions as the first approximation of the first evaluation. However, the algorithm continues reevaluating the program as long as there are new dimensions after each iteration. Next, the algorithm determines if there are any changes in the dimensionalities of

the inputs, with no further relevant dimensions to those variables in each equation. Also, those changes can be realized based on the program's structure and the iteration time of the process. Therefore, if the dimensionalities of any variable show up only in the first approximation, then the re-evaluation will be ended. Otherwise, the re-evaluation is continued until there are no more changes.

### 5.3.2 Algorithm Examples and Results

Table 5.1 illustrates a set of atomic equations. We need to evaluate each equation to know the dimensionality for every variable. Then, we can figure out how many approximations and which dimensions are relevant in each approximation.

| **Atomic Equation** | **Dimensionality Re-evaluation** |
|---|---|
| `Y = H fby.c V` | `Y`'s dimensionality is the union of those of `H` and `V` plus dimension `c` |
| `X = first.e Y` | `X`'s dimensionality is all approximations of `Y` except dimension `e` |
| `Z = X + Y` | `Z`'s dimensionality is the union of the approximations of `X` and `Y`'s dimensionality |
| `N = next.d Y` | `N`'s dimensionality is the same as `Y`'s |
| `F = Y + R` | `F`'s dimensionality is the union of the approximations of `Y`'s dimensionality, and `R`'s dimensionality |
| `H = X fby.m Y` | `H`'s dimensionality is the union of those of `X` and `Y` plus dimension `m` |
| `V = R + W` | `V`'s dimensionality is the union of the approximations of `W`'s dimensionality, and `R`'s |
| `W = X asa.c Y` | `W`'s dimensionality is the union of the approximations of `X` and `Y`'s dimensionality, but not dimension `c` |

| | |
|---|---|
| `Q = init S` | `Q`'s dimensionality is all approximations of `S` minus `s` |
| `S = first.a X` | `S`'s dimensionality is all approximations of `X` except dimension `a` |
| `L = P + Y` | `L`'s dimensionality is the union of the approximations of `P` and `Y`'s dimensionalities |
| `P = next.d X` | `P`'s dimensionality is the same as `X`'s |
| `R = Y fby.k X` | `R`'s dimensionality is the union of those of `Y` and `X` plus dimension `k` |

Table 5.1: The Dimensionality of a Set of Atomic Equations

Table 5.1 shows the approximations of each variable of the input (atomic equations). However, table 5.2 below references the dimensionalities of these variables in a parallel process. It starts from variable `Y`, then `X` until the last one, `R`. The results of this process are shown after every iteration of the entire program. [1].

| Variables | First approximation of the relevant dim's | Second approximation | Third approximation | Fourth approximation | Fifth approximation | Sixth approximation | Seventh approximation |
|---|---|---|---|---|---|---|---|
| Y | { } | {c} | {c, m} | {c, m, k} | | | |
| X | { } | { } | {c} | {c, m} | {c, m, k} | | |
| Z | { } | { } | {c} | {c, m} | {c, m, k} | | |
| N | { } | { } | {c} | {c, m} | {c, m, k} | | |
| F | { } | { } | {c, k} | {c, m, k} | | | |
| H | { } | {m} | {m, c} | {m, c} | {m, c, k} | | |
| V | { } | { } | {k} | {k, c} | {k, c, m} | | |
| W | { } | { } | { } | {m} | {m, k} | | |
| Q | { } | { } | { } | { } | {c} | {c, m} | {c, m, k} |
| S | { } | { } | { } | {c} | {c, m} | {c, m, k} | |
| L | { } | { } | {c} | {c, m} | {c, m, k} | | |
| P | { } | { } | { } | {c} | {c, m} | {c, m, k} | |
| R | { } | {k} | {k, c} | {k, c, m} | | | |

Table 5.2: Sequence of Approximations for the Equations in Table (5.1)

Based on our assumptions, the first approximation of the relevant dimensions is the empty set {} for all variables in the program, as is shown in table 5.2. Nevertheless, the dimensionalities of the variables can vary in the second approximation, or the third ..., etc.

Therefore, we consider any change in the program as a parallel evaluation of the whole program. For example, this equation.

```
Y = H fby.c V
```

The operator `fby.c` makes dimension `c` relevant to the variable `Y`. Meanwhile, we can begin to define the relevant dimensions based on this equation.
From tables 5.1 and 5.2, the dimensionality shows up in the second approximation in the `c` dimension, which is a relevant dimension to the variable `Y`. Also, `Y` includes the dimensionalities of the variables `H` and `V`. Thus, the algorithm adds the `m` dimension to `Y` in the third iteration. Then, also in the fourth iteration, the dimension `k` is added to the variable `Y`. The fifth approximation and so on are empty because there is no more change in the dimensionality of those variables. However, the algorithm still reevaluates the dimensionalities of the rest of the variables as parallel inputs until no more dimensions can be generated. As a result, all relevant dimensions can be determined in this multidimensional data flow program.

## 5.4 Conclusions

In the evaluations presented in section 5.2, the program in our third algorithm (MD-DA) is generated by Lucid equations and in reference to the GLU version of their data flow equations. However, our work on these equations is based on a new assumption. In this algorithm, we consider all dimensionalities that the variables may have in an equation, but not only present the two dimensions, time `t`, and space `s` dimensions, like what we have in the second algorithm, in chapter 4.

The additional explanations regarding MD-DA in section 5.3 and shown in the flow chart in section 5.3.1 display the separate steps of this algorithm process in sequential order, and provide a method of defining those steps. Section 5.3.2 illustrates clear examples with the dimensionalities for every variable in the multidimensional data flow program. Consequently, after iterations of the parallel evaluation for this program, we can determine the relevant dimensions for each variable defined in those atomic equations. That matches the basic idea of the third algorithm in our model (DAM); as well as, we can exclude the dimensions irrelevant to those variables.

# Chapter 6

# The Dimensional Analysis of Data Flow Programs that include User Defined Functions (DADFP-UDFs) Algorithm

The DADFP-UDF Algorithm is the last goal in our model, DAM. We will determine the dimensionalities of the variables in data flow programs that include user defined functions. The GLU colleagues never solved this problem.

We have created a new version of the PyLucid interpreter, the DAM Interpreter, in which we evaluate data flow programs over the domain of dimensionalities. In other words, instead of evaluating the operations over actual numbers, stings, lists etc we use as our values the four dimenionalities {}, {s}, {t}, and {s,t}. Thus, once the proogran has, as usual, been converted using Yaghi code to a set of atomic equations. This version has no user defined functions and can be educed directly.

We overlooked this scheme for a long time because for programs with any kind of recursion, the evaluation over the domain of dimensionalities does not terminate. This is because `if-then-else-fi` is approximated by rules such as:

`if t` then `t`

else `s,t fi` = `s,t fi`

Finally we added counters for the number of evaluations and the length of the place code sequences. If we evaluate the program with caps (bounds) on these metrics, the evaluation terminates. However in principle the result may be incomplete. For example, it may return {`t`} as the dimensionality of the program when the real dimensionality is {`t, s`}. To avoid this we simply run the program again with bigger caps until the result settles down.

We experimented extensively with the DAM interpreter and confirmed that not only does it work, it settles down for relatively small values of the cap. As an added bonus, it (obviously) handles programs without UDFs. In other words, it serves as an implementation of the algorithms in chapters 3 and 4 and could easily be extended to the multiple dimensions case described in chapter 5

## 6.1 The Calculation of Dimensionalities

As we mentioned previously in chapter 4 the filters in PyLucid denote functions from streams to streams. If we have a `sum` filter which transforms a stream, such as 1, 4, 5, 6, 8 . . . into the stream 1, 5, 10, 16, . . . , then, this filter acts as if it stores the sum in internal memory. However, this does not look helpful for translating programs into data flow because the standard data flow model demands we create a data-push system that always requires implementing filters with additional internal memory for Object-Oriented programs.

This requires a lot of work since it is not only about the results as numbers for the data flow programs, but we also need to know the results in terms of the dimensionality of these programs. Nevertheless, we simplified this issue to be working efficiently for User Defined Functions (UDFs).

In this dissertation, in addition to using the PyLucid compiler or interpreter, We also translate the data flow program that includes user-defined functions into atomic equations by using Yaghi code. That is the main reason for creating the DAM interpreter for our model, DAM.

### 6.1.1 Atomic Equations

The atomic equation forms are shown in these examples. For example, if `I` is defined as following:

```
I = J + K
```

Then the PyLucid interpreter demands the values of `J` and `K` and then evaluates them with no change of the registers where the particular space and time coordinates of these variables are stored. Then, it returns the `sum` of these values. Also, if:

```
I = next J
```

Then, the Time register is bumped by 1; and then `J` will be evaluated. After that, the Time register will be decrementing. In other words, the time register restores its initial value. However, if:

```
I = J sby K
```

Then the Space register is examined. If it is 0, then `J` will be evaluated; otherwise, the space register will be decremented, and `K` will be evaluated. After that, the Space register will be incrementing.

More examples are in the Appendix of this dissertation.

### 6.1.2 Yaghi Code

The approach of PyLucid is that a first-order program can be transformed into a set of zero-order definitions that contain context manipulation operations. Rondogiannis and Wadge formalized Yaghi Code, and said that "for each function `f` defined in the source functional program,
1- Number the textual occurrences of calls to `f` in the program, starting at 0, including calls in the body of the definition of `f`.
2- Replace the $i$th call of `f` in the program by $\texttt{call}_i(\texttt{f})$. Remove the formal parameters from the definition of `f`, so that `f` is defined as an ordinary individual variable.
3- Introduce a new definition for each formal parameter of `f`. The right hand side of the definition is the operator `actuals` applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered." [27]

For the first-order program, we consider:

```
result = f(4) + f(5)
f(x) = g(x+1)
g(y) = y
```

Then, the compiler generates the following intensional program:

```
result = call0(f) + call1(f)
f = call0(g)
g = y
x = actuals(4, 5)
```

$\mathtt{y} = \mathtt{actuals(x+1)}$

The interpretation steps for this intensional program are shown based on these semantic rules below:

$$\mathtt{call}_i(\mathtt{a})(\mathtt{w}) = \quad \mathtt{a(i:w)}$$
$$(\mathtt{actuals}(\mathtt{a}_0,\ldots,\mathtt{a}_{n-1}))(\mathtt{i:w}) = (\mathtt{a}_i)(\mathtt{w})$$

First of all, the $\mathtt{call}_i$ and actuals are defined in terms of operations on finite lists of natural numbers, as tags or contexts. Then, the interpreter start to execute the program by asking the value of the variable result of the intensional program by empty *tag* []. After that, the operator $\mathtt{call}_i$ matches the operation of prefixing a *tag* w with i, whereas, actuals matches the head i of a *tag*, and uses it to choose its $i$th argument. Further actuals can take intensions $\mathtt{a}$, $\mathtt{a}_0$, ..., $\mathtt{a}_{n-1}$, and let " : " to denote the consing operation on lists of the semantic equations as it's introduced in Yaghi code as following:

$$\begin{aligned}
&\mathtt{Eval}(\mathtt{call}_0(\mathtt{f}) + \mathtt{call}_1(\mathtt{f}),[])\\
&= \mathtt{Eval}(\mathtt{call}_0(\mathtt{f}),\, []) + \mathtt{Eval}(\mathtt{call}_1(\mathtt{f}),\, [])\\
&= \mathtt{Eval(f},\, [0]) + \mathtt{Eval(f},\, [1])\\
&= \mathtt{Eval}(\mathtt{call}_0(\mathtt{g}),\, [0]) + \mathtt{Eval}(\mathtt{call}_0(\mathtt{g}),\, [1])\\
&= \mathtt{Eval(g},\, [0,0]) + \mathtt{Eval(g},\, [0,1])\\
&= \mathtt{Eval(y},\, [0,0]) + \mathtt{Eval(y},\, [0,1])\\
&= \mathtt{Eval(actuals(x} + 1),\, [0,0]) + \mathtt{Eval(actuals(x} + 1),\, [0,1])\\
&= \mathtt{Eval(x} + 1,\, [0]) + \mathtt{Eval(x} + 1,\, [1])\\
&= \mathtt{Eval(x},\, [0]) + \mathtt{Eval}(1,\, [0]) + \mathtt{Eval(x},\, [1]) + \mathtt{Eval}(1,\, [1])\\
&= \mathtt{Eval(x},\, [0]) + 1 + \mathtt{Eval(x},\, [1]) + 1)\\
&= \mathtt{Eval(actuals}(4,\, 5),\, [0]) + 1 + \mathtt{Eval(actuals}(4,\, 5),\, [1]) + 1\\
&= \mathtt{Eval}(4,\, []) + 1 + \mathtt{Eval}(5,\, []) + 1
\end{aligned}$$

$= 4 + 1 + 5 + 1$

$= 11$

## 6.2 User Defined Functions (UDFs)

A User Defined Function can denote a correspondence between infinite sequences of data; however, it could be thought of as a filter that is continually operating processes that transform data streams. The (DADFP-UDFs) algorithm can define a function concerning a body, meaning the expression on the right-hand side of the definition can use it without reference to this body. We can clarify that by this function in the example below.

```
f(a)
where
     f(n) = if n<2
     then 1
     else n*f(n-1) fi;
     a = 1 fby a+1;
end
```

In this UDF example even though the function `f` has no temporal operators, the time-sensitivity of `a` is passed on by `f`. The DAM interpreter can evaluate the function `f` to know if the output is sensitive to a dimension, and there are no further relevant dimensions to the variables in this program.

## 6.3 The Algorithm Interpreter and Examples

We mentioned that PyLucid works as a compiler and an interpreter; however, there is still one more issue that we need to know to determine the dimensionality in terms of the variables and expressions in the program, not only the numbers. To deal with this issue, we need to compute the value for a set of dimensions and exclude those irrelevant. Thus, the `tag` (label) for a value stored in a warehouse should not have irrelevant dimensions. Otherwise, the same result would be saved many times with tags that whose only difference is that they are stored as irrelevant dimensions. We resolved this problem with the DAM interpreter which successfully works to determine the dimensionality of the variables in data flow programs.

For example, for the program above we can evaluate it by applying the Yaghi code version of the program over a small domain of dimensionalities {{}, {t}, {s}, {s,t}} with subset ordering.

As a result, it produces the output "`t`", which means the program varies in the time dimension, but not the space dimension, and that is correct.

## 6.4 Conclusions

In Section 6.1, we know how to determine the dimensionality by using a new idea which is to create a new interpreter, the DAM interpreter, using Yaghi code. Then, section 6.1.2 illustrates Yaghi code with good details that include a program as an example that shows the rules that we use to adapt the interpreter's program. Section 6.2 shows a data flow program as an example for the user defined function. It also explains the process of running this program by using the DAM interpreter and the potential output in terms of finding the dimensionality of this program.

Section 6.3 shows the result of running a data flow program that includes a UDF. For that, we used our new technique of interpreting, which works based on the standard steps of the Yaghi code method. The output shows the relevant dimension to that program, which is the time dimension `t`.

If we need to know the dimensionality of a data flow program that includes a user defined function, then we run this program on the PyLucid compiler and interpreter, as well as the DAM interpreter. Consequently, we can know the numbers evaluating a data flow program, and also the relevant dimensions which result from on that program by using (DADFP-UDFs) Algorithm.

# Chapter 7

# Conclusions

## 7.1 Summary of the Dissertation

In this dissertation, Dimensional Analysis of Data-Flow Programs (DADFP), we have presented a novel Dimensional Analysis Model (DAM). We also explained a new idea that uses the Lucid interpreter and works to detect the dimensionalities of the variables in a data flow program. The DAM of data flow programs is a system that can calculate or determine the dimensionalities of the variables in those programs and then output their relevant dimensions.

The algorithms of our system DAM can improve the productivity of programmers by using the basics of data flow programmming. The approach of our model, DAM is motivated by specific problems experienced in the evaluation of Lucid programs that start from one, two or multidimensional programs, and then programs with user-defined functions.

The DAM system fulfills the main objective of this dissertation and helps researchers to further DADFP.

We also mentioned that the PyLucid works as a compiler and interpreter for the first and second algorithms, but it needs to improve its functioning. Thus, we created the DAM interpreter to detect the dimensionality of an entire data flow

program that includes User Defined Functions (UDFs) after translating them into atomic equations by using Yaghi code.

Furthermore, We proved the comprehensiveness of the DAM of data flow programs, in which we verified the DA algorithms, starting from one-dimension to two-dimensions, then multiple-dimensions, and then a data flow program which includes user-defined functions. Thus, we can exclude the dimensions which are irrelevant to a variable in a data flow network and include the relevant ones. That also reduces the redundancy in that data.

The structure of these algorithms is supported by assumptions and clear examples, based on previous studies of data flow programming languages, such as Lucid and GLU.
In terms of using user-defined functions (UFDs), we have established, tested, and evaluated the dimensional analysis of data flow prograsm containing user-defined functions. Thus, we think about expanding our scope to include the possibility of User Declared Dimensions (UDDs).

## 7.2 Future Work

Even though we extensively improved the Dimensional Analysis of Data-Flow Programs (DADFP) for more dimensional analysis algorithms, our system, DAM has certain limitations. In this regard, User Declared Dimensions (UDDs) is essentially as expressive as the final stage of DAM. Therefore, this stage could be, potentially, part of the future work to improve our results.

Our model, DAM. includes UDFs, but we also should work on the dimensional analysis of individual UDFs. If we have a function `foo(X, Y)`, then we would like to recompute the dimensionalities based on new rules of the UDFs-DA until the approximations settle down, just like as we have done for the MD-DA algorithm

in chapter 5. However, we need to know how the dimensionality of `foo(X, Y)` depends on the dimensionalities of `X` and `Y`. Therefore, we consider abstracting some kind of matrix that captures the dimensionality behavior of the function, `foo`. We evaluate the program over a small domain of dimensionalities `{{}, {t}, {s}, {s,t}}` with subset ordering.

The GLU developers never found a good solution to analyzing programs with User Defined Functions. However, we use standard problem solving techniques [23][26]. One is to solve simple examples by hand, and the other direction is to look at non-recursive definitions. Also, another simplification is functions with only one argument. For example, we know how to do this with built-in functions as following:

```
first.d
next.d
fby.d
```

We also know how they process dimensionalities, but what if the *programmer has defined their own function*?
The expression foo(X) can appear in an equation such as:

```
A = foo(B)
```

We propose considering classic Lucid with only the time dimension, in order to know how `foo` processes dimensionalities. For example:

```
foo(X) = first X +  next X
```

As well, if there is a small set of possibilities, then the dimensinal analysis of UDFs might be easier than we thought. For example:

```
R = foo
```

For more than one argument, such as:

```
R =  foo(X,Y,Z)
```

The dimensions are relevant to `foo` when the arguments are all constant, such as `foo(0,0,0)`. Therefore, for each input dimension $d$, for each argument. For example, `A` for the function like:

```
foo(A) = foo (X, Y, Z)
```

Then, we need to know if `A` is sensitive to these dimensions.
If it is sensitive to $d$, as well as, if there is possible output dimension for this function such as $e$, we also should know if the output of `foo` is sensitive to $e$.
Furthermore, `foo` mathematically is characterized by a function $foo*$, that takes dimensionalities $D_1$, $D_2$, and $D_3$ such that if `X`, `Y` and `Z` have dimensionalities $D_1$, $D_2$, and $D_3$; then the function `foo(X, Y, Z)` has dimensionality $foo*(D_1, D_2, D_3)$. More concisely, if $\mathscr{D}$ is the set of dimensionalities, then $foo*$: $\mathscr{D} \times \mathscr{D} \times \mathscr{D} \Longrightarrow \mathscr{D}$
Also, $foo*$ is monotonic and continuous, and distributive in that:

$$foo*(A \cup A' - -) = foo*(A - -) \cup foo*(A' - -)$$

An example of a possible basic fact about the function `foo` that is, `if s` is relevant to `Y`, `then t` is relevant to `foo(X,Y,Z)`.

As well as, the other basic facts like, `s` is relevant to `foo(0,0,0)` [1].

Nevertheless, potential future work may include giving more studies and challenges to figure out the exact set of the dimensionalities of UDF-DA in terms of not only the time and or space dimensions but many dimensions.

Ultimately, the case of multiple dimensions for the UDF-DA algorithm is more complicated; however, it is also practical for DADFP algorithms. It also might be helpful for modern applications that use frameworks based on the dimensionality of data flow programs, like SwiftUI.

# Appendices

Python code is used in the experiments to create the Lucid interpreter described in Section 5.2. Thus, one of the goals was to evaluate the overall usability of PyLucid, including all its features supporting code modularity and reuse. Hence, in terms of Python code, it is written in the too-long program, in which it is a style intended to maximize the utility of these features and to increase the programmer's productivity from One-Dimension to Two-Dimensions DA. However, we provide a good example of PyLucid, even in terms of UDF, and then last but not least is a simplified program.

## A Code Examples

We use a single destructor that returns components, and Python allows multiple returns. Thus instead of writing:

```
s = WhereSubject(w)

b = WhereBody(w)
```

We write:

```
s, b = WhereD(w)
```

Once the program passes the syntax check and is turned into a `where` clause the semantic checking begins.

Next, we verify that the `where` bodies have at most one definition per variable and that the formal parameters of functions are all distinct. Then, we check that function arities are consistent always have the same number of parameters. Finally, every variable besides input has a relevant definition.

Once the checking is done; we proceed to transform the program into a form that can be directly evaluated.

The first step is a systematic renaming of variables so that every logically different variable has a different name. The program below is an example:

```
X+I
Where
X = f(Y);
f(X) = X+Y*I
  where
  Y = 7;
  I = 2;
  end;
I=9;
end;
```

The result of renaming is:

```
X+I
where
X = f(Y);
f(X_01) = X_01* Y_01*I
   where
   Y_01 = 7;
   end;
I = 9;
Y = 3;
end;
```

## A.1 The Crucial Points

The crucial points are: first, the transformed program is still a PyLucid program; and second, it's semantically equivalent to the original. This will be true at each stage, so we will end up with a very simple (but long) program equivalent to the original source.

At several points we will 'dissolve' where clauses by simply dumping their definitions into the enclosing set of definitions. Thus:

```
A+B
where
A = 9;
```

```
B = 2 * A
   where
   A = 7;
   end;
end;
```

Then the inner where clause can be dissolved giving:

```
A+B*3
where
A = 9;
B = 2 * A_01
A_01 = 7;
end;
```

## A.2 The Elimination of User Defined Functions

The most important transformation is the elimination of user defined functions basically by transforming function calls into branching-time iteration. Then the program has only zero-order individual variables, and the rest of the `where` clauses; can be dissolved.

Finally, by introducing extra variables, we turn compound expressions into atomic expressions with only one operation. Also, the whole program is now a set of atomic equations. It means one individual variable on the left equated

to an expression consisting of an operation applied to individual variables, or a literal/constant. The program above becomes:

```
output = A+V_00;

V_00 = B*V01;

V_01 = 3;

A = 9;

B = V_02 * A_01;

V_02 = 2;

A_01 = 7;
```

Now, we need to use Yaghi code to be ready for the evaluation.

For the first-order program, we consider:

```
result = f(2) + f(7)

f(x) = g(x+1)

g(y) = y
```

Then, the compiler generates the following intensional program:

```
result = call_0(f) + call_1(f)

f = call_0(g)

g = y
```

$\mathsf{x} = \mathsf{actuals(2,\ 7)}$

$\mathsf{y} = \mathsf{actuals(x{+}1)}$

Then,

$$\mathsf{Eval}(\mathsf{call}_0(\mathsf{f}) + \mathsf{call}_1(\mathsf{f}),[])$$

$$= \mathsf{Eval}(\mathsf{call}_0(\mathsf{f}),\ []) + \mathsf{Eval}(\mathsf{call}_1(\mathsf{f}),\ [])$$

$$= \mathsf{Eval(f},\ [0]) + \mathsf{Eval(f},\ [1])$$

$$= \mathsf{Eval}(\mathsf{call}_0(\mathsf{g}),\ [0]) + \mathsf{Eval}(\mathsf{call}_0(\mathsf{g}),\ [1])$$

$$= \mathsf{Eval(g},\ [0,0]) + \mathsf{Eval(g},\ [0,1])$$

$$= \mathsf{Eval(y},\ [0,0]) + \mathsf{Eval(y},\ [0,1])$$

$$= \mathsf{Eval(actuals(x} + 1),\ [0,0]) + \mathsf{Eval(actuals(x} + 1),\ [0,1])$$

$$= \mathsf{Eval(x} + 1,\ [0]) + \mathsf{Eval(x} + 1,\ [1])$$

$$= \mathsf{Eval(x},\ [0]) + \mathsf{Eval}(1,\ [0]) + \mathsf{Eval(x},\ [1]) + \mathsf{Eval}(1,\ [1])$$

$$= \mathsf{Eval(x},\ [0]) + 1 + \mathsf{Eval(x},\ [1]) + 1)$$

$$= \mathsf{Eval(actuals}(2,\ 7),\ [0]) + 1 + \mathsf{Eval(actuals}(2,\ 7),\ [1]) + 1$$

$$= \mathsf{Eval}(2,\ []) + 1 + \mathsf{Eval}(7,\ []) + 1$$

$$= 2 + 1 + 7 + 1$$

$$= 11$$

Now, it is ready for the evaluation.

# B A PyLucid Experiment

A simple pylucid program to calculate e as following:

```
Sum
where
    i = 1 fby i + 1;
    term = 1 fby term / i;
    sum = 0 fby sum + term;
    columns = 1;
    rows = 12;
    numformat = '%8.6f';
end
```

## B.1 Program Analysis

The last three equations are not part of the program but specify what the output will look like.

This is the output:

```
0.000000
1.000000
2.000000
2.500000
```

```
2.666667

2.708333

2.716667

2.718056

2.718254

2.718279

2.718282

2.718282
```

Number of evals: 90

warehouse saves 33 warehouse finds 30

Number of place codes generated 0

Number of calculations 42

Vars fetched { 'term', 'sum', 'i'}

The last lines are diagnostic information.

Also, this is the assembly language form of the program:

```
output: id sum

numformat: " '%8.6f'

rows: " 12

columns: " 1

 sum: fby V05 V06

 term: fby V03 V04

 i: fby V00 V02
```

```
V02: + i V01

V01: " 1

V00: " 1

V04: / term i

V03: " 1

V06: + sum term

V05: " 0
```

# Bibliography

[1] Dimensional Analysis multiple dimensions. https://billwadge.wordpress.com/?s=shennat. Accessed: 2021-03-12.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] William B Ackerman. Data flow languages. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 1087–1095. IEEE, 1979.

[4] Jarryd P Beck, John Plaice, and William W Wadge. Multidimensional infinite data in the language lucid. *Math. Struct. Comput. Sci.*, 25(7):1546–1568, 2015.

[5] Jonathan Bowen. Algorithms, software and hardware of parallel computers: J mikloško and ve kotov (eds) springer-verlag, berlin, frg (1984) dm 89 pp

395, 1985.

[6] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes*, pages 28–41, 2008.

[7] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W Wah, and Jianyoung Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 323–334. Elsevier, 2002.

[8] J Chudik, G David, VE Kotov, NN Mirenkov, J Ondas, I Plander, and VA Valkovskii. *Algorithms, software and hardware of parallel computers*. Springer Science & Business Media, 2013.

[9] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[10] Philippe Dumont and Pierre Boulet. Another multidimensional synchronous dataflow: Simulating array-ol in ptolemy ii. 2005.

[11] Antony A Faustini and William W Wadge. An eductive interpreter for the language lucid. *ACM SIGPLAN Notices*, 22(7):86–91, 1987.

[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[13] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE transactions on software engineering*, 18(9):785–793, 1992.

[14] John Hershberger, Nisheeth Shrivastava, Subhash Suri, and Csaba D Toth. Adaptive spatial partitioning for multidimensional data streams. *Algorithmica*, 46(1):97–117, 2006.

[15] Jon Hoffman. *Swift 4 Protocol-Oriented Programming: Bring predictability, performance, and productivity to your Swift applications*. Packt Publishing Ltd, 2017.

[16] Jon Hoffman. *Mastering Swift 5.3: Upgrade your knowledge and become an expert in the latest version of the Swift programming language*. Packt Publishing Ltd, 2020.

[17] Raganswamy Jagannathan and Chris Dodd. Glu programmer's guide. *SRI International, Menlo Park, California, Tech. Rep*, 1996.

[18] Jacob Leverich and Christos Kozyrakis. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.

[19] Svetlana Mansmann and Marc H Scholl. Empowering the olap technology to support complex dimension hierarchies. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(4):31–50, 2007.

[20] Praveen K Murthy and Edward A Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, 2002.

[21] Walid A Najjar, Edward A Lee, and Guang R Gao. Advances in the dataflow computational model. *Parallel computing*, 25(13-14):1907–1929, 1999.

[22] Eduardo Eloy Loza Pacheco, Mayra Lorena Díaz Sosa, Christian Carlos Delgado Elizondo, and Miguel Jesús Torres Ruiz. Swift ui and their integration to mapkit technology as a framework for representing spatial information in mobile applications. In *GIS LATAM Conference*, pages 80–91. Springer, 2020.

[23] Nikolaos S Papaspyrou and Ioannis T Kassios. Glu embedded in c++: a marriage between multidimensional and object-oriented programming. *Software: Practice and Experience*, 34(7):609–630, 2004.

[24] John Persano, Said M Mikki, and Yahia MM Antar. Gradient population optimization: A tensorflow-based heterogeneous non-von-neumann paradigm for large-scale search. *IEEE Access*, 6:77097–77122, 2018.

[25] John Plaice. Multidimensional lucid: Design, semantics and implementation. In *International Workshop on Distributed Communities on the Web*, pages 154–160. Springer, 2000.

[26] John Plaice, Blanca Mancilla, and Gabriel Ditu. From lucid to translucid: Iteration, dataflow, intensional and cartesian programming. *Mathematics in Computer Science*, 2(1):37–61, 2008.

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

[28] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[29] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Communications of the ACM*, 51(11):75–84, 2008.

[30] Siraphob Theeracheep and Jaruloj Chongstitvatana. Multiplication of medium-density matrices using tensorflow on multicore cpus. *Tehnički glasnik*, 13(4):286–290, 2019.

[31] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.

[32] Jayant Varma. Data and combine. In *SwiftUI for Absolute Beginners*, pages 51–66. Springer, 2019.

[33] Jayant Varma. *SwiftUI for Absolute Beginners*. Springer, 2019.

[34] William W Wadge. An extensional treatment of dataflow deadlock. *Theoretical computer science*, 13(1):3–15, 1981.

[35] William W Wadge and Edward A Ashcroft. *LUCID, the dataflow programming language*, volume 198. Academic Press London, 1985.

[36] William W Wadge and Abdulmonem I Shennat. Dimensional analysis of dataflow programming. In *Proceedings of the Future Technologies Conference*, pages 732–739. Springer, 2020.

[37] Mohamed Zahran. Heterogeneous computing: Here to stay. *Communications of the ACM*, 60(3):42–45, 2017.

[1][36][2][3][4][5][6][7][10][11][12][13][14][15][16][17][18][19][20][21][22][23][24][25]
[26][28][29][30][31][32][33][34][35][37][9][8][27]