ACCEPTED
FACULTY OF GRADUATE STUDIES

DEAN

DATE

INDEXICAL PARALLEL PROGRAMMING

by

Weichang Du

B.Sc., Beijing Polytechnical University, 1983
M.Sc., University of Victoria, 1987

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. M. R. Levy, Departmental Member (Department of Computer Science)

Dr. G. G. Miller, Outside Member (Department of Mathematics)

Dr. C. G. Morgan, Outside Member (Department of Philosophy)

Dr. R. Jagannathan, External Examiner (SRI International)

Supervisor: Dr. William W. Wadge

Abstract

Indexical programming means programming languages and/or computational models based on indexical logic and possible world semantics. Indexical languages can be considered as the result of enriching conventional languages by allowing constructs to vary according to an implicit context or index. Programs written in an indexical language define the way in which objects vary from context to context, using context switching or indexical operators to combine meanings of objects from different contexts.

Based on indexical semantics, in indexical programs, **context parallelism** means that computations of objects at different contexts can be performed in parallel, and **indexical communication** means that parallel computation tasks at different contexts communicate with each other through indexical operators provided by the indexical language.

The dissertation defines the indexical functional language mLucid - a multidimensional extension of the programming language Lucid proposed by Ashcroft and Wadge. The language enriches the functional language ISWIM by incorporating functional semantics with indexical semantics. The indexical semantics of mLucid is based on the context space consisting of points in an arbitrary n-dimensional integer space. The meanings of objects, called *intensions*, in mLucid are functions from these contexts to data values. The language provides five primitive indexical operators, *origin*, *next*, *prev*, *fby* and *before* to switch context along a designated dimension.

The dimensionality of an intension in the indexical semantics of mLucid is defined as the set of dimensions that determines the range of the context space in which the intension varies. An abstract interpretation are defined that maps mLucid expressions to approximations of dimensionalities. Context parallelism and indexical communication in mLucid programs are defined by a semantics-based dependency relation between the values of variables at different contexts.

In parallel programming, the context space of mLucid is divided into a time dimen-

sion and space dimensions. The time dimension can be used to specify time steps in synchronous computations, or to specify indices of data streams in asynchronous computations. The space dimensions can be used to specify process-to-processor mappings. The dissertation shows that mLucid supports several parallel programming models, including systolic programming, multidimensional dataflow programming, and data parallel programming.

Examiners:

Dr. W. W. Wadge, Supervisor (Department of Computer Science)

Dr. G. C. Shoja, Departmental Member (Department of Computer Science)

Dr. M. R. Levy, Departmental Member (Department of Computer Science)

Dr. G. G. Miller, Outside Member (Department of Mathematics)

Dr. C. G. Morgan, Outside Member (Department of Philosophy)

Dr. R. Jagannathan, External Examiner (SRI International)

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor Bill Wadge who, through his insights and careful guidance, has made this dissertation possible. He has provided me with timely encouragement and advice as well as an outstanding amount of freedom to pursue research issues I considered important while at the same time critically appraising my work. Most importantly, Bill has that invaluable asset of the good supervisor, an ever open door.

I would like to thank my other committee members, Dr. G. Shoja, Dr. M. Levy, Dr. G. Miller and Dr. C. Morgan, for their conscientious reading of the dissertation and suggestions.

Special thanks go to Dr. Bill McColl for suggesting the topic of this dissertation and discussions in the early stages of this research.

I would also like to thank all members of the Department of Computer Science at University of Victoria for the friendly and pleasant working environment, especially members of the functional programming group in the department with whom I have had very stimulating discussions.

I was supported financially at University of Victoria by a University of Victoria Fellowship and a BC Advanced Systems Graduate Scholarship.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Why Parallel Programming

As the use of computers affects increasingly broader segments of the world, many of the problems to which people apply computers grow continually larger and more complex. Demands for faster and larger computer systems increase steadily. Computer architectures have followed two general approaches in response to these demands. The first approach uses novel technology in conventional serial computer architectures. In the past decade, computer technology has advanced rapidly and the speed of serial computers has increased hundreds of times, but new problems that need even faster computers to solve are still emerging. The demand for faster and larger computer systems seems to outpace the technology. Also in the conceivable future, technologies for developing computer hardware will approach the physical limit, the speed of light, for transmitting electronic signals. This raises the question, to satisfy these demands, "can we still make a great leap in performance while the rate of technology improvement remains relatively constant?"

The second approach exploits the parallelism inherent in many problems. It allows many parts of an application program to be computed on different computing elements simultaneously and to combine the results of the subcomputations as the final solution. The parallel approach seems to offer the best long-term strategy because as the problems grow, more and more opportunities arise to exploit the parallelism inherent in the problems

themselves.

Parallel computers are becoming increasingly widespread. The question is what to do with them. Parallelism is only useful in solving problems when it can be expressed in programs of a programming language, and the expression of parallelism can also be identified by the compilers of the language and implemented on parallel computers. The task of parallel programming research is to make the power of parallelism accessible to programmers and detectable by compilers.

1.2 Issues of Parallel Programming

There are three issues in parallel programming distinct from sequential programming.

- Expression of parallelism in a program of a programming language by identifying pieces of the program, i.e. computation tasks, that can be executed in parallel.
- Expression of communication and synchronization between computation tasks in parallel execution of the program.
- Given a parallel computer architecture, expression of a mapping from computation tasks onto computing elements or processors of the architecture.

Parallel programming can be classified into two categories: explicit parallel programming and implicit parallel programming. In explicit parallel programming, programs are written with explicit parallelism in so-called parallel programming languages. A parallel programming language provides special constructs for programmers to express one or more of the above issues explicitly in their programs. The meanings of the constructs are usually interpreted by the operational semantics of the language, instead of its mathematical semantics. In contrast, in implicit parallel programming, programs are written in conventional programming languages without explicit constructs for expressing any of the above issues. Parallelism and communication in programs of a language without explicit parallel constructs are automatically detected by the language's compilers for target parallel machines.

1.3 The Conflicting Goals of Parallel Programming

There are two main goals that should be achieved by a programming language. On the application side, programmers need a programming language that is abstract enough to allow them to concentrate on solving their problems without bothering about implementation details. On the implementation side, efficient compilation of programs in a programming language require that information on the relationships between program structures and target machine architectures can be easily exploited by compilers.

These two goals conflict because the former needs languages to be independent of underlying computer architectures, whereas the latter needs languages' operational semantics to be close to machine architectures on which their programs will be executed. The task of any programming language design is to harmonize the conflicting goals.

In parallel programming, expression of parallelism affects both sides of the problem. On the application side, the structure of a programming language must help programmers to write programs with more parallelism. On the implementation side, it should be possible and easy for compilers to detect parallelism in programs. Expression of communication affects implementations of parallel programs in the way that it must provide compilers with enough and clear information about how parallel tasks communicate. The conflicting goals appearing on the two sides of the problem causes a dilemma in the choice of languages for parallel programming. For application programmers, implicit parallel programming is preferred, whereas for efficient parallel implementations, explicit parallel programming is needed.

1.4 Problems of Existing Parallel Programming Paradigms

For implicit parallel programming, there are mainly two classes of programming languages: imperative languages and declarative languages. Implicit parallel programming in conventional languages like Fortran, Pascal and C is attractive to a certain extent. It absolves programmers, who are familiar with sequential programming, from thinking about parallelism, and allows them to run their old programs on new parallel computers without

rewriting. However, this approach has problems when high performance of sequential programs on parallel computers is considered, as sequentialism is implicit in the operational semantics of the language. A parallel compiler may not be able to discover the artificial sequentialization in a sequential program imposed by the programmer, and as a result, parallelism in the program cannot be fully exploited by analyzing the data dependency. Also, as sequential programmers are encouraged to write programs that are efficient in the sense of sequential implementation instead of parallel implementation, there may not be too much inherent parallelism in their programs.

Implicit parallel programming in declarative languages is promising, because declarative programs have implicit parallelism. For example, in functional programs, expression parallelism occurs when arguments of a function application and/or the arguments and the function itself can be evaluated in parallel. In logic programs, AND/OR parallelism occurs when two subgoals of a clause or two clauses can be computed in parallel. Implicit parallelism in declarative languages makes it easy for compilers to identify potential parallelism in declarative programs. However, there are still problems in implementing these languages efficiently on parallel computers. Communication between parallel computation tasks in most declarative languages is expressed implicitly through shared variables, which results in complexity in analysis of communication patterns in the programs and makes it difficult for the compilers to map parallel processes onto processors in parallel systems. Also, since the expression parallelism and AND/OR parallelism are usually fine-grain, granularity control is needed to compile the programs to parallel computers with coarse granularity. On the other hand, for parallel computers with massively parallel computation potential, the expression parallelism and the AND/OR parallelism are not sufficient to encourage programmers to write programs with massive parallelism.

For explicit parallel programming, existing parallel programming languages can roughly be classified into three categories: extensions of conventional languages such as Parallel Pascal [Ree84] and C* [Thi86]; specially designed imperative parallel languages such as CSP [Hoa85] and Linda [ACG86]; and extensions of declarative languages such as parallel functional languages [Hud88] and parallel logic languages [Sha86].

In parallel programming languages based on the imperative programming principle, explicit parallel constructs provide programmers with facilities to express parallelism and communication in their programs. In parallel programming languages based on the declarative programming principle, explicit annotations are needed for programmers to express control information on parallelism, communication and mapping, so that efficient implementations on particular parallel architectures can be achieved.

The common problem for explicit parallel programming is that it obviously diverges from the goal of freeing programmers from implementation details. Explicit parallelism puts a heavy burden on the programmers' shoulders. Explicitly expressing parallelism, communication and even process–to–processor mapping are irrelevant to the programmers' tasks for finding solutions for their problems. To make their programs run efficiently on parallel machines, they have to understand the operational meanings of the explicit constructs, and then use them to express complex parallel computation activities in the programs, at the same time concentrating on solving their problems. However, depending on the current development of compiler techniques, explicit parallel programming is still an effective approach to making imperative programs parallelized and allowing declarative programs to be implemented efficiently on parallel computers.

The problems of the existing parallel programming paradigms discussed above have raised a question. Is it possible to have a kind of programming language that is a better candidate for harmonizing the conflicting goals? Without altering or extending its semantics, this language could be used for both implicit and explicit parallel programming depending on programmers' desires. For implicit parallel programming, the language encourages programmers to write programs with large scale parallelism and regular communication. For explicit parallel programming, the language provides programmers with means to express precisely in their programs partitions of parallel computation tasks, communication and synchronization between the tasks and mapping from the tasks to processors. For efficient parallel implementations, the language provides compilers with easily detectable information on parallelism and communication in programs, so that the compilers can easily tailor needed information for efficient parallel implementations. Also,

to make the language application–oriented, precise and rigorous in specifying parallel computations, the language's expressive power for parallelism and communication and the ease of parallel implementation are implied in its mathematical semantics, without depending on some kind of irrelevant parallel operational meaning.

1.5 A Solution from Indexical Programming

This dissertation proposes a solution to harmonizing the conflicting goals of parallel programming based on the indexical programming paradigm.

Indexical programming means programming languages and/or computational models based on indexical logic and possible world semantics. Indexical programming languages can be considered as the result of enriching conventional programming languages in the functional, logic, or imperative paradigms. An indexical programming language is obtained by allowing constructs (expressions, predicates, commands) in the base language to vary according to an implicit context or index. The implicit context could be a moment in time, a point in a space, a node in a tree, a word in a language – all depending on the target application area. Programs written in an indexical programming language define the way in which objects vary from context to context, using context switching operators (such as "next", "up", or "parent") to combine meanings from different contexts.

Indexical semantics of indexical languages can be used to interpret parallelism and communication, called **context parallelism** and **indexical communication**, in indexical programs. Context parallelism in an indexical program means that computations of objects at different contexts can be performed in parallel. Indexical communication means that parallel computation tasks at different contexts communicate with each other through context–switching or indexical operators provided by the indexical language. Context parallelism and indexical communication are based on indexical operational semantics of the language, and are distinct from other kinds of parallelism and communication in the conventional languages.

In implicit parallel programming, context parallelism is specified by the definitions

for the meanings of objects at different contexts and indexical communication is specified through the use of indexical operators in the definitions. Indexical semantics, as a component of the mathematical semantics of an indexical program, is a part of the solution to the problem that the program solves. The indexical semantics can also be interpreted as the expression of parallelism and communication in the program. This duality of indexical semantics means two things. (1) On the application side, it implicitly forces programmers to write programs with context parallelism, when they are solving their problems. (2) On the implementation side, it provides compilers with a global view about where and when parallelism and communication occur in programs.

For example, given a vector A, to compute a vector B whose element at position p is the average of the elements of A at positions $p-1$ and $p+1$, we[1] can define B in an indexical language as follows, where the implicit context of the language is a point in one dimensional integer space:

```
B = (left A + right A)/2.
```

In the definition, *left* and *right* are indexical operators; at a point in space, they switch context from the current point to its neighbors on the left and right, respectively. The context parallelism specified in the definition is that the value of B at each point in the space can be computed in parallel. The indexical communication specified by the indexical operators in the definition is that to compute the value of B at a point p in space, p has to communicate with the points $p-1$ and $p+1$, so that A's elements at these points can be fetched.

In explicit parallel programming using indexical languages, a parallel computation task can be expressed as a subcomputation performed at a context. All the subcomputations at a context represent the computation activity of a processor. The parallelism among computations at different processors is expressed by context parallelism. The communication between the processors is expressed by indexical communication. By the rigorous mathematical meaning of indexical semantics, using context parallelism and indexical com-

[1]The uses of "we" in this dissertation are just a style of writing. In most of places it means the author himself.

munication, the programmer can precisely specify and formally verify parallel computation and communication behaviors and process–processor mappings in indexical programs.

For example, given lists A and B of the same length, consider the following parallel computation. First let processor 0 sort A and processor 1 sort B, then let processor 0 sum the sorted A and B pairwise. The following definitions in an indexical language specify the parallel computation, where the implicit context of the language is also a point in one dimensional integer space and the processor 0 and 1 correspond to the contexts 0 and 1, respectively.

```
SA = sort_list(A) at 0;
SB = sort_list(B) at 1;
SUM = sum_list(SA, right SB) at 0;
```

In the definitions, *at* is an indexical operator; at a point p in the space, `x at q` switches context from point p to point q and accesses x's value at q. The process–processor mapping specified by the indexical operator *at* is that `SA` and `SUM` are computed at context 0 corresponding to processor 0 and `SB` is computed at context 1 corresponding to processor 1. The parallelism specified in the definitions is that A and B can be sorted at contexts 0 and 1 in parallel. The communication specified by the indexical operator *right* in the definitions is that to compute the value of `SUM` at context 0, the context 0 has to communicate with context 1 so that `SB` can be fetched from context 1.

1.6 Synopsis

In Chapter 2, we introduce some background and survey related work. The background includes the concept of indexical logic and indexical programming, the dataflow programming language Lucid and its recent development and systolic arrays. The related work reviews the functional systolic programming language CRYSTAL, the parallel functional programming paradigm and the systolic logic programming paradigm.

In Chapter 3, we define the syntax and semantics of mLucid, including the formal semantics of mLucid's primitive indexical operators which serve as context switching op-

erators to combine values at different contexts.

In Chapter 4, we define the notion of dimensionalities of mLucid objects. The dimensionality of an mLucid object is a set of dimensions in the context space in which the value of the object varies. We also study the relationship between mLucid's indexical semantics and dimensionalities of its objects. We provide a formal approach to analyzing dimensionalities of various objects in mLucid programs based on an abstract interpretation technique.

In Chapter 5, we define the concept of context parallelism and indexical communication in an mLucid program by the dependency relation among values of variables at various contexts. We first define what is called direct dependency between two values of variables at two contexts. Based on the direct dependency relation, a data dependency graph, called variable–value dependency graph, of the program is defined. The context dependency graph of the program is then defined as a contraction of the variable–value dependency graph, in which only the dependency between values of variables at different contexts is considered. Based on the context dependency graph, context parallelism and indexical communication in the program are formally defined by the connectivity of the graph.

In Chapter 6, we describe specifications of systolic arrays in mLucid programs. We choose systolic arrays as an application of mLucid, in order to show the expressive power of the language for parallel programming, because systolic arrays as a type of special–purpose computers have both the temporal and spatial architecture–level parallelism. Moreover, their temporal regularity in terms of computations at processors and spatial regularity in terms of communication among the processors, are best suited to be specified by context parallelism and indexical communication of mLucid programs. We also describe a class of mLucid programs called SysLucid and show that the systolic array models and SysLucid programs are semantically equivalent. We then describe how parallel computation and communication activities in systolic arrays can be specified by the context parallelism and indexical communications of SysLucid programs.

In Chapter 7, we describe a multidimensional dataflow programming paradigm in mLucid. We show how to write mLucid programs with multidimensional pipeline parallelism

on a multidimensional abstract dataflow machine.

In Chapter 8, we study the relationship between context parallelism and data parallelism in mLucid programs. We describe how the common themes of data parallel programming, i.e. elementwise parallelism, parallel conditional, replication, reduction and permutation, are supported by mLucid constructs.

In the last chapter, we summarize the contributions of the dissertation, and discuss future work.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Indexical Logic and Indexical Programming

Indexical logic is a subset of intensional logic that originated in the study of 'intensional' phenomena in human reasoning, such as modality, knowledge, or flow of time. These all require a richer semantic picture than standard truth values in one static environment. Such a picture is provided by so–called *possible worlds semantics* [vB88].

In indexical logic, the meaning of a natural language expression depends on an implicit context, called a *possible world.* For example, the meaning of the expression *today's temperature* depends on a possible world – the day on which the expression is uttered. The essential idea of indexical logic is based on the distinction between *extension* and *intension* of an expression. The extension of an expression is the meaning of the expression in a given possible world. For instance, the temperature on July 1, 1991 in the above example is 20 c. The intension of an expression captures the way in which its extensions correspond to possible worlds; it is a function mapping possible worlds to extensions. For the above example, the intension of the expression is the set of pairs consisting of dates and degrees of temperature.

In order to reason about relationships among meanings of expressions in different

possible worlds, indexical logic also provides such operators, called *indexical operators*, that serve to switch contexts. Using indexical operators, the meaning of an expression in a possible world can be expressed as a combination of possibly the meaning of this expression and other expressions in other possible worlds. For example, the meaning of the expression *Today's temperature is five degree less than yesterday's temperature* can be expressed as a formula *yesterday(t) - 5*. In the formula, *t* represents today's temperature, and *yesterday* is an indexical operator that switches context from a possible world, namely today, to another possible world, namely yesterday. The result of the application *yesterday(t)* on a given date is the temperature of the day before the given date.

Indexical programming, originally proposed by Faustini and Wadge [FW86b], means programming in a language that at the same time is a formal system with indexical semantics. In an indexical programming language, the value of a syntactic object, such as an expression, a predicate or a command, depends on an implicit context; it is an intension (a map) from a set of possible worlds, called the *universe* of possible worlds, to a base value domain. A possible world can be broadly defined, such as a moment in time, a node in a tree, and a point in a space, depending on target applications of the language. A program in the language as an object also has indexical meaning, that is, it is a function mapping intensions of input variables to intension(s) as output. An indexical language provides users with a set of indexical operators whose definitions are based on the intended meaning of possible worlds and target applications. Using indexical operators, users can define the value of an object in a possible world as a combination of values of other objects and/or itself in other possible worlds.

2.2 The Dataflow Programming Language Lucid

Lucid is the first indexical programming language, though the concept of indexical programming was proposed 10 years after the language was originally introduced [AW76a] [AW76b] [AW79]. Lucid is an enrichment of Landin's functional programming language ISWIM [Lan66].

An ISWIM program is simply an expression called the *defining expression*. The output of the program is the value of the defining expression. An expression in ISWIM can be part of a *where* clause. The expression is then called the *subject* of the *where* clause. A *where* clause also has a body which consists of a set of equations, called the *body* of the clause. The left hand side of an equation in the body is a variable symbol with certain arity; and the right hand side of the equation is an expression, which may contain other *where* clauses. The body of a *where* clause defines a local environment for the value of its subject. For example, the following is an ISWIM program that computes the factorial of 5.

```
fac(5)
where
  fac(n) = if n eq 1 then 1 else n * fac(n-1) fi;
end.
```

ISWIM is not a single language; it is actually a parameterized family of languages, each member of which is determined by a choice of underlying data structures. The different members of this family look very similar; they are identical in their *outer syntax*, i.e. their *where* clause structures. A member of the ISWIM family is determined by an *algebra*.

Given a continuous algebra A, the member of the family determined by A is ISWIM(A). The language ISWIM(A), its syntax and semantics, is entirely determined by the algebra A. Given an algebra A, its signature S determines the syntax of ISWIM(A), where a signature is a collection of individual and operation constants together with an arity for each operation constant. An ISWIM(A) program is one in which the expressions are built up from nullary variables and from individual constants in S. Expressions are built up by applying operation symbols in S to appropriate number of operands, by applying function variables to argument lists and by forming *where* clauses. The individual and function variables defined in a *where* clause are called the *locals* of the clause, where each local of a clause has only one definition in the clause.

Given an algebra A with signature S, the semantics of a program in ISWIM(A) consisting of expression E with respect to an interpretation I is defined as follows. Here an

interpretation is a mathematical object which extends A by giving meanings to individual and function variables as well as to the constants. If E consists of an individual variable or constant c, then the value of E with respect to I is the value which I assigns to c. If E consists of an n-ary operation constant or function variable G together with actual parameters $E_1, E_2, \cdots, E_n$, then the value of E with respect to I is $g(v_1, v_2, \cdots, v_n)$, where g is the operation over the universe of A which I assigns to G and each v_i is the meaning of E_i with respect to I. If E consists of a *where* clause then the value of E with respect to I is the value of the subject of E with respect to the least interpretation I' such that (a) each definition in the clause is true with respect to I' and (b) I' differs from I at most in the values it assigns to variables having definitions in the clause.

Lucid inherited ISWIM's syntax and extended its semantics. In Lucid the value of an expression is no longer a single datum. Instead, it is a stream of data items. Each datum in the stream corresponds to a time point indexed by a nonnegative integer. The first datum in the stream corresponds to time 0. In terms of indexical programming, in Lucid's indexical semantics the context space, or the universe of possible worlds, consists of all time points, that is, all nonnegative integers. The value of an expression in a Lucid program, hence the meaning of the entire program, depends on the implicit context – a moment in time.

The indexical language mLucid defined in this dissertation is a multidimensional extension of Lucid, so that Lucid is considered as a special case of mLucid. The formal semantics of mLucid is defined in the next chapter.

When programming in Lucid, programmers think in terms of streams and filters. A simple Lucid program corresponds to a filter with n input variables and one output. The following is an example of a trivial Lucid program:

```
a + b
where
  a = 1.1;
  b = 3.89;
end
```

Note that the syntactic object 1.1 denotes the stream or sequence $< 1.1, 1.1, 1.1, \cdots >$. Thus the definition `a = 1.1` defines the variable `a` to be an infinite sequence of 1.1's. In general, the output expression on the right hand side can contain references to variables that have no definition. These free variables are the program's input streams. For example, the program

```
a + b
```

has two input streams and performs the pointwise addition of the streams. Let the input streams for a and b be $< 2,4,6,8,10, \cdots >$ and $< 1,3,5,7,9, \cdots >$. Then the stream corresponding to `a + b` is $< 3,7,11,15,19, \cdots >$

Lucid provides three primitive indexical operators for context switching: **fby, next** and **first**. These operators permit streams to vary with the stream index.

The two arguments of the binary filter *fby* (read as "followed by") are combined by taking the first component of the first argument and appending to it the stream corresponding to the second argument. Syntactically, *fby* is an infix operator, and thus the program

```
x
where
  x = 1 fby 2;
end
```

defines x to be the stream $< 1,2,2,2,2, \cdots >$. Note that no part of the second argument is lost; its components are in some sense pushed back by 1 in index ordering. Using *fby* an infinite stream can be defined recursively, e.g.

```
x
where
  x = 1 fby x + 1;
end.
```

This program defines x (the output) to be the stream $< 1,2,3,4,5, \cdots >$, an infinitely varying sequence. The first argument of *fby* is the initial value of the stream based on

which successive future values to be generated. Note that since Lucid, as a functional language, is referentially transparent, the variable x denotes the same stream everywhere in the program. Thus, in the definition `x = 1 fby x + 1`, the x on the right hand side of the *fby* is the same as the one being defined which begins with 1. Since x is defined to be 1 at index 0 and 1 is defined to be 1 at index 0, x's value at index 1, which is 2, can be produced according to the definition (from x+1). Obviously this argument can be applied ad infinitum.

Unary filter *next* removes the index 0 component of its argument stream. In the following program, the term *next x* defines the stream $< 2, 3, 4, 5, 6, \cdots >$:

```
next x
where
  x = 1 fby x + 1;
end
```

The following program defines the Fibonacci sequence $< 1, 1, 2, 3, 5, 8, 13, \cdots >$:

```
fib
where
  fib = 1 fby 1 fby fib + next fib;
end
```

Unary filter *first* simply takes the index 0 component of its argument stream and produces a constant stream. In the following program, the term *first x* defines the stream $< 1, 1, 1, 1, \cdots >$:

```
first x
where
  x = 1 fby x + 1;
end
```

The following program defines the factorial of an input number n:

```
first fac
```

```
where
  fac = if x < 1 then 1 else x * next fac fi;
  x = n fby x - 1;
end
```

A Lucid program is a set of equations that is really a textual form of a directed graph. The nodes in the graph are the operations in the text (for example, + and *fby*) and arcs are variables such as x or expressions such as $x + 1$. For example, the following program

```
x
where
  x = 1 fby x + 1;
end
```

is the textual form of the net in Figure 2.1. Thus, all Lucid programs have a unique

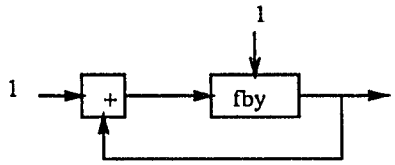

Figure 2.1: A dataflow network

representation as a network of operators. [AJ86] has termed such dataflow networks corresponding to Lucid operators *operator nets.*

The evaluation of Lucid programs is based on a computation model called **eduction** or **tagged demand driven dataflow** [FW86a][AFH85]. Unlike traditional dataflow [Den80][AK81], eduction can be thought of as a form of dataflow with two-way traffic in communication lines. Data flows in one direction from producers to consumers in the usual way. In the other direction, demands are sent from consumers upstream to producers. Both demands sent upstream and data sent downstream are tagged with the stream index.

The computation of a Lucid program is demand driven in that it computes the value of the program's output, or the value of the defining expression, at the stream indices that

the user demands. The computation of any particular stream index leads to the demands of values of various program variables. These variables may be defined in terms of context switching operators that can cause demands for variables at stream indices different from the index associated with the original demand.

2.3 Recent Development of Lucid

Two views of Lucid exist in the computer science community. The oldest view is that Lucid is a language for writing and proving properties about programs. This view is still valid even though the language has evolved considerably since the earliest publication [AW76a]. The second and more recent view is that Lucid is a stream–oriented (dataflow) language. This view is largely based on [WA85]. This second view has been taken by many as a final definition of the language. This is certainly not the case.

From a historical perspective the recent development of Lucid can be traced back to the chapter entitled "beyond Lucid" in [WA85] and to [Ash85] on massive parallelism in Lucid, but it differs from the original versions in these early articles. The development mainly consists of two independent but closely related projects at University of Victoria [DW88][DW90][Du91][Wad91b], some of whose recent results constitute this dissertation, and at Arizona State University and SRI international [AF89][AJ89][Ash90][Ash91][FJ91] [JF91].

The early and recent work both concentrate on multidimensional extensions of Lucid. In original Lucid the implicit index was simply time or iteration index. Consequently, a multidimensional problem would have to be solved using time and some form of monolithic data structure such as *list* or *string* [WA85]. Multidimensional problems can be expressed directly in multidimensional extensions of Lucid without the need for monolithic data structures. This has important implications for parallel and distributed implementations as described in this dissertation.

Originally, multidimensional Lucid objects were defined and manipulated using the operators *initial, rest* and *cby* [WA85][Ash85]. This was elegant since it complemented

the *first*, *next* and *fby* of stream oriented Lucid. Moreover, the proposed operators were information preserving. In other words, a multidimensional object could be defined and manipulated without any of the components being lost.

Unfortunately, using this approach to multidimensionality, programmers found difficulty to write programs, as they had to think of multidimensional object monolithically. The operators preserved information by manipulating data in a nested hyperspace. For example, the first argument of the operator *cby* might itself be a multidimensional object that would be "twisted" by the operator into a higher dimension to preserve it. The operator *initial* would then perform an "untwisting" operation to extract the data into its original "shape".

The work described in this dissertation and that in [FJ91] are based on a distributed view of multidimensionality. In this view, a multidimensional space is composed of orthogonal dimensions, and each dimension has associated with its operators for defining data on a per dimension basis.

Unlike mLucid (described in Chapter 3), which uses an absolute approach to extending Lucid, the language described in [FJ91] (called *Indexical Lucid*) extends the multidimensionality of Lucid using a relative approach. Their approach is to allow the user to extend the dimensions explicitly by giving appropriate names to the new dimensions as they are needed. The language extends Lucid's *where* clause to so-called *indexed where clause*. An indexed *where* clause is one that contains declarations of indices for new dimensions through use of an *index declaration* (or dimension declaration). The scope of the indices introduced by the index declaration is the same as the scope of the variables defined in an ordinary *where* clause. For example the following indexed *where* clause introduces two new indices or dimensions *height* and *width*.

```
a + b
where
  index height, width;
  a = ...
  b = ...
```

```
end
```

In the above program, the variables a and b may be defined to vary over two new dimensions *height* and *width*, in addition to time dimension that is always implicit in Indexical Lucid.

All of the temporal operators in Lucid are extended in Indexical Lucid to any user–declared dimension h as follows:

Conventional Lucid Operators	**Conventional Lucid Operators expanded**	**Extended Indexical Lucid Operators for Dimension h**
first	first.time	first.h
next	next.time	next.h
fby	fby.time	fby.h

As in mLucid, each of the above extended Lucid operators works on its particular dimension in the same way that the ordinary Lucid operators worked on the time dimension.

In Indexical Lucid, index variables that returns the indices of corresponding dimensions at a point in the multidimensional space have the same name as those used for dimension names. All index variables vary over the natural numbers. The following is an example of simple Indexical Lucid that defines a unit matrix:

```
...
where
  index i, j;
  unitMatrix = if i eq j then 1 else 0 fi;
end.
```

2.4 Systolic Arrays

An application of the indexical language mLucid for parallel programming is to specify systolic arrays. Systolic arrays, initially proposed by H.T.Kung [KL78][Kun82], are a

kind of high-performance, special-purpose computer systems for solving computing–bound problems in many areas, such as signal/image processing and scientific computing. Systolic arrays are becoming increasingly attractive because of the continuous advances in VLSI technology.

A systolic array consists of a network of processing elements (or PEs) connected together by a regular interconnection with data streams flowing into and out of the array, much like the flow of blood to and from the heart. Data streams flow into the array through PEs at the boundaries of the network. Each PE in the array processes every datum in one or more streams and passes the result to its neighboring PEs. The final results produced by the array either stay in the local memory of the PEs or, as data streams, flow out of the network through the boundary PEs. The computation in the array is synchronized; the data is rhythmically computed (timed by a global clock) and passed through the network.

Functionalities of PEs and the interconnection topology of a systolic array are specified by an algorithm, called a *systolic algorithm*, which the the systolic array design is based on. In this dissertation, we only consider systolic arrays at the level of their hardware–independent computation and communication behaviors. In this sense, we use the terms, *systolic arrays* and *systolic algorithms*, synonymously in later discussions.

Systolic arrays have the following essential characteristics. PEs in a systolic array have (1) functional modularity in terms of that they usually perform homogeneous computations, and (2) regular communication in terms of that they are usually connected via some regular network topology, such as one–dimensional linear array, ring, mesh, two–dimensional hexagonal array, triangular array, torus or binary tree. There is no global storage in the array; each PE has access to its local storage. PEs can communicate through the network with some small number of neighboring PEs via message passing.

For example, Figure 2.2 shows a systolic array for computing matrix multiplication. The systolic algorithm that the array is based on is described as follows.

Algorithm :

Input : An $m \times n$ matrix A and an $n \times l$ matrix B read into an $m \times l$ array of PEs.

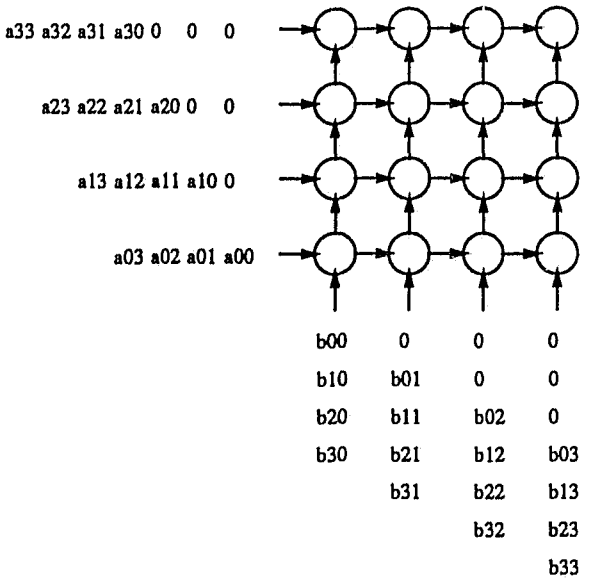

Figure 2.2: A matrix multiplication systolic array

Output : The product $C = A \times B$ of the input matrices A and B. The output is found in the array when the computation terminates; each PE at position (i,j) holds an element $C_{i,j}$ of the product.

Method : Each C_{ij} at PE_{ij} is initially 0. At each time step, PE_{ij} does the following:

1. inputs two data items from $A_{i,j-1}$ at the PE to its left and $B_{i-1,j}$ from the PE below, or from the input matrices, respectively;
2. multiplies the two input data items and adds the result to the value stored in C_{ij};
3. passes the input data items to the PEs to its right and above along the original dimensions in which they came, so that they can be inputs for the neighboring PEs at the next time step;
4. for each C_{ij}, after $n + i + j - 2$ time steps the result C_{ij} is held by PE_{ij}. □

One of problems in systolic array designs concerns the specification and verification of systolic arrays. For purposes of implementation and proof, rigorous notation other

than informal pictures or "snapshot" for specifying systolic arrays is desirable. The search for programming languages that permit complete specifications of the computation and communication activities on the systolic arrays represents a very challenging field of programming language research [Kun82][Kun87]. This dissertation takes up this challenge.

2.5 The Functional Language Crystal

In terms of expressing systolic algorithms, some features of the indexical language mLucid defined in this dissertation are similar to the functional language Crystal [Che86c][Che86a] [Che86b].

In Crystal, a program consists of a system of recursion equations similar to other functional languages, but it is interpreted differently from the traditional functional interpretation. A Crystal program has the following general form:

$$
\begin{aligned}
F_1 &= \phi_1(F_1(\tau_{11}(v)), F_2(\tau_{12}(v)), \cdots, F_n(\tau_{1n}(v))),\\
F_2 &= \phi_2(F_1(\tau_{21}(v)), F_2(\tau_{22}(v)), \cdots, F_n(\tau_{2n}(v))),\\
&\cdots\\
F_n &= \phi_n(F_1(\tau_{n1}(v)), F_2(\tau_{n2}(v)), \cdots, F_n(\tau_{nn}(v))).
\end{aligned}
$$

where

- $v = [v_1, v_2, \cdots, v_n] \in A$, with A a Cartesian product $A_1 \times A_2 \times \cdots \times A_n$ where each A_i is a subset of the set of integers. The set A, considered as a discrete structure, is similar to the context space in mLucid's indexical semantics.

- Functions τ_{ij}, called *communication functions*, map from A to A. They are similar to indexical operators in mLucid.

- Functional variables $F_1, F_2, \cdots, F_n$, range over continuous functions from $(A, \sqsubseteq)$ to $(D, \sqsubseteq)$, where D is some value domain and $(A, \sqsubseteq)$ and $(D, \sqsubseteq)$ are flat domains. F_i is called a *multiple data stream* and is similar to a variable in mLucid whose value depends on a context – an element in A.

- Functions $\phi_1, \phi_2, \cdots, \phi_n$, are continuous functions over the value domain $(D \sqsubseteq)$. ϕ_i called *processing function* is similar to a pointwise function in mLucid.

Thus the above system of recursion equations is a system of fixed–point equations, and on its right hand side,

$$\lambda(F_1, F_2, \cdots, F_n).\lambda v.[\phi_1(\cdots), \phi_2(\cdots), \cdots, \phi_n(\cdots)]$$

is a continuous function from E^n to E^n, where $E = [(A, \sqsubseteq) \rightarrow (D, \sqsubseteq)]$, the domain of continuous functions from $(A, \sqsubseteq)$ to $(D, \sqsubseteq)$.

For example, the following is a Crystal program for integer partition. The number of partitions of an integer k into integers less than or equal to m can be obtained by applying the following recursively defined function C to the pair of integers(m,k):

$$C(i,j) = \begin{cases} i = 1 \rightarrow 1 \\ i > 1 \rightarrow \begin{cases} i > j \rightarrow C(i-1, j) \\ i = j \rightarrow C(i-1, j) + 1 \\ i < j \rightarrow C(i-1, j) + C(i, j-i) \end{cases} \end{cases}$$

where

- $v \in A = N^2$, where N is the set of positive integers.
- Depending on v, $\tau(i,j) = (i-1, j)$ or $(i, j-1)$.
- The functional variable C is a continuous function from $(N^2, \sqsubseteq)$ to $(N, \sqsubseteq)$.
- Depending on v, the function ϕ is the addition or identity function.

In Crystal, an operational interpretation is given to the equations, and parallelism similar to context parallelism in mLucid is introduced. Instead of being considered as actual parameters of a function application like *C(i,j)*, the pair *(i,j)* is considered as an index pair for a process in an ensemble of parallel processes. Each pair *(i,j)* in the set A corresponds to a process, and A is called a process structure. The process structure of a Crystal program can be seen as a Directed Acyclic Graph (DAG). A DAG consists

of nodes, where each node corresponds to an index pair *(i,j)* in the set A. Each directed edge comes out of a node whose index pair appears on the right hand side of an equation in the system, and goes into a node whose index pair appears on the left hand side of the same equation. The directed edges of the DAG define the data dependency relation of the algorithm expressed by the program. The dependency relation ($u \prec v$) between two nodes u and v in the DAG holds, if there is a directed edge from u to v. The transitive closure of the relation on all such pairs is a partial order. The computation of a Crystal program starts from executions of the processes corresponding to the source nodes of the partial order, and is followed by other processes after their predecessors have been executed.

A major difference between Crystal and mLucid is their mathematical semantics. The semantics of Crystal is **extensional**, while that of mLucid is **intensional**. In Crystal, since indices of dimensions appear explicitly as parameters in function definitions, functions have extensionality, that is, the value of a function application corresponds to an explicit context. In other words, a function application with particular indexical actual parameters is considered as an individual object, whose value is a datum in the given data domain. By contrast, in mLucid, since context is implicit, functions have intensionality, that is, the value of a function application as an intension depends on the implicit context without referring to explicit indices. The semantic difference of the two languages results in different programming styles. In Crystal programs, indices are explicitly and directly manipulated; context switching is expressed by changing indices. This is convenient when communication patterns in the programs are irregular. In mLucid programs, indices are implicitly and indirectly manipulated by indexical operators; context switching is expressed by applications of indexical operators. When communication patterns in the programs are regular, in terms of parallel processing, the intensional approach reflects clearer global structures of the programs than the extensional one. Also, a Crystal program must be rewritten if it is to be used on objects with different dimensionalities than the targeted objects. In contrast, an mLucid program can be used on objects with various dimensionalities without changing the program, because different dimensionalities of the objects only affect the program's implicit context but not the program's text itself.

Furthermore, using indexical operators is more expressive power in terms of specifying communications in parallel programs than explicitly manipulating indices. This is shown in the indexical parallel functional programming technique described in Chapter 9.

2.6 Parallel Functional Programming

The indexical parallel functional programming described in Chapter 9 is inspired by the parallel functional programming paradigm proposed by [HS86b][Hud86a][Hud86b] [Hud88][Hud89].

One advantage of functional languages is that parallelism in a functional program is implicit, which is called *expression parallelism*; it is manifested solely through data dependencies and the semantics of primitive operators. Expression parallelism in a functional language has two forms [Gol88]:

- *Horizontal* parallelism occurs when two or more arguments of a function application are evaluated in parallel.
- *Vertical* parallelism occurs when an argument is evaluated in parallel with the function application to which it is being passed.

In most implementations of functional languages, parallelism in functional programs is detected by compilers and processes are assigned to parallel processors automatically. Although in certain constrained classes of functional languages like Crystal the mapping of processes to processors can be determined optimally, in the general case the optimal strategy is undecidable. But what if a programmer knows a good mapping strategy for a program executing on a particular machine, but the compiler is not smart enough to determine it? To meet this need, some researchers have designed extensions of functional languages, called *parafunctional programming languages*[HS86b][Hud86a][Hud86b]. The extensions amount to a metalanguage (e.g. annotations) for programmers to express control information on process–to–processor mappings..

The fundamental idea behind process–to–processor mapping is as follows. Consider the expression $e_1 + e_2$. The strict semantics of + allows the subexpressions e_1 and e_2 to

be executed in parallel. Suppose that the programmer wishes to express precisely *where* (i.e. on which processor) the subexpressions are to be evaluated. He/she may do so by annotating the subexpressions with appropriate mapping information. An expression annotated in this way is called a *mapped expression.* It has the form:

```
exp on proc
```

which declares that exp is to be computed on the processor identified by proc. The expression exp is the body of the mapped expression and represents the value of which the overall expression will evaluate. The expression proc must evaluate to a processor identifier (pid), which is in simple case an integer. For example, the expression e_1 + e_2 can be annotated as follows:

$(e_1$ on 0) + (e_2 on 1)

where 0 and 1 are pids. There is a reserved identifier called *self* that can be used in annotations of a mapped expression. When the expression is evaluated, *self* returns the pid of the currently executing processor. For example, suppose that there is a ring of n processors that are numbered consecutively, the above expression can be rewritten as

```
(e1 on left self) + (e2 on right self)
        where left pid = mod (pid-1) n
              right pid = mode (pid+1) n
```

which denotes the computation of the two subexpressions in parallel on the two neighboring processors, with the sum being computed on self.

As another example, in the following parafunctional program for computing a list of factorials on a ring of processors, successive recursive calls require different processes to be executed on different processors.

```
(map fac [2,3,4] on 0
 where map f []     = []
       ,   f(x:xs) = f x : ((map f xs) on (right self)))
```

Note that the recursive call to map is mapped onto the processor to the right of the current one, and thus the elements 2, 6, 24 in the result list are computed on processors 0, 1 and 2, respectively, if the initial pid is 0.

2.7 Systolic Logic Programming

The idea of indexical systolic programming as a general parallel programming paradigm described in Chapter 7, is inspired by the systolic logic programming proposed by [Sha87].

Systolic logic programming, based on a similar motivation to parafunctional programming, is a framework for constructing general–purpose parallel computers, which incorporates the essence of the systolic approach in a software–oriented methodology of algorithm design and implementation. The framework is based on two rather general principles of systolic arrays:

1. localized communication,

2. overlap and balanced computation with communication.

Systolic logic programming consists of four components: an abstract machine, a programming language, a process-to-processor mapping notation, and an algorithm development and programming methodology.

The abstract machine is an infinite *processing surface*: A regular arrangement of processors, each of which has some local memory, can timeshare between several software processes, and can communicate with the neighbors connected to it. Basically the processing surface is an infinite two–dimensional grid of processors, each connected to four neighbors, but other arrangements with higher–connectivity, topology and dimensions are also possible.

The programming language is Concurrent Prolog [Sha86]. Parallelism in the language comes from the AND–parallel evaluation of the goals of a conjunction and from OR–parallel evaluation of the guards of a guarded Horn clause [Ued85]. There is no sequential AND–operator in the language, so every goal of a conjunction creates a new parallel process. Parallel processes communicate through shared logical variables. Synchronization

is based on suspension on read–only variables, which are marked by suffixing them with "?".

The process–to–processor mapping notation is a LOGO–like Turtle program. Each process, like a Turtle, has a position on the processing surface and a heading. A fixed–instruction Turtle program T is associated with every process P, as P@T (read 'execute P at T'). Initially, the position and heading of process P is inherited from the process that invoked it. Its final position and heading are determined by executing the associated turtle program T. The process then runs on the processor at its final position.

The algorithm development and programming methodology identifies two separate, though interrelated activities: the design and implementation of process structures and process behaviors, and design and implementation of the mapping of these structures onto the processing surface. The methodology consists of the following steps:

1. Compose a pure logic program that defines the desired input/output relation.
2. Convert the program into Concurrent Prolog by adding the appropriate read–only annotation and commit operators [Sha86].
3. Identify a systolic algorithm from the program's behavior.
4. Compose a Turtle program that maps processes in execution of the logic program onto processors on the processing surface.

For example, the following is a Turtle–annotated Concurrent Prolog program for multiplying two matrices, based on the classic systolic algorithm (Figure 2.2). It assumes that each of the two input matrices is represented by a stream of streams of their columns and rows, respectively. It produces a stream of streams of the rows of the output matrix. The program operates by spawning a rectangular grid of *ip* processes for computing the inner products of each row and column.

```
mm([], _, []).
mm([X|Xs], Ys, [Z|Zs]) <-
  vm(X, Ys?, Z)@right,
```

```
    mm(Xs?, Ys, Zs)@forward.

vm(_, [], []).
vm(Xs, [Y|Ys], [Z|Zs]) <-
    ip(Xs?, Y, Z)
    vm(Xs, Ys?, Zs)@forward.


ip([X|Xs], [Y|Ys], Z) <-
    Z:= (X * Y) + Z1, ip(Xs?, Ys?, Z1).
ip([], [], 0).
```

Both parafunctional programming and systolic logic programming intend to do explicitly parallel programming in languages with only implicit parallelism. To solve the problem, explicit constructs, i.e. annotations, have to be introduced, so that programmers can use them to express control information such as process–to–processor mapping in functional and logic programs. These explicit constructs are given operational meanings, which obviously diverges from the original mathematical semantics of the programs.

Indexical languages, being semantic enrichment of functional and logic languages, inherit expression and AND/OR parallelism from the base languages. Also, by their indexical semantics, indexical languages introduce a new type of implicit parallelism – context parallelism associated with indexical communication. Since indexical communication is explicit in indexical programs, indexical languages like mLucid can be used for explicitly parallel programming without introducing new constructs, where indexical operations are used as *meaning-preserving* annotations. This will be shown in Chapter 9.

Besides their semantic difference, there are other differences in terms of parallel programming between indexical programming and the annotated functional and logic programming.

Monolithic vs. Distributed For solving multidimensional problems, in the annotation approach, although process–to–processor mapping can be expressed, monolithic data

structures like lists and arrays are passed between processors in the program. In implementations, this results in that either a shared memory model is assumed or that compilers have to detect relations among elements of the structure and distribute them to appropriate processors. By contrast, in indexical programming, in many cases no monolithic structures are needed, and data is naturally distributed to various contexts along with decomposed subproblems.

Dynamic vs. Static Parallel processes are dynamically spawned by (usually recursive) calls in parallel functional and logic programming, even if these process creations are conceivable by the programmer at programming time. In contrast, parallel processes are statically created in indexical programs, when the programmer defines values of objects at various contexts.

Shared variable vs. Indexical operation Communication between parallel processes is implicit in functional and logic parallel programs; it is expressed through shared variables. In contrast, indexical communication between parallel processes in indexical programs is explicit. Explicit communication is easier to be detected by compilers.

Chapter 3

THE INDEXICAL FUNCTIONAL LANGUAGE mLucid

3.1 Syntax and Semantics of mLucid

3.1.1 Syntax

mLucid is a multidimensional extension of the programming language Lucid. Like Lucid, mLucid enriches the semantics of the functional language ISWIM by introducing indexical semantics. In this dissertation, by ISWIM we mean the syntax and semantics shared by all languages in the ISWIM family, and by mLucid we mean the extension of the common syntax and semantics of the ISWIM languages. Therefore, like ISWIM, mLucid is considered to be a parameterized family of indexical languages, each member of which corresponds to a given extensional algebra. All the program examples given in the dissertation are written in a particular mLucid language that is an extension of the programming language pLucid[WA85].

The following is an abbreviation of mLucid's syntax in an extended BNF.

$$
\begin{array}{lll}
< program > & ::= & < dimensionalities >< exp > \\
< dimensionalities > & ::= & dimensionality\ \{< dimensionality >;\} \\
< dimensionality > & ::= & \{< identifier >,\} :' \{'\{< constant >,\}'\}' \\
< exp > & ::= & < constant > \\
 & & \mid < identifier > \\
 & & \mid \theta_1\ < exp > \\
 & & \mid < exp >\ \theta_2\ < exp > \\
 & & \mid \gamma_1\ (< constant >)(< exp >) \\
 & & \mid \gamma_2\ (< constant >)(< exp >, < exp >) \\
 & & \mid if\ < exp >\ then\ < exp >\ else\ < exp >\ fi \\
 & & \mid < cond - exp > \\
 & & \mid < function - call > \\
 & & \mid < where - clause > \\
< cond - exp > & ::= & cond < cbody > end \\
< cbody > & ::= & \{< exp >: < exp >;\} < defaultcase > \\
< defaultcase > & ::= & default : < exp >; \\
< function - call > & ::= & < identifier > (< actuals - list >) \\
 & & \mid < identifier > (< constant >)(< actuals - list >) \\
< actuals - list > & ::= & < exp > \mid < exp >, < actuals - list > \\
< where - clause > & ::= & < exp >\ where\ < definition - list >\ end \\
< definition - list > & ::= & \{< definition >;\} \\
< definition > & ::= & < simple - def > \mid < func - def > \\
< simple - def > & ::= & < identifier > = < exp > \\
< func - def > & ::= & < identifier > (< formals >) =< exp > \mid \\
 & & < identifier > (< constant >)(< formals >) =< exp > \\
< formals > & ::= & < identifier > \mid < identifier >, < formals >
\end{array}
$$

In the above syntax, the symbol θ_n represents an original n-ary operator of ISWIM and γ_n represents an n-ary indexical operator of mLucid.

3.1.2 Indexical Semantics of Expressions

The context space, or the universe of possible worlds, of mLucid is the set of points in an ω-dimensional Euclidean integer space, where ω is the first infinite ordinal number. Each dimension in the context space infinitely extends toward both the positive and negative directions. The value of a syntactic object, an expression, in an mLucid program depends on a point in the context space; it is an intension from the context space to an extension of ISWIM's data domain.

Let D be the flat data domain of ISWIM, mLucid extends D by adding a special data object called *eod* (for "end of data") to it, forming a new domain D_+, where *eod* is at the same level as other base values like integers and reals in the domain.

We define the formal indexical semantics of an expression in mLucid as follow.

Definition 3.1 *The value of an expression in an mLucid program is an intension that is a function*

$$intension \ : \ ID$$

where $ID = Z^\omega \rightarrow D_+$ *and* Z *is the domain of integers.*

Z^ω in the above definition of intensions of expressions specifies the largest number of possible dimensions constituting the context space. In practice, however, in most cases the value of an expression in an mLucid program varies only at points along a finite number of dimensions in the context space, along others it is constant. This number can be determined by analyzing the program, as described in Chapter 4. To simplify the presentation, in the following descriptions, we use the notation mLucid(n) to denote the indexical language with the same syntax and semantics as mLucid, except that its context space is Z^n instead of Z^ω, for a particular n. The formal definition of mLucid(n) is given in Chapter 4. For example, given D as the domain of reals, the value of expression $x + y$ can be the sum of two reals, two vectors of reals and two matrices of reals in mLucid(0), mLucid(1) and mLucid(2), respectively.

3.1.3 Pointwise Extension of ISWIM Operators

mLucid also extends the semantics of ISWIM operators from functions in $D^k \to D$ $(k \geq 1)$ to functions in $ID^k \to ID$ in a pointwise manner. The extension takes two steps. At the first step, we extend the meaning of each operator θ_i of ISWIM to the special data object *eod*. Informally, there are two general rules for this extension.

1. If the result of the operation is a combination of the values of two operands, such as $x + y$, the operator can have either of the following two meanings.

   - It is strict to *eod*, i.e. as long as one of the operands is *eod* then the result is always *eod*. In this case, *eod* is treated in the same way as $\perp$ in ISWIM.
   - It is not strict to *eod*. The result, which is a *non-eod* value, depends on the non-*eod* operand(s), if there are any; otherwise it is *eod*. For example, the result of the operation *(x + eod)* is x's value instead of *eod*.

   In the following definition, we give the first meaning to the operator with the same syntax θ_i, and the second meaning to the operator with the syntax θ_i^*.

2. If the result of the operation is the value of one of the operands, such as *if p then x else y*, we also treat *eod* in the same way as $\perp$ in ISWIM.

We formally define the extension as follows.

Definition 3.2 *Let θ_i^s be an i-ary operator in ISWIM and θ_i and θ_i^* be the extended operators from θ_i^s. Let $e, e_0, e_1, e_2 \in D_+$.*

$$\theta_1\ e = \theta_1^*\ e = \begin{cases} \theta_1^s\ e & \text{if } e \neq eod \\ eod & \text{otherwise} \end{cases}$$

$$e_1\ \theta_2\ e_2 = \begin{cases} e_1\ \theta_2^s\ e_2 & \text{if } e_1 \neq eod \wedge e_2 \neq eod \\ eod & \text{otherwise} \end{cases}$$

$$e_1\ \theta_2^*\ e_2 = \begin{cases} e_1\ \theta_2^s\ e_2 & \text{if } e_1 \neq eod \wedge e_2 \neq eod \\ e_1 & \text{if } e_1 \neq eod \wedge e_2 = eod \\ e_2 & \text{if } e_1 = eod \wedge e_2 \neq eod \\ eod & \text{otherwise} \end{cases}$$

$$\text{if } e_0 \text{ then } e_1 \text{ else } e_2 = \begin{cases} e_1 & \text{if } e_0 = true \\ e_2 & \text{if } e_0 = false \\ eod & \text{if } e_0 = eod \end{cases}$$

From now on, by ISWIM's flat data domain D we mean the extended data domain D_+, and by ISWIM's operators we mean such extended operators.

At the second step, we define the meanings of ISWIM operators in terms of mLucid's indexical semantics. In an mLucid program, the application of an ISWIM operator to its operands whose values are intensions is evaluated pointwise. The result of the application at each context is the result of the operator applying to the values of the operands at that context. For example, for the expression $x + y$ in an mLucid(1) program, the operator '+' sums the values of variables x and y at each context in the one dimensional space, and returns an intension – a vector with infinitely many elements, whose value at a context is the result of the addition at the context. In mLucid, we term such pointwise extended ISWIM operators *pointwise operators*, as they do not switch context when applied to the operands at a context. Formally, we define the second step of the extension as follows.

Definition 3.3 *Let I_{iswim} and I_{mLucid} be the interpretations of ISWIM and mLucid that assign meanings to constant symbols of the languages, respectively. Let θ be an extended ISWIM operator in Definition 3.2*

$$I_{iswim}(\theta) : D^k \rightarrow D \qquad (1 \leq k \leq 3).$$

The formal semantics of the pointwise operator θ in mLucid is a function

$$I_{mLucid}(\theta) : ID^k \rightarrow ID$$

such that

$$\forall p \in Z^\omega \quad I_{mLucid}(\theta)(e_1, \cdots, e_k)(p) = I_{iswim}(\theta)(e_1(p), \cdots, e_k(p))$$

where $e_i \in ID$ *for* $1 \leq i \leq k$.

3.1.4 Pointwise Extension of Constants

In mLucid, a constant in ISWIM is extended pointwise to all contexts in the context space; it is an intension that at all contexts has the same value as the constant in ISWIM's data domain D. The formal definition of the semantics of a constant symbol in mLucid is given as follows.

Definition 3.4 *Let c be a constant symbol in both ISWIM and mLucid and*

$$I_{iswim}(c) \in D.$$

The semantics of c in mLucid is an intension

$$I_{mLucid}(c) : Z^\omega \to D$$

such that

$$\forall p \in Z^\omega \quad I_{mLucid}(c)(p) = I_{iswim}(c).$$

In the following semantic descriptions, we do not distinguish $I_{mLucid}(c)$ which is a constant intension, and $I_{mLucid}(c)(p)$ which is a scalar value, unless it is necessary.

Definition 3.5 (Pointwise Expression) *An expression in mLucid is pointwise if it is*

- *a constant,*
- *an identifier, or*
- *composed of a pointwise operator together with pointwise expressions as the operands.*

Definition 3.6 (Pointwise Function) *A closed function in mLucid defined by*

$$f(x_1, x_2, \cdots, x_n) = e$$

is pointwise if e is pointwise.

3.1.5 Functional Semantics of mLucid

The functional semantics of mLucid is determined by its two formal semantic components, fixed point semantics and indexical semantics. Because of the introduction of indexical semantics, the meaning of an mLucid program P becomes a mapping from intensions that are values of input variables to an intension that is the value of the program's defining expression, i.e.

$$P: \ ID^k \rightarrow ID \qquad (k \in N)$$

where N is the set of natural numbers. To derive the functional semantics of mLucid programs, we need to extend the definition of the fixed point semantics of ISWIM in accordance with indexical semantics of mLucid.

The functional semantics of an ISWIM program is a mapping from a set of values in D associated with input variables to a value in D associated with the defining expression of the program. This functional semantics is determined by the fixed point semantics of the language as follows. An ISWIM program defines a set of equations – the definitions for all non–input variables. Let *Env* be a domain of environments that map variable symbols in the program to values in D, that is,

$$Env = V \rightarrow D$$

where V is the set of all variable symbols in the program, and we assume that they are all distinct. Elements in *Env* form a complete partial order such that

$$env_1 \sqsubseteq env_2 \ \longleftrightarrow \ \forall v \in V \ env_1(v) \sqsubseteq env_2(v) \qquad env_1, env_2 \in Env$$

in terms of the least value $\perp$ in the flat domain D. The equations defining variables in the program can be interpreted as defining a continuous function Φ from *Env* to *Env*. The least fixed point $\hat{env}$ of the function Φ

$$\hat{env} = \Phi(\hat{env})$$

determines the functional semantics of the program. Let *exp* be the subject of the defining expression of the program. The value of *exp*, or the output of the program, is uniquely

determined by $e\hat{n}v$ in the way that all occurrences of each variable symbol v in exp are substituted by the value $e\hat{n}v(v)$.

In mLucid, we extend the fixed point semantics of ISWIM by redefining the domain of environments *Env* and the complete partial order among elements of the redefined *Env*. An environment *env* in mLucid *Env* maps variable symbols in an mLucid program to *intensions* in ID, that is,

$$Env : V \rightarrow ID.$$

Accordingly, the complete partial order among elements in *Env* becomes

$$\begin{aligned}
&env_1 \sqsubseteq env_2 \longleftrightarrow \\
&\forall v \in V\ env_1(v) \sqsubseteq env_2(v) \longleftrightarrow \\
&\forall v \in V\ \forall p \in Z^{\omega}\ env_1(v)(p) \sqsubseteq env_2(v)(p) \\
&where \quad env_1, env2 \in Env
\end{aligned}$$

The least fixed point $e\hat{n}v$ of the function Φ for an mLucid program can be obtained using Kleene's recursion theorem [Kle81]. In particular, $e\hat{n}v = \bigsqcup_{i=0}^{\infty} \Phi^i(env_0)$ where the initial environment env_0 assigns $\perp$ to the value of each non–input variable at every context, and gives certain intensions to input variables of the program.

In the following, based on the above fixed point semantics for mLucid programs, we define a formal denotational semantics for mLucid. For clarity, we assume that the following syntactic transformation has occurred on the mLucid source program. The equations defining functions have been mapped into "curried" lambda expressions. For example,

$$\text{fac(n)} = \text{if n eq 1 then 1 else n} * \text{fac(n-1) fi}$$

will be converted into

$$\text{fac} = \lambda\text{n. if n eq 1 then 1 else n} * \text{fac(n-1) fi.}$$

mLucid's denotational semantics is captured by the semantic function

$$E : Exp \rightarrow (Env \rightarrow (Z^{\omega} \rightarrow D)),$$

where *Exp* is the syntactic domain of mLucid expressions. For convenience, we will omit the details of the syntactic and semantic domains, as well as errors. In the following

definition, $E[exp]\ env\ p$ denotes the value of exp at the context p in the environment env.

$$E[< constant >]\ env\ p = I_{mLucid}(< constant >)$$
$$E[< identifier >]\ env\ p = env(< identifier >)(p)$$
$$E[\theta_1\ < exp >]\ env\ p = I_{mLucid}(\theta_1)(E[< exp >]\ env)\ p$$
$$E[< exp_1 >\ \theta_2\ < exp_2 >]\ env\ p =$$
$$I_{mLucid}(\theta_2)(E[< exp_1 >]\ env\ ,\ E[< exp_2 >]\ env)\ p$$
$$E[if\ < exp_0 >\ then\ < exp_1 >\ else\ < exp_2 >\ fi]\ env\ p =$$
$$if\ (E[< exp_0 >]\ env\ p)\ then\ E[< exp_1 >]\ env\ p\ else\ E[< exp_2 >]\ env\ p$$
$$E[\lambda x.\ < exp >]\ env\ p = \lambda x'.\ E[< exp >]\ [x'/x]env\ p$$
$$E[< exp_1 >\ < exp_2 >]\ env\ p = (E[< exp_1 >]\ env\ p)(E[< exp_2 >]\ env\ p)$$
$$E[< exp >\ where\ f_i\ = < exp_i >\ ; \cdots end]\ env\ p = E[< exp >]\ e\hat{n}v\ p$$
$$where\ \ e\hat{n}v = [\cdots E[< exp_i >]\ e\hat{n}v/f_i \cdots]env.$$

In the above definitions, the notation *[v/x]env* denotes an environment that maps variable symbols to intensions as same as *env* except that for the variable symbol x it returns the intension v. The semantics of a complete mLucid program is given by the semantic function E_P, defined by:

$$E_P[prog]\ p\ =\ E[prog]\ env_0\ p.$$

Notice that in the denotational semantics of mLucid we have defined so far, the context parameter p is redundant. For there is no context–switching operations involved in the semantics. All operations defined so far are pointwise in terms of the context space of the program. However, even only using this partially defined mLucid, can we write more concise programs for solving some problems than using ISWIM or other functional language. For example, the following is an mLucid(2) program without context switching operators. It computes the matrix sum of a list of matrices. In the program, the data type of input variable *mlist* is *list*, and each element of *mlist* is a matrix represented by an intension varying in the two dimensions.

```
SumList(mlist)
where
```

```
SumList(x) = if isnil(tl(x)) then hd(x) else hd(x) + SumList(tl(x)) fi;
end.
```

3.2 Semantics of Primitive Indexical Operators

mLucid has five primitive indexical operators: **origin**, **next**, **prev** (for "previous"), **fby** (for "followed by") and **before**. The operators switch context from a point to another point along a given dimension in the context space.

The operator **origin** switches context from any point along a given dimension to the origin (coordinate 0) of the dimension. Let d be a nonnegative integer indicating a dimension and x be an intension, The result of the indexical operation *origin(d)(x)* is the intension whose value at all points along dimension d in the context space is x's value at the origin of the dimension. For example, let x represent an $n \times m$ matrix X in mLucid(2); x's value at the context with coordinates i and j for dimension 1 and 2 is element $x_{i,j}(0 \leq i \leq n-1, 0 \leq j \leq m-1)$ of X; x's value at all other contexts is *eod*. Then the result of *origin(1)(x)* at any context p is x's value at the first row (row 0) and the same column as p, i.e. $x_{0,j}$ where j is p's coordinate for dimension 2.

In the definitions of formal semantics of mLucid indexical operators, we use the following notations.

Given an intension $x \in ID$, a context $p \in Z^\omega$, a dimension indicator $d \in N$ and an integer $k \in Z$, let

- $x_p = x(p)$ denote the value of x at p,
- $[k/d]p \in Z^\omega$ denote the context whose coordinate for dimension d is k and whose other coordinates are as same as p's,
- $p(d) \in Z$ denote p's coordinate for dimension d.

The formal semantics of *origin* is defined as follow.

Definition 3.7 *origin is a function*

$$origin : N \to (ID \to ID).$$

The indexical semantics of the operator **origin** *is defined by:*

$$origin = \lambda d.\lambda x.\lambda p.\ x_{[0/d]p}.$$

The denotational semantics of the operation **origin(d)(exp)** *is defined by:*

$$E[origin(< constant >)(< exp >)]\ env\ p = E[< exp >]\ env\ [0/I_{mLucid}(< constant >)]p.$$

The operators **next** and **prev** switch context from a point to its neighboring points along a given dimension toward the positive and negative directions. The result of the indexical operations *next(d)(x)* and *prev(d)(x)* are the intensions whose values at a point p in the context space are x's values at the points next to p along dimension d in the positive and negative directions, respectively. For example, let x represent a matrix in mLucid(2) as in the last example. The operation *next(1)(x)* (or *prev(1)(x)*) results in that the layout of the matrix represented by x is shifted by one point along the axis of dimension 1 toward the negative (or the positive) direction.

The formal semantics of *next* and *prev* is defined as follows.

Definition 3.8 **next** *and* **prev** *are functions*

$$next : N \rightarrow (ID \rightarrow ID)$$

$$prev : N \rightarrow (ID \rightarrow ID).$$

The indexical semantics of the operators **next** *and* **prev** *is defined by:*

$$next = \lambda d.\lambda x.\lambda p.\ x_{[p(d)+1/d]p}$$

$$prev = \lambda d.\lambda x.\lambda p.\ x_{[p(d)-1/d]p}.$$

The denotational semantics of operations **next(d)(exp)** *and* **prev(d)(exp)** *is defined by:*

$$E[next(< constant >)(< exp >)]\ env\ p = E[< exp >]\ env\ [p(d) + 1/d]p$$
$$where\ d = I_{mLucid}(< constant >)$$

$$E[prev(< constant >)(< exp >)]\ env\ p = E[< exp >]\ env\ [p(d) - 1/d]p$$
$$where\ d = I_{mLucid}(< constant >)$$

The operators **fby** and **before** switch context according to the position of a point in the context space. *fby* behaves as the *identity* operator at contexts on the non–positive side of a given dimension, and behaves as the *prev* operator at contexts on the positive side of the dimension. The result of the indexical operation *fby(d)(x,y)* is an intension. It can be considered as the concatenation of one half value of x on the non–positive side of dimension d and one half value of y' on the positive side of dimension d, where y' is the result of shifting the value of y one point along dimension d toward the positive direction.

Conversely, *before* behaves as the *identity* operator at contexts on the non–negative side of a given dimension, and behaves as the *next* operator at contexts in the negative side of the dimension. The result of the indexical operation *before(d)(x,y)* is an intension. It can be considered as the concatenation of one half value of x' on the negative side of dimension d and one half value of y on the non–negative side of dimension d, where x' is the result of shifting the value of x one point along dimension d toward the negative direction.

The formal semantics of *fby* and *before* is defined as follows.

Definition 3.9 **fby** *and* **before** *are functions*

$$fby : N \to ((ID \times ID) \to ID)$$

$$before : N \to ((ID \times ID) \to ID).$$

The indexical semantics of the operators **fby** *and* **before** *is defined by:*

$$fby = \lambda d.\lambda(x,y).\lambda p.\ if\ p(d) \leq 0\ then\ x_p\ else\ y_{[p(d)-1/d]p}$$

$$before = \lambda d.\lambda(x,y).\lambda p.\ if\ p(d) < 0\ then\ x_{[p(d)+1/d]p}\ else\ y_p$$

The denotational semantics of the operations $\mathbf{fby(d)(exp_1, exp_2)}$ *and* $\mathbf{before(d)(exp_1, exp_2)}$ *is defined by:*

$$E[fby(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$if\ p(d) \leq 0\ then\ E[< exp_1 >]\ env\ p\ else\ E[< exp_2 >]\ env\ [p(d) - 1/d]p$$
$$where\ \ d = I_{mLucid}(< constant >)$$

$$E[before(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$if\ p(d) < 0\ then\ E[< exp_1 >]\ env\ [p(d) + 1/d]p\ else\ E[< exp_2 >]\ env\ p$$
$$where\ \ d = I_{mLucid}(< constant >).$$

Using indexical operators *fby* and *before*, a special operator **index**, which is built–in in mLucid, can be defined. *index* is a function

$$index : N \rightarrow ID.$$

The application of *index* to a dimension indicator d is the intension whose value at a context p is p's coordinate for dimension d. *index* is defined by the recursive definition

```
index(d) = fby(d)(before(d)(index(d)−1,0), index(d)+1)
```

Informally, the above definition defines that the value of *index(d)* at the contexts on the origin of dimension d in the context space is zero. At each of other contexts, its value is either an increment or decrement of itself at the neighboring context in the positive or negative direction along the dimension, depending the position of the context in the context space. The formal indexical semantics and denotational semantics of the operator **index** is defined by:

$$index = \lambda d.\lambda p.\ p(d)$$

$$E[index(< constant >)]\ env\ p = p(I_{mLucid}(< constant >)).$$

There are other built–in indexical operators in mLucid, which are defined by the primitive indexical operators. We will define them in the course of the dissertation before they are used in programming examples.

Chapter 4

DIMENSIONALITY ANALYSIS

4.1 Definition

In general, the value of an mLucid expression as an intension may be different at different contexts in the context space having an arbitrary number of dimensions. However, in a particular mLucid program, the value of an expression as an intension may only vary in a finite number of dimensions in the context space, and keeps constant in the rest of the dimensions. By constant in a dimension, we mean that an expression has the same value at all points along that dimension.

We define the dimensionality *DIM(x)* of an intension x as the set of dimensions in which x's value varies and in all the other dimensions outside of the set it is constant.

Definition 4.1 (Dimensionality) *The dimensionality DIM(x) of an intension $x \in ID$ is the set of nonnegative integers, such that*

$$DIM(x) = \{d \in N \mid \exists p \in Z^{\omega}\ \exists k \in Z\ x_p \neq x_{[k/d]p}\}$$

For example, a constant c in mLucid has dimensionality DIM(c) = {}. An intension v representing a non–constant vector laid in dimension 0 has dimensionality DIM(v) = {0}. An intension m representing a matrix with distinct rows and columns laid in dimensions 0 and 1 has dimensionality DIM(m) = {0, 1}.

The notion of dimensionality is important both for deriving the meanings of mLucid programs and for implementing the programs. In programming, the value of an expression consisting of operations on intensions with different dimensionalities may be different. For example, for the intensions c, v and m described above, the addition of each pair of them has the following (informal) meaning.

$c + v$	add c to each element of v	where DIM$(c + v) = \{0\}$
$c + m$	add c to each element of m	where DIM$(c + m) = \{0,1\}$
$v + m$	add v to each column of m	where DIM$(v + m) \subseteq \{0,1\}$

In implementation, the value of an expression e at a context p only depends on p's coordinates for the dimensions in e's dimensionality. In other words, e's value at a context p is known as soon as the value at any context p_0 is evaluated, where p's coordinates for the dimensions in e's dimensionality is the same as p_0's. For example, for constant c only a single value c_{p_0} is needed, where $\forall d \in N\ p_0(d) = 0$. For vector v only values v_{p_0} along dimension 0 are needed, where $\forall d \in N\ d \neq 0 \rightarrow p_0(d) = 0$. For matrix m only values m_{p_0} in the subspace consisting of dimensions 0 and 1 are needed, where $\forall d \in N\ d \neq 0 \wedge d \neq 1 \rightarrow p_0(d) = 0$.

We will prove that an intension x is constant in dimensions outside of *DIM(x)*, that is, for all contexts p and q in the context space, as long as the coordinates of p and q for dimensions in *DIM(x)* agree the value of x at p and q is always same. First we need to introduce the notion of *continuous intensions.* The motivation for defining the continuity of an intension is concerned with the computability of the intension. Consider an intension x in the ω–dimensional context space whose value at a context p depends on an infinite number of coordinates of p. In other words, there is a point q in the context space such that the value of x at p and q may be different even if any finite number of the coordinates of p and q agree. Intuitively, x_p is not computable because an infinite number of coordinates have to be examined to compute x_p. For example, let x_p be the minimum of those integers i such that there are infinitely many occurrences of i among all the ω coordinates of p if i exists; otherwise it is $\perp$. Consider the following contexts in the

context space

$$p_0 =< 0,1,1,1,\cdots >$$
$$p_1 =< 0,0,1,1,1,\cdots >$$
$$p_2 =< 0,0,0,1,1,1,\cdots >$$
$$\cdots$$

Formally, p_i for $0 \leq i < \infty$ is defined as

$$\forall d \in N \; p_i(d) = \begin{cases} 0 & \text{if } d \leq i \\ 1 & \text{otherwise} \end{cases}$$

x's value at all these points p_i is not $\perp$, but not computable. For all finite i, x_{p_i} is 1, that is,

$$\lim_{i \to \infty} x_{p_i} = \lim_{i \to \infty} 1 = 1,$$

while

$$x_{(\lim_{i \to \infty} p_i)} = 0.$$

To define the continuity of an intension at a context, and hence in the context space, we first define the *local dimension dependence set* of the intension. The local dimension dependence set is a family of sets of dimensions such that the value of the intension at the context can be determined by the coordinates for the dimensions in any element of the family. Therefore, the local continuity of the intension can be defined as the existence of a finite element in the local dimension dependence set. Hence, the continuity of the intension can be defined as the local continuity at every context in the context space.

Definition 4.2 (Local Dimension Dependence Set) *For an intension $x \in ID$ and a context $p \in Z^\omega$, the family $DDS_p(x)$ of sets of dimensions is defined by*

$$DDS_p(x) = \{D \subseteq N \mid \forall q \in Z^\omega \; (\forall d \in D \; p(d) = q(d) \rightarrow x_p = x_q)\}.$$

Proposition 4.1 *For all intensions $x \in ID$ and all contexts $p \in Z^\omega$, $DDS_p(x)$ is not empty and $N \in DDS_p(x)$.*

Proof: Trivial. □

Definition 4.3 (Continuity at a context) *An intension $x \in ID$ is continuous at a context $p \in Z^\omega$ if and only if (i) $DDS_p(x)$ has a finite element and (ii) $x_p \neq \perp$.*

Definition 4.4 (Continuity) *An intension $x \in ID$ is continuous if and only if $\forall p \in Z^\omega$ x is continuous at p.*

In the following, we define the global dimension dependence set of an intension as a family of sets of dimensions, such that the value of the intension at *all* contexts can be determined by the coordinates for the dimensions in any element of the family. We then prove that the dimensionality of a continuous intension is the least element of the global dimension dependence set of the intension.

Definition 4.5 (Dimension Dependence Set) *For an intension $x \in ID$, the family DDS(x) of sets of dimensions is defined by*

$$\begin{aligned} DDS(x) = \quad & \{D \subseteq N \mid \forall p, q \in Z^\omega \\ & (\forall d \in D\ p(d) = q(d) \ \rightarrow \ x_p = x_q)\}. \end{aligned}$$

Proposition 4.2 *The dimension dependence set DDS(x) of an intension $x \in ID$ is the intersection of the local dimension dependence sets $DDS_p(x)$ for all contexts $p \in Z^\omega$.*

Proof: Trivial. □

Lemma 4.1

$$\begin{aligned} & \forall x \in ID\ \forall p \in Z^\omega\ \forall d \in N\ \forall k \in Z \\ & d \notin DIM(x) \ \rightarrow \ x_p = x_{[k/d]p}. \end{aligned}$$

Proof: Straightforward from the definition of DIM(x). □

Lemma 4.2 *For a continuous intension $x \in ID$ and a context $p \in Z^\omega$, let $D_p \in DDS_p(x)$ be finite. Let $\overline{DIM(x)}_p = D_p - DIM(x)$.*

$$\forall d \in \overline{DIM(x)}_p\ \forall k \in Z\ D_p \in DDS_{p'}(x)$$

where $p' = [k/d]p$.

Proof: Let $q' \in Z^\omega$ and $q = [p(d)/d]q'$, $\forall d' \in D_p$ if $p'(d') = q'(d')$ then $p(d') = q(d')$. By Lemma 4.1, $x_{p'} = x_p$ and $x_q = x_{q'}$. By the definition of D_p, $x_p = x_q$. □

Theorem 4.3 *For a continuous intension $x \in ID$, the dimensionality DIM(x) of x is the least element of DDS(x), i.e.*

$$DIM(x) = \bigcap DDS(x) \in DDS(x).$$

Proof:

1. We prove that $DIM(x) \in DDS(x)$. For $p \in Z^\omega$, let $D_p \in DDSp(x)$ be finite. Let $\overline{DIM(x)}_{D_p} = D_p - DIM(x)$. For $p, q \in Z^\omega$, let

$$diff(x, p, q, D_p) = \overline{DIM(x)}_{D_p} - \{d \in N \mid p(d) \neq q(d)\}.$$

   (a) When $|diff(x, p, q, D_p)| = 0$

$$(\forall d \in DIM(x)\ p(d) = q(d)) \Rightarrow (\forall d \in D_p\ p(d) = q(d)) \Rightarrow x_p = x_q.$$

   (b) Assume that when $|diff(x, p, q, D_p)| = n$

$$(\forall d \in DIM(x)\ p(d) = q(d)) \rightarrow x_p = x_q.$$

   (c) When $|diff(x, p, q, D_p)| = n + 1$, let $d' \in diff(x, p, q, D_p)$ and $p' = [q(d')/d']p$. By Lemma 4.1, $x_p = x_{p'}$. By lemma 4.2, $D_p \in DDS_{p'}(x)$, and $|diff(x, p', q, D_p)| = n$. Therefore

$$(\forall d \in DIM(x)\ p(d) = q(d)) \Rightarrow$$
$$(\forall d \in DIM(x)\ p'(d) = q(d)) \Rightarrow$$
$$x_{p'} = x_q \Rightarrow x_p = x_q.$$

2. We prove $\forall D \in DDS(x)\ DIM(x) \subseteq D$. Let $d \in DIM(x)$ but $d \notin D$. According to Definition 4.1, $\exists p \in Z^\omega$ and $\exists k \in Z$ such that $x_p \neq x_{p'}$ where $p' = [k/d]p$. Therefore, $\forall d \in D\ p(d) = p'(d)\ \wedge\ x_p \neq x_{p'}$ This is in contradiction with the definition of D. □

The dimensionality *DIM(x)* of an intension x captures a global view of how x varies in the context space. That is, as long as there are two contexts along a dimension in the

context space at which x's value is different, we consider that x varies in the dimension, even if there are many points along the dimension at which x's value is constant. To capture a local view of x's variation in dimensions with respect to a context p, we define the local dimensionality of x at p as the set of the dimensions in which there is context q such that x at p is different than x at q. We give the formal definition of the local dimensionality as follows.

Definition 4.6 (Local Dimensionality) *The local dimensionality $DIM_p(x)$ of an intension $x \in ID$ and a context $p \in Z^\omega$ is the set of nonnegative integers*

$$DIM_p(x) = \{d \in N \mid \exists k \in Z\ x_p \neq x_{[k/d]p}\}$$

Theorem 4.4 *For an intension $x \in ID$ and a context $p \in Z^\omega$,*

$$DIM_p(x) = \bigcap DDS_p(x).$$

Proof: Let $D_0 = \bigcap DDS_p(x)$.

1. We prove $D_0 \subseteq DIM_p(x)$. Let $d \in D_0$ but $d \notin DIM_p(x)$. By the definition of $DIM_p(x)$, $\forall k \in Z\ x_p = x_{[k/d]p}$. Let $D = N - \{d\}$, then $D \in DDS_p(x)$. But $D_0 \not\subseteq D$. This is in contradiction with that D_0 is the intersection of $DDS_p(x)$.

2. We prove $D_0 \supseteq DIM_p(x)$. Let $d \notin D_0$ but $d \in DIM_p(x)$. By the definition of $DIM_p(x)$, let $k \in Z$ such that $x_p \neq x_{[k/d]p}$. Since $d \notin D_0$, let $D \in DDS_p(x)$ such that $d \notin D$ and $\forall d' \in D\ p(d') = ([k/d]p)(d')$, but $x_p \neq x_{[k/d]p}$. This is in contradiction with that $D \in DDS_p(x)$. □

Corollary 4.5 *If an intension $x \in ID$ is continuous at $p \in Z^\omega$, $DIM_p(x)$ is finite.*

Notice that $DIM_p(x)$ may not be in $DDS_p(x)$, or $DDS_p(x)$ may not be closed under intersection. For example, let intension x be defined as

$$x = \lambda p. \begin{cases} 0 & \text{if } p(0) = 0 \vee p(1) = 0 \\ 1 & \text{otherwise} \end{cases}$$

At a context p with $p(0) = 0$ and $p(1) = 0$, $DDS_p(x)$ consists of the sets containing dimension 0 or/and 1, but its intersection $DIM_p(x)$ is $\{\}$.

Proposition 4.3 *For an intension $x \in ID$,*

$$DIM(x) = \bigcup_{p \in Z^\omega} DIM_p(x).$$

Proof: Straightforward from the definitions of *DIM(x)* and $DIM_p(x)$. □

Corollary 4.6 *If the dimensionality DIM(x) of an intension $x \in ID$ is finite, then for all contexts $p \in Z^\omega$ the local dimensionality $DIM_p(x)$ is also finite.*

Notice that the dimensionality of an intension can be infinite even if at all contexts in the context space its local dimensionalities are finite. For example, let intension x be defined as

$$x = \lambda p.p(p(3)).$$

At each context p, $DIM_p(x) = \{3, p(3)\}$, but $DIM(x) = N$.

4.2 Dimensionalities of Primitive Operations

The result of an operation may have different dimensionality than that of its operand(s). For instance, the dimensionalities for expressions $c + v$, $c + m$ and $v + m$ in the previous examples. The following propositions show how the dimensionality of a primitive operation depends on the semantics of the operator and dimensionalities of the operands.

Proposition 4.4 *Let θ_1 be a unary pointwise operator. Given a continuous intension $x \in ID$,*

1. *the intension $(\theta_1\ x)$ is continuous,*
2. $$DIM(\theta_1\ x) \subseteq DIM(x).$$

Proof: Since

$$\forall p, p' \in Z^\omega \quad x_p = x_{p'} \longrightarrow (\theta_1\ x)_p = (\theta_1\ x)_{p'}$$

1. for all $p \in Z^\omega$, let D *finite* $\in DDS_p(x)$ then $D \in DDS_p(\theta_1\ x)$;

2. for all $d \in DIM(\theta_1\ x)$, if $(\theta_1\ x)_p \neq (\theta_1\ x)_{p'}$ then $x_p \neq x_{p'}$ where $p' = [k/d]p$ for some $k \in Z$. □

The inclusion in the other direction, $DIM(\theta_1\ x) \supseteq DIM(x)$, does not hold. For example, suppose that *DIM(x)*={0} and x's value at contexts with positive and non-positive coordinates for dimension 0 is 1 and -1, respectively. The result of the unary pointwise operation *abs(x)* has dimensionality {}, and hence $DIM(\theta_1\ x) \not\supseteq DIM(x)$.

Proposition 4.5 *Let θ_2 be a binary pointwise operator. Given two continuous intensions $x \in ID$ and $y \in ID$,*

1. *the intension $(x\ \theta_2\ y)$ is continuous,*
2. $$DIM(x\ \theta_2\ y) \subseteq DIM(x) \cup DIM(y).$$

Proof: Since

$$\forall p, p' \in Z^\omega \quad x_p = x_{p'} \wedge y_p = y_{p'} \longrightarrow (x\ \theta_2\ y)_p = (x\ \theta_2\ y)_{p'}$$

1. for all $p \in Z^\omega$, let $D_x\ finite\ \in DDSp(x)$ and $D_y\ finite\ \in DDS_p(y)$, then $(D_x \cup D_y) \in DDS_p(x\ \theta_2\ y)$;

2. for all $d \in DIM(x\ \theta_2\ y)$, if $(x\ \theta_2\ y)_p \neq (x\ \theta_2\ y)_{p'}$ then $x_p \neq x_{p'} \vee y_p \neq y_{p'}$ where $p' = [k/d]p$ for some $k \in Z$. □

The inclusion in the other direction, $DIM(x\ \theta_2\ y) \supseteq DIM(x) \cup DIM(y)$, does not hold. For example, suppose that *DIM(x)*={0} and x's value at contexts with positive and non-positive coordinates for dimension 0 is 1 and -1, respectively. *DIM(y)*={0} and y's value at contexts with positive and non-positive coordinates for dimension 0 is -1 and 1, respectively. The result of binary pointwise operation $x * y$ has dimensionality {}, instead of {0}.

Proposition 4.6 *Given three continuous intensions $c \in Z^\omega \rightarrow \{true, false\}$, $x \in ID$ and $y \in ID$,*

1. *the intension* $(if\ c\ then\ x\ else\ y)$ *is continuous,*
2. $$DIM(if\ c\ then\ x\ else\ y) \subseteq DIM(c) \cup DIM(x) \cup DIM(y).$$

Proof: Since

$$\forall p, p' \in Z^\omega \quad c_p = c_{p'} \wedge x_p = x_{p'} \wedge y_p = y_{p'} \longrightarrow$$
$$(if\ c\ then\ x\ else\ y)_p = (if\ c\ then\ x\ else\ y)_{p'}$$

1. for all $p \in Z^\omega$, let $D_c\ finite\ \in DDS_p(x)$, $D_x\ finite\ \in DDS_p(y)$ and $D_y\ finite\ \in DDS_p(y)$, then $(D_c \cup D_x \cup D_y) \in DDS_p(if\ c\ then\ x\ else\ y)$;
2. for all $d \in DIM(if\ c\ then\ x\ else\ y)$, if $(if\ c\ then\ x\ else\ y)_p \neq (if\ c\ then\ x\ else\ y)_{p'}$, then $c_p \neq c_{p'} \vee x_p \neq x_{p'} \vee y_p \neq y_{p'}$, where $p' = [k/d]p$ for some $k \in Z$. □

The inclusion in the other direction, $DIM(if\ c\ then\ x\ else\ y) \supseteq DIM(c) \cup DIM(x) \cup DIM(y)$, does not hold. For example, suppose that *DIM(c)*=$\{0\}$ and c's value at contexts with positive and non-positive coordinates for dimension 0 is *true* and *false*, respectively. *DIM(x)*=$\{0\}$ and x's value at contexts with positive and non-positive coordinates for dimension 0 is 1 and -1, respectively. *DIM(y)*=$\{0\}$ and y's value at contexts with positive and non-positive coordinates for dimension 0 be -1 and 1, respectively. The result of conditional pointwise operation *if c then x else y* has dimensionality $\{\}$, instead of $\{0\}$.

Proposition 4.7 *Given a dimensional indicator* $d \in N$ *and a continuous intension* $x \in ID$,

1. *the intension* `origin(d)(x)` *is continuous,*
2. $$DIM(origin(d)(x)) \subseteq DIM(x) - \{d\}.$$

Proof: Let $y = origin(d)(x)$.

1. For all $p \in Z^\omega$, let $D \in DDS_{[0/d]p}(x)$ be finite.

$$\forall q \in Z^\omega\ (\forall d' \in D\ p(d') = q(d')) \rightarrow y_p = x_{[0/d]p} = x_{[0/d]q} = y_q.$$

2. (a) Prove $d \notin DIM(y)$.

$$\forall p \in Z^\omega \forall k \in Z \; y_p = y_{[k/d]p} = x_{[0/d]p}.$$

(b) Prove $DIM(y) \subseteq DIM(x)$. Let $d' \in DIM(y)$ and $d' \neq d$. Let $p \in Z^\omega$ and $k \in Z$ such that $y_p \neq y_{[k/d']p}$, then $x_{[0/d]p} \neq (x_{[0/d][k/d']p})$. Therefore $d' \in DIM(x)$. □

The inclusion in the other direction, $DIM(origin(d)(x)) \supseteq DIM(x) - \{d\}$, does not hold. For example, suppose that *DIM(x)*={0,1} and x's value at each context is the product of the coordinates of the context for dimensions 0 and 1. The result of indexical operation *origin(0)(x)* has dimensionality {}, instead of {1}.

Proposition 4.8 *Given a dimensional indicator $d \in N$ and a continuous intension $x \in ID$,*

1. *the intension* `next(d)(x)` *is continuous,*
2.

$$DIM(next(d)(x)) = DIM(x).$$

Proof: Let $y = next(d)(x)$.

1. For all $p \in Z^\omega$, let $D \in DDS_{[p(d)+1/d]p}(x)$ be finite.

$$\forall q \in Z^\omega \; (\forall d' \in D \; p(d') = q(d')) \; \rightarrow \; y_p = x_{[p(d)+1/d]p} = x_{[q(d)+1/d]q} = y_q.$$

2. Let $p \in Z^\omega$, $d' \in N$, and $k \in Z$. If $d = d'$,

$$y_p \neq y_{[k/d']p} \; \leftrightarrow \; x_{[p(d)+1/d]p} \neq x_{[k+1/d]p}$$

If $d \neq d'$, let $p' = [p(d) + 1/d]p$

$$y_p \neq y_{[k/d']p} \; \leftrightarrow \; x_{p'} \neq x_{[k/d']p'}$$

Therefore $DIM(next(d)(x)) = DIM(x)$. □

Proposition 4.9 *Given a dimensional indicator $d \in N$ and a continuous intension $x \in ID$,*

1. *the intension* `prev(d)(x)` *is continuous,*
2. $$DIM(prev(d)(x)) = DIM(x).$$

Proof: Similar to the proof of Proposition 4.7. □

Proposition 4.10 *Given a dimensional indicator $d \in N$ and two continuous intensions $x \in ID$ and $y \in ID$,*

1. *the intension* `fby(d)(x,y)` *is continuous,*
2. $$DIM(fby(d)(x,y)) \subseteq DIM(x) \cup DIM(y) \cup \{d\}.$$

Proof: Let $z = fby(d)(x,y)$.

1. Let $p \in Z^\omega$, $D_x \in DDS_p(x)$ and $D_y \in DDS_{[p(d)-1/d]p}(y)$ be finite. Let $D = D_x \cup D_y \cup \{d\}$. If $p(d) \leq 0$,

   $$\forall q \in Z^\omega\ (\forall d' \in D\ p(d') = q(d')) \rightarrow z_p = x_p = x_q = z_q$$

   If $p(d) > 0$,

   $$\forall q \in Z^\omega\ (\forall d' \in D\ p(d') = q(d')) \rightarrow z_p = y_{[p(d)-1]p} = y_{[q(d)-1]q} = z_q.$$

2. Let $d' \in DIM(z) \wedge d' \neq d$, $p \in Z^\omega$ and $k \in Z$ such that $z_p \neq z_{[k/d']p}$.

   $$p(d) \leq 0 \rightarrow x_p = z_p \neq z_{[k/d']p} = x_{[k/d']p}$$
   $$p(d) > 0 \rightarrow y_{p'} = z_p \neq z_{[k/d']p} = y_{[k/d']p'}$$

   where $p' = [p(d) - 1/d]p$. □

The inclusion in the other direction, $DIM(fby(d)(x,y)) \supseteq DIM(x) \cup DIM(y) \cup \{d\}$, does not hold. For example, suppose that *DIM(x)*=$\{0\}$ and x's value at contexts with positive and non–positive coordinates for dimension 0 is 1 and -1, respectively. *DIM(y)*=$\{0\}$ and y's value at contexts with non–negative and negative coordinates for dimension 0 is -1 and 1, respectively. The result of indexical operation *fby(0)(x,y)* has dimensionality $\{\}$, instead of $\{0\}$.

Proposition 4.11 *Given a dimensional indicator $d \in N$ and two continuous intensions $x \in ID$ and $y \in ID$,*

1. *the intension* `before(d)(x,y)` *is continuous,*
2. $$DIM(before(d)(x,y)) \subseteq DIM(x) \cup DIM(y) \cup \{d\}.$$

Proof: Similar to the proof of Proposition 4.9. □

The proper inclusions in Propositions 4.3–4.6 and 4.9–4.10 do not hold. A trivial example is when all x, y and c have dimensionality $\{\}$ and have different constant values. In this case

$$DIM(\theta_1\ x) = DIM(x) = \{\}$$
$$DIM(x\ \theta_2\ y) = DIM(x) \cup DIM(y) = \{\}$$
$$DIM(if\ c\ then\ x\ else\ y) = DIM(c) \cup DIM(x) \cup DIM(y) = \{\}$$
$$DIM(origin(d)(x)) = DIM(x) - \{d\} = \{\}$$
$$DIM(fby(d)(x,y)) = DIM(x) \cup DIM(y) \cup \{d\} = \{d\}$$
$$DIM(before(d)(x,y)) = DIM(x) \cup DIM(y) \cup \{d\} = \{d\}.$$

4.3 Declaration

We define the maximum dimensionality of an input variable as the set of dimensions in which its input value is allowed to vary. The maximum dimensionality of an input variable can be declared through a construct of mLucid, called *dimensionality declaration.* The actual dimensionality of an input variable is a subset of the declared maximum dimensionality. The declaration is of the form

dimensionality x,y:{0,1}; z:{};

Its syntax is defined in Section 3.1.1 of Chapter 3. The formal meaning of the declaration is defined as follows.

Definition 4.7 (*Allowable* **Initial Environment**) *Let X be the set of free variables of a program P, and each $x \in X$ be declared in P with a finite set of nonnegative integers $Dim_{max}(x)$. For all initial environments $env_0 \in Env$ such that the denotational semantics of P is defined by $E[P]$ env_0 p, env_0 is* **allowable** *if for all $x \in X$ $env_0(x)$ is continuous and*

$$DIM(env_0(x)) \subseteq Dim_{max}(x)$$

For example, the following mLucid program with dimensionality declaration

```
dimensionality x,y: {1,2};
x + y
```

may compute the results in Table 4.1, depending on the actual dimensionalities of x and y, where x and y take integer values.

DIM(x)	DIM(y)	DIM(x+y)	x+y
{}	{}	{}	sum two integers
{1}	{}	{1}	add an integer to each element of a vector
{1}	{1}	{1}	sum two vectors
{1}	{2}	{1,2}	sum each pairs of elements of two vectors
{1}	{1,2}	{1}	add a vector to each column (or row) of a matrix
{1,2}	{1,2}	{1,2}	sum two matrices

Table 4.1: A dimensionality example

A smaller actual dimensionality of an input variable than its declared maximum dimensionality can be specified by an input convention that associates an input value with

dimensionality. When a value $v \in D$ of an input variable x at a context p is input, the value can be associated with a set C of pairs (d,s) where d is a dimension indicator and s is a sign (a boolean value). The dimension indicator d denotes a dimension, which must be in $Dim_{max}(x)$, and the sign s denotes a direction in the dimension, positive or negative. The convention defines that v is x's value at p and all other contexts p' such that p and p' have the same coordinates for the dimensions in $Dim_{max}(x) - \{d \mid (d,s) \in C\}$, and p' is on the positive or negative side of p in each dimension d in $(d,s) \in C$ depending on the sign s. The input convention is formally defined as follows.

Definition 4.8 (Input Convention) *The input convention of an input variable x of a program is a set of pairs (v,C), where $v \in D$ be an input value for the value x_p of x at context $p \in Z^\omega$, and $C \subset N \times \{true, false\}$ such that*

$$\begin{aligned}
&\forall (d,s) \in C \;\; d \in Dim_{max}(x) \wedge \\
&\forall p' \in Z^\omega \\
&((\exists (d,s) \in C \;\; s = false \wedge p'(d) \leq p(d) \;\vee\; s = true \wedge p'(d) \geq p(d)) \wedge \\
&(\forall d \in Dim_{max}(x) - \pi_1(C) \;\; p(d) = p'(d))) \longrightarrow x_{p'} = v
\end{aligned}$$

where π_i is the projection operator for the i^{th} element of an n-tuple.

For example, let (i,j) denotes a context with coordinates i and j for dimensions 1 and 2. Table 4.2 shows possible input conventions associated with input values of variables x and y at contexts (i,j) in Table 4.1. Here we use the intuitive notation d and -d to represent the pairs (d,true) and (d, false) in the input convention.

Using input conventions, we can also define the boundary for the value of an input variable representing a finite data object, by filling the special value *eod* at contexts outside of the range of the object. For example, for an input variable representing an $m \times n$ matrix, where the matrix is laid in the area from (0,0) through (m-1,n-1) in dimensions 1 and 2, its boundary can be defined by inputing *eod* associated with dimensionalities at the four points: (-1,-1):{-1,2}, (-1,n):{1,2}, (m,n):{1,-2}, (m,-1):{-1,-2}.

In the following, we prove that values of all variables in an mLucid program with an allowable initial environment are continuous and their dimensionalities are defined by the

DIM(x)	DIM(y)	DIM(x+y)	C_x at	C_y at
{}	{}	{}	(0,0): {-2,-1,1,2}	(0,0): {-2,-1,1,2}
{1}	{}	{1}	(i,0): {-2,2}	(0,0): {-2,-1,1,2}
{1}	{1}	{1}	(i,0): {-2,2}	(i,0): {-2,2}
{1}	{2}	{1,2}	(i,0): {-2,2}	(0,j): {-1,1}
{1}	{1,2}	{1,2}	(i,0): {-2,2}	(i,j): {}
{1,2}	{1,2}	{1,2}	(i,j): {}	(i,j): {}

Table 4.2: An input convention example

program.

Lemma 4.7 *Let $[ID] \subset ID$ be the set of all continuous intensions. $[ID]$ is a subdomain of ID.*

Proof: Let $\langle x^i \rangle_{i \in \omega}$ be an ω–chain of $[ID]$

$$x^0 \sqsubseteq x^1 \sqsubseteq \cdots \sqsubseteq x^i \sqsubseteq \cdots$$

Let $x^\omega = \bigsqcup \langle x^i \rangle_{i \in \omega}$ is the least upper bound of the chain. We prove that $x^\omega \in [ID]$. For all $p \in Z^\omega$, if ${x^\omega}_p \neq \perp$, then there must exist x^i such that ${x^i}_p = {x^\omega}_p$, $\forall p' \in Z^\omega$ ${x^i}_{p'} \neq \perp \rightarrow {x^i}_{p'} = {x^\omega}_{p'}$ and $DDS_p(x^i) = DDS_p(x^\omega)$, so $DDS_p(x^\omega)$ has a finite element. □

Lemma 4.8 *Let f and g be two functions whose applications to all continuous intensions are continuous. Applications of the composition $f \circ g$ to all continuous intensions are also continuous.*

Proof: Without any loss of generality, let f and g be two unary operators. For all $x \in [ID]$, $f \circ g(x) = f(g(x))$ is also continuous, since $g(x) \in [ID]$. □

Theorem 4.9 *The result of a program P with an allowable initial environment env_0 is continuous.*

Proof: Straightforward from Definition 4.6, Proposition 4.3 – 4.10 and Lemma 4.7 – 4.8. □

From now on, when we refer to the domain of intensions *ID* we mean its continuous subdomain *[ID]*, and we assume that all initial environments are allowable.

Proposition 4.12 *Given a program P and an initial environment* env_0, *let x be a nullary variable defined in P. Let* v_x *be the value of x defined by P and* env_0. *Let* $v_{P'}$ *be the value of the program P' where P' is a transformation of P by substituting x's defining expression for the subject of P.*

1. $v_x = v_{P'}$, *hence* $DIM(v_x) = DIM(v_{P'})$.

2. v_x *is continuous.*

Proof:

1. Straightforward from the denotational semantics of P.

2. Straightforward from Theorem 4.9. □

4.4 Analysis

An approximation to the dimensionality of the value of a variable in an mLucid program can be derived at compile–time, using the abstract interpretation technique [AH87]. The abstract domain in which the program is interpreted consists of all possible dimensionalities of intensions, that is, all subsets of nonnegative integers. The derivation is called *dimensionality analysis* of the program. In general, it is impossible to derive the exact dimensionality of the value of a variable, either at compile–time because the value is a dynamic object, or at run time because the value is an infinite object. The result of dimensionality analysis is an approximation to the dimensionalities of variables in a program at compile–time.

To simplify the following description, we assume that the following transformations have been done before the analysis.

- For a program with nested where–clauses, we use the standard semantics–preserving transformation technique for functional languages, called λ–*lifting* [FH88], to transform it into one with only a single outermost where–clause.
- To analyze the dimensionality of the output of a program, we also define a special variable *result* in the where–clause of the transformed program using the subject of the defining expression of the original program.

For example, the following mLucid program with two nested where–clauses

```
f(x,10)
where
  f(x,y) = g(z)
  where
    z = prev(1)(x) + next(1)(x);
    g(s) = fby(1)(y,s) + k;
         where k = s * x; end;
  end;
end
```

is transformed into the following program with only the outermost where–clause

```
result
where
  result = f(x,10)
  f(x,y) = g(z(x),x,y)
  z(x) = prev(1)(x) + next(1)(x);
  g(s,x,y) = fby(1)(y,s) + k(s,x);
  k(s,x) = s * x;
end
```

In the above programs, dimensionality declarations are omitted.

In the following, we define the maximum dimensionality of an mLucid program as the union of the dimensionalities of all variables in the program.

Definition 4.9 *Given a program P, let X be the set of nullary variables in P. Given an initial environments* $env_0 \in X \rightarrow ID$, *let* v_x *be the value of a variable* $x \in X$ *defined by P and* env_0. *The maximum dimensionality DIMMAX(P) of P with respect to* env_0 *is defined by*

$$DIMMAX(P) = (\bigcup_{x \in X} DIM(v_x)).$$

An upper bound of the maximum dimensionality of an mLucid program with respect to all initial environments can be determined by syntactically analyzing the program. In mLucid, by Propositions 4.3 – 4.10, the indexical operators *fby* and *before* are the only ones whose results may have larger dimensionalities than the union of dimensionalities of their operands. Thus the maximum dimensionality of an mLucid program must be smaller than the union of the dimension indicators of all *fby* and *before* operations in the program and the declared maximum dimensionalities of input variables.

Theorem 4.10 *Given a program P, let X be the set of input variables in P and* $D = \{ v_d \mid$ *fby(d)(x,y) in* $P \vee$ *before(d)(x,y) in P* $\}$, *where* v_d *is the value of d. Let DimMax(P)* $= \bigcup_{x \in X} Dim_{max}(x) \cup D$.

$$DIMMAX(P) \subseteq DimMax(P).$$

Proof: We prove the theorem by structural induction on the standard denotational semantics of the program.

The value of P is defined by the denotational semantics

$$E[P]\ env_0\ p$$

where

$$\forall x \in X\ DIM(env_0(x)) \subseteq Dim_{max}(x) \wedge \forall x \notin X\ DIM(env_0(x)) = \{\}.$$

From Propositions 4.3–4.10, we can conclude that for all k-ary mLucid operators f

$$\forall env \in Env\ \ DIM(E[f(exp_1, \cdots, exp_k)]\ env) \subseteq (\bigcup_{i=1}^{k} DIM(E[exp_i]\ env)) \cup D$$

Therefore

$$\forall env \in Env$$
$$(\forall 1 \leq i \leq k\ DIM(E[exp_i]\ env) \subseteq DimMax(P)) \longrightarrow$$
$$(DIM(E[f(exp_1, \cdots, exp_k)]\ env) \subseteq DimMax(P)).\square$$

In Section 3.1.2 of Chapter 3, we gave the notion that a program is written in mLucid(n). Now we can formally define the notion by the upper bound of the maximum dimensionality of the program, that is, by a program P in mLucid(n), we mean that DimMax(P) = n.

In the following, we describe how the dimensionality of a variable in a program can be approximated, using the abstract interpretation technique.

Given a program P, we define the concrete domain D^c as the domain of intensions, i.e. $D^c = ID$. We define the abstract domain D^a over which the abstract interpretation is carried out to be the collection of all subsets of the maximum dimensionality of P, i.e. $D^a = \{dim \subseteq DimMax(P)\}$. In D^a, the ordering among elements is the subset relation and $\perp$ is the least element below the empty set $\{\}$. The abstraction map is defined as

$$\alpha : D^c \rightarrow D^a$$
$$\alpha(x) = DIM(x)$$

We must also define the effect of the above abstraction mapping on mLucid operators. The image of an operator op is a new operator $op^{\#}$. We use the set-union '$\cup$' and set-exclusion '$-$' as the basic dimensionality operators to define $op^{\#}$. According to Propositions 4.3–4.10, these operators acting in the abstract domain are defined as

$$\begin{aligned}
\theta_1^{\#}(x) &= x \\
\theta_2^{\#}(x,y) &= x \cup y \\
if-then-else^{\#}(p,x,y) &= p \cup x \cup y \\
origin^{\#}(d)(x) &= x - \{d\} \\
next^{\#}(d)(x) &= x \\
prev^{\#}(d)(x) &= x \\
fby^{\#}(d)(x,y) &= x \cup y \cup \{d\} \\
before^{\#}(d)(x,y) &= x \cup y \cup \{d\}.
\end{aligned}$$

The abstract operators must be consistent with the way in which intensions have been mapped by α. This is usually expressed as a safety condition

$$\alpha(op(x_1,\cdots,x_k)) \subseteq op^{\#}(\alpha(x_1),\cdots,\alpha(x_k)) \quad 1 \leq k \leq 3$$

which is guaranteed by the propositions.

Let P_v be the transformed program, as defined in Proposition 4.11, for deriving the dimensionality of the value of a variable v in a program P. We define the abstract semantics for P_v as follows. We define the abstract semantic function for expressions as $E^{\#} : Exp \rightarrow Env^{\#} \rightarrow D^a$, where $Env^{\#} : Id \rightarrow D^a$ is the domain of environments mapping identifiers to dimensionalities.

$E^{\#}[< constant >]\ env^{\#} = \{\}$

$E^{\#}[< identifier >]\ env^{\#} = env^{\#}(identifier)$

$E^{\#}[\theta_1\ < exp >]\ env^{\#} = E^{\#}[< exp >]\ env^{\#}$

$E^{\#}[< exp_1 >\ \theta_2\ < exp_2 >]\ env^{\#} = E^{\#}[< exp_1 >]\ env^{\#} \cup E^{\#}[< exp_2 >]\ env^{\#}$

$E^{\#}[if\ < exp_0 >\ then\ < exp_1 >\ else\ < exp_2 >\ fi]\ env^{\#} =$

$\quad E^{\#}[< exp_0 >]\ env^{\#} \cup E^{\#}[< exp_1 >]\ env^{\#} \cup E^{\#}[< exp_2 >]\ env^{\#}$

$E^{\#}[origin(< constant >)(< exp >)]\ env^{\#} = E^{\#}[< exp >]\ env^{\#} - \{I_{mLucid}(constant)\}$

$E^{\#}[next(< constant >)(< exp >)]\ env^{\#} = E^{\#}[< exp >]\ env^{\#}$

$E^{\#}[prev(< constant >)(< exp >)]\ env^{\#} = E^{\#}[< exp >]\ env^{\#}$

$E^{\#}[fby(< constant >)(< exp_1 >, < exp_2 >)]\ env^{\#} =$

$\quad E^{\#}[< exp_1 >]\ env^{\#} \cup E^{\#}[< exp_2 >]\ env^{\#} \cup \{I_{mLucid}(constant)\}$

$E^{\#}[before(< constant >)(< exp_1 >, < exp_2 >)]\ env^{\#} =$

$\quad E^{\#}[< exp_1 >]\ env^{\#} \cup E^{\#}[< exp_2 >]\ env^{\#} \cup \{I_{mLucid}(constant)\}$

$E^{\#}[\lambda x.\ < exp >]\ env^{\#} = \lambda x'.E^{\#}[< exp >]\ [x'/x]env^{\#}$

$E^{\#}[< exp_1 >< exp_2 >]\ env^{\#} = (E^{\#}[< exp_1 >]\ env^{\#})(E^{\#}[< exp_2 >]\ env^{\#})$

$E^{\#}[< exp >\ where\ f_i\ = < exp_i >; \cdots end]\ env^{\#} = E^{\#}[< exp >]\hat{env}^{\#}$

$\quad where\ \hat{env}^{\#} = [\cdots E^{\#}[< exp_i >]\ \hat{env}^{\#}/f_i \cdots]env^{\#}.$

The approximation to the dimensionality of v, $Dim(v)$, is defined by the abstract

semantic function $Ep^{\#} : Exp \rightarrow D^a$ as

$$Dim(v) = Ep^{\#}[P_v] = E^{\#}[P_v]env_0^{\#}$$

where $env_0^{\#}$ maps free variables in P to their declared maximum dimensionalities and maps other variables to {}. From now on, when we refer to the dimensionality of a variable in a program, we mean its approximation defined above, unless it is necessary to distinguish it with the *real* dimensionality of the variable.

In the following, we give an example program for computing the product of two matrices, and analyze the dimensionalities of the variables in the program.

```
dimensionality A:{0,1}; B:{0,2};

asa(0)(C, iseod(C))
where
  C = fby(0)(0, C + A * B);
end
```

The program takes two input intensions, A and B, representing two $m \times n$ and $n \times l$ matrices, respectively. A is laid on the plane consisting of dimensions 0 and 1, and each row is laid along dimension 0 starting at the origin. B is laid on the plane consisting of dimensions 0 and 2, and each column is laid along dimension 0 starting at the origin.

In the program, the dimensionality of C is {0,1,2} because the least fixed point of C's abstract semantics

$$C = A \cup B \cup C \cup \{0\}$$

is $\{0,1\} \cup \{0,2\} \cup \{0\} = \{0,1,2\}$.

iseod is a built–in pointwise predicate that returns *true* at a context if the value of its operand is *eod* at the context; otherwise it returns false.

The indexical operator *asa* (for "as soon as") is built–in in mLucid

$$asa : N \rightarrow ID \times ID \rightarrow ID$$

which is defined by the primitive indexical operators as follows

```
asa(d)(x,t) = if origin(d)(t) then origin(d)(x)
                 else asa(d)(next(d)(x), next(d)(t))
              fi.
```

Informally, at a context p, the operation returns the value of x at the context p', such that p and p' are lined up in dimension d, p' is at the positive side of d's origin, and t has the first true value at p' from d's origin through p'.

The formal indexical semantics of *asa* is defined by:

$$asa = \lambda d.\lambda(x,t).\lambda p.\ x_{[k/d]p}$$
$$where\ \ k \in Z\ \wedge\ k \geq 0\ \wedge\ t_{[k/d]p}\ \wedge\ \forall 0 \leq i < k\ \neg t_{[i/d]p}.$$

The formal denotational semantics of *asa* is defined by:

$$E[asa(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$E[< exp_1 >]\ env\ [k/I_{mLucid}(< constant >)]p$$
$$where$$
$$k \in Z\ \wedge\ k \geq 0\ \wedge$$
$$t_{[k/d]p}\ \wedge\ \forall 0 \leq i < k\ \neg t_{[i/d]p}.$$

In the abstract semantics, the least fixed point of $asa^{\#}$ is

$$fix(asa^{\#}) = \lambda d.\lambda x.\lambda t.\ (t - \{d\}) \cup (x - \{d\}).$$

The result of the program, the product of A and B, is laid on the plane consisting of dimensions 1 and 2. The result's dimensionality can be derived by applying the abstract operator $asa^{\#}$ to the dimensionalities of C and *iseod(C)*, that is

$$\{1,2\} = asa^{\#}(0)(\{0,1,2\},\{0,1,2\}).$$

There is a standard algorithm [AH87] to compute fixed points of variables in an abstract program. The algorithm has two phases. In the initial phase, all the variables are assigned appropriate abstract values. In the iteration phase, at each step the new abstract value of each variable is computed based on the current values of the variables.

The iteration proceeds until the fixed points of all the variables are found, that is, no new value of any variable is produced at a step.

In our case, the fixed points are the approximate dimensionalities of the variables in an mLucid program defined by the above abstract semantics. The initial abstract values assigned to all input variables are their declared maximum dimensionalities, to all other nullary variables are $\{\}$, and to all k–ary variables are the function $\lambda(x_1, x_2, \cdots, x_k).\{\}$. Since by Theorem 4.10 the maximum dimensionality of any mLucid program is finite, the fixed points of all the variables will be reached eventually during the iteration.

In most cases, however, the algorithm is very slow to reach the fixed points, mainly because of user–defined functions in the program. Without any loss of generality, consider a unary function f in the program. f works on the concrete domain, i.e. $f : D^c \rightarrow D^c$ where $D^c = ID$. Let $f^\#$ be the function abstracted from f and $f^\#$ works on the abstract domain, i.e. $f^\# : D^a \rightarrow D^a$ where $D^a = \{dim \subseteq MaxDim(P)\}$. According to the abstract semantics, $f^\#$ is defined as $f^\#(\varepsilon) = \bigcup\{\alpha(f(x)) \mid \alpha(x) \subseteq \varepsilon\}$. Let $n = |MaxDim(P)|$, then there are possibly 2^n values of ε that $f^\#$ may apply to in the program. In general, for a k–ary abstract function $f^\#$ there are 2^{kn} values of ε that $f^\#$ may apply to in the program. This causes a large number of computation steps involved to reach the fixed point of $f^\#$.

In the following, we describe an approach to speeding up the iteration algorithm by further abstracting abstract functions $f^\#$.

Consider the following observation on the abstract functions of the primitive operators defined earlier. They can all be written in a uniform form $x - \pi \cup \eta$, where x is the union of its operands, and π and η are sets of dimensions independent of the operands.

$$\begin{aligned}
\theta_1^\#(x) &= x - \{\} \cup \{\} \\
\theta_2^\#(x, y) &= (x \cup y) - \{\} \cup \{\} \\
if - then - else^\#(p, x, y) &= (p \cup x \cup y) - \{\} \cup \{\}
\end{aligned}$$

$$
\begin{aligned}
origin^{\#}(d)(x) &= x - \{d\} \cup \{\} \\
next^{\#}(d)(x) &= x - \{\} \cup \{\} \\
prev^{\#}(d)(x) &= x - \{\} \cup \{\} \\
fby^{\#}(d)(x,y) &= (x \cup y) - \{\} \cup \{d\} \\
before^{\#}(d)(x,y) &= (x \cup y) - \{\} \cup \{d\}.
\end{aligned}
$$

We can also prove that if two functions are in the above *exclusion-union* form, so is their composition. Without losing generality, in the following we prove unary functions.

Proposition 4.13 *Let $f^{\#}$ and $g^{\#}$ be two set functions defined by*

$$
\begin{aligned}
f^{\#}(x) &= x - \pi_1 \cup \eta_1 \\
g^{\#}(y) &= y - \pi_2 \cup \eta_2
\end{aligned}
$$

where π_1 and η_1 are disjoint, so are π_2 and η_2. The composition of $f^{\#}$ and $g^{\#}$, $f^{\#} \circ g^{\#}$, can be defined by

$$
f^{\#} \circ g^{\#}(z) = z - \pi \cup \eta
$$

where $\pi = (\pi_2 - \eta_1) \cup \pi_1$, $\eta = (\eta_2 - \pi_1) \cup \eta_1$, and π and η are disjoint.

Proof:

$$
\begin{aligned}
f^{\#}(g^{\#}(x)) = &\ (x - \pi_2 \cup \eta_2) - \pi_1 \cup \eta_1 \\
= &\ x - (\pi_2 \cup \pi_1) \cup (\eta_2 - \pi_1 \cup \eta_1) \\
= &\ x - ((\pi_2 - \eta_1) \cup \pi_1) \cup ((\eta_2 - \pi_1) \cup \eta_1) \quad \square
\end{aligned}
$$

Based on the above observation and proposition, by structural induction on the defining expression of any abstract function $f^{\#}$ in the program, $f^{\#}$ can be defined in the *exclusion-union* form and the value of $f^{\#}$ is determined by the sets π and η. Since π and η are independent of the arguments of the function, at each iteration step, the value of $f^{\#}$, i.e. the sets π and η, can be computed by the formulae derived in Proposition 4.12. At the initial phase, π is assigned the maximum dimensionality $MaxDim(P)$ of the program and η is assigned $\{\}$, so that $f^{\#}$ returns $\{\}$ for all possible arguments initially. At any later stage of the iteration, the value of π can be considered as the complement set of the dimensions that have been dropped off from the initial set during the iteration. The two

formulae defining the new values of π and η in Proposition 4.12 show that, during the iteration, the sizes of the two sets will never be reduced, so that the fixed points of the two sets will reach eventually, hence the fixed point of $f^{\#}$.

There is another approach to speeding up the iteration, due to [Wad91a] [Esc91]. It is based on the following observation.

- A dimension in an argument of an abstract function $f^{\#}$ is independent of other dimensions in the same argument in terms of the result dimensionality, that is,

$$f^{\#}(x \cup x', y) = f^{\#}(x, y) \cup f^{\#}(x', y).$$

- An argument of an abstract function $f^{\#}$ is independent of other arguments in terms of the result dimensionality, that is,

$$f^{\#}(x, y) = f^{\#}(x, \{\}) \cup f^{\#}(\{\}, y).$$

Thus, the value of a k–ary abstract function $f^{\#}(x_1, x_2, \cdots, x_k)$ can be represented by k $n \times n$ boolean matrices and a boolean vector V with n elements, where n is the maximum dimensionality of the program. Each of the matrices, M_i, is associated with a formal parameter x_i. The true value in an entry (m, l) in M_i denotes that dimension l is in the result dimensionality if dimension m is in x_i. The true value in an entry l in V denotes that dimension l is in the result dimensionality, which is independent of the arguments but may depend on global variables in $f^{\#}$'s definition. At the initial phase, all entries of the matrices and vector are assigned false representing $f^{\#}(x_1, x_2, \cdots, x_k) = \{\}$. Let x_i be also represented as boolean vectors. At each iteration step, the result dimensionality is represented by the boolean sum of the products of vector x_i and matrix M_i and V, that is,

$$f^{\#}(x_1, x_2, \cdots, x_k) = x_1 M_1 + x_2 M_2 + \cdots + x_k M_k + V.$$

Meanwhile each matrix M_i is updated according to the result.

Chapter 5

CONTEXT PARALLELISM AND INDEXICAL COMMUNICATION

In this chapter, we first define context parallelism in an mLucid program by a kind of data dependency, *variable-value dependency*, among values of variables at various contexts in the program. We then study the relationship between the dependence and the indexical operations in the program, and define indexical communication by this relationship.

5.1 Direct Variable–Value Dependence

In an mLucid program, the value of a variable at a context may depend on values of other variables and/or itself at the same and/or different contexts. This dependence relation is determined by the variable's defining expression. The dependence forms a hierarchy. At the first level of the hierarchy, there are values of the variables in the defining expression at some contexts. At the second level of the hierarchy, there are values of the variables that are in the defining expressions of the variables at the first level at some contexts and so on.

For the value of a variable at a context in the program, we are especially interested in

its dependence at the first level in the hierarchy, because if we can determine it by some rules, we can build the next level by recursively applying the same rules to each of the variable values at the current level. In this sense, we define that the value x_p of a variable x at a context p in an mLucid program P *directly* depends on the values of some variables at some contexts if, given these value–context pairs, x_p can be determined by x's defining expression.

From the operational point of view, suppose that the values of all variables in the program at all contexts, but not x_p, are available before the evaluation of x_p. During the evaluation of x_p, or the evaluation of the value of x's defining expression at p, the values of some variables at some contexts are fetched from the environment. We say that these values being fetched are directly depended on by x_p.

In the following, we formally define the direct dependence for the value of a variable at a context in an mLucid program in terms of the program's denotational semantics. In the definition, we assume that all local definitions in nested where-clauses in the program have been λ–lifted, so that there is only a single where-clause in the program. We also assume that the definition for each k-ary variable symbol ($k > 0$), as a function identifier, is closed, i.e. there are no free variables in the definition except the formal parameters and possibly the variable symbol itself. A non–closed function definition can be transformed into a closed one by parameterizing the free variables. To define the direct variable–value dependence of the subject of the program's defining expression, we also assume that the special variable *result* has been added to the program, where *result* is defined by the subject in the where-clause, and the subject of the modified program becomes the expression *result*.

Before we give the formal definition, we need to prove that the semantics of expressions in mLucid is functionally sequential. Informally, a k–ary function $f : ID^k \rightarrow ID$ is functionally sequential if the result of every application of f at every context depends on a unique set of the values of its arguments at some contexts.

Definition 5.1 (Function Dependence Set) *Given a function* $f : ID^k \rightarrow ID$ ($k > 0$), *intensions* $x_1, \cdots, x_k \in ID$, *and a context* $p \in Z^\omega$, *if* $f(x_1, \cdots, x_k)_p \neq \perp$, *the family*

$FDS(f, < x_1, \cdots, x_k >, p)$ of sets of intensions, indexed by i for $1 \leq i \leq k$, is defined by

$$FDS(f, < x_1, \cdots, x_k >, p)_i = \{x_i' \sqsubseteq x_i \mid f(x_1, \cdots, x_i', \cdots, x_k)_p = f(x_1, \cdots, x_i, \cdots, x_k)_p\}.$$

Definition 5.2 (Functionally Sequential Function) *A function $f : ID^k \rightarrow ID$ $(k > 0)$ is functionally sequential if and only if $\forall x_1, \cdots, x_k \in ID\ \forall p \in Z^\omega$, if $f(x_1, \cdots, x_k)_p \neq \perp$, then for $1 \leq i \leq k$ $FDS(f, < x_1, \cdots, x_k >, p)_i$ has a least element, that is,*

$$\mu FDS(f, < x_1, \cdots, x_k >, p)_i = \sqcap FDS(f, < x_1, \cdots, x_k >, p)_i \ \in \ FDS(f, < x_1, \cdots, x_k >, p)_i$$

An example of non–sequential functions is related to the so–called *paror* (for "parallel or") operator, which is defined as follow [Yag84]:

$$paror: \quad ID \times ID \rightarrow ID$$

$$paror = \lambda(x, y).\ \lambda p \begin{cases} true & \text{if } x_p \vee y_p \\ false & \text{if } \neg x_p \wedge \neg y_p \\ \perp & \text{otherwise} \end{cases}$$

Consider a function f being defined as

```
f(x) = paror(x, next(0)(x));
```

in mLucid. At a context $p \in Z^\omega$, the true value $f(x)_p$ depends on the true value of either x_p or $x_{[p(0)+1/0]p}$, even if one of them is $\perp$. Therefore, all boolean–valued intensions with true value at p and/or $[p(0) + 1/0]p$ are in $FDS(f, x, p)$, but there is no least element in the set.

It has been proved by [Yag84] that for all continuous functionally sequential functions f, all intensions $x_1, \cdots, x_k$ and all contexts p, if $f(x_1, \cdots, x_k)$ at p is not $\perp$ then there are a finite number of contexts q such that $(\mu FDS(f, < x_1, \cdots, x_k >, p)_i)_q \neq \perp$ for $(1 \leq i \leq k)$.

We prove that all mLucid expressions are functionally sequential by the following lemmas and theorem. Notice that the binary primitive pointwise operators in mLucid, inherited from ISWIM, are either strict or functionally sequential. A strict pointwise

operation, like +,−,*, or /, at a context depends on the both operands at the context. A functionally sequential pointwise operator, like *and* or *or*, is defined by a conditional on the operands so that the operation depends on them sequentially. For example, the operator *or* in mLucid is defined by

$$\text{x or y} = \text{if x then true else y.}$$

Lemma 5.1 *All the primitive operators in mLucid are functionally sequential.*

Proof: By the semantics of the primitive operators, we have

1. For each unary operator θ_1

$$\mu FDS(\theta_1, x, p) = \lambda q. \begin{cases} x_q & \text{if } q = p \\ \bot & \text{otherwise} \end{cases}$$

2. For each strict binary operator θ_2,

$$\mu FDS(\theta_2, < x_1, x_2 >, p)_i = \lambda q. \begin{cases} (x_i)_q & \text{if } q = p \\ \bot & \text{otherwise} \end{cases} \quad (i = 1, 2)$$

3. For the conditional operator if–then–else–fi,

$$\mu FDS(if, < t, x, y >, p)_1 = \lambda q. \begin{cases} t_q & \text{if } q = p \\ \bot & \text{otherwise} \end{cases}$$

$$\mu FDS(if, < t, x, y >, p)_2 = \lambda q. \begin{cases} x_q & \text{if } q = p \wedge t_q \\ \bot & \text{otherwise} \end{cases}$$

$$\mu FDS(if, < t, x, y >, p)_3 = \lambda q. \begin{cases} y_q & \text{if } q = p \wedge \neg t_q \\ \bot & \text{otherwise} \end{cases}$$

4. For the indexical operator origin,

$$\mu FDS(origin(d), x, p) = \lambda q. \begin{cases} x_q & \text{if } q = [0/d]p \\ \bot & \text{otherwise} \end{cases}$$

5.

$$\mu FDS(next(d), x, p) = \lambda q. \begin{cases} x_q & \text{if } q = [p(d) + 1/d]p \\ \perp & \text{otherwise} \end{cases}$$

6. For the indexical operator prev,

$$\mu FDS(prev(d), x, p) = \lambda q. \begin{cases} x_q & \text{if } q = [p(d) - 1/d]p \\ \perp & \text{otherwise} \end{cases}$$

7. For the indexical operator fby,

$$\mu FDS(fby(d), < x, y >, p)_1 = \lambda q. \begin{cases} x_q & \text{if } q = p \ \wedge \ p(d) \leq 0 \\ \perp & \text{otherwise} \end{cases}$$

$$\mu FDS(fby(d), < x, y >, p)_2 = \lambda q. \begin{cases} y_q & \text{if } q = [p(d) - 1/d]p \ \wedge \ p(d) > 0 \\ \perp & \text{otherwise} \end{cases}$$

8. For the indexical operator before,

$$\mu FDS(before(d), < x, y >, p)_1 = \lambda q. \begin{cases} y_q & \text{if } q = p \ \wedge \ p(d) \geq 0 \\ \perp & \text{otherwise} \end{cases}$$

$$\mu FDS(before(d), < x, y >, p)_2 = \lambda q. \begin{cases} x_q & \text{if } q = [p(d) + 1/d]p \ \wedge \ p(d) < 0 \\ \perp & \text{otherwise} \end{cases}$$ □

Without losing generality, we prove that functionally sequential unary functions are closed under composition and chain union.

Lemma 5.2 *Given two unary functionally sequential functions f and g, $f \circ g$ is also functionally sequential.*

Proof: For an intension $x \in ID$ and a context $p \in Z^\omega$, let x_0 be defined by

$$x_0 = \lambda q. \begin{cases} x_q & \text{if } \exists p' \ \mu FDS(f, g(x), p)_{p'} \neq \perp \ \wedge \ \mu FDS(g, x, p')_q \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

1. We prove $x_0 \in FDS(f \circ g, x, p)$. By the definition of x_0, $x_0 \sqsubseteq x$ and $g(x_0) = \mu FDS(f, g(x), p)$, therefore

$$f(g(x_0))_p = f(\mu FDS(f, g(x), p))_p = f(g(x))_p$$

2. We prove $\forall x' \in FDS(f \circ g, x, p)\ x_0 \sqsubseteq x'$. Assume that $x' \in FDS(f \circ g, x, p)$ but $x_0 \not\sqsubseteq x'$. Let $q \in Z^\omega$ be a context such that $x_{0q} = x_q$ and $x'_q = \perp$. Then $\exists p'\ \mu FDS(f, g(x), p)_{p'} \neq \perp \wedge \mu FDS(g, x, p')_q \neq \perp \wedge x'_q = \perp$. By the definition of μFDS, $g(x')_{p'} = \perp$, hence $f(g(x'))_p = \perp$. This is in contradiction with $x' \in FDS(f \circ g, x, p)$. □

Lemma 5.3 *Given an ω-chain of unary functionally sequential functions*

$$\langle f_i \rangle_{i \in \omega} = f_0 \sqsubseteq f_1 \sqsubseteq \cdots \sqsubseteq f_n \sqsubseteq \cdots$$

The least upper bound of the chain

$$f_\omega = \bigsqcup_{i \in \omega} f_i$$

is also functionally sequential.

Proof: For an intension $x \in ID$ and a context $p \in Z^\omega$, if $f_\omega(x)_p \neq \perp$ there must exist a f_k such that $\forall 0 \leq i < k\ f_i(x)_p = \perp$ and $\forall j \geq k\ f_j(x)_p = f_\omega(x)_p$. For each $j \geq k$ let $x_j = \mu FDS(f_j, x, p)$, so that $f_j(x_j)_p = f_j(x)_p = f_\omega(x)_p$. Since $f_j(x_j) \sqsubseteq f_{j+1}(x_j)$, $f_{j+1}(x_j)_p = f_j(x_j)_p$, i.e. $x_j \in FDS(f_{j+1}, x, p)$. Therefore, $\forall j \geq k\ x_j \sqsupseteq x_{j+1}$. Since for each x_j there are a finite number of contexts $q \in Z^\omega$ such that $x_{jq} \neq \perp$, there must exist an n $(k \leq n)$ such that $\forall k \leq j \leq n\ x_j \sqsupseteq x_n$ and $\forall j > n\ x_j = x_n$. We prove $x_n = \mu FDS(f_\omega, x, p) \in FDS(f_\omega, x, p)$.

1.

$$f_n(x_n)_p = f_\omega(x)_p \Rightarrow x_n \in FDS(f_\omega, x, p).$$

2.

$$(\forall x' \sqsubset x_n\ \forall i \in \omega\ f_i(x')_p = \perp) \Rightarrow x_n = \mu FDS(f_\omega, x, p).\ \square$$

Theorem 5.4 *The semantics of an expression in mLucid is functionally sequential.*

Proof: By Lemma 5.1 – 5.3. □

The property of functional sequentiality guarantees that the direct variable–value dependence of the value of a variable at a context is unique in all mLucid programs.

Definition 5.3 (Direct Variable–value Dependence) *Given a program P and an initial environment env_0, let V be the set of nullary variables defined in P. For a variable $v \in V$, let e be v's defining expression in P, and $u \in V^k$ for some $0 \leq k \leq |V|$ be the vector of nullary variables appearing in e. Let function f*

$$f: \ ID^k \longrightarrow ID$$

be defined as

$$f(x_1, \cdots, x_k) \ = \ E[e]\,[x_1/u_1, \cdots, x_k/u_k]e\hat{n}v$$

where the semantic function E and the environment $e\hat{n}v$ are defined in Section 3.1.5 of Chapter 3.

Let $x_i = e\hat{n}v(u_i)$ for $(1 \leq i \leq k)$. Given a context $p \in Z^\omega$, if $e\hat{n}v(v)_p \neq \perp$, the Direct Variable–value Dependence DVD(v,p) with respect to P and the initial environment env_0 is a finite subset of $V \times Z^\omega$ defined by

$$DVD(v,p) = \{(u_i, q) \in V \times Z^\omega \mid 1 \leq i \leq k \ \wedge \ (\mu FDS(f, < x_1, \cdots, x_k >, p)_i)_q \neq \perp\}.$$

The direct variable–value dependence can be derived by a kind of non–standard semantics for mLucid. For a variable v defined in the program, if there are no function applications involved in v's defining expression e, the non–standard semantic domain S for the derivation of *DVD(v,p)* can simply be defined as the set VZ of finite subsets of $V \times Z^\omega$. However, if there is a user–defined function application $g(e_1, \cdots, e_k)$ in e, where g is in the syntax domain, the value v_p of v at p depends on not only g's value (as function), but also the values of variables appearing in $e_1, \cdots, e_k$ at some contexts. If there are indexical operations on formal parameters in g's definition, these contexts can only be determined *after* applying g to its arguments. Thus we need to introduce the second

component F to the non-standard semantic domain that carries the higher order behavior of a function value. F has the type

$$F: \ (S \longrightarrow S)$$

Therefore, the type of the non-standard domain S is

$$S: \ VZ + F.$$

The environment Env_d in the non-standard semantics has the type

$$Env_d: \ V \rightarrow Z^\omega \rightarrow S.$$

Given a program P, to derive *DVD(v,p)* for the value of a nullary variable v in P at a context p, we first transform P to P_v such that P_v has the same body of the where-clause as in P, but the subject of P_v is changed to v's defining expression.

The non-standard semantic function for expressions

$$E: \ Exp \rightarrow Env_d \rightarrow Z^\omega \rightarrow S$$

is defined as follows.

The value of a constant expression does not depend on any variable value at any context.

$$E[< constant >] \ env_d \ p = \{\}.$$

For an expression consisting of only one variable, its value at a context depends on itself.

$$E[< id >] \ env_d \ p = if \ Oracle(< id >, p) \neq \perp$$
$$then \ \{(id, p)\} \ else \ \perp$$

Notice that the *Oracle* in the definition is a shorthand for carrying the standard semantics around and consulting it for the conditionals. Because of the involvement of the standard semantics, the derivation of *DVD(v,p)* according to the non-standard semantics cannot be performed at compile-time.

For an expression consisting of a unary pointwise operation, its value at a context depends on what its operand depends on at the same context.

$$E[\theta_1 \ < exp >] \ env_d \ p = E[< exp >] \ env_d \ p.$$

For an expression consisting of a strict binary pointwise operation, its value at a context depends on what its operands depend on at the same context.

$$E[< exp_1 > \ \theta_2 \ < exp_2 >]\ env_d\ p = \ vz_1 \cup vz_2$$
$$where$$
$$vz_1 \ = \ E[< exp_1 >]\ env_d\ p$$
$$vz_2 \ = \ E[< exp_2 >]\ env_d\ p.$$

Notice that we treat the functionally sequential binary operators as conditionals.

For an expression consisting of a conditional operation, its value at a context depends on what the predicate and consequent depend on if the predicate in the standard semantics is true, or what the predicate and alternate depend on otherwise.

$$E[if \ < exp_0 > \ then \ < exp_1 > \ else \ < exp_2 > \ fi]\ env_d\ p =$$
$$if\ Oracle(< exp_0 >, p) \ = \ True$$
$$then\ vz_0 \cup vz_1\ else\ vz_0 \cup vz_2$$
$$where$$
$$vz_0 = E[< exp_0 >]\ env_d\ p$$
$$vz_1 = E[< exp_1 >]\ env_d\ p$$
$$vz_2 = E[< exp_2 >]\ env_d\ p\ .$$

For an expression consisting of an *origin(d)* indexical operation, according to its indexical semantics, its value at a context p depends on what its operand depends on at the context $[0/d]p$.

$$E[origin(< constant >)(< exp >)]\ env_d\ p = E[< exp >]\ env_d\ [0/d]p$$
$$where\ d = I_{mlucid}(< constant >).$$

For an expression consisting of a *next(d)* or *prev(d)* indexical operation, according to its indexical semantics, its value at a context p depends on what its operand depends on at the context $[p(d) + 1/d]p$ or $[p(d) - 1/d]p$.

$$E[next(< constant >)(< exp >)]\ env_d\ p = E[< exp >]\ env_d\ [p(d) + 1/d]p$$
$$where\ d = I_{mlucid}(< constant >)$$

$$E[prev(< constant >)(< exp >)]\ env_d\ p = E[< exp >]\ env_d\ [p(d) - 1/d]p$$
$$where\ d = I_{mlucid}(< constant >)$$

For an expression consisting of a *fby(d)* or *before(d)* indexical operation, according to its indexical semantics, its value at a context p depends on what one of its operands depends on at the context either p or $[p(d) - 1/d]p$ for *fby*, or either $[p(d) + 1/d]p$ or p for *before*, depending on p's position in the context space.

$$E[fby(< constant >)(< exp_1 >, < exp_2 >)]\ env_d\ p =$$
$$if\ p(d) \leq 0\ then\ E[< exp_1 >]\ env_d\ p$$
$$else\ E[< exp_2 >]\ env_d\ [p(d) - 1/d]p$$
$$where\ d = I_{mlucid}(< constant >)$$

$$E[before(< constant >)(< exp_1 >, < exp_2 >)]\ env_d\ p =$$
$$if\ p(d) \geq 0\ then\ E[< exp_2 >]\ env_d\ p$$
$$else\ E[< exp_1 >]\ env_d\ [p(d) + 1/d]p$$
$$where\ d = I_{mlucid}(< constant >).$$

For an expression with a bound variable, which defines a function, its value does not depend on any free variable since it is closed by the assumption. However its higher order behavior must be carried in the environment for expressions involving applications of the function.

$$E[\lambda x. < exp >]\ env_d\ p = \lambda x'.\ E[< exp >]\ [x'/x]env_d\ p.$$

The dependence of an application of a function at a context is determined by the function with non-standard semantics to the dependence of the argument.

$$E[< exp_1 > < exp_2 >]\ env_d\ p = vz$$
$$where$$
$$f = E[< exp_1 >]\ env_d\ p$$
$$vz = f(E[< exp_2 >]\ env_d\ p).$$

The non-standard semantic function for expressions with where-clauses is defined as

follow. Notice that by the assumption, there is at most one where–clause in P_v.

$$E[< exp > \ where \ \cdots f_i \ = < exp_i > ; \cdots end] \ env_d \ p = E[< exp >] \ \hat{env}_d \ p$$
$$where \ \ \hat{env}_d = [\cdots E[< exp_i >] \ \hat{env}_d \ p/f_i \cdots] env_d.$$

The value of P_v in the non–standard semantics is denoted by the semantic function

$$E_p[P_v] \ p = E[P_v] \ env_{d_0} \ p$$

where env_{d_0} is defined as

$$env_{d_0}(v)(p) = \begin{cases} \{\} & \text{if } v \text{ is an input variable} \\ \perp & \text{otherwise} \end{cases}$$

The direct variable–value dependence *DVD(v,p)* is

$$DVD(v,p) \ = \ E_p[P_v] \ p.$$

Based on mLucid's standard denotational semantics, such derived DVD by the above non–standard semantics can be verified by the structural induction on the non–standard semantic functions. The correctness of the derived dependence is guaranteed by carrying the standard semantics for non–strict conditionals, where the derivation depends on actually evaluating the program. The least–element property of the dependence is guaranteed by the least fixed point solution of the non–standard semantics.

For example, in the matrix multiplication program in Section 4.4 of Chapter 4, there are two variables C and *result*. Using the non–standard semantics, we can derive

$$DVD(C, \ p) = \begin{cases} \{(C,p'),(A,p'),(B,p')\} & \text{if } p(0) > 0 \\ \{\} & \text{otherwise} \end{cases}$$

where $p' = [p(0) - 1/0]p$, and

$$DVD(result, \ p) \ = \ \{(C,[0/0]p), \ (C,[1/0]p), \cdots, (C,[n/0]p)\}$$

where A and B are input matrices to the program with sizes $m \times n$ and $n \times l$, respectively.

5.2 Variable–value Dependence Graph

Based on the direct variable–value dependence, we can construct the *variable–value dependence graph* of an mLucid program. A node in the graph denotes the value of a variable at a context. An edge from a source node to a destination node in the graph denotes that the value denoted by the source directly depends on the value denoted by the destination. The variable–value dependence graph is formally defined as follows.

Definition 5.4 (Variable–value Dependence Graph) *Given a program P in mLucid(n) and an initial environment env_0, let V be the set of nullary variables in P. The Variable–value Dependence Graph VDG(P) is a directed graph (N, E).*

$$N = V \times Z^n$$

is the set of vertices.

$$E = \{\langle (v_1, p_1), (v_2, p_2)\rangle \in N \times N \mid (v_2, p_2) \in DVD(v_1, p_1)\}.$$

is the set of edges.

For example, Figure 5.1 shows the VDG of the matrix multiplication program. In the

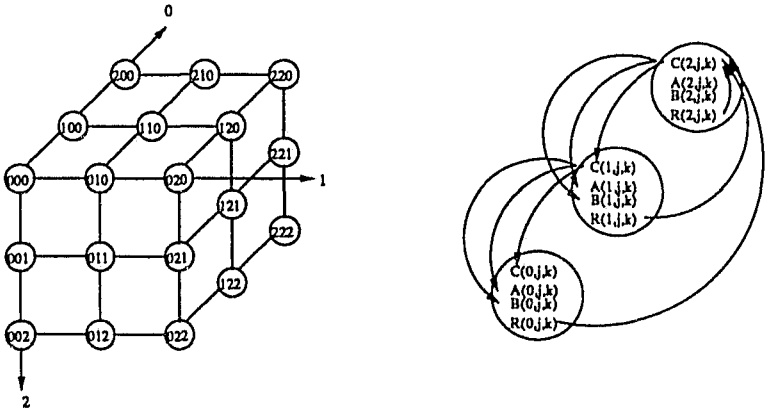

Figure 5.1: The variable–value dependence graph of a matrix multiplication program

graph, each context is represented by its coordinates for dimensions 0, 1 and 2, as the program's maximum dimensionality is {0,1,2}.

The VDG of any computable mLucid program must be acyclic, because otherwise the computations for the values of variables at the contexts represented by the nodes in a cycle of the graph will not terminate, resulting in $\perp$. We prove the acyclic property of VDG in the following theorem.

Lemma 5.5 *Given a program P in mLucid(n) and an initial environment env_0, for all nullary variables v in P and all contexts $p \in Z^n$, if $v_p \neq \perp$*

$$(v\ ,p\ ) \notin DVD(v,p).$$

Proof: Straightforward from the fixed point semantics □

Theorem 5.6 *Given a program P and an initial environment env_0, VDG(P) is acyclic.*

Proof: For all paths $(v,p),(v_1,p_1),\cdots,(v_n,p_n)$ in $VDG(P)$, we prove that $(v,p) \neq (v_n,p_n)$, by induction on their lengths.

For a path $((v,p),(v_1,p_1))$ with length one, Lemma 5.5 proves that $(v,p) \neq (v_1,p_1)$.

Assume that for all paths $(v,p),(v_1,p_1),\cdots,(v_{n-1},p_{n-1})$ with length $n-1$, $(v,p) \neq (v_{n-1},p_{n-1})$.

Consider a path $t = (v,p),(v_1,p_1),(v_2,p_2),\cdots,(v_n,p_n)$ with length n. Let v be defined by $v = e$ in P. We redefine v by $v = e'$, where e' is the same as e except that all the occurrences of v_1 in e have been substituted by v_1's defining expression. By the denotational semantics, e' and e are equivalent in terms of defining v's value, and $(v_2,p_2) \in DVD(v,p)$ with respect e'. Let $t' = (v,p),(v_2,p_2),\cdots,(v_n,p_n)$ with length $n-1$, then t' is in the VDG(P') where $P' = P$ except that the equation $v = e$ in P is changed to $v = e'$ in P'. $(v,p) \neq (v_n,p_n)$ in t' by the assumption. Therefore, $(v,p) \neq (v_n,p_n)$ in t. □

The VDG of a program represents a kind of data dependence in the program, where the *data* is the value of a variable at some context. Since the VDG is acyclic, it defines a partial order among the values of variables in the program at various contexts. During the computation of the program, the partial order determines the evaluation order for the values. The value of a variable at a context represented by a node n in the VDG can be

evaluated, only if all the values of variables at contexts represented by the nodes that have incoming edges from n are evaluated.

During the execution of a program, if the values of two variables at two contexts, which are represented by two nodes in the program's VDG, do not depend on each other, that is, there is no path from one of the nodes to the other in the graph, the two values can be evaluated in parallel. Because of indexical semantics of the program, this kind of parallelism is not the same as expression parallelism in functional programs. Recall how expression parallelism in a functional program is defined:

1. All arguments of a function application can be evaluated in parallel, if there is no data dependencies among them;

2. The function application itself can be evaluated with its arguments in parallel, subject to the availability of the arguments.

In an mLucid program, if all function applications are pointwise, we may simply extend the definition of expression parallelism pointwise. The **pointwise expression parallelism** in an mLucid program can be defined, informally, as

1. All arguments of a function application *at each context* can be evaluated in parallel, if they do not depend on each other at the context;

2. The function application *at each context* can be evaluated with its arguments in parallel, subject to the availability of the arguments at the context.

However, because of indexical operations in the program, at a context, the evaluations for the values of arguments in a function application as well as the application itself may result in evaluations at contexts other than the given one. For example, for the function application *f(next(0)(x), prev(0)(x))*, if the application itself occurs at context p, then the evaluations for the arguments occur at the contexts $[p(0) + 1]p$ and $[p(0) - 1]p$, instead of p. We may call this kind of expression parallelism involving "multi-contexts" **context–expression** parallelism. The data dependence represented by edges of a program's VDG restricts the context–expression parallelism in the program.

5.3 Context Dependence Graph and Context Parallelism

In this dissertation, we are in particular interested in a special type of context-expression parallelism in mLucid programs, called **context parallelism**, because of the indexical nature of the language. Context parallelism occurs in executing a program when the values of variables at *different* contexts can be evaluated in parallel. Context parallelism in the program is restricted by a subset of the program's variable–value dependence, that is, the dependence between the values of variables at *different* contexts.

We define context parallelism in an mLucid program at three levels, based on different abstractions of the program's VDG. At the first level called *fine context parallelism*, we consider that two values of the same variable or two different variables at two different contexts do not depend on each other, so that the two values can be evaluated in parallel at the two contexts. At the second level for which we use the same term as *context parallelism*, we consider that values of all variables at two contexts do not depend on each other, so that the two sets of the values at the two contexts can be evaluated in parallel Given a partition on the context space, At the third level called *coarse context parallelism*, we consider that values of all variables at two classes of contexts do not depend on each other, so that the two sets of the values at the two classes can be evaluated in parallel.

In the following, we formally define the three levels of context parallelism based on the direct variable–value dependencies among values of variables in an mLucid program. Without losing generality, we assume that mLucid programs in the following definitions are closed, i.e. they do not have input variables.

Definition 5.5 (Fine Context Dependence Graph) *Given a program P in mLucid(n), let V be the set of nullary variables in P. The Fine Context Dependence Graph FCDG(P) is a directed multigraph* (Z^n, E). *E as the set of edges is defined by*

$$E = \{(p, v, u, q) \mid p, q \in Z^n \wedge p \neq q \wedge v, u \in V \wedge (u, q) \in DVD(v, p)\}.$$

Informally, the above definition states that a node in a fine context dependence graph is a context in the context space of the program; two nodes, p and q, in the graph can

be connected through one or more directed edges each of which is labeled by two variable symbols, v and u, in the program, if the value of v at context p directly depends on the value of u at context q.

Definition 5.6 (Fine Context Parallelism) *Given a program P, let V be the set of nullary variables in P. Given two variables $v, u \in V$ and two contexts $p, q \in Z^n$ where $p \neq q$. There exists fine context parallelism between the pairs (v, p) and (u, p), if there is no path in FCDG(P) starting with an edge of the form (p, v, x, p') and ending at the edge of the form (q', y, u, q), and vice versa, for all $x, y \in V$ and $p', q' \in Z^n$.*

Informally, the above definition states the following. There is the parallelism between computations for the values of variables v and u in the program at contexts p and q, if the nodes p and q in the program's fine context dependence graph are not connected through p's outgoing edge labeled by v and q's incoming edge labeled by u, and vice versa.

Definition 5.7 (Context Dependence Graph) *Given a program P in mLucid(n). let V be the set of nullary variables in P. The Context Dependence Graph CDG(P) is a directed graph (Z^n, E). E as the set of edges is defined by*

$$E = \{ (p, q) \mid p, q \in Z^n \wedge p \neq q \wedge \exists v, u \in V. (u, q) \in DVD(v, p) \}.$$

Informally, the above definition states the following. A node in a context dependence graph is a context in the context space of the program. Two nodes, p and q, in the graph can be connected through a directed edge if there exist variables v and u in the program such that the value of v at context p directly depends on the value of u at context q.

Definition 5.8 (Context Parallelism) *Given a program P in mLucid(n), let V be the set of nullary variables in P. Given two contexts $p, q \in Z^n$ where $p \neq q$. There exists context parallelism between p and q if there is no path in CDG(P) starting at the node p and ending at the node q, and vice versa.*

Informally, the above definition states the following. There is the parallelism between computations for the values of *all* variables in the program at contexts p and q, if the nodes p and q in the program's context dependence graph are not connected.

To define coarse context dependence graphs of mLucid programs, we first define partitions on the context spaces of the programs.

Definition 5.9 (Partition on Context Space) *Given an n-dimensional context space, a partition* Π *on the context space is a function*

$$\Pi : Z^n \rightarrow S$$

where S is a family of disjoint subsets of Z^n *such that*

$$\forall s, s' \in S \; s \cap s' = \phi \;\; \wedge \;\; \bigcup S = Z^n.$$

Definition 5.10 (Coarse Context Dependence Graph) *Given a program P in mLucid(n), let V be the set of nullary variables in P. Given a partition* $\Pi : Z^n \rightarrow S$. *The Coarse Context Dependence Graph CCDG(P) is a directed graph (S,E). E as the set of edges is defined by*

$$\begin{aligned} E \;=\; & \{ (s, s') \mid s, s' \in S \;\wedge\; s \neq s' \;\wedge \\ & \exists p \in s \; \exists q \in s' \exists v, u \in V. \\ & (u, q) \in DVD(v, p) \; \}. \end{aligned}$$

Informally, the above definition states the following. A node in a coarse context dependence graph is an equivalent class of contexts in the context space of the program determined by the given partition. Two nodes, s and s', in the graph can be connected through a directed edge if there are variables v and u in the program and contexts p and q in the classes s and s' respectively, such that the value of v at context p directly depends on the value of u at context q.

Definition 5.11 (Coarse Context Parallelism) *Given a program P in mLucid(n), let V be the set of nullary variables in P. Given a partition* $\Pi : Z^n \rightarrow S$. *Given two elements of S, s and s', where* $s \neq s'$. *There exists coarse context parallelism between s and s' if there is no path in CCDG(P) starting at the node s and ending at the node s', and vice versa.*

Informally, the above definition states the following. There is the parallelism between computations for the values of *all* variables in the program at *all* contexts in two disjoint classes s and s' of contexts, if the nodes s and s' in the program's coarse context dependence graph are not connected.

For example, Figures 5.2 – 5.4 show the FCDG, CDG and CCDG of the matrix multiplication program defined in Chapter 4, which we rewrite as follows:

```
dimensionality A:{0,1}; B:{0,2};

result
where
  result = asa(0)(C, iseod(C));
  C = fby(0)(0, C + A * B);
end.
```

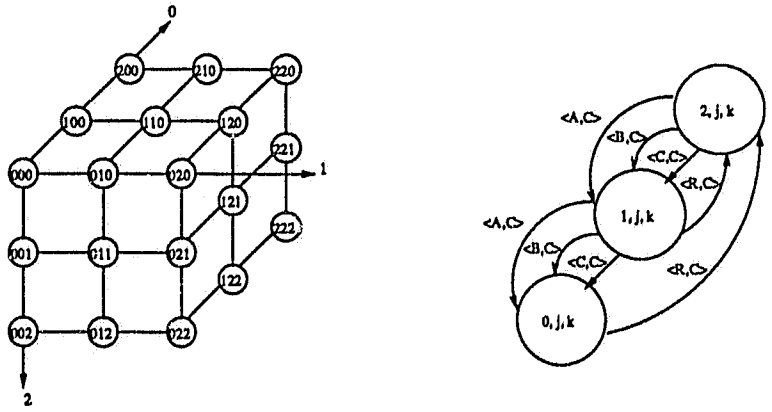

Figure 5.2: The fine context dependence graph of a matrix multiplication program

For the CCDG, we assume that contexts in the 3-dimensional contexts space is partitioned by the coordinates for dimensions 1 and 2, that is, $\Pi(p) = \{q \in Z^3 \mid q(1) = p(1) \wedge q(2) = p(2)\}$.

If we decompose the definition for the variable C in the program into two definitions:

```
C = 0 fby C + AB;
```

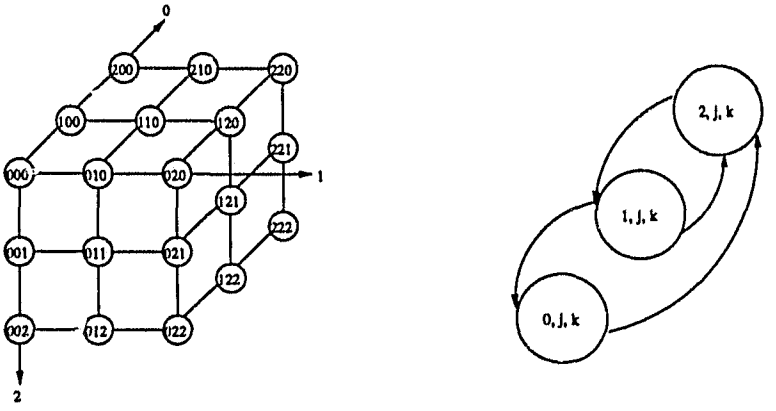

Figure 5.3: The context dependence graph of a matrix multiplication program

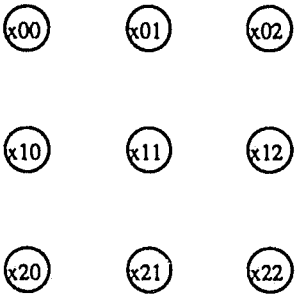

Figure 5.4: The coarse context dependence graph of a matrix multiplication program

```
AB = A * B;
```

from the FCDG we can find two kinds of fine context parallelism in the program. One is among AB's values at all different contexts, and the other is between C's values (also *result*'s values) at any two contexts whose coordinate for dimension 1 or 2 is different. From the CDG, we can find that context parallelism in the program exists between the contexts p and q, if $p(d) \neq q(d)$ for some $d = 1, 2$. From the CCDG, we can find that coarse context parallelism exists between every pair of the equivalent classes defined by the given partition.

In a program, if there is a level of context parallelism between every pair of nodes in the corresponding dependence graph, we say that the program has *maximum context parallelism* at that level. For instance, the matrix multiplication program above has maximum coarse context parallelism with respect to the given partition. In the following, we give the formal definition of maximum context parallelism for each level.

Definition 5.12 (Maximum Fine Context Parallelism) *Given a program P, let (N,E) = FCDG(P). There exists maximum fine context parallelism in P if* $E = \phi$.

Definition 5.13 (Maximum Context Parallelism) *Given a program P, let (N,E) = CDG(P). There exists maximum context parallelism in P if* $E = \phi$.

Notice that maximum context parallelism in a program is the formal definition of the program's pointwise expression parallelism, which we informally defined in the last section.

Definition 5.14 (Pointwise Expression Parallelism) *Given a program P, there exists pointwise expression parallelism in P if and only if there exists maximum context parallelism in P.*

Definition 5.15 (Maximum Coarse Context Parallelism) *Given a program P and a partition* Π *on P's context space, let (N,E) = CCDG(P) with respect to* Π. *There exists maximum coarse context parallelism in P if* $E = \phi$.

Proposition 5.1 *Given a program P, there exists maximum context parallelism in P if and only there exists maximum fine context parallelism in P.*

Proof: Trivial. □

Proposition 5.2 *Given a program P, there exists maximum coarse context parallelism in P if there exists maximum context parallelism in P.*

Proof: Trivial. □

5.4 Indexical Communication

Context parallelism in an mLucid program is limited by the data dependencies among values of variables at different contexts. This dependency is caused by using indexical operators in the program, since it is the only way in which the value of a variable at a context can be defined as a combination of values of variables at other contexts. In other words, if there are no indexical operations in the program, the program has maximum context parallelism. For this, we give the following propositions.

Proposition 5.3 *Given a program P, there exists maximum context parallelism in P if there is no appearance of indexical operators in P.*

Proof: Straightforward from the indexical semantics. □

Proposition 5.4 *Given a program P in mLucid(n) and a partition* Π *on* Z^n*, there exists maximum coarse context parallelism, if during the evaluation of P there is no indexical operation that switchs context p to context q such that* $\Pi(p) \neq \Pi(q)$.

Proof: Straightforward from the indexical semantics. □

Notice that Proposition 5.3 is based on the syntactic properties, whereas Proposition 5.4 is based on the semantic properties. For a particular partition of the context space, we may also give the assertion on the relation between indexical operators and maximum coarse context parallelism in the program based on the syntactic domain. For example, in a simple case that the given partition divides contexts in the context space by their coordinates for dimensions, by checking if certain dimension indicators appear in the program, we can determine whether the program has maximum coarse context parallelism with respect to the partition.

Proposition 5.5 *Given a program P in mLucid(n), and a set of dimensions* $D \subseteq \{d \mid 0 \leq d \leq n-1\}$. *Let the partition* Π *be defined as.*

$$\Pi(p) = \{q \in Z^n \mid \forall d \in D\ p(d) = q(d)\}$$

There exists maximum coarse context parallelism in P if there is no appearance of dimension indicators $d \in D$ *associated with any indexical operators.*

Proof: Straightforward from the indexical semantics. □

From the point of view of parallel computation, we may give a different interpretation to the context dependence in a program. For the fine context dependence, consider the evaluations for the values of two variables at two different contexts as two independent computations. The fine context dependence between the two values means that the computation for one of the values needs to *communicate* with the computation for the other. For the context dependence, consider the evaluations for the values of all variables at two different contexts as two independent computations. The context dependence between the two contexts means that the computation at one of the contexts needs to *communicate* with the computation at the other context. For the coarse context dependence, consider the evaluations for the values of all variables at two different classes of contexts as two independent computations. The coarse context dependence between the two classes means that the computation in one of the classes needs to *communicate* with the computation in the other class.

By indexical semantics, the above described communications are specified by indexical operators in the program. Therefore we term this type of communications **indexical communications.** Indexical communication corresponding to each level of context parallelism can be defined formally by a variation of the context dependence graph at that level. In the variation, a canonical composition of indexical operators is added as a label to each edge in the graph. In the following, we first define canonical compositions of indexical operators, then give the formal definitions of indexical communications.

Definition 5.16 (Canonical Composition of Indexical Operators) *Given two contexts $p, q \in Z^n$, a Canonical Composition of Indexical Operators*

$$\begin{aligned} CCIO(p,q) \subseteq \\ & \{(next(d))^k \mid 0 \leq d < n \ \wedge \ k \in N\} \cup \\ & \{(prev(d))^k \mid 0 \leq d < n \ \wedge \ k \in N\} \end{aligned}$$

where $(next(d))^k$ and $(prev(d))^k$ are k-tuples $(next(d)_1, \cdots, next(d)_k)$ and $(prev(d)_1, \cdots, prev(d)_k)$, respectively. CCIO(p,q) is defined by

$$\begin{aligned} CCIO(p,q) = \ & \{(next(d))^k \mid 0 \leq d < n \ \wedge \ p(d) - q(d) < 0 \ \wedge \ k = q(d) - p(d)\} \cup \\ & \{(prev(d))^k \mid 0 \leq d < n \ \wedge \ p(d) - q(d) > 0 \ \wedge \ k = p(d) - q(d)\}. \end{aligned}$$

Definition 5.17 ((fine) Indexical Communication) *Given a program P in mLucid(n), let V be the set of nullary variables in P. The Indexical Communication corresponding to the fine context parallelism in P is a directed multigraph FIC(P) = (Z^n, E). E as the set of edges is defined by*

$$\begin{aligned} E = \ & \{(p, x, CCIO(p,q), y, q) \mid p, q \in Z^n \ \wedge \ x, y \in V \ \wedge \\ & (p, x, y, q) \in E' \ \ where \ \ (N', E') = FCDG(P)\}. \end{aligned}$$

Definition 5.18 (Indexical Communication) *Given a program P in mLucid(n), the Indexical Communication corresponding to the context parallelism in P is a directed graph IC(P) = (Z^n, E). E as the set of edges is defined by*

$$E = \{(p, CCIO(p,q), q) \mid p, q \in Z^n \ \wedge \ (p,q) \in E' \ \ where \ \ (N', E') = CDG(P)\}.$$

Definition 5.19 ((coarse) Indexical Communication) *Given a program P in mLucid(n), and a partition Π on the context space Z^n, let (N,E') = CCDG(P) with respect to Π. The Indexical Communication corresponding to the coarse context parallelism in P with respect to Π is a directed graph CIC(P) = (N, E). E as the set of edges is defined by*

$$\begin{aligned} E = \ & \{(n_1, I, n_2) \mid n_1, n_2 \in N \ \wedge \ (n_1, n_2) \in E' \ \wedge \\ & I = \{x \mid (p, x, q) \in E'' \ \wedge \ p \in n_1 \ \wedge \ q \in n_2 \ \wedge \ (N'', E'') = IC(P)\}\}. \end{aligned}$$

Notice that in the definition of CIC(P), the indexical communication between two classes of contexts is defined as the set of canonical compositions of indexical operators. Each of the compositions is the indexical operations that switch context from a context in one class to a context in the other class.

For example, Figure 5.5 shows the indexical communication (IC) with respect to the context parallelism in the matrix multiplication program.

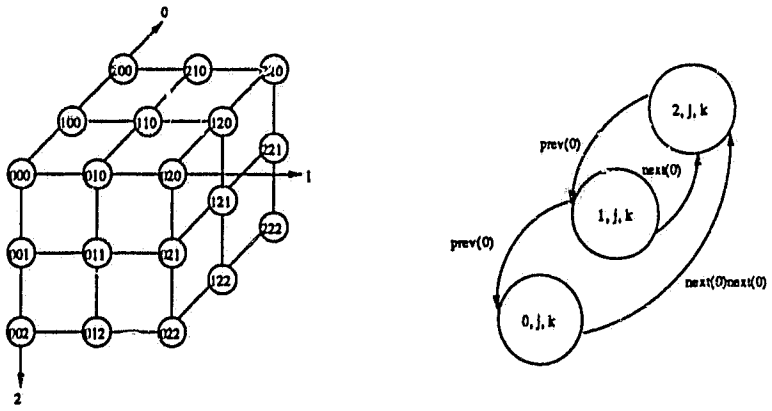

Figure 5.5: The indexical communication of a matrix multiplication program

In executing a program, we can consider that a computation task at a context consists of subcomputation tasks each of which evaluates a needed value of a variable at the context. During the execution, a subcomputation task may communicate with another subcomputation at the same context, which is determined by the dependency between pointwise subexpressions, instead of indexical communication. In this sense, we consider that the communication between subcomputation tasks is the *closest* if it remains in the same context. We consider that the communication between two computation tasks corresponding to two different contexts is *closer* if there is a direct edge from one to the other in the indexical communication graphs of the program.

For example, in the matrix multiplication program, the values of the variables A, B and AB at each context are closest. The contexts p and $[p(0)+1]p$ are closer. All the equivalent classes are not close, as there is no indexical communication between any of them.

Chapter 6

SPECIFICATION OF SYSTOLIC ARRAYS

In this chapter, we first introduce a mathematical model for systolic arrays and describe a syntactically restricted form of mLucid called SysLucid for specifying systolic arrays. We then show how parallel computations and communications in systolic arrays can be expressed by context parallelism and indexical communication in SysLucid programs.

6.1 A Mathematical Model of Systolic Arrays

In the following, we introduce a mathematical model for systolic arrays in two levels: general systolic networks and systolic arrays, which is based on [Lei83][MR84][PS88].

A **general systolic network** (GSN) is defined as a directed multigraph $GSN = (N, E)$, where the nodes $n \in N$ represent processing elements (PEs for short) and the edges $e \in E$ represent communication paths between PEs. Each edge e is a 6-tuple (p, o, c, m, i, q), where

1. p, q are two PEs;
2. the two variable symbols, o attached to the source p and i attached to the sink q, represent an output port of the PE at the source and an input port of the PE at the

sink, respectively;

3. the nonnegative integer m denotes the communication delay along the path; and

4. the constant c in the data domain on which the array works denotes the initial data on the path.

Nodes without incoming or outgoing edges represent the host that sends input data to and collects results from the network. The meaning of a variable symbol f for an input/output port of a PE varies with a clock, i.e. it is a function $f : T \rightarrow D$ where T is the set of time points (natural numbers) and D is the data domain on which the PEs work. For an edge (p, o, c, m, i, q), the value of an input port i is defined by

$$i(t) = \begin{cases} c & \text{if } t < d \\ o(t - m) & \text{otherwise.} \end{cases}$$

The value of an output port o of a PE at a time point depends on the values of some input ports at the same PE at the same time. The semantics of the GSN is the solutions for all the input/output ports at all the PEs in the network.

A systolic array is said to have *spatial regularity* if homogeneous PEs in the network have a regular layout. This regularity can be exploited by placing the PEs on an n–dimensional grid and each PE is identified by its index in the grid.

A **Systolic Array** SA is a GSN associated with a function Γ mapping nodes of the GSN to an n–dimensional grid. It has the following properties. Let (p, o, c, m, i, q) and (q, o, c', m', i, s) be two connected edges in the GSN. The following conditions should hold true:

1. $\Gamma(p) - \Gamma(q) = \Gamma(q) - \Gamma(s)$, where '–'is the vector difference.

2. $m = m'$ and $c = c'$.

These two conditions restrict that the index Γ should be chosen so that the PEs on the lines performing same function applications, for computing the values of their output ports, should be placed in a uniform distance and have the same delay on their communication paths.

For example, Figure 6.1 shows the SA for the systolic matrix multiplication in Figure 2.2, where each node is represented by its index (i,j). In the graph, an edge connecting

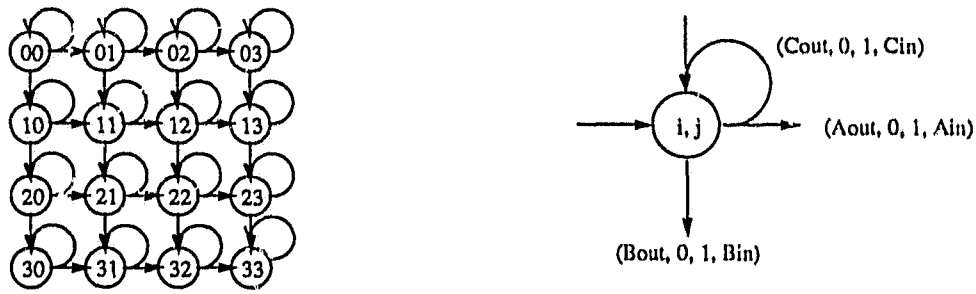

Figure 6.1: A systolic array for matrix multiplication

two nodes, u and v, has either of the form (u,Aout,0,1,Ain,v) or (u,Bout,0,1,Bin,v). An edge connecting the same node v has the form (v,Cout,0,1,Cin,v). The input ports of each node, Ain, Bin and Cin, are defined as

$$Ain_{(i,j)}(t) = \begin{cases} 0 & \text{if } t < 1 \\ Aout_{(i,j-1)}(t-1) & \text{otherwise} \end{cases}$$

$$Bin(t)_{(i,j)} = \begin{cases} 0 & \text{if } t < 1 \\ Bout_{(i-1,j)}(t-1) & \text{otherwise} \end{cases}$$

$$Cin(t)_{(i,j)} = \begin{cases} 0 & \text{if } t < 1 \\ Cout_{(i,j)}(t-1) & \text{otherwise} \end{cases}$$

where $X_{(i,j)}$ means that the input port X at node (i,j). The output ports of each PE, Aout, Bout and Cout, are defined as

$$Aout_{(i,j)}(t) = Ain_{(i,j)}(t)$$

$$Bout_{(i,j)}(t) = Bin_{(i,j)}(t)$$

$$Cout_{(i,j)}(t) = Ain_{(i,j)}(t) * Bin_{(i,j)}(t) + Cin_{(i,j)}(t)$$

The output ports of each host node, $Aout_{(i,0)} and Bout_{(0,j)}$, are defined as

$$Aout_{(i,0)}(t) = A_i(t)$$

$$Bout_{(0,j)}(t) \;=\; B_j(t)$$

where A_i and B_j are input data streams with type $T \to Z$.

6.2 Description

We first define two generic indexical operators, **next(d)(i,x)** and **prev(d)(i,x)**, which are built–in in mLucid by their primitive counterparts. The generic indexical operators, *next* and *prev*, have types

$$next: \;\; N \;\to\; ((ID \times ID) \;\to\; ID)$$

$$prev: \;\; N \;\to\; ((ID \times ID) \;\to\; ID)$$

and defined by

```
next(d)(i,x) = if i = 0 then x else next(d)(i-1, next(d)(x)) fi;

prev(d)(i,x) = if i = 0 then x else prev(d)(i-1, prev(d)(x)) fi;
```

where $i \in (Z^\omega \to N)$. Their indexical semantics and denotational semantics are defined as follows.

$$next = \lambda d.\lambda(i,x).\lambda p.\; x_{[p(d)+i_p/d]p}$$

$$prev = \lambda d.\lambda(i,x).\lambda p.\; x_{[p(d)-i_p/d]p}$$

$$E[next(< constant >)(< exp_1 >, < exp_2 >)]\; env\; p = E[< exp_2 >]\; env\; [p(d)+i)/d]p$$
$$where \;\; i = E[< exp_1 >]\; env\; p; \quad d = I_{mLucid}(< constant >).$$

$$E[prev(< constant >)(< exp_1 >, < exp_2 >)]\; env\; p = E[< exp_2 >]\; env\; [p(d)-i)/d]p$$
$$where \;\; i = E[< exp_1 >]\; env\; p; \quad d = I_{mLucid}(< constant >).$$

To specify the space and time behavior of systolic arrays, we first consider that the context space of a program in mLucid(n+1) consists of a **temporal** subspace and a **spatial** subspace. The temporal subspace is one–dimensional and consists of dimension 0 of the context space, which we also call the **time dimension.** The spatial subspace is

n–dimensional and consists of dimensions k for $1 \leq k \leq n$ of the context space, which we also call the **space dimensions**.

With respect to the time dimension and the space dimensions, the coordinates of a context p in the context space are divided into two vectors, t and s. t representing a point in the time dimension is a singleton consisting of p's coordinate for dimension 0. s representing a point in the spatial subspace consists of all other n coordinates of p. In this sense, we represent a context p as a pair (t,s).

Given a point t in the time dimension, there are infinitely many contexts in the context space associated with t, that is, all the contexts (t,s) with the same t. In other words, each point in the time dimension is associated with a spatial subspace; at different time points the subspaces associated with them may differ. On the other hand, given a point s in the spatial subspace, there are also infinitely many contexts associated with s, that is, all the contexts (t,s) with the same s. In other words, each point in the spatial subspace is associated with a time dimension; at different space points the time dimension associated with them may differ.

In the following description, to distinguish indexical operations on the two subspaces, we call indexical operations that switch context in the time dimension *temporal operations*, and use the suffix notation γ_time to represent $\gamma(0)$, where γ is an indexical operator. We call indexical operations that switch context in the space dimensions *spatial operations*.

With respect to the model of systolic arrays and the above division of the context space, SysLucid as a class of mLucid programs has the following syntactic restrictions.

1. 0–ary variables defined in the where–clause of a program in SysLucid can be classified into three classes: **result**, **output–port** and **input–port**.

2. A variable r in the result class never appears on the right hand side of any equation in the program, and in r's defining expression only variables in the output–port class can appear. The subject of the program's defining expression is the list of all variables in the result class.

3. A variable o in the output–port class is defined by a pointwise expression in which only variables in the input–port class can appear.

4. A variable i in the input–port class is defined by a conditional on the space indices. it has the most general form:

$$
\begin{aligned}
i = \; & \mathit{cond} \\
& \ldots \\
& < \mathit{condition} >_k : < \mathit{exp} >_k \\
& \ldots \\
& \mathit{end};
\end{aligned}
$$

$< condition >_k$ consists of spatial index operators, integer constants and pointwise arithmetic operators. The expression $< exp >_k$ has the form

```
if index_time < m then c else prev_time(m,switch(o)) fi;
```

where m is a nonnegative integer constant, c is a constant. *switch(o)* is a composition of spatial generic operators *next(d)(l,x)* and *prev(d)(l,x)*, where $d \neq 0$, l as an offset argument consists of index operators, integer constants and pointwise arithmetic operators, and o is an output–port variable.

Notice that the *if–then–else–fi* construct is a special case of *cond* with only two cases. Also, here we consider the indexical operator *fby* as a conditional defined by the index operator:

```
fby(d)(x,y) = if index(d) <= 0 then x else prev(d)(y) fi.
```

In the following, we show that a program in SysLucid is semantically equivalent to a GSN.

1. Given a GSN = (V,E) with the sets of input/output ports I and O, a SysLucid program in mLucid(k+1) for some $k > 0$ can be defined as follows.

   - Let I and O be the *input–port* class and the *output–port* class of the program.

- Each node v in V corresponds to a point in the k–dimensional spatial subspace. v is represented by the vector $(v_1, v_2, \cdots, v_k)$.
- Let points in the time dimension correspond to the discrete clock T.
- Each variable i in the input–port class is defined by a conditional with $|V|$ cases, each of which corresponds to a node v. The condition of each case corresponding to v represented by $(v_1, v_2, \cdots, v_k)$ is defined as

  $$\text{index(1) eq } v_1 \text{ and index(2) eq } v_2 \text{ and } \cdots \text{ and index(k) eq } v_k$$

  Let $(u, o, c, m, i, v) \in E$ and $n = u - v$, where $u - v$ is the vector difference of the vector representations of u and v. If u is an interior node, the consequent of the case is defined by

  ```
  if index_time < m then c else prev_time(d, switch(o)) fi;
  ```

  where *switch*(o) is the composition of k generic indexical operators on o, each of which is either *next(j)*(n_j,x) if $n_j \geq 0$ or *prev(j)*(n_j,x) otherwise, where $1 \leq j \leq k$. If u is a host, the consequent is defined by a free variable.
- Let F be the set of functions defining the functionalities of output ports in O. Each variable o in the output–port class is defined by

  $$o = f'_o(I_o);$$

  where f'_o is a pointwise function that has the same semantics as $f_o \in F$ associated with o and $I_o \subseteq I$.

According to the denotational semantics of mLucid, in the above defined program, the value of each variable in the input–port or output–port class at a space point corresponding a node in GSN is the solution of the corresponding input or output port of that node.

2. Given a SysLucid program P in mLucid(k+1), consider the program's coarse context dependence graph CCDG(P) with the partition Π defined by

   $$\Pi(p) = \{q \in Z^{k+1} \mid \forall 1 \leq d \leq k \; p(d) = q(d)\}.$$

CCDG(P) = (V, E) is independent of all initial environments, since, according to the definition of SysLucid programs, the offset argument l of every generic indexical operation *next(d)(l,x)* or *prev(d)(l,x)* in *switch(o)*, which determines the context dependence in the program, is independent of input values. An infinite GSN (V, E') with input/output ports I and O can be defined as follows:

- For each $o \in O$, f_o is the semantics of the expression that defines o in the program.
- For each $(u, v) \in E$, $(u, o, c, m, i, v) \in E'$ for each $i \in I$ and $o \in O$, where m and c are in i's defining expression corresponding to the space index of v.

From the above construction, each $i \in I$ or $o \in O$ at a node in CCDG(P) and the constructed GSN has the same definition for its semantics, hence P and the GSN are semantically equivalent. A finite GSN is a finite subgraph of the infinite GSN constructed above.

A systolic array SA = (V,E,Γ) is semantically equivalent to a SysLucid program P with the following further syntactical constraint corresponding to the constraints in the definition of SA. That is, the defining expression of each input–port variable i consists of a single case

```
if index_time < m then c else prev_time(m,switch(o)) fi;
```

and the offset argument l of each of the generic indexical operations *next(d)(l,x)* and *prev(d)(l,x)* in *switch*(o) is constant.

The following is an operational interpretation of a SysLucid program in accordance with the mathematic model of a systolic array.

1. Each PE in the array corresponds to a point in the spatial subspace of the program's context space.
2. The discrete clock of the array corresponds to points in the time dimension, where we only consider the nonnegative direction of the dimension.

3. All variables in the result class denote outputs of the array, which are data streams.

4. The functionality of an input/output port of each PE is specified by the value of a variable in the input/output port classes at the space point representing the PE.

5. A variable in the output–port class denotes an output port of each PE in the array. The defining pointwise expression of the variable specifies a scalar function to compute the value of the output port from values of some input ports at the same PE at the same time.

6. A variable i in the input–port class denotes an input port of each PE in the array. In the defining expression of the variable, each case k corresponds to a set of PEs whose space indices satisfy $< condition >_k$. The communication path of input port i at each of the PEs at space point p to an output port o of a PE at space point q is specified by $< exp >_k$. In $< exp >_k$, d denotes the delay of the path, c denotes the initial data on the path, *switch(o)* specifies the communication between the PEs, and the temporal operation *prev_time* specifies the delay of the communication.

For example, Program 6.1 shows a SysLucid program, specifying the systolic array in Figure 2.2. In the program, the result class consists of the variable `result`. The input–port class consists of the variables `Ain`, `Bin`, and `Cin`. The output–port class consists of the variables `Aout`, `Bout`, and `Cout`.

```
dimensionality A: {0,1}; B: {0,2}; n: {};

result
where
result = asa_time(Cout, iseod(Aout));

Ain = if index(2) eq 0 then if index_time < 1 then 0 else prev_time(A) fi
      else if index_time < 1 then 0 else prev_time(prev(2)(Aout)) fi fi;
Bin = if index(1) eq 0 then if index_time < 1 then 0 else prev_time(B) fi
```

```
      else if index_time < 1 then 0 else prev_time(prev(1)(Bout)) fi fi;
Cin = if index_time < 1 then 0 else prev_time(Cout) fi;

Aout = Ain;
Bout = Bin;
Cout = Ain * Bin + Cin;
end
```

Program 6.1a A SysLucid program for systolic matrix multiplication

Using the indexical operator *fby* instead of explicit uses of index operators, the above SysLucid program can be simplified as follows.

```
dimensionality A: {0,1}; B: {0,2}; n: {};

result
where
  result = asa_time(Cout, iseod(Aout));

  Ain = fby_time(0, fby(1)(A, Aout));
  Bin = fby_time(0, fby(2)(B, Bout));
  Cin = fby_time(0, Cout);

  Aout = Ain;
  Bout = Bin;
  Cout = Ain * Bin + Cin;
end
```

Program 6.1b A simplified SysLucid program for systolic matrix multiplication

Context parallelism and indexical communication in a SysLucid program specify the parallel computation and communication behavior of a systolic array. Parallel processing

in a systolic array combines two types of parallelism in architecture level: temporal parallelism and spatial parallelism. In a network of PEs, temporal parallelism, or so called pipeline parallelism, can be stated as follows. An output port of each PE in the network produces a sequence of results, i.e. a data stream. The i^{th} data item produced by an output port o_1 at PE_1 at time point t is passed to PE_1's neighboring processor PE_2. Using the received input, an output port o_2 of PE_2 produces its i^{th} data item at next time point $t+1$; and at the same time $(t+1)$ o_1 at PE_1 produces its $(i+1)^{th}$ item. For example, in the matrix multiplication systolic array (Figure 6.1), the PEs in the same horizontal or vertical lines have pipeline parallelism. On the other hand, spatial parallelism means that at a time point t the output ports of two or more PEs in the network can compute their i^{th} items in parallel. For example, in the matrix multiplication systolic array, the PEs in each diagonal have spatial parallelism.

Context parallelism in a SysLucid program specifies both pipeline parallelism and spatial parallelism in the corresponding systolic array. The pipeline parallelism in the systolic array is specified by the context parallelism in the SysLucid program between contexts (t,s) and (t',s'), where s, s' represent two neighboring PEs and t,t' represent two time points, such that in the context dependence graph (t',s') depends on (t-k,s) for some $k > 0$. The spatial parallelism in the systolic array is specified by the coarse context parallelism in the SysLucid program between two classes of contexts s and s' representing PEs at all time points.

For example, Figure 6.2 shows the context dependence graph of the SysLucid program specifying the matrix multiplication systolic array (Program 6.1). In the graph, there are two types of context parallelism. One is between a node indexed by a time point t and a space point v and other nodes indexed by the time points $t' \geq t$ and the space points that are v's neighbors. The other is among nodes that do not have the temporal dependence relation, that is, the context parallelism among nodes (t, v) and (t', v') for all t and t'.

In accordance with pipeline parallelism and spatial parallelism, we may also classify indexical communication in SysLucid programs into three categories: *temporal communication*, *pipeline communication*, and *spatial communication.*

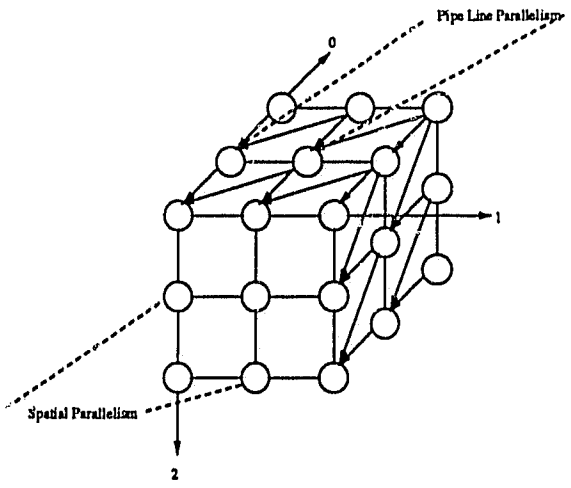

Figure 6.2: The context dependence graph of a systolic matrix multiplication program

Temporal communication in a SysLucid program involves indexical operations that switch context only in the time dimension. Temporal communication specifies the transformation of two internal states of a PE in the array, each of which corresponds to a time point. For example, in the systolic matrix multiplication program, there is temporal communication, at the same space point representing a PE, between the value of variable *Cout* at time point t and that of variable *Cin* at time point $t + 1$.

Spatial communication in the program involves indexical operations that switch context only in the space dimensions. Spatial communication specifies the communication of two directly connected PEs in the array without delay.

Pipeline communication in the program involves indexical operations that switch context in both the time dimension and some of space dimensions. Pipeline communication specifies the communication of two directly connected PEs in the array with delay. For example, in the systolic matrix multiplication program, there is pipeline communication between the value of variable *Aout* at a space point representing a PE at a time point t and that of variable *Ain* at the PE's right neighboring point at time point $t + 1$.

6.3 Examples

In the following we show how to specify a family of systolic arrays for the convolution problem in SysLucid programs. These systolic arrays were originally proposed by H.T. Kung in [Kun82]. The problem is defined as below.

Problem 6.1 Convolution Computation

Given a sequence of weights $(w_0, w_1, \cdots, w_k)$ *and an input sequence* $(x_0, x_1, \cdots, x_n, \cdots)$, *compute the result sequence* $(y_0, y_1, \cdot, y_{n-k}, \cdots)$ *defined by*

$$y_i = \sum_{j=0}^{k} w_j \times x_{i+j} \qquad (i \geq 0).$$

We consider the convolution problem because it is a simple problem with a variety of systolic solutions and representative of a wide class of computations suited for systolic arrays. The convolution problem can be viewed as a problem of combining two data streams, w and x, in a certain manner to form a resultant data stream of y. This type of computation is common to many computation problems, such as filtering, pattern matching, correlation, interpolation, polynomial evaluation, and polynomial multiplication and division. For example, if multiplication and addition are interpreted as comparison and boolean *and*, respectively, the convolution problem becomes the pattern matching problem.

In order to have intuitive meanings, in the following SysLucid programs specifying the convolution systolic arrays, we use the notations, γ_time and γ_space, to represent the $\gamma(0)$ and $\gamma(1)$, respectively, where γ is any of indexical operators that we have defined so far.

In each of the following SysLucid programs, variables with postfix "in" compose the input–port class, variables with postfix "out" compose the output–port class, the variable *result* is the only element in the result class.

Example 6.1 *Systolic Array B1: (broadcast inputs, move results, weights stay)*

The systolic array and its PE definition are depicted in Figure 6.3. Weights are preloaded to the PEs, one at each PE, and stay at the PEs throughout the computation. Partial

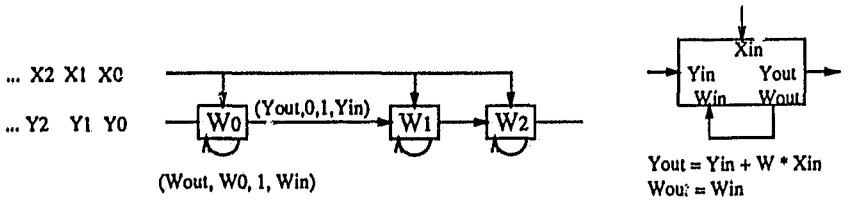

Figure 6.3. Systolic array B1 for convolution

results y_i move systolically from PE to PE in the left-to-right direction, that is, each of them moves over the PE to its right at the next time point. At each time point t, one x_t is broadcasted to all the PEs; and y_t initialized as zero enters the leftmost PE. At time point 0, w_0x_0 is accumulated in y_0 at PE_0, and at time point 1, w_0x_1 and w_1x_1 are accumulated in y_1 and y_0 at PE_0 and PE_1, respectively. Starting from time point k, the final values $y_0, y_1, \cdots$ of y are out from PE_k at rate of one y_i per time point. The following SysLucid program specifies B1.

```
dimensionality x:{0}; w:{1}; k:{};

result
where
   result = asa_space(Yout, index_space eq k);

   Xin = x;
   Yin = fby_space(0, prev_time(Yout));
   Win = fby_time(w, Wout);

   Yout = Yin + Win * Xin;
   Wout = Win;
end
```

In the above program, the broadcast of the input data stream x, which is represented by the input variable x with dimensionality {0}, to the input port *Xin* of each PE is

specified by the combination of variables with different dimensionalities. The variable *Xin* has dimensionality {0} defined by the equation $Xin = x$. The variable *Yin*'s dimensionality includes dimension 1, because of the use of indexical operator *fby_space* in its definition. The definition of *Yout* combines the two variables, so that its result has dimensionality {0,1}. This combination results in that, at each time point t, the value of the output port *Yout* at each PE depends on the value of the input port *Xin* at the same PE at the same time point t; however, all *Xin's* values at t is the same as the t^{th} element of the input data stream x.

The stay of the weight w at each PE is specified by defining the pair of input and output ports, *Win* and *Wout*. The correctness of the specification is guaranteed by the following proposition.

Proposition 6.1 *Given an intension $x \in ID$ and a dimension indicator $d \in N$, if $d \notin DIM(x)$, then the following two definitions of y are equivalent.*

$$y = x$$
$$y = fby(d)(x, y)$$

Proof: Straightforward from the definition of DIM and indexical semantics of *fby*.

Example 6.2 *Systolic Network B2: (broadcast inputs, move weights, results stay)*

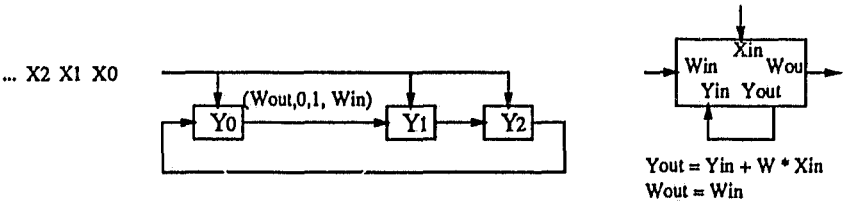

Figure 6.4: Systolic array B2 for convolution

Figure 6.4 shows the systolic network B2 in which each y_i stays at a PE to accumulate its items and the weights circulate around the array. The first weight w_0 is associated with a tag bit that signals y_i to reset its content so that a new accumulation can start. The accumulation result of y_i is not outputted until all the weights have passed the PE.

Unlike B1, B2 can only produce a finite sequence of y_i's from y_0 to y_n computed from a finite input stream x_i with n+k elements and a sequence of weights with k elements. The following SysLucid program specifies B2.

```
dimensionality x:{0}; w:{1}; n:{};

result
where
  result = asa_time(Yout, (Wout neq 0) and (next_time(Wout) eq 0));

  Xin = x;
  Yin = fby_time(0, Yout);
  Win = fby_time(w, fby_space(next_space(n,Wout), Wout));

  Yout = Yin + Win * Xin;
  Wout = Win;
end
```

In the above program, we assume that the input sequence w of weights, represented by the input variable w with dimensionality $\{1\}$, has $n+1$ elements; among them there are k *real* weights which have been laid in correct positions in the sequence, and others are filled with 0. The input port *Win* at each PE is defined as follows. At the initial time, it has an initial value from the input sequence w; at every next time point it receives a value from the output port *Wout* at either its right neighbor if the PE is not the leftmost one, or the rightmost PE otherwise.

The final result produced by *Yout* of each PE is specified by the definition of the variable *result*. It is the *Yout*'s value at the time point at which the *Wout* of the same PE has a nonzero value, but at next time point it has a zero value.

Example 6.3 Systolic Array R1*: (results stay, inputs and weights move in opposite directions)*

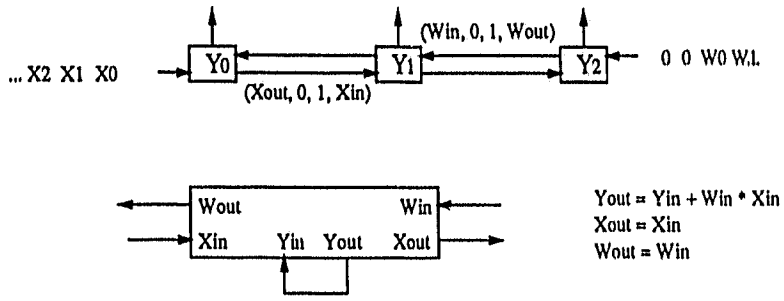

Figure 6.5: Systolic array R1 for convolution

Figure 6.5 shows the systolic array R1 in which each partial result y_i stays at a PE to accumulate its items. The x and w move systolically in opposite directions such that when an x_j meets a $w_{j'}$ at a PE, they are multiplied and the result product is accumulated in the y_i staying at that PE. To ensure that each x_i of x is able to meet every w_j of w, consecutive elements in the data stream x are separated by two cycles and so are those in w. like B2, R1 can only produce a finite output stream y from y_0 to y_n computed from a finite input data stream x from x_0 to x_{n+k} and a sequence of weights. The following SysLucid program specifies R1.

```
dimensionality x:{0}; w:{0}; n:{};

result
where
  result = asa_time(Yout, (Wout neq 0) and (next_time(Wout) eq 0));

  Xin = fby_space(x, fby_time(0, Xout));
  Yin = fby_time(0, Yout);
  Win = if index_space eq n then w else fby_time(0, next_space(Wout)) fi;

  Xout = Xin;
  Yout = Yin + Win * Xin;
  Wout = Win;
```

end

In the above program, the input port *Xin* at the leftmost PE receives the input data stream x; the input port *Win* at the rightmost PE receives the input data stream w. Other input ports, *Xin* and *Win*, are initially 0 and at each next time point they receive data items from the *Xout* ports and *Wout* ports at their left and right neighbors.

In the program, to satisfy the synchronization requirement of the array, we assume that the input data streams x and w have been rearranged as follows. Two consecutive elements in the streams are separated by a zero. A number of zero's are added to the head of x (or w) to guarantee that the corresponding elements of the two streams meet at the right PEs during the computation. By defining two indexical functions, *insert* and *prefix*, in mLucid, we can also specify these rearrangements of the data streams.

The function *prefix(d)(c,n,x)* takes a constant c, a nonnegative integer n and an intension x. It shifts x's values at contexts on the positive side of dimension d n positions along the dimension toward the positive direction, and fills c at the empty contexts left by the shift. The function is defined by

```
prefix(d)(c,n,x) = prev(d)(n-1, fby(d)(c,x));
```

The function *insert(d)(c,n,x)* takes a constant c, a nonnegative integer n and an intension x. It stretches x's value at contexts on the positive side of dimension d along the dimension, leaving n empty positions between the values originally at two neighboring contexts in dimension d, and fills c at the empty contexts lefted by the stretch. The function is defined by

```
insert(d)(c,n,x) = fby(d)(x, prefix(d)(c,n,insert(d)(c,n,next(d)(x))));
```

For example, in the systolic array R1, let x0 and w0 be the original data streams without the rearrangement. The rearranged input data streams x and w in the program can be defined as

```
x = prefix_time(0, n, insert_time(0,1,x0));
w = insert_time(0,1,w0).
```

Example 6.4 *Systolic Array R2: (results stay, inputs and weights move in the same directions but at different speeds)*

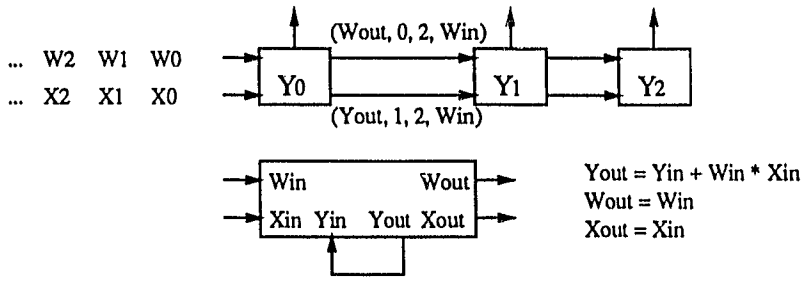

Figure 6.6: Systolic array R2 for convolution

Figure 6.6 shows the systolic array R2 in which each partial result y_i stays at a PE to accumulate its items. The both input data streams, x and w, move from left to right systolically, but the elements x_i of x move twice as fast as that of w. More precisely, to pass w's elements between two neighboring PEs, two time units of delay are needed during the communication between the input/output ports of two PEs, while to pass x's only one unit is needed. Like R1, R2 can only produce a finite data stream y from y_0 to y_n computed from a finite input data stream x from x_0 to x_{n+k} and a finite sequence of weights. The following SysLucid program specifies R2.

```
dimensionality x0:{0}; w:{0};

result
where
  result = asa_time(Yout, (Wout neq 0) and next_time(Wout) eq 0);

  Xin = fby_space(x, fby_time(0, Xout));
  Yin = fby_time(0, Yout);
  Win = fby_space(w, fby_time(0, fby_time(0,Wout)));

  Xout = Xin;
```

```
  Yout = Yin + Win * Xin;
  Wout = Win;

  x = insert(0,1,x0);  //rearrange input stream x0
end
```

In the above program, the two-time-unit delay of the data stream w through the array is specified using the indexical operator *fby_time* twice in *Win*'s definition. The definition says that at the leftmost PE the input port *Win* receives the input stream w; at each of other PEs, *Win* initially has value 0 at the first two time points, then it receives data items from *Wout* at its left neighbor that were produced two time points ago.

Example 6.5 *Systolic Array W1: (weights stay, inputs and results move in opposite directions)*

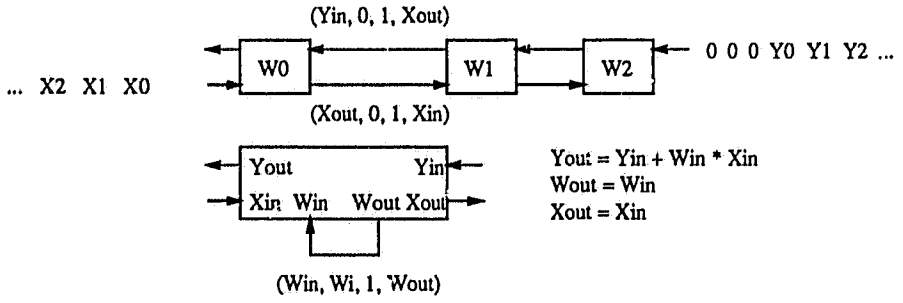

Figure 6.7: Systolic array W1 for convolution

Figure 6.7 shows the systolic array W1 in which the sequence of weights w stay, one element at each PE, while the input/output streams x and y move systolically in opposite directions. Similar to the systolic array R1, consecutive x_i's and y_i's are separated by two time units. Notice that because of the communication path for moving y_i to the PE at the leftmost boundary of the array, there is no need to collect each y_i from each PE. Instead, y's are output from the leftmost PE when the last input, x_{i+k}, enters the PE. The following SysLucid program specifies W1.

```
dimensionality x0:{0}; w:{1}; k:{};
```

```
result
where
  result = origin_space(Yout);

  Xin = fby_space(x, fby_time(0, Xout));
  Yin = if index_space eq k then 0 else fby_time(0, next_space(Yout)) fi;
  Win = fby_time(w, Wout);

  Xout = Xin;
  Yout = Yin + Win * Xin;
  Wout = Win;

  x = insert(0,1,x0);      // rearrange input stream x0
end
```

Example 6.6 *Systolic Array W2: (weights stay, inputs and results move in the same direction but at different speeds)*

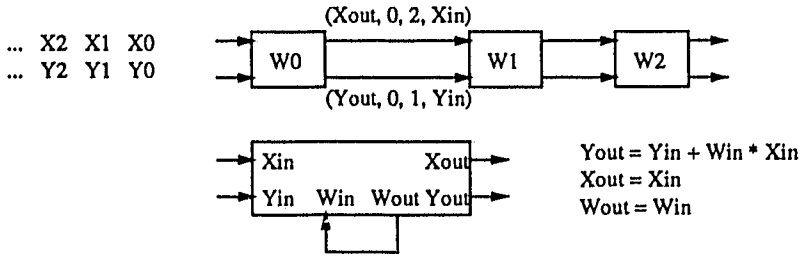

Figure 6.8: Systolic array W2 for convolution

Figure 6.8 shows the systolic array W2 in which the sequence of weights stay, one element at each PE, while both the input and output streams x and y move systolically from left to right through the array. Similar to R2, each x_i in x moves twice as fast as y_i in y along the communication paths in the array. The output stream of the array is output from the

rightmost PE. The following SysLucid program specifies W2.

```
dimensionality x:{0}; w:{1}; k:{};

result
where
  result = asa_space(Yout, index_space eq k);

  Xin = fby_space(x, fby_time(0, Xout));
  Yin = fby_space(0, fby_time(0, fby_time(0,Yout)));
  Win = fby_time(w, Wout);

  Xout = Xin;
  Yout = Yin + Win * Xin;
  Wout = Win;
end.
```

Chapter 7

MULTIDIMENSIONAL DATAFLOW PROGRAMMING

7.1 Overview

Using mLucid to specify parallel computations and communications of systolic arrays can be considered as a form of explicitly parallel programming. In an mLucid program specifying a systolic array, we specify how a problem is decomposed into small computation tasks that compute values of variables at various contexts, and how the tasks are distributed to PEs represented by space points in the spatial subspace of the program's context space. We consider that the decomposition and distribution are explicit, because they can be determined by the syntactic structure of the program. A computation task or process at a context is specified when the program is written, instead of being dynamically spawned during the execution of the program. We also specify the communication between PEs by spatial operations. The specification is explicit in the program, because there exists a communication path between two PEs only if there is a spatial operation explicitly defined in the program that switches the space points representing the PEs from one to the other. Here we may have two views of indexical operations in the program.

- From the point of view of semantics, an indexical operation specifies data dependencies between values of objects at two different contexts.
- From the point of view of parallel execution, an indexical operation can be treated as an explicit construct for message–passing between PEs when it is a spatial operation, or for synchronizing computations on PEs when it is a temporal operation.

On the other hand, however, mLucid as an indexical functional language has implicit parallelism, and all constructs including indexical operators in the language have mathematical meaning. Uses of the constructs in an mLucid program are part of the user's efforts for problem–solving, instead of only specifying parallelism and communication. In this sense, we may have another view of an mLucid program that specifies a systolic array, that is, it expresses an algorithm for solving the problem that the systolic array solves. The algorithm, hence the program, is based on the principle of systolic computation, but the program is not necessarily implemented on a systolic array. It can be considered as a general parallel program with implicit pipeline and spatial parallelism and can be implemented on any parallel architecture. This idea, also called *soft systolic programming*, was originally proposed by [Sha87] for *systolic logic programming*.

In this chapter we propose a parallel programming technique, which we term **multidimensional dataflow programming**. We use the term multidimensional dataflow programming, instead of systolic programming, because (i) systolic computation is essentially a kind of synchronized dataflow computation; (ii) we want to emphasize the asynchronous nature of dataflow computation; (iii) in terms of dataflow programming, mLucid extends Lucid to allow multidimensional data streams.

Multidimensional dataflow programming as an extension of the original dataflow programming is concerned with specifying dataflow networks that have modularity and multidimensionality. Informally, the modularity means that a dataflow network is built on few types of dataflow sub–networks, which we term **dataflow modules**, and composed of many copies of the modules. The multidimensionality means that in the network these sub–networks are connected in a regular multidimensional structure, and multiple data streams flow through the sub–networks along their multidimensional communication

paths. We term this kind of dataflow networks **multidimensional dataflow networks.**

7.2 Description

mLucid as a multidimensional dataflow language provides constructs, i.e. indexical operators, to specify both dataflow modules and their multidimensional interconnections. In multidimensional dataflow programming, like in specifying systolic arrays, we divide the context space of an mLucid program into two subspaces, the time dimension and the spatial subspace. Accordingly, we classify indexical operators in mLucid into two categories, temporal operators that work on the time dimension , i.e. dimension 0, and spatial operators that work on spatial dimensions. In specifying a dataflow module, we define a set of equations in mLucid that specifies a (zero-dimensional) dataflow sub-network. The defining expression of each of the equations contains only pointwise and temporal operators. Free variables in the equations correspond to the input ports of the network. To specify the interconnection of dataflow sub-networks duplicated from the defined dataflow modules, we use spatial operators to set up the communication path between an input port of a sub-network located at a space point to an output port of a sub-network located at another space point. This way, an mLucid program specifies a multidimensional dataflow network.

Based on this idea, we will describe an indexical multidimensional dataflow programming technique.

The technique consists of an abstract machine on which multidimensional dataflow programs are designed, a programming language, namely mLucid, for programming the abstract machine, and a programming methodology for developing multidimensional dataflow programs. Here we do not need explicit notations or constructs for mapping processes to processors on the abstract machine. Indexical operators in mLucid programs implicitly express the process-to-processor mapping.

The abstract machine is an infinite *processing space* consisting of infinitely many processors. Each processor is located at a point in an n-dimensional grid, where n is arbitrary.

Through grid edges, each processor at a gridpoint is connected to the processors at its $2n$ neighboring points. At a processor, an input port receives a data stream from an output port of a neighboring processor; an outport sends a data stream to an input port of a neighboring processor. Each processor is a dataflow machine that executes dataflow networks. The task of multidimensional dataflow programming on the abstract machine is to define, for each of the processors on the machine, a dataflow network and communication paths to other processors through input/output port connections.

The methodology for developing multidimensional dataflow programs in mLucid consists of the following steps.

1. Define a set of dataflow modules. Each module is defined by a set of mLucid equations without spatial operators.
2. Identify the input/output ports for each of the defined dataflow modules.
3. Map the processors in the processing space to the defined dataflow modules.
4. Define a set of communication routes. Each route is defined by an mLucid expression with spatial operators.
5. Map the input ports of the processors to the defined set of communication routes.
6. Identify the input/output ports of the boundary processors that receive/send data streams from/to the outside world.

7.3 Examples

In the following, we explain the methodology in detail while designing two multidimensional dataflow programs for generating prime numbers and solving linear equations.

Consider generating a sequence of prime numbers using the well-known algorithm called "sieve of Erastothenes". The algorithm consists of a certain number of iterations, depending on how many prime numbers we want to generate. Each iteration step generates one prime number. The initial value of the iteration is the sequence of consecutive natural

numbers starting from 2. Each iteration step outputs the first number of the sequence created at the previous step as a prime number, and creates a new sequence by removing all the numbers in the current sequence that are multiples of the generated prime at this step.

The algorithm can be implemented in a multidimensional dataflow network. The network consists of two dataflow modules. The first module, which we call *initializer* generates and outputs a data stream consisting of the consecutive natural numbers starting from 2, which we call *candidates*, and a constant stream consisting of the first prime number 2, which we call *prime*. We specify the *initializer* in the following set of mLucid equations

```
prime = origin_time(candidates);
candidates = fby_time(2, candidates + 1).
```

The module does not have input port, so that there is no free variable in the equations. It has two output ports represented by the variables *prime* and *candidates*. Figure 7.1 shows the *initializer* module.

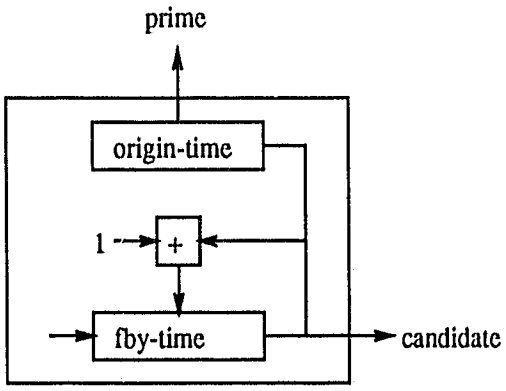

Figure 7.1: The initializer module of a multidimensional dataflow network for prime number generation

The second module, which we call *generator*, takes an input data stream, and produces two data streams, which we also call *candidates* and *prime*, respectively. The *candidates* is created from the input stream in the way that those elements of the input stream whose

values are multiples of the first element of the input stream, which is a prime, are filtered out. The *prime* is a constant stream; every element in the stream has the same value equal to the first element of the *candidates*.

Before specifying the *generator module* in mLucid, we define an indexical operator called **whenever**, which is built-in in mLucid.

$$whenever : N \rightarrow (ID \times ID \rightarrow ID).$$

```
whenever(d)(x,t) = if origin(d)(t)
        then fby(d)(origin(d)(x), whenever(d)(next(d)(x), next(d)(t)))
        else whenever(d)(next(d)(x), next(d)(t)) fi.
```

Informally, at a context p on the positive side of dimension d, the operator returns the value of x at the context q, such that p and q is lined up in dimension d, and t has the $(p(d)+1)^{th}$ true value at q from d's origin through q.

The formal indexical semantics of *whenever* is defined by:

$$whenever = \lambda d.\lambda(x,t).\lambda p.\ x_{[k/d]p}$$
$$where\ \ k \in N\ \wedge\ t_{[k/d]p}\ \wedge$$
$$|\{i \mid 0 \leq i < k\ \wedge t_{[i/d]p}\}| = \begin{cases} p(d) & \text{if } p(d) \geq 0 \\ 0 & otherwise \end{cases}$$

The denotational semantics of *whenever* is defined by:

$$E[whenever(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$E[< exp_1 >]\ env\ [k/I_{mLucid}(< constant >)]p$$
$$where$$
$$k \in N\ \wedge\ t_{[k/d]p}\ \wedge$$
$$|\{i \mid 0 \leq i < k\ \wedge t_{[i/d]p}\}| = \begin{cases} p(d) & \text{if } p(d) \geq 0 \\ 0 & otherwise \end{cases}$$

As a filter in a dataflow network, *whenever(0)* or *whenever_time* takes two input data streams, x and t, where t is a stream of boolean values, and produces a data stream from

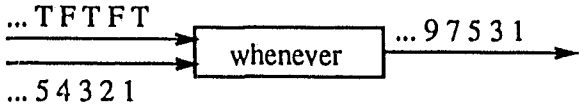

Figure 7.2: The behavior of whenever filter

x by removing those elements of x if their corresponding elements of t at the same stream indices are false. Figure 7.2 illustrates the behavior of the *whenever_time* filter.

Using *whenever_time* and other temporal and pointwise operators, we specify the dataflow module *generator* in the following two mLucid equations:

```
prime = origin_time(candidates);
candidates = whenever_time(x, (x mod origin_time(x)) neq 0).
```

The module has an input port represented by the variable x and two output ports represented by the variables *prime* and *candidates*. Figure 7.3 shows the *generator* module.

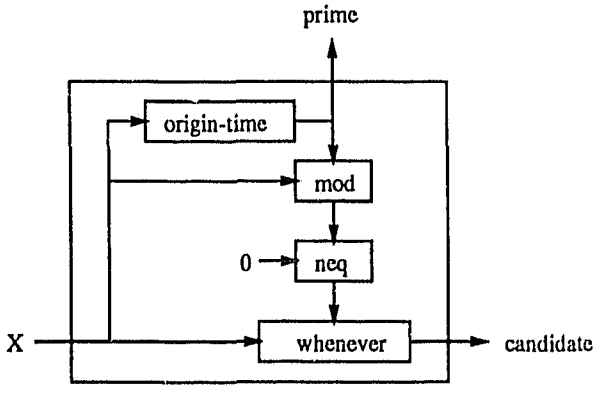

Figure 7.3: The generator module of a multidimensional dataflow network for prime number generation

The prime generator as a multidimensional dataflow network can be programmed on the one–dimensional processing space. We map the processor at the point in the processing space with index 0 to the *initializer* module and the processors at points with index $i > 0$ to the *generator* module. Figure 7.4 shows the processing space after the mapping.

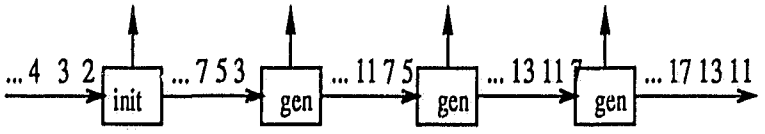

Figure 7.4: A multidimensional dataflow network for generating prime numbers

Based on the above mapping, we specify the communication route for the input port x of each processor that executes the *generator* module in the following mLucid definition

```
x = prev(1)(candidate).
```

Finally, we combine all the above specifications together and write a multidimensional dataflow program for generating prime numbers as follows.

```
prime
where
  prime = origin_time(candidates);
  candidates = fby(1)(fby_time(2, candidates + 1),
    whenever_time(candidates,
                  (candidates mod origin_time(candidates)) neq 0));
end.
```

Program 7.1: A multidimensional dataflow program for prime generation

In the following, we design a multidimensional dataflow program for solving linear equations.

Given a set of linear equations

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 = b_0$$
$$a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

A parallel algorithm for solving the equations based on Gauss elimination is as follows [Sha87]. The algorithm is designed on a triangular array of processors. There are two types of processors in the array: pivot (in circle) and cell (in square). The algorithm has two phases: elimination and back–substitution. Figure 7.5a and 7.5b show the directions of dataflow in the two phases, respectively.

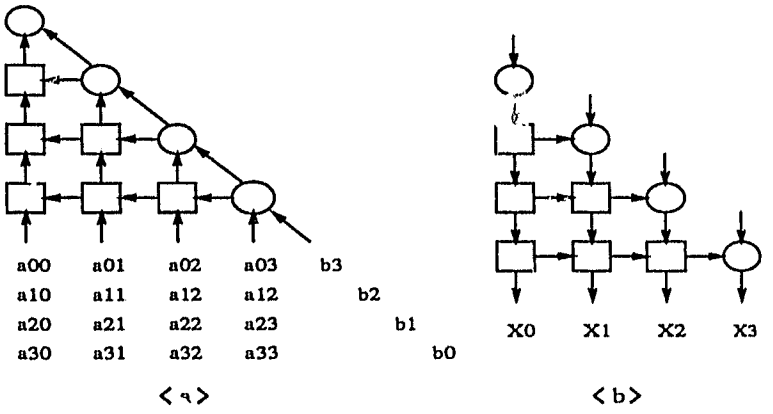

Figure 7.5: The dataflow direction of (a) the elimination and (b) the back–substitution phases of a multidimensional dataflow network for Gauss elimination

In the elimination phase, the coefficient matrix A of the linear equations is input at the bottom processors; each of the processors receives an input data stream consisting of a column of A. The vector B as a data stream is input into the pivot processor at the bottom. The functionalities of a pivot processor and a cell processor are shown in Figure 7.6a and 7.6b.

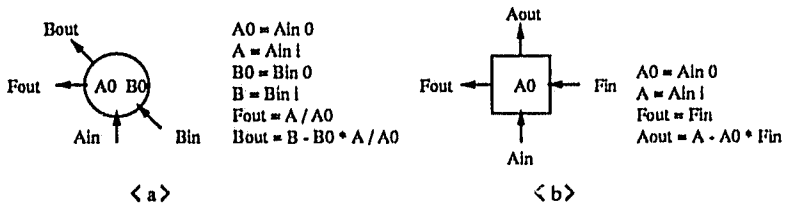

Figure 7.6: The functionality of the (a) pivot and (b) cell processors in the elimination phase

In the figure, for a symbol S denoting an input data stream, S_0 is the first element

of S, which is stored at the processor after it is input, and S_i is the rest of S after S_0 is removed. From the definitions of the two types of processors and their connections in the array, we can see that at each horizontal line i ($0 \leq i \leq n$) of the array from bottom up, the variable x_{n-i} in the $n+1$ equations is eliminated. Hence, when the data reach the top pivot processor of the array ($i = n$), the value of x_0 is computed. In this phase, data flows from bottom up and from right to left.

In the back–substitution phase, the computed x_i flows back from top down. When x_i arrives at a cell processor at horizontal line $n - i - 1$, the processor computes the product of x_i and the stored coefficient A_0, adds the product to the partial sum of the products received from its left neighbor, and passes the new partial sum to its right neighbor. Eventually, the pivot processor in the line receives the sum of all the products from its left neighboring cell processor, and uses it to solve x_{i+1}. In this phase, data flows from top down and from left to right. The final solutions of variables x_i's ($0 \leq i \leq n$) flow out from the processors at the bottom of the array. Figure 7.7a and 7.7b show the functionalities of a cell processor and a pivot processor in this phase, respectively.

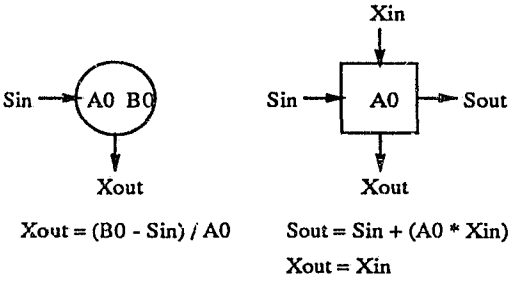

Figure 7.7: The functionality of the (a) pivot and (b) cell processors in the back–substitution phase

To implement the algorithm as a multidimensional dataflow network, we first implement the two types of processors as dataflow modules. For the *pivot* processor, Figure 7.8a shows the dataflow network implementing the elimination phase of the computation on the processor. In the network, we use the filter *origin_time* to implement A_0 and B_0 in the algorithm. The output of the filter is a constant stream with the value of A_0 or B_0, as if it would be stored in the processor. We specify the dataflow network as a set of

mLucid equations as follows:

```
A0 = origin_time(Ain);
Ai = next_time(Ain);
B0 = origin_time(Bin);
Bi = next_time(Bin);
Fout = Ai / A0;
Bout = Bi - B0 * Ai / A0.
```

Figure 7.8b shows the dataflow network implementing the back–substitution phase of the *pivot* processor. The following mLucid equations specify the dataflow network.

```
A0 = origin_time(Ain);
B0 = origin_time(Bin);
Xout = (B0 - Sin) / A0.
```

The composition of the above specified two dataflow networks, shown in Figure 7.8c, is the dataflow module implementing the *pivot* processor. Accordingly, we combine the two sets of equations as follows to specify the module.

```
A0 = origin_time(Ain);
Ai = next_time(Ain);
B0 = origin_time(Bin);
Bi = next_time(Bin);
Fout = Ai / A0;
Bout = Bi - B0 * Ai / A0.
Xout = (B0 - Sin) / A0.
```

There are three input ports in the module represented by the variables *Ain*, *Bin* and *Sin*, and three output ports in the module represented by the variables *Fout*, *Bout*, and *Xout*.

Similarly, Figure 7.9a and 7.9b show the dataflow networks implementing the elimination and back–substitution phases of the *cell* processor. In the dataflow networks, the

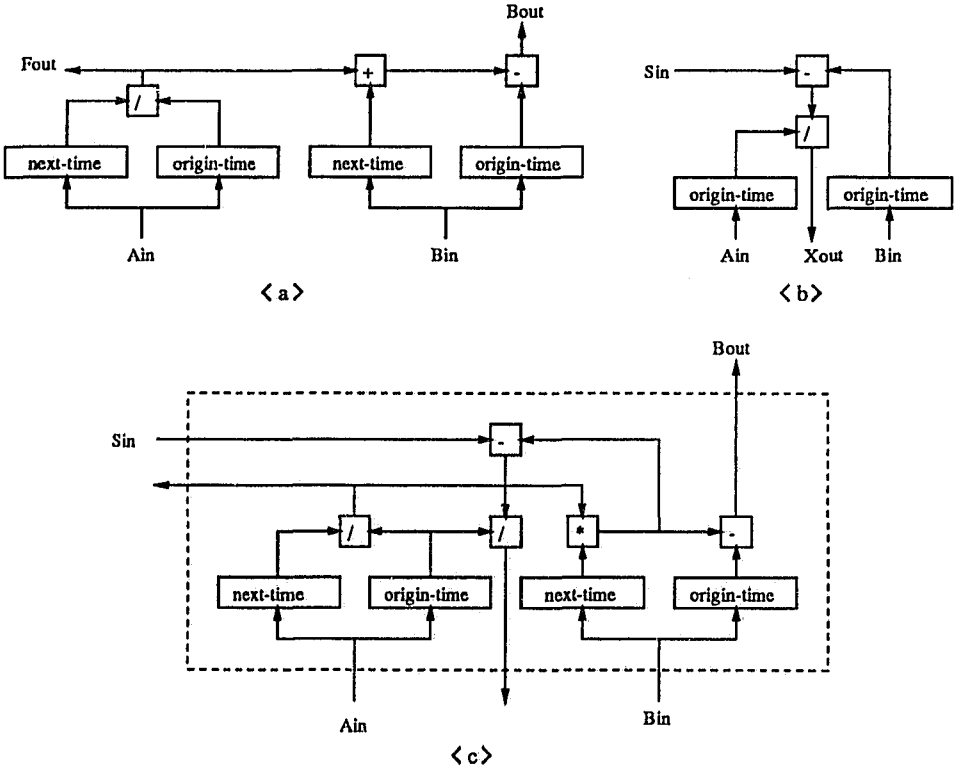

Figure 7.8: The pivot dataflow module for Gauss elimination

filter *id* is an identity operator that outputs the same data stream as its input stream. The dataflow network for the elimination phase can be specified by the set of equations:

```
A0 = origin_time(Ain);
Ai = next_time(Ain);
Fout = Fin;
Aout = Ai - A0 * Fin.
```

The dataflow network for the back-substitution phase can be specified by the set of equations:

```
A0 = origin_time(Ain);
Sout = Sin + (A0 * Xin);
Xout = Xin;
```

Figure 7.9c shows the dataflow module implementing the *cell* processor composed of the above specified two dataflow networks. Accordingly, we combine the two sets of equations as follows to specify the module.

```
A0 = origin_time(Ain);
Ai = next_time(Ain);
Fout = Fin;
Aout = Ai - A0 * Fin.
Sout = Sin + (A0 * Xin);
Xout = Xin;
```

There are four input ports in the module represented by the variables *Ain*, *Fin*, *Sin* and *Xin*, and four output ports in the module represented by the variables *Fout*, *Aout*, *Sout* and *Xout*.

At the next step, we map the processors in the two-dimensional processing space to the two dataflow modules, *pivot* and *cell*, in the same layout as in Figure 7.5, and locate the bottom and left boundaries of the array at the axes of dimensions 1 and 2, respectively. Based on the layout, we then specify the communication routes for input ports of the

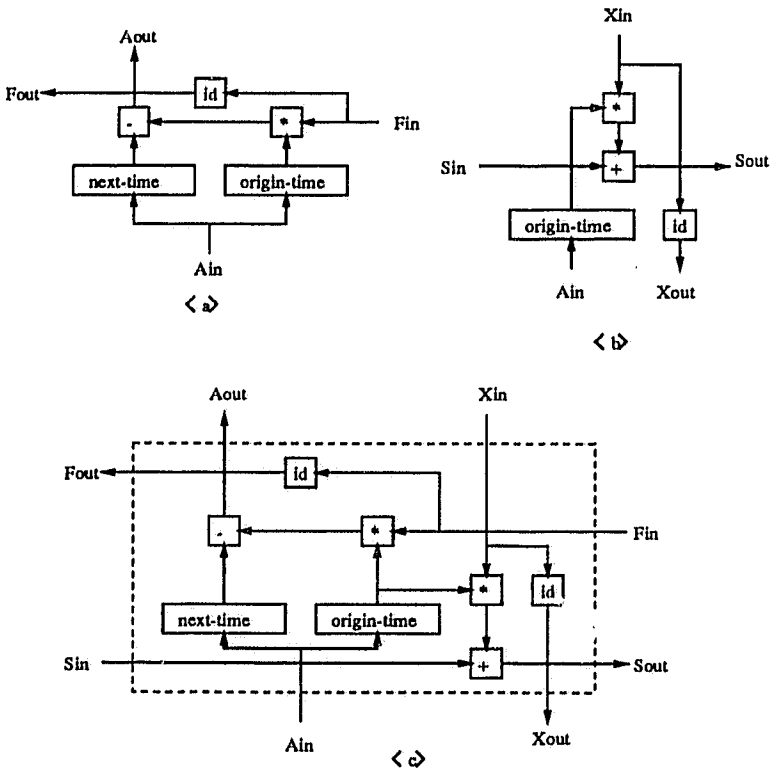

Figure 7.9: The cell dataflow module for Gauss elimination

dataflow network at a processor in the processing space. For the input ports *Ain*, *Bin* and *Sin* of each interior *pivot* processor, their communication routes can be specified by the mLucid definitions

```
Ain = prev(2)(Aout)
Bin = prev(2)(next(1)(Bout))
Sin = prev(1)(Sout).
```

Given two input streams to the processing space represented by variables A and B, the input ports *Ain* and *Bin* of the bottom *pivot* processor can be specified by the mLucid definitions

```
Ain = A
Bin = B.
```

According to the layout of the pivot processors in the array, we combine the definitions of the input ports at the interior and boundary as follows:

```
Ain = fby(2)(A, Aout)
Bin = fby(2)(B, next(1)(Bout))
Sin = prev(1)(Sout).
```

Following the same procedure, we can define the input ports of *cell* processors by the following definitions:

```
Ain = fby(2)(A, Aout)
Fin = next(1)(Fout)
Sin = fby(1)(0, Sout)
Xin = next(2)(Xout).
```

Finally, by combining all the above definitions together, we have the following multi-dimensional dataflow program for solving linear equations.

```
dimensionality A:{0,1}; B:{0}; n:{};
```

```
origin_time(origin(2)(Xout))
where
  A0 = origin_time(Ain);
  Ai = next_time(Ain);
  B0 = origin_time(Bin);
  Bi = next_time(Bin);
  Aout = Ai - A0 * Fin;
  Bout = Bi - B0 * Ai / A0;
  Fout = if diagonal then Ai / A0 else fin fi;
  Sout = Sin + (A0 * Xin);
  Xout = if diagonal then (B0 - Sin) / A0 else Xin fi;
  Ain = fby(2)(A, Aout);
  Bin = fby(2)(B, next(1)(Bout));
  Fin = next(1)(Fout);
  Sin = fby(1)(0, Sout);
  Xin = next(2)(Xout);
  diagonal = n - index(1) eq index(2);
end.
```

Program 7.2: A multidimensional dataflow program for Gauss elimination

There are two major differences between a systolic array and a multidimensional dataflow network designed in the processing space.

Synchronous vs. Asynchronous In a systolic array, computation at a PE and communication between PEs are synchronized based on a global clock. In the SysLucid program specifying the array, the synchronization is specified by the definitions that explicitly refer to the global clock, i.e. the index of the time dimension, and by the temporal operations that specify the delay of the communication.

In a multidimensional dataflow network, dataflow computation at a processor and communication between processors are not synchronized. The computation depends

on the availability of data on the incoming arcs of filters in the dataflow network that is assigned to the processor. The communication depends on the availability of data on the communication paths between input/output ports of the processors. Notice that here we assume that the dataflow computation is data driven. In the mLucid program specifying the multidimensional dataflow network, the asynchronization is reflected by implicit references to the index of the time dimension and no temporal operation that specifies the delay of the communication between the processors.

Scalar computation vs. Stream computation In a systolic array, computation at each PE consists of only scalar operations; at each time point the same operations are repeatedly performed. In the SysLucid program specifying the array, the scalar computation is specified by only allowing pointwise operations in the definitions of the output ports of a PE. In other words, the computation at a PE in the systolic array is specified by a dataflow network consisting of only pointwise filters.

For a multidimensional dataflow network, computation in a dataflow subnetwork at each processor consists of not only scalar operations but also stream operations, such as *whenever_time*, whose meanings depend on the entire input streams. In the mLucid program specifying the network, the stream computation is specified by the temporal operations in the definitions of dataflow modules.

There are two levels of parallelism in a multidimensional data flow program. At the outer level, like in a SysLucid program, there are pipeline parallelism and spatial parallelism among the processors. The both kinds of parallelism are specified by context parallelism in the program. Notice that here we assume that the communication path between two processors is pipelined, otherwise there may be also parallel data passing through the path. At the inner level, there is parallelism introduced by the dataflow computation at each processor. The parallelism is specified by the expression parallelism and context parallelism in the equations defining dataflow modules, where the context parallelism occurs if the tagged dataflow computation is assumed.

Chapter 8

DATA PARALLEL PROGRAMMING

8.1 Overview

Data parallel programming is to design programs for fine–grained parallel computers with general communications. Parallelism in these programs comes from simultaneous operations across large sets of data, which is called *data parallelism* [HS86a].

There are five basic requirements for programming languages that support data–parallel programming in terms of expressing data parallelism in programs: *elementwise parallelism*, *parallel conditional*, *replication*, *reduction*, and *permutation*, which are also called *common themes* of the data–parallel languages [Ste88].

There is a natural match between data parallelism and context parallelism in mLucid programs. In indexical programming, the value of an object at various contexts corresponds to a collection of data; parallel operations on the object at different contexts correspond to parallel operations across the collection of data in data parallel programming. An mLucid program can have massive context parallelism, as its context space is theoretically infinitely large and computations on values of objects at different contexts can be independent of each other. Synchronous computations for values of objects at various contexts can also be expressed in mLucid programs, like in specifying systolic arrays,

by dividing the context space into the time dimension and the spatial subspace.

8.2 Description

In this section, we describe how the five common themes of data-parallel languages can be implemented by constructs in mLucid.

Considering that collections of data are values of expressions at different contexts in an mLucid program, elementwise parallelism in the program is the context parallelism, or pointwise expression parallelism, caused by pointwise operations on the values at these contexts.

Every conditional in an mLucid program is considered to be a parallel conditional, as it is a pointwise operation on values of the operands at each context, resulting in context parallelism.

There are several ways to implement replications of data in mLucid. A one-to-many replication, also called *scalar extension*, broadcasts and combines a single element to every element of a collection of data. A scalar extension can be impelmented by an mLucid expression that combines an intension of zero dimensionality with intensions of multiple dimensionalities. For example, if variable A with dimensionality {1,2} in an mLucid program represents a matrix, then the operation `A + 1` adds one to every element of the matrix.

A regular few-to-many replication, also called *spread*, spreads elements of an array with a lower rank to an array with a higher rank and combines the elements of the two arrays. A spread operation can be implemented by an mLucid expression that combines intensions with different dimensionalities. For example, in an mLucid program, let variable A with dimensionality {1,2} represent a matrix in dimensions 1 and 2. Let variable B with dimensionality {1} represent a vector in dimensions 1. The operation `A + B` spreads vector B to every column (or row) of the matrix and adds the corresponding elements of A and B together.

In the following, we define an indexical operator called **upon**, which is built-in in

mLucid. The operator can be used as a replication operator in data parallel programming to replicate, regularly or irregularly, a collection of elements.

The indexical operator **upon(d)(x,t)** takes an intension x and a boolean–valued intension t. t partitions all contexts in the context space along dimension d into segments, such that every two neighboring segments of contexts in dimension d is separated by a context at which t has the value *true (tt)*. The operation maps the value of x at each context to a segment of points specified by t along dimension d and replicates the value to all the points in the segment. For example, let x and t0, t1 and t2 be defined as

```
x = index(1);
t0 = true;
t1 = before(1)(not t1, fby(1)(true, not t1));
t2 = before(1)(not t2, fby(1)(false, not t2)).
```

and they all have dimensionality {1}. Their values in dimension 1 are illustrated as follows:

$$x = (\cdots,\text{-}3,\text{-}2,\text{-}1,\underline{0},1,2,3,\cdots)$$

$$t0 = (\cdots,\text{tt},\text{tt},\text{tt},\underline{\text{tt}},\text{tt},\text{tt},\text{tt},\cdots)$$
$$t1 = (\cdots,\text{ff},\text{tt},\text{ff},\underline{\text{tt}},\text{ff},\text{tt},\text{ff},\cdots)$$
$$t2 = (\cdots,\text{tt},\text{ff},\text{tt},\underline{\text{ff}},\text{tt},\text{ff},\text{tt},\cdots).$$

The results of the operations

$$\text{upon}(1)(x,t0) = (\cdots,\text{-}3,\text{-}2,\text{-}1,\underline{0},1,2,3,\cdots)$$
$$\text{upon}(1)(x,t1) = (\cdots,\text{-}3,\text{-}3,\text{-}2,\text{-}2,\text{-}1,\text{-}1,\ \underline{0},1,1,2,2,3,3,\cdots)$$
$$\text{upon}(1)(x,t2) = (\cdots,\text{-}3,\text{-}3,\text{-}2,\text{-}2,\text{-}1,\text{-}1,0,\ \underline{0},0,1,1,2,2,3,3,\cdots).$$

where the values of the variables at index 0 are underlined.

The operator is defined as follows:

```
upon(d)(x,t) = before(d)(B, fby(d)(origin(d)(x), F))
  where
    F = if t then upon(d)(next(d)(x), next(d)(t))
```

```
            else upon(d)(x, next(d)(t)) fi;
    B = if t then upon(d)(prev(d)(x), prev(d)(t))
            else upon(d)(x, prev(d)(t)) fi;
end.
```

The indexical semantics of *upon* is defined by

$$upon = \lambda d.\lambda(x,t).\lambda p.\ x_{[k/d]p}$$

$$where\ k = \begin{cases} p(d) & p(d) = 0 \\ |\{i \mid 0 \le i \le p(d)\ \wedge t_{[i/d]p}\}| & p(d) > 0 \\ -|\{i \mid p(d) \le i \le 0\ \wedge t_{[i/d]p}\}| & p(d) < 0 \end{cases}$$

The denotational semantics of *upon* is defined by

$$E[upon(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$E[< exp_1 >]\ env\ [k/I_{mLucid}(< constant >)]p$$

$$where\ k = \begin{cases} p(d) & p(d) = 0 \\ |\{i \mid 0 \le i \le p(d)\ \wedge t_{[i/d]p}\}| & p(d) > 0 \\ -|\{i \mid p(d) \le i \le 0\ \wedge t_{[i/d]p}\}| & p(d) < 0 \end{cases}$$

In order to express arbitrary permutations and replications, we also define another indexical operator **at**, which is built-in in mLucid. At a context p, the indexical operator **at(d)(i,x)** (1) takes an integer-valued intension i and an intension x, (2) switches context to an absolute point q along dimension d such that q's coordinate for dimension d is the value of i at p, and (3) returns the value of x at that q. The operator is defined by

```
at(d)(i,x) = if i >= 0 then origin(d)(next(d)(i,x))
                       else origin(d)(prev(d)(abs(i),x)) fi.
```

The indexical semantics of *at* is defined by:

$$at = \lambda d.\lambda(i,x).\lambda p.\ x_{[i_p/d]p}$$

The denotational semantics of *at* is defined by:

$$E[at(< constant >)(< exp_1 >, < exp_2 >)]\ env\ p =$$
$$E[< exp_2 >]\ env\ [E[< exp_1 >]\ env\ p/I_{mLucid}(< constant >)]p$$

For example, let variables A and B represent two one–dimensional arrays in dimension 1, and integer–valued variable V represent a vector also in dimension 1 whose value at each context is an array subscript. The mLucid definition

```
A = at(1)(V, B)
```

defines A to be a permutation of B if the value of V at any two different contexts is different; otherwise it defines irregular replications of B's elements.

There are several ways to express reductions of data in mLucid. A many–to–one reduction, also called *aggregation*, reduces a collection of elements to a scalar value by combining all the elements in the collection. An aggregation function can be implemented in mLucid by such indexical operations that reduces dimensionalities of the operands, such as *origin* and *asa*. Notice that the operation `at(d)(i,x)` is also an aggregation operation if i does not vary in dimension d, i.e. $d \notin Dim(i)$.

In the following, we define a second order function *aggregate* in mLucid. Using the function, various aggregation operators that reduce dimensions of values can be defined. The *aggregate* function has type

$$aggregate: N \rightarrow (ID \times ID \rightarrow ID) \times ID \times ID \rightarrow ID$$

Given a dimension indicator d, a binary pointwise, associative function f, an initial value *init* and an intension x, the function is defined by

```
aggregate(d)(f,init,x) = asa(d)(s, iseod(x))
  where s = fby(d)(init, f(s,x)) end;
```

Notice that the `aggregate` function works only for the nonnegative direction of dimension d. In the definition, the indexical operation *asa(d)(s,iseod(x))* reduces dimension d from the dimensionality of the result. We list the definitions of some first order aggregation operators below, which are defined by the *aggregate* function.

SUM(d)(x) = aggregate(d)(+,0,x)
PRODUCT(d)(x) = aggregate(d)(*,1,x)
MAX(d)(x) = aggregate(d)(max,-∞,x)
MIN(d)(x) = aggregate(d)(min,∞,x)
COUNT(d)(x) = aggregate(d)(+,0, if iseod(x) then x else 1 fi)
ANY(d)(x) = aggregate(d)(or,false,x)
ALL(d)(x) = aggregate(d)(and,true,x).

We will define a more efficient *aggregate* function in a later example using parallel prefix.

The indexical operator **whenever** defined in Chapter 7 can be considered as a selection operator in data parallel programming. Given a dimension indicator d, an intension x and a boolean–valued intension t, the operation **whenever(d)(x,t)** selects x's value at the contexts at which t has true value, and moves the selected values along dimension d towards the origin of d. The operator works only for the nonnegative direction of the dimension.

We can also define a second order function **collapse** to implement many–to–few reductions along a dimension. The *collapse* function has type

$$collapse: N \rightarrow (ID \times ID \rightarrow ID) \times ID \times ID \rightarrow ID$$

Given a dimension indicator d, a binary pointwise, associative function f, an initial value *init* and two intensions x, y, the function is defined as

```
collapse(d)(f,init,x,y) = fby(d)(first, collapse(d)(f,init,restx,resty))
  where
    first = aggregate(d)(f,init, whenever(d)(y, isx));
    restx = whenever(d)(x, not isx);
    resty = whenever(d)(y, not isx);
    isx = x eq origin(d)(x);
  end
```

Notice that the `collapse` function as defined works only for the nonnegative direction of dimension d. For each of the distinct values v of x at contexts along dimension d, the

function *collapse* reduces the value of y at the contexts at which x's value is equal to v. For example, let

$$x = (a, b, a, c, b, a)$$
$$y = (1, 2, 3, 4, 5, 6)$$

the results of the following operations are

$$\text{collapse(d)(add,0,x,y)} = (10, 7, 4)$$
$$\text{collapse(d)(count,0,x,y)} = (3, 2, 1)$$

where *count* is defined as $\lambda(x, y).\ x + 1$.

Indexical operators **next** and **prev** can be considered as regular permutation operators. They shift the values of their operands along a dimension. For an intension x representing a finite object surrounded by *eod*'s at the boundaries of each dimension, we can define two regular shift–wrap operators **nextw** and **prevw** that shift x's value toward the negative and positive directions along a dimension, respectively. The values that reach a boundary of the dimension are taken from the opposite boundary of the dimension.

```
nextw(d)(x) = if iseod(next(d)(x)) then basa(d)(x, iseod(prev(d)(x)))
                                   else next(d)(x) fi;
prevw(d)(x) = if iseod(prev(d)(x)) then fasa(d)(x, iseod(next(d)(x)))
                                   else prev(d)(x) fi;
```

where **fasa(d)(x,t)** (for "forward as soon as") and **basa(d)(x,t)** (for "backward as soon as") are similar to *asa(d)(x,t)*, but they search the desired values from the current context forward and backward. along dimension d until the the first true value of t is found, instead of always starting from the origin of the dimension. They are defined as follows:

```
basa(d)(x,t) = if t then x else basa(prev(d)(x), prev(d)(t)) fi;
fasa(d)(x,t) = if t then x else basa(next(d)(x), next(d)(t)) fi;
```

8.3 Examples

In the first example, we develop an mLucid program expressing the *parallel prefix* data parallel algorithm [HS86a]. Given an array A with 2^n elements, the algorithm computes

all prefixes of the elements in n iteration steps. Consider the prefixes are all partial sums of array A. The result of the prefix computation is an array B of size 2^n. Each element B[i] ($0 \leq i \leq 2^n - 1$) of B holds the sum of the elements A[k] of A for $0 \leq k \leq i$. In particular, the element B[$2^n - 1$] of B holds the sum of the entire array. The computation is distributed to 2^n processors. The processors are indexed from 0 through $2^n - 1$ and connected in a general communication network. At each iteration step, a processor can communicate with any one of other processors, but no two processors are allowed to communicate with the same processor. Initially, the elements B[i] are allocated to the local memory of processor i and are assigned A[i]. At iteration step k ($1 \leq k \leq n$), each processor i communicates with processor j and adds the value of B[j] in the local memory of the processor j to its own B[i] where $i = j + 2^{k-1}$ for $j \geq 0$ and $i \leq 2^n - 1$. After n steps, all the processors i hold the desired partial sums in B[i]. Figure 8.1 illustrates the computation with 16 processors.

The algorithm can be formally defined by a recurrence equation as follows:

$$B_i^k = B_i^{k-1} + B_{i-2^{k-1}}^{k-1} \quad (2^{k-1} \leq i \leq 2^n - 1,\ 1 \leq k \leq n)$$

where $B_i^0 = A[i]$.

The following mLucid program expresses the partial sum algorithm, where, as in previous examples, we use the suffix "_time" to represent the dimension indicator $d = 0$ for the time dimension.

```
dimensionality A:{1}; n:{};

at_time(B, n)
where
  B = fby_time(A, if eligible then B + prev(1)(offset, B)) else B fi;
  offset = fby_time(1, 2**offset);
  eligible = index(1) >= offset;
end
```

Program 8.1a: A data parallel program for partial sum

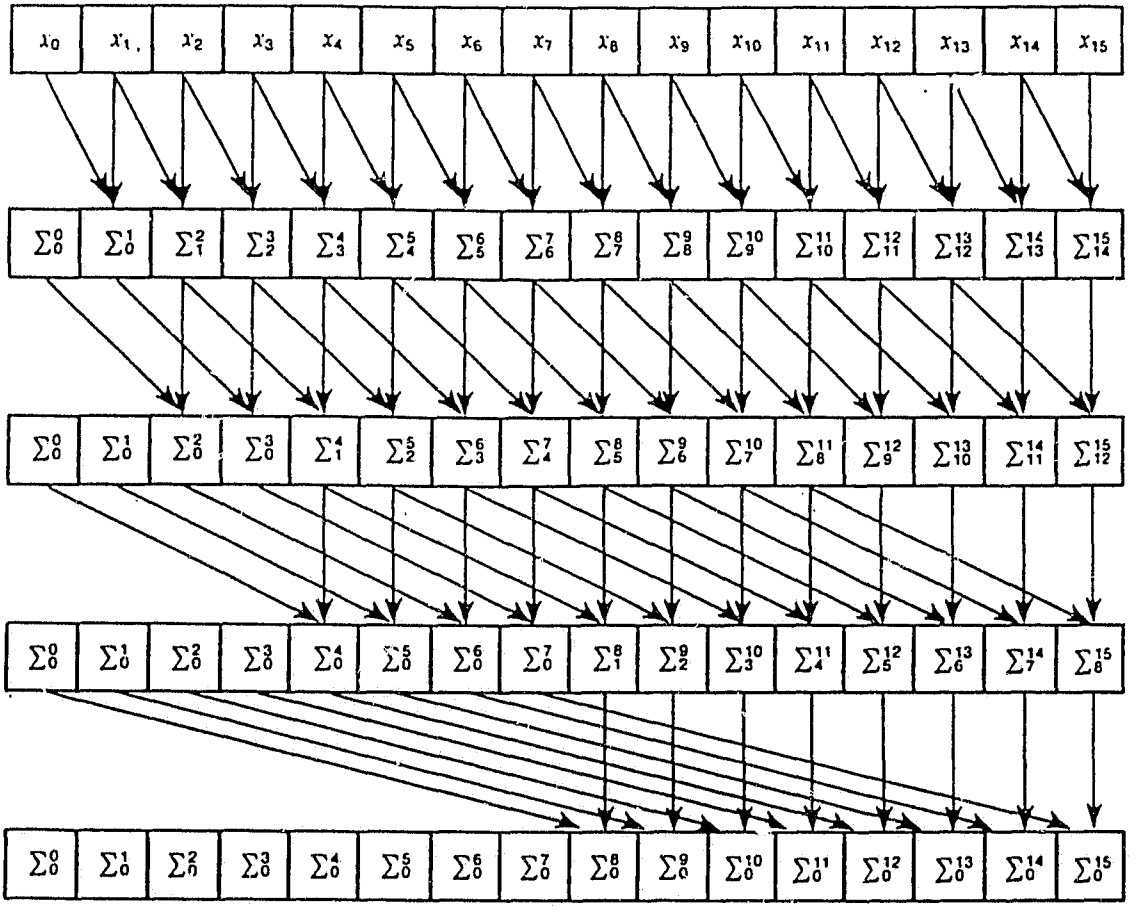

Figure 8.1: Computing partial sums of an array of 16 elements

In the program, the data parallelism of the algorithm is expressed by the expression *B + prev(1)(offset, B)* which has elementwise parallelism. The synchronization is expressed by the temporal operator *fby_time* in the definitions for *B* and *offset*. *offset* is a temporal sequence with dimensionality {0}; at each iteration step represented by a time point in the time dimension, its value is the distance between each pair of processors that communicate with each other at that step. The broadcast of the communication route, specified by *offset*, at each step is expressed by the spread operation *prev(1)(offset,B)*, as *B's* dimensionality is a super set of *offset*'s. Also the indexical operation *prev(1)(offset,B)* expresses the communication from an eligible processor to another processor with distance *offset*, at each step. Notice that for input array A with higher dimensionality, say a matrix, the same program expresses the partial sums of each row or column of the matrix.

The program can be improved further at two places. For an input array whose size is unknown and whose boundaries are surrounded by *eod*'s, we can rewrite the program as follows.

```
dimensionality A:{1};

asa_time(B, iseod(next_time(prev(1)(offset,A))))
where
  B = fby_time(A, B + prev(1)(offset, B));
  offset = fby_time(1, 2**offset);
end
```

Program 8.1b: An index-independent data parallel program for partial sum

In the program, the computation at each of the processors terminates as soon as it tries to communicate with a non-existing processor, that is, switching context to a point at which the fetched value is *eod*. It is expressed by *iseod(next_time(prev(1)(offset,A)))*. Notice that the result of the program is irrelevant to how A is laid in dimension 1.

We can also generalize the partial sum program to a second order function *parfix* to express a general parallel prefix algorithm. Given a dimension indicator d, a binary

pointwise, associative function f, and an intension x, the function is defined by

```
parfix(d)(f,x) = asa_time(s, iseod(next_time(prev(d)(offset,x))))
where
  s = fby_time(x, f(s, prev(d)(offset, s)));
  offset = fby_time(1, 2**offset);
end
```

Using the *parfix* function, we can redefine a more efficient *aggregate* function as follows.

```
aggregate(d)(f,init,x) =
  f(init, asa(d)(parfix(d)(f,x), iseod(next(d)(x)))).
```

In the second example, we will develop a program for finding the shortest path from vertex A to vertex B of a given graph. The following is a data parallel algorithm for solving the problem [Hil85][Sab88]:

1. Label vertex A with 0.
2. Label all vertices except A with plus infinity.
3. Re–label every vertex, except A, with 1 plus the minimum of its neighbors' labels. Repeat this step until the labels stabilize.
4. Terminate. The label of B is the answer.

To implement the algorithm in an mLucid program, given a graph, we use three input variables *vertex*, *from_node* and *to_node* to represent all the vertices, the vertices that are sources of edges, and the vertices that are sinks of edges in the graph, respectively. *vertex* as a vector has dimensionality {1}. *from_node* and *to_node* as a pair of vectors have dimensionality {2}. The pair of the values of *from_node* and *to_node* at a context represents an edge in the graph. For example, Figure 8.2 shows a graph and its representation.

There are also two input variables *source* and *sink* representing the source and sink vertices between which we want to find the shortest path.

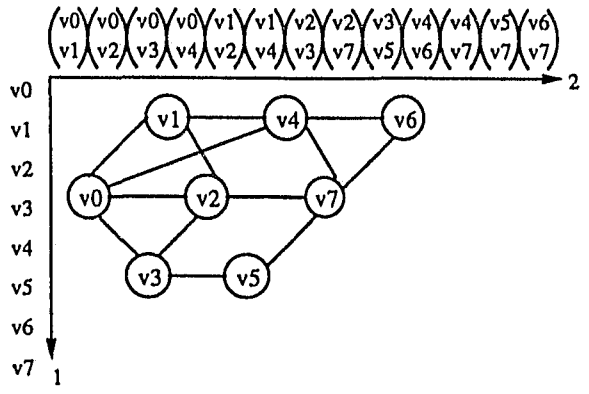

Figure 8.2: A graph represented by variable values

The following is the mLucid program implementing the algorithm.

```
dimensionality vertex:{1}; from_node, to_node:{2}; source, sink:{};

asa(1)(asa_time(label, stable), vertex eq sink)
where
  label = if vertex eq source then 0 else fby_time(INFINITY, newlabel) fi;
  newlabel = 1 + MIN(2)(neighbor);
  neighbor = whenever(2)(edgelabel, from_node eq vertex);
  edgelabel = asa(1)(label, to_node eq vertex);
  stable = ALL(1)(next_time(label) eq label).
end
```

Program 8.2: A data parallel program for shortest paths

In the program, we use variable *label*, corresponding to *vertex*, to hold the labels of the vertices at each step. The value of *label* is always 0 if it corresponds to the source vertex; otherwise it is INFINITY at initial time, and then at each time step it is the value of *newlabel* computed at the previous time step. At a time step, the value of *newlabel* corresponding to a vertex v is 1 plus the minimum value of the vector *neighbor* in dimension 2; *neighbor* represents the labels of v's neighboring vertices at the time step.

The minimum is reduced by the aggregation operator MIN. Given the vector *edgelabel*, in dimension 2, consisting of labels each of which corresponds to the *to_node* of each edge, for each vertex v the selection operator *whenever* selects the labels of v's neighbors from *edgelabel* by checking the *from_node* of each edge. Since *edgelabel* is constant in dimension 1, it is automatically spread to dimension 1 during the execution of *whenever*, so that the selections for all vertices can be done in parallel. The *edgelabel* vector is computed by executing the search operator *asa* for the *to_node* of each edge. Since *label* is constant in dimension 2, during the execution, it is automatically spread to dimension 2, so that the search can be done in parallel. Finally, the stability condition for terminating the execution of the program is defined by the aggregation operator ALL that checks if there is no change on the labels of vertices at two consecutive time steps.

Chapter 9

CONCLUSION AND FUTURE WORK

9.1 Summary

This dissertation has studied parallel programming in the indexical programming language mLucid.

mLucid as a multidimensional extension of the programming language Lucid enriches its base functional language ISWIM by incorporating functional semantics with indexical semantics. The context space in which the indexical semantics of the language is defined consists of points in an arbitrary n-dimensional integer space. The meanings of objects in mLucid implicitly vary with contexts. The original ISWIM operators are extended pointwise; they apply to the values of objects at every context in the context space, without consulting with values at other contexts. mLucid provides five primitive indexical operators *origin*, *next*, *prev*, *fby* and *before* to switch context from one to another, so that values at different contexts can be combined. The formal semantics of mLucid is defined by an extended fixed point semantics, in which the meanings of objects are intensions instead of single data values.

An important notion in the indexical semantics of mLucid is dimensionality. The dimensionality of an intension in the indexical semantics of mLucid is the set of dimensions

that determines the range of the context space in which the intension varies. The definition of dimensionality relies on the notion of continuous intensions. The value of a continuous intension at a context depends on a finite set of dimension coordinates of the context. A continuous intension has the same value at the contexts whose coordinates for the dimensions in the dimensionality of the intension are same. An approximation to the dimensionalities of objects in mLucid programs is derived at compile–time by an abstract interpretation of the programs, which maps mLucid expressions to dimensionalities.

Context parallelism and indexical communication in an mLucid program is formally defined by a semantics–based dependency graph of the program in which the dependencies among variable values at different contexts are defined. The definition of variable–value dependence relies on the notion of functional sequentiality of mLucid functions. Each application of a functionally sequential function at a context depends on the values of the operands at a unique set of contexts. Based on the dependence graph, context parallelism is defined among unconnected contexts in the graph; indexical communication is defined by indexical operations.

Context parallelism and indexical communication in mLucid programs can be used to specify parallel computations and communications of systolic arrays. The semantics of a systolic array can be defined by the semantics of a SysLucid program. The context space of the program consists of the time dimension and the spatial subspace. Each PE in the array corresponds to a space point in the spatial subspace; the functionality of the PE is the values of variables at the corresponding space point at all time points. The semantics of the array is the values of variables at all the space points corresponding to the PEs at all time. Indexical communication specifies the connectivity of the PEs in the array and synchronization among computations of the neighboring PEs. Context parallelism represents both the pipeline parallelism and spatial parallelism in the array. The pipeline parallelism is the parallel evaluations of the values of variables at two neighboring space points at different time. The spatial parallelism is the parallel evaluations at unconnected space points.

A generalization of the systolic programming is the multidimensional dataflow pro-

gramming, which allows multiple data streams flow into and out from a multidimensional dataflow network. In multidimensional dataflow programming, the definitions of variables at a space point specify a dataflow subnetwork; temporal operations in the definitions specify the stream operations in the subnetwork; spatial operations in the definitions specify communication between the subnetwork and others at different space points.

mLucid also supports data parallel programming with massive parallelism. In mLucid programs, pointwise operations have elementwise parallelism, and indexical operations implement reductions, replications and permutations of collections of elements.

9.2 Conclusions

- Indexical programming is distinct from other parallel programming models owing to its indexical nature. It introduces context parallelism and indexical communication based on indexical semantics.
- Indexical programming enriches the expressive power of conventional programming models especially with respect to parallelism and communications.
- Indexical programming can be used for specifying decompositions of parallel tasks, mappings from tasks to processors, and synchronization and communications among tasks, in terms of context parallelism and indexical communication.
- Indexical programming can be used by application programmers to write problem-solving programs with large scale inherent parallelism, namely context parallelism, and regular communication patterns defined by explicit indexical operations.
- Indexical programming can be used as a formal model to study various parallel programming paradigms by giving context parallelism and indexical communication different interpretations.

9.3 Limitations

The expressive power of indexical programming in mLucid is constrained by both the functional semantics of the base language and the indexical semantics. In the following, we discuss the limitations incurred by the language's indexical semantics.

The shape of the context space of an mLucid program is fixed; it is an n–dimensional integer space. This constraint makes mLucid suitable for expressing parallel algorithms whose communication networks can be easily embedded into the multidimensional grid–like context space. For others, complex index conversions have to be defined in the program using provided indexical operators, causing the structure of the program to become obscure.

There are no user–defined dimensions in mLucid. All dimensions in the context space are fixed and represented by natural numbers. Once a program is written, the context space is static and cannot be changed during the execution of the program. A dynamic context space is especially useful when extra dimensions are needed only for some local computations. This limitation results in that the language cannot directly express nested parallelism. Nested parallelism occurs among parallel computations in the nested context space, that is, each point in the outer context space represents another inner space. To express nested parallelism in mLucid, the inner context space has to be flattened by the programmer using extra dimensions.

Since in mLucid we treat the value of an object as a whole (intension) in the context space, the language is suited to express parallel algorithms with homogeneous computations and regular communications; otherwise programs will look tedious because of many conditionals on indices.

mLucid as a multidimensional dataflow language is very good at expressing infinite objects like data streams. However, when it deals with finite objects, *eod* has to be used explicitly by the programmer to set boundaries. The advantage of this method is that the boundaries can be set dynamically, but it also increases difficulties for compilers to detect the boundaries at compile–time. In this sense, lazy or demand driven evaluation of mLucid programs seems essential.

9.4 Contributions

The research described in this dissertation has the following contributions.

- It defines the programming language mLucid and describes the formal indexical and denotational semantics of mLucid.
- It defines the notion of continuity and dimensionality of mLucid semantic objects, and studies their relationship.
- It describes an abstract interpretation approach to deriving approximations to dimensionalities from mLucid programs.
- It defines the notion of semantics–based functional sequentiality, and proves the functional sequentiality of mLucid programs.
- It defines the semantics–based dependence relation of variable values in mLucid programs and hence defines the notion of context parallelism and index communication based on that relation.
- It describes a methodology for specifying systolic arrays.
- It describes a multidimensional dataflow programming paradigm.
- It describes data parallel programming in mLucid.

9.5 Future Work

The author is currently investigating two other parallel programming paradigms in mLucid: (1) *parallel functional programming* and (2) *distributed programming*, as well as parallel implementation strategies of mLucid.

9.5.1 Parallel Functional Programming

mLucid as an indexical functional language has two semantic components: indexical semantics and functional semantics. In terms of parallel programming, we have so far

emphasized the relationship between the indexical semantics and the context parallelism and indexical communication in mLucid programs. By contrast, expression parallelism in the programs introduced by the language's functional semantics is secondary.

In parallel functional programming, we exchange the roles of the two semantic components of mLucid. We consider that mLucid is first a functional language in which functional programs with implicit expression parallelism are written. Then, using mLucid's enriched indexical semantics, we investigate an annotation–based parallel functional programming methodology.

There is a subset of mLucid programs that are pure functional programs, that is, the meanings of the programs do not depend on the language's indexical semantics. A pure functional mLucid program is in mLucid(0), in which there are no indexical operators and all the input variables have zero dimensionality. Consider adding some dimensionality–reducing/preserving indexical operators, such as *origin*, *next* and *prev*, to some expressions of a pure functional mLucid program. By the dimensionality analysis, the modified program is still in mLucid(0) and has the same functional semantics as the original program. We can treat these indexical operators as *meaning–preserving* annotations, which we term *indexical annotations*.

From the problem–solving point of view, the above indexical annotations are meaningless, though they have well–defined mathematical semantics. However, we can interpret these indexical annotations with operational meanings, so that the programmer can use them to specify explicitly some control information to help parallel compilers produce more efficient code. Especially, we are interested in using indexical annotations to specify communication mechanisms and process–to–processor mappings in pure functional programs.

To do this, we first extend the singleton context space of an mLucid(0) program with indexical annotations to the multidimensional context space consisting of the dimension indicators of the indexical annotations in the program. Given a set of processors, then let each of the processors correspond to a context in the extended context space of the annotated mLucid(0) program. The indexical annotations in the program explicitly express

the process–to–processor mapping and the communication between processes at different processors without altering the semantics of the corresponding pure functional program.

For example, let the extended context space is two-dimensional. Let two expressions e_1 and e_2 have dimensionality {}. At a context (i,j) in the extended context space corresponding to processor p_{ij}, where i and j are coordinates for dimensions 1 and 2 respectively, we can interpret the annotated function application in the program

$$\textbf{next(2)(f(prev(1)}(e_1)\textbf{, next(1)}\ e_2\textbf{))}$$

as follows. The evaluations for the arguments e_1 and e_2 are performed at processors $p_{i-1,j+1}$ and $p_{i+1,j+1}$, respectively, and the application is performed at processor $p_{i,j+1}$, then the result of the application will be sent back to p_{ij}.

9.5.2 Distributed Programming

mLucid in general supports distributed programming. We map each processor in a distributed system to a context. The processor is identified by the index of the context it corresponds to. Processes, or evaluations of values of expressions, at a context are mapped to the corresponding processor. In the case that computations at the processors are synchronized by iterations, each processor may correspond to a class of contexts or a space point, with respect to the partition that divides the context space into the temporal dimension representing iteration steps and the spatial subspace for locating the processors.

The point–to–point message passing and broadcast communications among processes at different processors can be both expressed in mLucid programs. Indexical communication through indexical operators are point–to–point message passing. In this case, an indexical operation in a process at a processor explicitly sends a request to another processor to invoke a process at that processor. Broadcast and multicast can be expressed implicitly by combining expressions with different dimensionalities. An expression with 0–dimensionality has constant value at every processor. The value of the expression can be evaluated at a processor and then be broadcast to all other processors. It can also be evaluated repeatedly at all processors even though it has the same value everywhere. The value of an expression with a fewer dimensionality than the context space in which

the processors are located are implicitly multicast to the processors along the dimensions in which the value is constant. Another point view of the dimension spread is to treat a variable with a fewer dimensionality as one logically shared by all processors along the dimensions in which the variable is constant. The value of the variable is evaluated and stored at one of them and retrieved by others.

The grain size of a processor depends on how often processes executed at a processor communicate with processes at other processors. It can be controlled by the programmer. We call a process consisting of only pointwise operations *closed process*, because it does not communicate with other processes outside of the processor when it is executed at a processor. The larger the closed processes at a processor, the larger the grain size of the processor.

Based on the above observation on the relationship between indexical programming and distributed programming, the following is a simple strategy for writing distributed programs in mLucid.

1. Choose a context space that is suited to embed the communication network of the underlying distributed system.
2. Map processors in the system to contexts in the context space.
3. Based on the layout of the processors, define indexical operators that specify communications among neighboring processors in the network.
4. Allocate processes to each processor by defining the values of variables at the context corresponding to the processor.
5. Use the sizes of closed processes to control grain sizes of the processors.

9.5.3 Parallel Implementation Strategies

The space–time mapping methodology[Che86a] is a possible strategy for implementing mLucid programs on general purpose multiprocessors, provided that the context dependence graphs of the programs are acyclic. The methodology partitions computations at

each context in an mLucid program is a task. Each task t is assigned an order number called *time* in accordance with the partial order defined by the program's context dependence graph, so that all other tasks that t depends on have smaller order numbers than t's. A set of tasks are mapped onto a processor if their assigned times are pairwise different and they are local to each other according to the indexical communication graph of the program; executions of the tasks on the processor are sequentially scheduled according to their assigned times. By this scheduling strategy, maximum parallelism and efficient communications among the processors are expected.

A more general implementation strategy is to simulate the distributed eduction abstract machine [AFH85] [HFA85] on general purpose parallel computers. The eduction machine is based on the tagged demand driven computation model. The distributed eduction machine consists of a set of processors, each of which is indexed by a point in an n–dimensional space and connected to all other processors in the set. Given an mLucid program, each processor stores the definitions and the evaluated values of variables at the corresponding context. A computation in a processor is fired by a demand for the value of some variable at the corresponding context from another processor or from the outside world. During the computation, when the value of a variable at another context is needed, which is specified by an indexical operation, the processor sends a demand to the corresponding processor and suspends the computation until the needed value is available. During the waiting time, the processor can carry on with computations for other demands. The implementation on a general purpose multiprocessor maps a subset of the virtual eductive processors onto a physical processor and simulates computations performed by each virtual processor.

This dissertation does not address the implementation issues, because so far the strategies have not been implemented. Prototype parallel implementations of mLucid based on these and other strategies are the first consideration of extending this research.

9.5.4 Others

In the dissertation, we have studied the relations between indexical programming and several parallel computation models: systolic computation, dataflow computation, data parallel computation. The next conceivable step of the research is to study a theory on these and other parallel computation models based on indexical logic. Using the theory we can formally define and reason about these different computation models.

The expressive power of mLucid can be increased by allowing user-defined context spaces, so that programmers can have a degree of freedom to choose a suitable structure of context spaces for their problems. Syntax and semantics of the extension will be investigated and the impact of the extension to the language's indexical semantics will also be studied.

Currently, indexical languages are mainly based on functional and logic programming languages, so that they can have declarative semantics. However, an investigation on enriching imperative languages with indexical semantics also seems needed. Potential benefits from the extension includes increasing the expressive power and abstraction levels of imperative languages, and giving formal meanings to parallel and communication constructs of imperative parallel languages.

Bibliography

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug. 1986.

[AF89] E.A. Ashcroft and A.A. Faustini. Adding intensionality to functional programs. In *The 1989 International Symposium on Lucid and Intensional Programming*, 1989.

[AFH85] E.A. Ashcroft, A.A. Faustini, and B. Huey. Eduction - a model of parallel computation and the programming language lucid. In *the Phoenix Conference on Computers and Communications*, pages 9–15. IEEE, 1985.

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[AJ86] E.A. Ashcroft and R Jagannathan. Operator nets. In J.V. Woods, editor, *Fifth Generation Computer Architecture*, pages 177–202. North-Holland, 1986.

[AJ89] E.A. Ashcroft and R. Jagannathan. An intensional parallel processing language for application programming. In *The 1989 International Symposium on Lucid and Intensional Programming*, 1989.

[AK81] Arvind and V. Kathail. A multiple processor dataflow machine that supports generalized procedures. In *Proc. 8th Ann. Symp. Arch.*, pages 291–302, May 1981.

[Ash85] E.A. Ashcroft. Ferds – massive parallelism in lucid. In *the Phoenix Conference on Computers and Communications*, pages 16–21. IEEE, 1985.

[Ash90] E.A. Ashcroft. Parallel programming in lucid. In *The 1990 International Symposium on Lucid and Intensional Programming*, 1990.

[Ash91] E.A. Ashcroft. Lucid is secon–order: Explaining lucid and intensionality to functional programmers. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 1–8, April 1991.

[AW76a] E.A. Ashcroft and W.W. Wadge. Lucid – a formall system for writing and proving programs. *SIAM J. Comp.*, 5(3):336–354, Sept. 1976.

[AW76b] E.A. Ashcroft and W.W. Wadge. Lucid, a nonprocedural language with iteration. *Comm. ACM*, 20(7):519–526, July 1976.

[AW79] E.A. Ashcroft and W.W. Wadge. Structured lucid. Technical Report CS–79–21, Department of Computer Science, University of Waterloo, Ontario, Canada, 1979.

[Che86a] M.C. Chen. A design methodology for synthesizeing parallel algorithm and architectures. *Journal of Parallel and Distributed Computing*, pages 461–491, 1986.

[Che86b] M.C. Chen. A parallel language and its compilation to multiprocessor machines or vlsi. In *Proc. 13th ACM POPL*, pages 131–139, 1986.

[Che86c] M.C. Chen. Transformations of parallel programs in crystal. *Information Processing*, pages 455–462, 1986.

[Den80] J.B. Dennis. Data flow supercomputer. *Computer*, 13(11):48–56, Nov. 1980.

[Du91] W. Du. Two indexical parallel programming techniques. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 9–18, April 1991.

[DW88] W. Du and W.W. Wadge. An intensional language as the basis of a 3-d spreadsheet. In *Proc. International Conference of Computer Languages*, pages 2–9, Oct. 1988.

[DW90] W. Du and W.W. Wadge. A 3d spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, May 1990.

[Esc91] C. Escalante. Dimensionality analysis for variables in a lucid–like language. Project Report, Department of Computer Science, University of Victoria, 1991.

[FH88] A.J. Field and P.G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[FJ91] A.A. Faustini and R. Jagannathan. Indexical lucid. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 19–34, April 1991.

[FW86a] A.A. Faustini and W.W. Wadge. An eductive interpreter for plucid. Technical Report TR–004–86, Department of Computer Science, Arizona State University, 1986.

[FW86b] A.A. Faustini and W.W. Wadge. Intensional programming. Technical Report DCS–55–IR, Department of Computer Science, University of Victoria, British Columbia, Canada, 1986.

[Gol88] B.F. Goldberg. Multiprocessor execution of functional programs. Technical Report YALEU/DCS/RR–618, Department of Computer Science, Yale University, 1988.

[HFA85] B. Huey, A.A. Faustini, and E.A. Ashcroft. An eduction engine for lucid – a highly parallel computer architecture. In *the Phoenix Conference on Computers and Communications*, pages 156–160. IEEE, 1985.

[Hil85] W.D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, Massachusetts, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HS86a] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170-1183, December 1986.

[HS86b] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *ACM Symp. Princ. Programming Languages*, pages 243–254, 1986.

[Hud86a] P. Hudak. Denotational semantics of a parafunctional programming language. *International Journal of Parallel Programming*, pages 103–125, April 1986.

[Hud86b] P. Hudak. Parafunctional programming. *IEEE Computer*, pages 60–71, Aug. 1986.

[Hud88] P. Hudak. Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, pages 54–61, Jan. 1988.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, pages 359–411, Sept. 1989.

[JF91] R. Jagannathan and A.A. Faustini. Tournament computation in glu. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 98–109, April 1991.

[KL78] H.T. Kung and C.E. Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Symposium*, pages 256–282, 1978.

[Kle81] S.C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3:52-67, 1981.

[Kun82] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, Jan. 1982.

[Kun87] S.Y. Kung. *VLSI Array Procesors.* Prentice Hall, 1987.

[Lan66] P.J. Landin. The next 700 programming languages. *Comm. ACM*, 9:157–166, 1966.

[Lei83] C.E. Leiserson. *Area-Efficient VLSI Computation.* Addison Wesley, 1983.

[MR84] R.G. Melhem and W.C. Rheinboldt. A mathematical model for verification of systolic networks. *SIAM J. Comput.*, 13(3):541–565, Aug. 1984.

[PS88] S. Purushothaman and P.A. Subrahmanyam. Reasoning about systolic algorithms. *Journal of Parallel and Distributed Computing*, 5:669–699, 1988.

[Ree84] A.P. Reeves. Parallel pascal: An extended pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, 1984.

[Sab88] G.W. Sabot. *The Paralation Model.* The MIT Press, Cambridge, Massachusetts, 1988.

[Sha86] E. Shapiro. Concurrent prolog: A progress report. *IEEE Computer*, 19(8):44–58, Aug. 1986.

[Sha87] E. Shapiro. Systolic programming: A paradigm of parallel processing. In Ehud Shapiro, editor, *Concurrent Prolog*, volume 1, pages 207–242. The MIT Press, 1987.

[Ste88] G.L. Steele. Languages for massively parallel computers. In *Proceedings of 1988 IEEE Second Symposium On The Frontiers of Massively Parallel Computations*, pages 3–13, Oct. 1988.

[Thi86] Thinking Machine Corporation. Introduction to date level parallelism. Technical Report TR–86–14, April 1986.

[Ued85] K. Ueda. Guarded horn clauses. Technical Report TR–103, ICOT, 1985.

[vB88] J. van Benthem. *A Manual of Intensional Logic.* Center for the Study of Language and Information, 1988.

[WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[Wad91a] W.W. Wadge. Personal Communication, 1991.

[Wad91b] W.W. Wadge. High order lucid. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 62–69, April 1991.

[Yag84] A.A. Yaghi. *The Intensional Implementation Technique for Functional Languages.* PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.