From \textbf{Alpha} to Imperative Code: 
A Transformational Compiler for an 
Array Based Functional Language

Doran K. Wilde$^1$

Ph.D. Dissertation 
Oregon State University

July 1995

Abstract

Practical parallel programming demands that the details of distributing data to processors and inter-
processor communication be managed by the compiler. These tasks quickly become too difficult for 
a programmer to do by hand for all but the simplest parallel programs. Yet, many parallel languages 
still require the programmer to manage much of the the parallelism.

I discuss the synthesis of parallel imperative code from algorithms written in a functional language 
called Alpha. Alpha is based on systems of affine recurrence equations and was designed to specify 
algorithms for regular array architectures. Being a functional language, Alpha implicitly supports the 
expression of both concurrency and communication. Thus, the programmer is freed from having 
to explicitly manage the parallelism.

Using the information derived from static analysis, Alpha can be transformed into a form suitable 
for generating imperative parallel code through a series of provably correct program transformations. 
The kinds of analysis needed to generate efficient code for array-based functional programs are a 
generalization of dependency analysis, usage analysis, and scheduling techniques used in systolic 
array synthesis.

$^1$email: wilde@ee.byu.edu

This work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632 and by NSF Grant No. MIP-890852.
# Contents

1 Introduction
- 1.1 Synthesizing compiler approach ........................................... 2
- 1.2 Approach to be used to solve the problem .................................. 3
- 1.3 Tools Used .............................................................................. 4
  - 1.3.1 The Alpha Environment .................................................. 4
  - 1.3.2 Mathematica ......................................................................... 4
  - 1.3.3 The Polyhedral Library ...................................................... 4
- 1.4 Related Work ........................................................................... 5
  - 1.4.1 Crystal ................................................................................. 5
  - 1.4.2 Systolic Array Synthesis ..................................................... 5
  - 1.4.3 Static Analysis of Imperative Programs ................................. 6
  - 1.4.4 Scheduling ........................................................................... 6
- 1.5 Goals ......................................................................................... 7
- 1.6 Contributions ........................................................................... 7
- 1.7 Impact ....................................................................................... 8
- 1.8 Overview of thesis ................................................................. 8

2 The Alpha language
- 2.1 Systems of Affine Recurrence Equations ............................... 10
- 2.2 Alpha : An applicative equational language .......................... 11
- 2.3 Alpha System Declarations .................................................. 12
- 2.4 Variable Declarations .......................................................... 13
  - 2.4.1 Type Checking ..................................................................... 13
- 2.5 Domain variables .................................................................... 13
- 2.6 Denotational and type semantics ......................................... 15
- 2.7 Equations and Expressions ................................................... 15
  - 2.7.1 Variables ............................................................................. 16
  - 2.7.2 Constants ........................................................................... 17
  - 2.7.3 Pointwise Operators .......................................................... 17
- 2.8 Domain operators ..................................................................... 17
  - 2.8.1 Case Operator ....................................................................... 18
  - 2.8.2 Restrict Operator ................................................................. 18
  - 2.8.3 Dependence Operator .......................................................... 19
  - 2.8.4 Reduction Operator ............................................................. 19
- 2.9 An Extended Semantics ......................................................... 20
- 2.10 Conclusion ............................................................................... 20
3 Loop Nest Synthesis using the Polyhedral Library 22
  3.1 Introduction .............................................. 22
  3.2 The Polyhedron Scanning Problem .......................... 23
    3.2.1 Introduction to Parameterized Polyhedra .............. 23
    3.2.2 The Polyhedron Scanning Problem .................... 24
  3.3 Example ................................................... 24
  3.4 Description of new method ................................ 25
    3.4.1 Add Rays to a Domain .............................. 26
    3.4.2 Domain Simplify .................................... 26
    3.4.3 Loop Separation ................................... 27
  3.5 Conclusion ............................................... 27

4 Allocation of Memory for Polyhedra 28
  4.1 Reducing the memory allocation ............................ 30
  4.2 Open Problem ............................................. 30
  4.3 Conclusion .............................................. 31

5 The Naive Execution of ALPHA 32
  5.1 Introduction ............................................. 32
  5.2 The ALPHA Translator .................................... 32
  5.3 Improvements over naive code generation .................. 33
  5.4 Conclusion ............................................... 34

6 Analysis of ALPHA 36
  6.1 Dependency Analysis ..................................... 37
  6.2 Usage Analysis .......................................... 38
    6.2.1 Construction of the Usage Table ................... 38
  6.3 Schedule Analysis ....................................... 40
  6.4 Alignment and Virtual Allocation ......................... 41
  6.5 Communication Analysis, Partitioning and Processor Allocation .................. 42
  6.6 Conclusion ............................................... 44

7 ALPHA Transformations 45
  7.1 Related Work ............................................ 45
  7.2 Transformation using the ALPHA Environment ............. 45
  7.3 Basic Transformations ................................... 46
    7.3.1 Proof of Basic Transformations ..................... 46
    7.3.2 Case Rules ......................................... 46
    7.3.3 Let Rules .......................................... 47
    7.3.4 Restriction Rules ................................. 47
    7.3.5 Dependence Function Rules ......................... 48
    7.3.6 Reduction Rules ................................. 48
  7.4 Change of Basis ......................................... 49
  7.5 Substitution ............................................. 49
  7.6 Add Local Variable ...................................... 49
  7.7 Serialization ............................................ 50
  7.8 Normalization .......................................... 50
  7.9 Other Normalizations ................................... 50
  7.10 Conclusion .............................................. 50

8 Conclusion 

9 Appendix
Chapter 1

Introduction

Parallel architectures are intended to obtain high performance at reasonable cost. However, the complexity of managing the communication and synchronization between parallel processes has been a major obstacle in programming these architectures and exploiting their potential. This complexity has also made it much more difficult to reason about parallel programs than about sequential programs. How to effectively and efficiently program parallel machines has thus become a major challenge. In the paragraphs below, I discuss several approaches that are being taken.

Parallel extensions to imperative languages The first parallel programs were written in standard imperative languages (such as C and Fortran) extended with architecture specific communication primitives. Communication was explicitly coded in the program and the management of parallelism was done entirely by the programmer without any aid from the compiler. Experience has proven that writing this kind of parallel program by hand is very error prone, particularly involving communication or synchronization, and it is equally difficult to debug. Furthermore, this kind of program is targeted for a specific parallel computer system and lacks portability.

Parallel languages Parallel languages allow a programmer to specify programs for an abstract parallel architecture and help the programmer to manage the parallelism. Examples of these kinds of languages are data parallel languages such as High Performance Fortran (HPF) [38] and C* [32]. Special effort is made in these languages to facilitate communication by making it implicit, and thus eliminating the source of many programming errors. A parallel language compiler compiles a program written in a parallel language and specializes it for a specific target architecture. By coding for an abstract parallel architecture, a single program may be compiled for any number of different targets without the programmer having to know details of the target machines. Sometimes parallel compiler directives are used to guide compilation. Even though the communication is implicit, the parallelism is still explicit. The programmer is still responsible for finding and exploiting parallelism and the parallelization of an algorithm is still performed by hand. Programming in a parallel language is better than programming in an extended imperative language for a specific architecture, but is still error prone and difficult to debug.

Parallelizing compilers Another approach to this problem has been the use of parallelizing compilers. A parallelizing compiler starts with a sequential program and tries to extract the parallelism in it to produce an explicitly parallel program. The advantage of this approach is that it can compile existing time-proven sequential programs. The disadvantage is that automatic systems by themselves are not powerful enough to find all the parallelism in complex sequential programs. The compiler must try to abstract a sequential program back to a (possibly) parallel algorithm. Then the compiler must go through a synthesis phase where the algorithm is retargeted for a particular parallel architecture.

When writing a program for a sequential computer, a programmer must totally sequentialize an algorithm. An algorithm which is well suited for a sequential implementation is most often chosen as a
basis for the program. Often, other algorithms may be better if one were considering implementation on a parallel computer. A parallelizing compiler starts with a sequential program and tries to undo the programmer’s sequentializing in order to find any possible parallelism it can exploit. But, there may be a limit to the amount of parallelism available when an inherently sequential algorithm is used as the base for the source program.

**Functional languages** Backus’ 1978 Turing Award lecture [2] generated a lot of interest in the functional (or applicative) languages as an alternative to conventional programming languages. It has long been known that functional programming languages are inherently parallel and there exist methods to test and debug functional programs on sequential computers [83]. Functional programs do not suffer from problems stemming from false dependencies (write- and anti-dependencies) which are an artifact of programs written in an imperative language. Functional languages implicitly support both the expression of concurrency and communication, and the programmer is freed from having to explicitly manage either communication or parallelism. However the transformation of a functional program to an efficient parallel program has proven to be a very difficult problem. Research in this area has shown that the problem of selecting the useful parallelism in functional programs is (almost) as difficult as detecting parallelism in imperative programs [81].

### 1.1 Synthesizing compiler approach

In this dissertation, I demonstrate synthesizing parallel code from algorithms written in a functional language. This new parallel programming methodology separates the issues of algorithm design from the detailed management of architecture dependent parallelism and communication. It allows the programmer to focus on designing the algorithm while delegating the management of parallelism and communication to the compiler. In this methodology, the algorithm is written as a high level specification which is then compiled into a data parallel imperative language, and finally the data parallel program may be compiled for the desired parallel architecture.

As the high level specification language, I am using ALPHA [53], a language which embodies the formal model of systems of recurrence equations. The recurrence equation model directly exposes potential parallelism since it only forces sequential dependencies on variables which are functionally dependent. Algorithms may be represented at a very high level in ALPHA, close to how one might specify them mathematically and are in an equational and fully parallel form. Writing programs in ALPHA allows the programmer to focus on the essentials of the algorithm. ALPHA source programs tend to be short, intuitive, and easy to verify.

In this dissertation, I demonstrate how ALPHA can be compiled into an imperative data parallel language. A data parallel language is an appropriate target form because recurrence relations are themselves inherently data parallel. Data parallel programs are machine-independent, however, the efficiency of a data parallel program is still machine independent. A data parallel program may be “tuned” for a target architecture in order to improve its performance [32]. The compiler can be given information about the target architecture to guide the derivation toward a data parallel program which will be tailored for the target architecture.

ALPHA was designed originally for systolic array synthesis and its synthesis capabilities were demonstrated by Dezan and Sie in their respective theses [19, 82]. ALPHA restricts the kinds of programs that it can represent, though not to the point of being trivial or useless. For instance, it is able to represent mathematical types of algorithms such as the kinds currently being proposed for signal and video processing. In exchange for reduced expressive power, one can make use of a set of powerful static analysis tools that enable the automation of some transformations and facilitate the compilation process. Many of the well understood transformations in systolic array design are becoming automated in this way. The synthesizing compiler is also based on a transformational approach. ALPHA is transformed into a form suitable for execution on a parallel computer by a series of provably correct program transformations.
1.2 Approach to be used to solve the problem

Being a functional language, ALPHA exposes all the parallelism of an algorithm. Communication and parallelism are represented implicitly. Mapping an algorithm which is already described in a parallel form (such as a system of affine recurrence equations) to another (perhaps less) parallel form involves mapping variables, operations, and dependencies to nested loop structures, instructions, processors, and communication links.

From the functional programming community, we know that all functional programs have a naive compilation using an applicative cache. However, the naive execution of ALPHA programs is generally inefficient in both speed and use of memory. The naive execution of ALPHA is described in chapter 5. From the systolic array community, we have techniques to compute timing functions for program variables. These functions schedule the order in which computations of variables need to occur to avoid context switching. The approach taken in this thesis is to use scheduling information to transform an ALPHA program to run efficiently.

I make use of the ALPHA language formalism and transformation tools which have already been shown useful for the synthesis of systolic arrays. I have also developed new transformations useful for parallel program synthesis and thus have made a contribution in the tools for transforming high-level programs into parallel programs.

Each transformation operates on an entire ALPHA program, producing a restructured, but equivalent ALPHA program. ALPHA has the expressive power to represent a program at many levels. Through the application of these transformations, the ALPHA environment becomes a synthesis tool. Figure 1.1 illustrates the idea of a synthesis trajectory within a program design space using discrete transformation steps. The design space is narrow at the top and wide at the bottom to represent that the number of design alternatives increases as the design abstraction decreases. The trajectories represent possible paths taken by the synthesizing compiler. All transformations work toward synthesizing a target program. Some transformations are typically chosen by hand by the designer, as she evolves the program towards her design goal.

Figure 1.2 is an overview of the synthesis process supported by the ALPHA environment. The analysis/ transformation synthesis loop is followed by the code generation phase. To simplify the final mapping
to parallel code (code generation), an ALPHA program is transformed into a form very close to the structure of the final code. This is the model I use to compile an ALPHA program. In this dissertation, I describe what transformations need to be performed to an ALPHA program to prepare it for code generation and describe a code generator.

1.3 Tools Used

1.3.1 The ALPHA Environment

This work relies heavily on the ALPHA system environment, in which a number of useful program transformations have already been developed. The compiler uses this capability to transform the source program into a format suitable for code generation. The ALPHA environment provides a toolchest from which different transformations can be used to transform a program from its current state toward some target state. Additional transformations can also be written and easily integrated into the system. A parser and unparsers (pretty printers) are also provided to translate an ALPHA source program to an abstract syntax tree and back again.

1.3.2 Mathematica

The ALPHA environment is built on top of the symbolic algebra system Mathematica [92] where the abstract syntax tree of an ALPHA program can be symbolically manipulated. Mathematica supports imperative, functional, and rule-based programming paradigms built on top of a symbolic calculator. The rule processor applies a set of rewrite rules to transform an ALPHA program (in abstract syntax tree form) to another equivalent ALPHA program. The ALPHA environment contains many kinds of transformations which are all performed using the symbolic calculator of Mathematica.

1.3.3 The Polyhedral Library

Transformations also rely heavily on a polyhedral library which provides the capability for doing fundamental operations on polyhedra and is described in my master’s thesis [88]. This is a library of functions which operates on objects called domains made of unions of convex polyhedra which are used to describe the domains of computation of variables in ALPHA. Examples of domain operations which can be performed by the library are: Image, Preimage, Intersection, Difference, Union, ConvexHull, and Simplify.
1.4 Related Work

1.4.1 Crystal

Crystal is another purely functional language similar to Alpha that provides special data types index domains which embody the geometric shape of data structures and data fields which generalize the notion of distributed data structures. It was developed by Marina Chen and her students at Yale University. In [10, 13, 12] she gave an overview of the language Crystal and developed a design methodology for synthesizing efficient systolic algorithms and architectures from problem definitions. In [9], she described the semantics and algebraic properties of Crystal. She also described the Crystal parallelizing compiler which performs a sequence of optimizations tuned for particular parallel machines, leading to the generation of efficient target code with explicit communications or synchronization. The approach used is to classify source programs according to the communication primitives and their cost on the target machine, and then map data structures to distributed memory and generate parallel code with explicit communication commands. In his thesis, Yang [94] presents a parallel programming methodology based on Crystal and transformational program derivation. He formalized the parallelizing process in terms of aligning, partitioning, broadcast-removing, communication-aggregating, scheduling, and optimizing high-level functional specifications for target parallel architectures.

Alpha and Crystal have many similarities: both are strongly typed pure functional languages and in both languages, variables are based on index domains and act as functions which map points in the index domain to values in a data field. However, there are also significant differences. Index field computations in Alpha are based on an algebra of index domain operations on unions of convex polyhedra whereas Crystal index domains are based on communication forms, (hypercube domains, tree domains, etc). Alpha is based on denotational semantics which allow the index domain of any Alpha expression to be statically computed, whereas Crystal is not.

The Alpha language is more restricted than Crystal. The restrictions in Alpha guarantee closure on a set of important program transformations, and also ensure that any Alpha program may be put into a predefined normal form as will be described in chapter 7. These restrictions also allow an Alpha program to be analyzed to obtain timing and communication information.

1.4.2 Systolic Array Synthesis

Karp, et al. [36] introduced the notion of systems of uniform recurrence equations and showed how they could be described using a dependence graph. Properties relating to the scheduling and parallelism of these systems were given. This paper became the basis for the work in systolic array synthesis which followed 15 years later. An article by Kung and Leiserson [42] first introduced systolic arrays. A tutorial article by Kung [41] defined the term systolic arrays, showed their usefulness, and motivated new research in the field.

Systolic array synthesis began by studying uniform recurrence equations (URE) which are defined in chapter 2 of this thesis. Quinton [62, 63] proposed a method of automatically deriving systolic arrays from URE’s. He gave a method for finding a timing function and an allocation function given a set of uniform dependencies. Chen [11] presented an approach where the initial specification is transformed into what are defined as first-order recurrences (which turned out to be a subset of URE’s) and an inductive technique was applied to map these onto a space-time domain. Cappello et al. [6] also studied space time transformation of URE based designs. Miranker et al. [55] demonstrated that computation can be interpreted in terms of a space time graph. He studied transformations of URE based arrays and show how computation could be put in terms of uniform recurrence equations by means of pipelining. Fortes [26, 27] also studied transformational techniques to design systolic arrays starting from algorithms. Rao [77] and Van Dongen [84] showed that all systolic circuits could be modeled with URE’s. Van Dongen [84] also showed how general linear recurrences could be transformed into URE’s by pipelining the data and then transforming them into a systolic recurrence using a change of basis.

Later the URE model was generalized to affine recurrence equations or ARE. Rajopadhye [70, 75] proposed the class of affine recurrence equations (ARE) as a superset of URE and extended many of the
previous results with URE's to ARE's. He discussed techniques for finding timing and allocation functions for ARE's. He also studied some of the more practical issues of doing I/O to a systolic array [71] and the synthesis of control signals to control systolic array processors [72]. Quinton [64] studied mapping affine recurrences onto parallel architectures and with Van Dongen [85], gave the sufficient and necessary conditions for scheduling ARE's. Huang [33] proposed synthesizing systolic arrays by performing provably correct transformations on an initial specification expressed as a program. The approach involved an exhaustive (or heuristic) search over all possible reorderings of computations. Roychowdhury et al. [78] discussed the localization of recurrence equations— that is, the conversion from ARE to URE.

The convergence in the fields of systolic design and parallelizing compilers has been noticed by many researchers. Formal systems for derivation of systolic arrays have been developed [29, 33, 57, 62] and continue to be improved. The techniques developed for synthesizing systolic arrays have begun to be applied to the generation of parallel programs [3, 68].

1.4.3 Static Analysis of Imperative Programs

Parallelizing compilers start from sequential imperative programs and produce parallel programs. Sequential programs are encumbered with false dependencies which hide the parallelism. In order to extract the parallelism of a program, an analyzer must ‘undo’ the sequential code generation to try to abstract a parallel algorithm from the sequential program. Then a parallel program is generated from the result of the analysis.

Research in this area may have had early roots in the work of Kuck [39]. During the 80’s, much research was done in creating vectorizing compilers, which successfully exploited the capabilities of large vector computers. Many of those researchers turned their attention to parallelizing compilers, which thus have a legacy in older vectorizing compilers [91]. One example, the PAF (Paralléliseur Automatique pour FORTRAN) by Raji-Werth and Feautrier [68, 87] performs dataflow analysis on FORTRAN program (actually, from a proper subset of FORTRAN programs that are amenable to such analysis), and derives a single assignment program from it. True data dependencies are found and are used as a basis for computing a new space-time mapping after the manner of systolic array techniques. Even after analysis, the resulting algorithm may still carry the effects of having once been a sequential program.

Barnett [3] used systolic techniques to mechanically transform simple algorithmic programs to parallel programs. However, the problem domain was severely restricted: it had to have a linear schedule, variables were defined over restricted domains, algorithms had to be uniform (systolic type of neighboring connections only), strides of loops were restricted to ±1, and loop bounds had to be linear (not piecewise linear).

This thesis treats a related problem of generating a parallel program from an abstract algorithm. The difference is that this thesis does not need to perform the difficult abstraction step to arrive at an abstract algorithm. Here, the source is already an algorithmic specification. Another difference is that the specification that I start with is at a much higher level of abstraction than what is achieved by a parallelizing compiler. One last difference is that the parallelizing compiler can use the scheduling information abstracted out of the sequential program to reschedule the program for parallel computation. Having started from a sequential program, this schedule always exists. These compilers find parallelism by doing loop exchanges and other transformations on nested loops which preserve the original schedule. In contrast, the proposed compiler does not have a schedule to start with, and one must therefore be derived.

1.4.4 Scheduling

In a pioneering paper, Lamport [43] described finding a schedule by finding a set of timing hyperplanes. In his PhD thesis, Kuhn [40] made use of dependence analysis and the dependence graph. Moldovan [56] gave a method of scheduling URE’s based on dependency analysis. Quinton[63] proposed a method of automatically deriving systolic arrays from uniform recurrence equations (URE). URE’s were proposed as a specification language for systolic arrays. A linear timing function was determined which satisfies
the causality constraints imposed by the dependencies. Second, an allocation function was chosen which maps the computation onto an array of processors.

Rajopadhya, Purushothanan, and Fujimoto [76] were the first to give a constructive solution to the ARE scheduling problem. Delosme and Ipsen [18] independently discussed the scheduling of ARE's at the about the same time. Quinton and Van Dongen [67] extended the solution of the ARE scheduling problem by giving both the necessary and sufficient conditions required for a schedule to exist. Yaacobly and Cappello [93] discussed the scheduling of ARE's where the linear part is non-singular. In [52], Mauras et al. extended existing techniques to schedule affine recurrence equations with a single global timing function to be able to independently schedule each variable in a system.

Lu and Chen [50, 51] proposed a loop transformation theory to extract parallelism from a nested loop program. A number of different scheduling paradigms were considered: uniform, subdomain-variant, and statement-variant. Feautrier [23] discussed the problem of finding closed form schedules as affine or piecewise affine functions of the iteration vector of a nested loop program. The scheduling problem was reduced to a small sized parametric integer programming problem. In a follow up article [25], he extended the algorithms given in [23] to problems whose schedules are polynomial, but not affine. Linear schedules were generalized to multidimensional affine schedules, with lexicographic ordering as a temporal succession, which were shown equivalent to polynomial schedules. A heuristic was given to produce a multidimensional schedule of the smallest dimension. In his thesis, Darte [16, 17] discussed finding optimal (in terms of total latency) variable dependent affine schedules by setting up and solving a linear programming problem. He then extended this work to the scheduling of parameterized systems. Most recently, he has shown that the derivation of multidimensional schedules by Feautrier [25] is equivalent to the decomposition method of Karp, Miller and Winograd [36], and that Feautrier’s heuristic in fact yields the optimal solution.

This work continues at the present day as researchers try to compute the nirvana of all schedules: an optimal, variable-dependent, parameterized, piecewise linear, multidimensional affine schedule. In this thesis, I simply acknowledge this extensive work and borrow from their results.

1.5 Goals

The following were goals of this work:

- Demonstrate the feasibility of the synthesizing compiler methodology, which is to synthesize an imperative data parallel program from the functional language Alpha using an analysis-guided, transformation-based compiler.

- Demonstrate that the functional language Alpha can be analyzed in order to optimize a program for a given target architecture. Show how this analysis can be used to formulate automatic synthesis as either an optimization or a constrained search problem. Report on the issues involved in doing such an analysis.

- Confirm that transformational program derivation is an effective tool for synthesizing compilers and show to what extent the selection of transformations can be automated. Demonstrate this by implementing the compiler.

- Demonstrate that an imperative data parallel language is an appropriate target when starting from the functional language Alpha. Describe the advantages and disadvantages of using a data parallel compiler as a back end to a parallelizing compiler. Describe tradeoffs made in the work done by the parallelizing compiler versus the data parallel compiler.

1.6 Contributions

This thesis makes the following contributions:
• The resulting compiler is embedded in an symbolic algebra system which allows the compiler to be extended. The system is thus useful to others who continue research in finding more efficient compilation methods and experiment with new optimization methods using the transformational approach.

• This research benefits the systolic array synthesis community where the transformational method is also applicable and where many of the compilation steps are similar to those in the proposed work. The proposed compiler also facilitates efficient simulation of systolic array specifications given in the Alpha language.

• This research benefits the functional language community by showing how a functional language which is restricted to be statically analyzable can be compiled into efficient imperative code (sequential or parallel).

• This research benefits the numerical analysis and digital signal processing communities by providing a compiler which is especially adapted to compiling data parallel algorithms of the type used in those fields.

1.7 Impact

I have demonstrated a methodology for programming parallel architecture machines, and contributed to the unlocking of their potential to deliver high performance at a reasonable cost. Using this methodology, a programmer is able to code a regular algorithm in a functional language in which parallelism is readily available, without having to explicitly manage communication, and compile this program for a parallel architecture.

Research in three somewhat disparate areas — functional programming, loop parallelization and systolic array synthesis has shared many concerns, without building on each others’ results. Functional languages have been clear but inefficient, imperative loop programs are often difficult to read, but efficient, and systolic arrays are overly specialized and often ignore the practical reality of partitioning, control generation, I/O, etc. I have applied some of the results of systolic array synthesis to the compilation of functional programming languages with a measure of success. I believe that there is still much to be gained by cross fertilization of these issues across the different communities.

1.8 Overview of thesis

Chapter 2 defines the Alpha language, and gives denotational and type semantics for it. To convert an Alpha program to imperative code, a variable which is defined over a domain must be computed in the body of an imperative loop nest which scans the domain. Chapter 3 presents a method to synthesize loop nests to scan polyhedral domains. The compiler must also be able to declare storage for the domains of Alpha variables. Chapter 4 deals with the issues involved in the allocation of memory to domain variables, in order to generate imperative declarations for program variables. Both loop nest generation and allocation of memory are fundamental to the generation of imperative code from Alpha. Chapter 5 demonstrates an automatic translation scheme which compiles Alpha into the imperative language C, which uses the results of the previous two chapters. The naive code generator described in chapter 5 generates recursive calls to functions to compute variables using a strategy which closely follows the denotational semantics of Alpha described in chapter 2.

Starting from this basis, I investigate techniques to improve the quality of the imperative code generated by the compiler. In chapter 6, I show that the same methods which are used for solving regular array synthesis problems can be applied to the compilation of Alpha. Dependency, usage and timing analyses of an Alpha program are discussed. Using dependency and timing analysis, an execution schedule giving relative times for computation of variables can be derived. If an Alpha program can be reordered according to the scheduled computation times of its variables, then non-recursive imperative code can
be generated. Chapter 7 presents the basis for a transformation system which can change the form of an Alpha program without changing its meaning. The Alpha environment implements these transformations. It is the application of these transformations which forms the basis of the synthesizing compiler for Alpha. In chapter 8, I show that given a schedule, and using a series of program transformations, Alpha statements can be separated, regrouped and reordered according to their scheduled execution times, and finally be transformed into a form suitable for sequential imperative code generation. In chapter 9, I take advantage of the static analyzability of Alpha to show how an Alpha program can be compiled to use memory most efficiently. In chapter 10, I show what additional capabilities are needed to extend the sequential code compiler, described in earlier chapters, so that Alpha can be compiled into imperative data parallel code. In conclusion, chapter 11 reviews the goals given in the introduction, presents some open problems, and summarizes the work done in this thesis.
Chapter 2

The ALPHA language

The ALPHA language was developed by Mauras [53] at IRISA in Rennes, France. It was a product of research in regular parallel array processors and systolic arrays. The ALPHA language is able to formally represent algorithms which have a high degree of regularity and parallelism such as systolic algorithms as defined by Kung [41]. It is based on the formalism of recurrence equations which has been often used in various forms by several authors [75, 31, 62] all of which were based on the introduction of the notion of uniform recurrence equations by Karp, Miller and Winograd [36].

2.1 Systems of Affine Recurrence Equations

ALPHA is based on the formalism of systems of affine recurrence equations. The definitions in this section review the basic concepts of systems of affine recurrence equations and are taken primarily from the work of Rajopadhye and Fujimoto [75], Yaacoby and Cappello [93], Delosme and Ipsen [18], and Quinton and Van Dongen [67].

In describing systems of affine recurrence equations, there are two equally valid points of view which can be taken. The first is a purely functional point of view in which every identifier is a function. A recurrence equation defines a function on the left hand side in terms of the functions on the right hand side. Alternately, each identifier can be thought of as a single assignment variable and equations equate the variable on the left hand side to a function of variables on the right.

**Definition 1 (Recurrence Equation)**

A Recurrence Equation over a domain $D$ is defined to be an equation of the form

$$f(z) = g(f_1(I_1(z)), f_2(I_2(z)), \ldots, f_k(I_k(z)))$$

where

- $f(z)$ is a variable indexed by $z$, an n-dimensional vector of integers. It can also be thought of as a function from points $z$ in its domain to values.
- $z \in D$, where $D$ is the (possibly parameterized) domain of variable $f$.
- $f_1, \ldots, f_k$ are variables found on the right hand side of the equation. They may include the variable $f$, or multiple instances of any variable.
- $I_i$ are index mapping functions (also called dependency mapping functions) which map $z \in D$ to $I_i(z) \in D_i$, where $D_1, \ldots, D_k$ are the (possibly parameterized) domains of variables $f_1, \ldots, f_k$, respectively.
- $g$ is a strict single-valued function whose complexity is $O(1)$ (can be executed in a fixed number of clock cycles) defining the right hand side of the equation.
A variation of an equation allows \( f \) to be defined in a finite number of disjoint “cases” consisting of subdomains:

\[
f(z) = \begin{cases} 
  z \in D_1 & \Rightarrow \ g_1(\ldots f_1(I_1(z))\ldots) \\
  z \in D_2 & \Rightarrow \ g_2(\ldots f_2(I_2(z))\ldots) \\
  \vdots
\end{cases}
\]

(2.1)

where the domain of variable \( f \) is \( D = \bigcup_i D_i \) and \((i \neq j) \rightarrow (D_i \cap D_j = \emptyset)\)

**Definition 2 (Dependency)**

For a system of recurrences, we say that a variable \( f_i \) at a point \( p \in D_i \) (directly) depends on variable \( f_j \) at \( q \), (denoted by \( p_i \rightarrow q_j \)), whenever \( f_j(q) \) occurs on the right hand side of the equation defining \( f_i(p) \). The transitive closure of this is called the dependency relation, denoted by \( p_i \rightarrow q_j \).

**Definition 3 (Uniform and Affine Recurrence Equations)**

A recurrence equation of the form defined above is called a Uniform Recurrence Equation (URE) if all of the dependency functions \( I_i \) are of the form \( I(z) = z + b \), where \( b \) is a constant \( n \)-dimensional vector. It is called an Affine Recurrence Equation (ARE) if \( I(z) = Az + b \), where \( A \) is a constant matrix, and \( b \) is a constant \( n \)-vector.

**Definition 4 (System of Affine Recurrence Equations, or SARE)**

A system of recurrence equations is a set of \( m \) such equations, defining the functions \( f_1 \ldots f_m \) over domains \( D_1 \ldots D_m \) respectively. The equations may be (mutually) recursive. Variables are designated as either input, output, or local variables of the system. Each variable (which is not a system input) appears on the left hand side of an equation once only and once only. Variables may appear on the right hand sides of equations as often as needed.

Such equations serve as a purely functional definition of a computation, and are in the form of a static program — a program whose dependency graph can be statically determined and analyzed (for any given instance of the parameters). Static programs require that all \( g_i \) be strict functions and that any conditional expressions be limited to linear inequalities involving the indices of the left hand side variable. By convention, it is assumed that boundary values (or input values) are all specified whenever needed for any function evaluation.

### 2.2 ALPHA : An applicative equational language

An equational language is a natural way to describe a SARE. When thinking about algorithms such as those used in signal processing or numerical analysis applications, a person naturally thinks in terms of mathematical equations. Mathematical notation has evolved over the centuries and obeys certain basic rules. (1.) Given a function and an input, the same output must be produced each time the function is evaluated. If the function is time varying, then time must be a parameter to the function. Turner [83] uses the term *static* to describe this property. (2.) Consistency in the use of names: a variable stands for the same value throughout its scope. This is called referential transparency. An immediate consequence of referential transparency is that equality is substitutive — equal expressions are always and everywhere interchangeable. This property is what gives mathematical notation its deductive power. ALPHA shares both of these properties with mathematical notation: it is both static and referentially transparent.

Using a language that shares the properties of mathematical notation eases the task of representing a mathematical algorithm as a program. Furthermore, such a method of describing algorithms has some interesting properties, as has been discovered by users of ALPHA. An equation specifies an assertion on a variable which must always be true. Reasoning about programs can thus be done in the context of the program itself, and relies essentially on the fact that ALPHA programs respect the substitution principle. This principle states that an equation \( X = Expression \) specifies a total synonymy between the variable on the left hand side of the equation and the expression on the right hand side of the equation. Thus any instance of a variable on the left hand side of any equation may be replaced with the
right hand side of its definition. Likewise, any sub-expression may be replaced with a variable identifier, provided that an equation exists, or one is introduced, in which that variable is defined to be equal to that sub-expression.

**ALPHA** has other properties which should be mentioned.

- **ALPHA** is a functional language. The only ordering of computations is that which is implied by data dependencies.

- **ALPHA** is a single assignment language. Each variable element can only ever hold a single value which is ultimately a function of system inputs.

- Every **ALPHA** program is a *static program*, meaning that its run time behavior can be analyzed at compile time.

- **ALPHA** does not support any notion of global variables. An execution of a system defined in **ALPHA** only affects the outputs of the system — there are never any side effects.

- **ALPHA** is strongly typed. Each variable must be predeclared with both a domain and data type attributed to that variable.

At this point, it can be stated that **ALPHA** adopts the classical principles of a functional language which is structured and strongly typed. An **ALPHA** program defines a function from its input variables to its output variables. This notion of a function is embedded in the definition of the **ALPHA** system construct.

### 2.3 **ALPHA** System Declarations

At the top level, an **ALPHA** program consists of a *system declaration* consisting of:

1. A system name with parameter declaration
2. A list of input variable declarations
3. A list of output variable declarations

which is followed by the *system definition*, consisting of:

1. A list of local variable declarations
2. A list of equations defining output and local variables.

This information appears in the following format in Alpha syntax.

```
| system <system-name> : <parameter-domain> |
|    ( <input-variable-declarations> )       |
|     returns ( <output-variable-declarations> ); |
| var  <local-variable-declarations>;        |
| let   <equations>                          |
| tel;                                       |
```

### 2.4 Variable Declarations

**Alpha** is strongly typed and each variable is declared with a type of the form \(<\text{domain} \leftrightarrow \text{datatype}>\). As in a classical typed system, each variable must be declared with its type and then uses of that variable must conform to its declaration.

A variable can be thought of as a function mapping integer points in a domain to values in the data type set:

\[ X : z \in \text{domain} \leftrightarrow X[z] \in \text{datatype} \]

The observation that a variable is also a function ties together the variable and functional views of recurrence equations. The type of a variable may thus be thought of as a prototype of the function of the variable, giving both the domain and range of the function.

The <datatype> is one of the three base data types: integer, boolean, and real. All data types are assumed to be infinite precision and include the special value \(\text{error}\) which means a value which is either not specified, or badly specified.

Each **Alpha** variable is associated with a fixed domain and has a value associated with each point in that domain. Scalar variables may also be declared, and hold a single value associated with the point in the trivial domain \(\mathbb{Z}^0\). The following syntax forms for a variable declaration are supported in **Alpha**, (the last is a scalar declaration):

\[ <\text{var-list}> : <\text{domain}> of <\text{datatype}>; \]
\[ <\text{var-list}> : <\text{datatype}>; \]

### 2.4.1 Type Checking

An important advantage of being a strongly typed language is that certain static analysis procedures can be performed which detect inconsistencies in the way a variable is declared and the way it is used. **Alpha** extends the classical type check to a more powerful check which uses polyhedral computation and permits the verification that a variable is defined and used in manner consistent with its declaration and that each and every element in the variable has exactly one value associated with it. These semantic checks are able to detect a large class of programming and logic errors.

### 2.5 Domain variables

Each **Alpha** variable is a function over a domain in \(\mathbb{Z}^n\). When specifying a system of affine recurrence equations, unions of convex polyhedra are used to describe the domains of computation of system variables.

The set of solution points which satisfy a mixed system of linear constraints form a polyhedron \(\mathcal{P}\) and serve as the *implicit definition* of the polyhedron

\[
\mathcal{P} = \{ x \mid Ax = b, Cx \geq d \} \tag{2.2}
\]
Some important polyhedral operators are used in the semantics of Alpha. They are defined here:

\[
\begin{align*}
\text{Image}(P, T) &= \{ x \mid x = Ty, y \in P \} \\
\text{Preimage}(P, T) &= \{ x \mid \exists y \in P : y = Tx \} \\
P_1 \cap P_2 &= \{ x \mid x \in P_1 \text{ and } x \in P_2 \} \\
P_1 \cup P_2 &= \{ x \mid x \in P_1 \text{ or } x \in P_2 \} \\
P_1 \setminus P_2 &= \{ x \mid x \in P_1 \text{ and not } x \in P_2 \}
\end{align*}
\]

The result of \text{Image} is in general not a polyhedron, but a linear bounded lattice [48]. However, under the restriction that \(T\) is unimodular, \text{Image}(P, T)\) is a polyhedron. The results of the union and difference operations are in general unions of polyhedra. These operations are all implemented in the polyhedral library [88] which is used extensively in the Alpha system. In computing the \text{Image} function, the library actually computes the convex hull of the result to force closure.

Whereas a polyhedron is a region containing an infinite number of rational (or real) points, a polyhedral domain, as the term is used in this thesis, refers to the lattice of integral points \(\mathbb{Z}^n\) which are inside a polyhedron (or union of polyhedra). Figure 2.2 illustrates this difference.

**Definition 5** A polyhedral domain of dimension \(n\) is defined as

\[
D : \{ z \mid z \in \mathbb{Z}^n, z \in P \} = \mathbb{Z}^n \cap P
\]

where \(P\) is a union of finitely many convex polyhedra of dimension \(n\).

In affine recurrence equations of the type considered here and in the Alpha language, every variable is declared over a domain as just described. Elements of a variable are in a one-to-one correspondence with points in a domain. Again, figure 2.2 illustrates this. Here, we formalize the definition of a variable.

**Definition 6** A variable \(X\) of type “datatype” declared over a domain \(D\) is defined as

\[
X = \{ X[z] \mid X[z] \in \text{datatype}, z \in D \}
\]

where \(X[z]\) is the element of \(X\) corresponding to the point \(z\) in domain \(D\) and “datatype” is either integer, boolean, or real.

The Alpha syntax for representing a single polyhedral domain is as follows:

\[
\{ \langle\text{index-list}\rangle \mid \langle\text{constraint-list}\rangle \}
\]
More complicated domains can be built up using the three domain operators: union, intersection, and difference. Union is written by combining two domains with a vertical bar: \( \{ \ldots \} \mid \{ \ldots \} \), intersection is written with an ampersand: \( \{ \ldots \} \& \{ \ldots \} \), and difference with an ampersand-tilde: \( \{ \ldots \} \&\sim \{ \ldots \} \).

Some examples of domains written in the Alpha syntax are given below:

\[
\begin{align*}
\{ x, y, z \mid -5 \leq x - y \leq 5; -5 \leq x + y \leq 5; z = 2x - 3y \} \\
\{ i, j, N \mid 0 \leq i \leq N-1; N \leq j \leq 2N-1 \} \quad \text{-- parameterized domain} \\
\{ i, j \mid 1 \geq 0 \} \quad \text{-- a plane in 3 space} \\
\{ i \mid 1 = 0 \} \quad \text{-- 2 dimensional universe domain} \\
\{ i, j \mid i=1; 0 \leq j \leq 2 \} \mid \{ i, j \mid i=3; 1 \leq j \leq 6 \} \quad \text{-- union of domains}
\end{align*}
\]

### 2.6 Denotational and type semantics

In the sections which follow, the syntax and semantics of each Alpha construct are described. Mauras [53] originally gave the denotational semantics of Alpha. In these sections, I give an equivalent semantics in terms of the abstract syntax tree. For each construct, both the denotational and type semantics are given in terms of functions which take as an argument a syntactic entity (in the form of an abstract syntax tree). This enables an interpreter for Alpha to be written directly from the semantics.

The type semantics are defined by the two functions \( \text{Domain()} \) and \( \text{Type()} \). The functions \( \text{Domain()} \), and \( \text{Type()} \) both take a single argument (a syntactic entity representing an Alpha expression), and they return the \langle\text{domain}\rangle\) and \langle\text{datatype}\rangle\) of their arguments respectively. Using these two functions, a static type checker can be written for Alpha.

Denotational semantics directly construct a definition of the function that each program in the language computes. This definition is built up hierarchically through definition of the function computed by each individual program construct [61]. In Alpha, every expression denotes a mapping from a point in the domain that expression to a value in the data type of that expression. The denotational semantics of Alpha are given by the \text{Eval()}\) function which takes as its first argument the abstract syntax tree of an Alpha expression, and as its second argument, a point in the domain of that expression. It then returns the value of the expression evaluated at that point. The \text{Eval()}\) function is given recursively in terms of other \text{Eval()}\) functions, the \text{error value}, and the \text{Input()}\) function. The \text{Input()}\) function retrieves input values from the “system” in an unspecified manner. (The parameter passing mechanism is unspecified). Using the denotational semantics, an interpreter which “executes” a program, that is, finds values for all of the outputs given the inputs, may be written.

These functions are used to prove theorems relating to the semantic equivalence of Alpha expressions. Many of these theorems are given in chapter 7. Their proofs are based on the definition of equivalent Alpha expressions, which is given here.

**Definition 7 (Equivalent Expressions)**

Two Alpha expressions \( \text{exp}_1 \) and \( \text{exp}_2 \) are equivalent if and only if they are of the same type and denotation, as established by the following three equalities:

\[
\begin{align*}
\text{Domain(} \text{exp}_1 \text{)} &= \text{Domain(} \text{exp}_2 \text{)} = D \\
\text{Type(} \text{exp}_1 \text{)} &= \text{Type(} \text{exp}_2 \text{)} \\
\forall z \in D : \text{Eval(} \text{exp}_1, z \text{)} &= \text{Eval(} \text{exp}_2, z \text{)}
\end{align*}
\]

### 2.7 Equations and Expressions

All local and output variables are defined by a set of equations.
var
  X : <domain> of <type>;
...
let
  X = <expression_1>;
...
  X = <expression_n>;
tel;

where:
X : <domain> of <type>; is called the *declaration* of X, and
X = <expression_1>; ⋯ X = <expression_n>; is called the *definition* of X. The semantics of an equation are as follows:

\[ \forall z \in \text{<domain>} : \]

\[
\text{Eval}(X, z) = \begin{cases} 
  \text{Eval(<expression_1>, z)} & \text{if } z \in \text{Domain(<expression_1>)} \\
  \ldots & \\
  \text{Eval(<expression_n>, z)} & \text{if } z \in \text{Domain(<expression_n>)} \\
  \text{error} & \text{otherwise}
\end{cases}
\]

As can be seen above, each variable X has certain type and definition information associated with it, namely: <domain>, <type>, <kind>, <expression>,

where

- <domain> is from the declaration of X
- <type> ∈ \{integer, boolean, real\} from the declaration of X
- <kind> ∈ \{input, output, local\} from the declaration of X
- <expression> the definition of X

Given this information, the following static *type checks* can be made

\[
\bigcup_i \text{Domain(<expression>_i)} \supseteq \text{<domain>}
\]

\[
\forall i : \text{Type(<expression>_i)} = \text{<type>}
\]

Other static checks may also be made:

1. *Declaration rule*. Every variable has one declaration as either an input, output, or local variable.
2. *Definition rule*. Every output and local variable has a definition. Input variables have no definition.
3. *Usage rule*. Every local and input variable appears in the definition of some other variable.

An expression in ALPHA is composed of variables, constants, and operations defined in the language. The semantics of these constructs are described in the following sections.

### 2.7.1 Variables

All variables must have a <domain> and <type> given in the variable declaration and (except for input variables) a defining <expression> given on the RHS of the variable definition. When a variable X is used, the following semantics apply <domain>, <type>, <kind>, <expression> are associated with X:

\[
\text{Eval}(X, z) = \begin{cases} 
  \text{Eval(<expression>, z)} & \text{if } z \in \text{<domain>} \text{ AND } \text{<kind>} = \text{local} \text{ or output} \\
  \text{Input}(X, z) & \text{if } z \in \text{<domain>} \text{ AND } \text{<kind>} = \text{input} \\
  \text{error} & \text{otherwise}
\end{cases}
\]

\[
\text{Domain}(X) = \text{<domain>}
\]

\[
\text{Type}(X) = \text{<type>}
\]
2.7.2 Constants

Constants (integer, boolean, and real) are defined over the scalar domain \( \mathbb{Z}^0 \), a zero dimensional domain consisting of a single value. This domain can be extended to a domain of any dimension.

\[
\begin{align*}
\text{Eval}(\text{constant}, z) &= \text{constant.value} \\
\text{Domain}(\text{constant}) &= \mathbb{Z}^0 \\
\text{Type}(\text{constant}) &= \text{constant.type}
\end{align*}
\]

Both the value and the type of a constant are derived from the syntax.

2.7.3 Pointwise Operators

A pointwise operator is a spatial generalization of the classical scalar operators which are applied to the domains of their arguments element by element. For example, if

\[
X = \begin{bmatrix} X_{11} & X_{21} & X_{31} \\ X_{12} & X_{22} & X_{32} \\ X_{13} & X_{23} & X_{33} \end{bmatrix} \quad Y = \begin{bmatrix} Y_{11} & Y_{21} & Y_{31} \\ Y_{12} & Y_{22} & Y_{32} \\ Y_{13} & Y_{23} & Y_{33} \end{bmatrix}
\]

then the expression \( X \leftrightarrow Y \) is computed as:

\[
X \leftrightarrow Y = \begin{bmatrix} X_{11} \leftrightarrow Y_{11} & X_{21} \leftrightarrow Y_{21} & X_{31} \leftrightarrow Y_{31} \\ X_{12} \leftrightarrow Y_{12} & X_{22} \leftrightarrow Y_{22} & X_{32} \leftrightarrow Y_{32} \\ X_{13} \leftrightarrow Y_{13} & X_{23} \leftrightarrow Y_{23} & X_{33} \leftrightarrow Y_{33} \end{bmatrix}
\]

Even if the domains of the arguments are not exactly the same, the definition of a scalar binary operator can still be extended to operate over the intersection of the two domain variables. In general, a binary pointwise operator \( op \) is defined as follows:

\[
\begin{align*}
\text{Eval}(\text{binop}(op, \text{Exp}_1, \text{Exp}_2), z) &= \begin{cases} 
\text{Eval}(\text{Exp}_1, z) \text{ op } \text{Eval}(\text{Exp}_2, z) & \text{if } z \in (\text{Domain}(\text{Exp}_1) \cap \text{Domain}(\text{Exp}_2)) \\
\text{error} & \text{otherwise}
\end{cases} \\
\text{Domain}(\text{binop}(op, \text{Exp}_1, \text{Exp}_2)) &= \text{Domain}(\text{Exp}_1) \cap \text{Domain}(\text{Exp}_2) \\
\text{Type}(\text{binop}(op, \text{Exp}_1, \text{Exp}_2)) &= \text{TypeTable}(op, \text{Type}(\text{Exp}_1), \text{Type}(\text{Exp}_2))
\end{align*}
\]

and as follows for unary operations:

\[
\begin{align*}
\text{Eval}(\text{unop}(op, \text{Exp}), z) &= \begin{cases} 
op \text{Eval}(\text{Exp}, z) & \text{if } z \in \text{Domain}(\text{Exp}) \\
\text{error} & \text{otherwise}
\end{cases} \\
\text{Domain}(\text{unop}(op, \text{Exp})) &= \text{Domain}(\text{Exp}) \\
\text{Type}(\text{unop}(op, \text{Exp})) &= \text{TypeTable}(op, \text{Type}(\text{Exp}))
\end{align*}
\]

where the function TypeTable, which maps an operator and the types of its arguments to a result type, is defined in table 2.1.

2.8 Domain operators

Domain operators explicitly manipulate the domains of variables. In this section, the case operator is presented which allows the piecewise composition of expressions. In connection with the case operator, the restrict operator is presented which permits the discrimination between different parts of the domain of an expression. And finally, the dependence operator is presented, which establishes an affine relation between the points of one domain and the points of another.
And for the *if — then — else* — operation:

```
<table>
<thead>
<tr>
<th>Source 1 Type</th>
<th>Source 2 Type</th>
<th>Source 3 Type</th>
<th>→ Destination Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>int/real/boolean</td>
<td>int/real/boolean</td>
<td>int/real/boolean</td>
</tr>
</tbody>
</table>
```

Table 2.1: Type Table for Pointwise Operations

### 2.8.1 Case Operator

The case operator pieces together a set of disjoint subexpressions. The expression:

```
case \( \text{Exp}_1 \)
\[ \vdots \]
\( \text{Exp}_n \)
esac
```

is defined as:

\[
\text{Eval}(\text{case}(\text{Exp}_1, \ldots, \text{Exp}_n), z) = \begin{cases} 
\text{Eval}(\text{Exp}_1, z) & \text{if } z \in D_1 = \text{Domain}(\text{Exp}_1) \\
\vdots & \\
\text{Eval}(\text{Exp}_n, z) & \text{if } z \in D_n = \text{Domain}(\text{Exp}_n) \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[
\text{Domain}(\text{case}(\text{Exp}_1, \ldots, \text{Exp}_n)) = \text{Domain}(\text{Exp}_1) \cup \cdots \cup \text{Domain}(\text{Exp}_n)
\]

\[
\text{Type}(\text{case}(\text{Exp}_1, \ldots, \text{Exp}_n)) = \text{Type}(\text{Exp}_i), \; i = 1 \cdots n
\]

with static checks:

\[
i \neq j \rightarrow \begin{cases} 
\text{Domain}(\text{Exp}_i) \cap \text{Domain}(\text{Exp}_j) = \{ \}
\end{cases}
\]

For each point \( z \in \text{Domain}(\text{case}) \), one and only one expression should exist such that \( z \in \text{Domain}(\text{Exp}_i) \). The value of the case statement is *error* for values of \( z \) which are not found in the domains of any of the subexpressions, and the expression is not defined if \( z \) is found in two or more subexpression domains. This situation can be detected with a static check.

### 2.8.2 Restrict Operator

The restriction operator takes a subset of an expression and is written as:

```
\text{Dom} : \text{Exp}
```

and is defined as:

\[
\text{Eval}(\text{restrict}(\text{Dom}, \text{Exp}), z) = \begin{cases} 
\text{Eval}(\text{Exp}, z) & \text{if } z \in (\text{Dom} \cap \text{Domain}(\text{Exp})) \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[
\text{Domain}(\text{restrict}(\text{Dom}, \text{Exp})) = \text{Dom} \cap \text{Domain}(\text{Exp})
\]

\[
\text{Type}(\text{restrict}(\text{Dom}, \text{Exp})) = \text{Type}(\text{Exp})
\]

18
The restrict operator defines a new expression equal to an ALPHA expression \( \text{Exp} \), but defined only over the restricted region (see figure 2.3). It is often used in connection with the case operator as follows:

\[
\text{case } \text{Dom}_1 : \text{Exp}_1 \\
\vdots \\
\text{Dom}_n : \text{Exp}_n \\
\text{esac}
\]

The restriction operators help to define the subdomains of the case expressions.

### 2.8.3 Dependence Operator

An affine dependence function is a function that maps each point \( z \) in a domain \( \mathcal{D} \) to a point \( A.z + b \) in a domain \( \mathcal{E} \), \( \text{dep} : \mathcal{D} \leftrightarrow \mathcal{E} \). The dependence operator composes an expression \( \text{Exp} \) with an affine dependence function defined by \( A.z + b \), where \( A \) is a constant matrix and \( b \) is a constant vector, and is written as:

\[
\text{Exp}.(z \rightarrow A.z + b)
\]

and is defined as:

\[
\text{Mat} = \begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}
\]

\[
\text{Eval}(\text{dep}(\text{Exp}, \text{Mat}), z) = \begin{cases} 
\text{Eval}(\text{Exp}, A.z + b) & : \text{if } z \in \text{Preimage}(\text{Domain}(\text{Exp}), \text{Mat}) \\
\text{error} & : \text{otherwise}
\end{cases}
\]

\[
\text{Domain}(\text{dep}(\text{Exp}, \text{Mat})) = \text{Preimage}(\text{Domain}(\text{Exp}), \text{Mat})
\]

\[
\text{Type}(\text{dep}(\text{Exp}, \text{Mat})) = \text{Type}(\text{Exp})
\]

Here are a few examples of dependency operations written in ALPHA syntax:

\[
X.(i,j \rightarrow j,i) \quad \text{The transpose of a 2-dimensional variable } X
\]

\[
X.(i \rightarrow i,1) \quad \text{The diagonal vector of a 2-dimensional variable } X
\]

\[
X.(i,j \rightarrow 2+i+j+3) \quad \text{A 1-dimensional variable } X \text{ being indexed by } 2 + i + j + 3
\]

### 2.8.4 Reduction Operator

The reduction operator is a high level construct that allows for a more abstract expression of an algorithm and enlarges the design space for the implementation of the algorithm. Many basic regular array
algorithms can be very simply expressed using this operator. It was introduced into the \textsc{Alpha} language by Le Verge [45] who showed that this construct preserved referential transparency, the substitution principle, and normalization.

A reduction operator performs a many to one projection of an \textsc{Alpha} expression, combining values mapped to a common result point with an associative and commutative binary operator. Its syntax is

\[ \text{reduce}(\oplus, (z \rightarrow A.z + b), \text{expression}) \]

The operator \( \oplus \) is any associative and commutative binary operator defined in the \textsc{Alpha} language and \textit{expression} is any \textsc{Alpha} expression. The function \((z \rightarrow A.z + b)\) is a \textit{projection function} \( f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m \) which is written like a dependence operator, where the matrix \( A \) is of dimension \( n \times m, n > m \). Not all projection functions are valid in a reduction operator. Conditions that qualify a valid function \((z \rightarrow A.z + b)\) are given in [45]. Examples of projection functions are: \((i, j, k \rightarrow i + j + k)\) and \((i, j \rightarrow j)\). The semantics of the reduction operator are defined as:

\[ \text{Mat} = \begin{pmatrix} A_{n \times m} & b \\ 0 & 1 \end{pmatrix} \]

\[ \text{Eval}(\text{reduce}(\oplus, (q \rightarrow Aq + b), \text{Exp}), z) = \begin{cases} \oplus((\text{Eval}(\text{Exp}, y) | Ay + b = z, y \in \text{Domain}(\text{Exp})) & \text{if } z \in \text{Image(\text{Domain}(\text{Exp}), \text{Mat})} \\ \text{Identity}(\oplus) & \text{otherwise} \end{cases} \]

\[ \text{Domain(\text{reduce}(\oplus, (q \rightarrow Aq + b), \text{Exp}))} = \text{Universe}(m) \]

\[ \text{Type(\text{reduce}(\oplus, (q \rightarrow Aq + b), \text{Exp}))} = \text{Type(\text{Exp})} \]

\[ \oplus(S) = \begin{cases} \oplus(S) & \text{if } S = \{ \} \\ x \oplus \oplus(S \setminus \{x\}) & \text{if } x \in S \end{cases} \]

An example of a reduction operation to do matrix multiplication \( \sum_{i=1}^{N} a[i, k] \ast b[k, j] \), written in \textsc{Alpha} syntax is:

\texttt{reduce( *, (u,v,w->u,v), a(i,j,k->i,k)*b(i,j,k->k,j) )}

\section*{2.9 An Extended Semantics}

By adding storage to the model, a more efficient evaluation strategy can be followed using the applicative caching model of Bird [4], and Keller and Sleep [37]. An alternative evaluation procedure, \texttt{Eval2(X,z)}, may be used in place of \texttt{Eval(X,z)} above to evaluate \textsc{Alpha} more efficiently. It stores precomputed values for points in \( X \) in an \( X \)-table which has been preinitialized with the value \text{undefined} (which value must be added to the semantics). The first time each point in \( X \) is evaluated, it is stored in this table. Should a point ever be needed again, it is not recomputed, but is instead obtained from the \( X \)-table. The table is accessed using two procedures: \texttt{Store(X,z,v)} which stores value \( v \) in the \( X \)-table at point \( z \), and \texttt{Get(X,z)} which retrieves and returns the value in the \( X \)-table at point \( z \).

\[ \text{Eval2}(X,z) = \begin{cases} \text{Get}(X,z) & \text{if } \text{Get}(X,z) \neq \text{undefined} \\ \text{Eval}(X,z), \text{Store}(X,z,v) & \text{if } \text{Get}(X,z) = \text{undefined} \end{cases} \]

\section*{2.10 Conclusion}

\textsc{Alpha} is a single assignment, equational language based on the formalism of systems of affine recurrence equations. Algorithms may be represented at a very high level in \textsc{Alpha}, close to how one might specify them mathematically and are in an equational and inherently parallel form. The \textsc{Alpha} language is somewhat restrictive in the class of algorithms it can easily represent, but is useful for programming mathematical types of algorithms such as the kinds currently being proposed for signal and video processing and a proper subset of the language (uniform \textsc{Alpha}) is Turing complete [79]. What is gained
by Alpha is the analyzability needed to facilitate the compilation process. Alpha can be transformed into a form suitable for execution on a parallel computer by a series of program transformations which can be independently proved. The derived program is therefore correct by construction (assuming the specification was correct).
Chapter 3

Loop Nest Synthesis using the Polyhedral Library

In ALPHA, equations are defined over a region of index space called a domain. To convert ALPHA programs to imperative code, equations which are defined over a domain must be elaborated in terms of imperative loops which scan the domain. The synthesis of loop nests to scan a domain is thus fundamental in the generation of imperative code from ALPHA.

3.1 Introduction

The spatial point of view of a loop nest goes back to the work of Kuck [39] who showed that the domain of nested loops with affine lower and upper bounds can be described in terms of a polyhedron (figure 3.1). Loop nest synthesis grew out of the earlier loop transformation theory [35, 90] where it was shown that all loop transformations could be performed by doing a reindexing of the underlying index domain, followed by a rescanning (perhaps in a different order) of the domain (figure 3.2). Loop nest synthesis is based on the polyhedral scanning problem which poses the problem of finding a set of nested do-loops which visit each integral point in a polyhedron.

Ancourt et al. [1] were the first to solve the polyhedral scanning problem. They used a method to compute the loop nests which is based on a Fourier-Motzkin pairwise elimination procedure. This method involves the projection of polyhedron along an axis to find the loop bounds. The main difficulty is that the Fourier-Motzkin elimination method creates redundant bound equations which must be eliminated afterward. Le Fur et al. [41] also use this method for their Pandore II compiler.

The traversal of a polyhedron by a set of nested loops can be thought of as a lexicographical ordering of the integer points in the polyhedron, where a point $a$ is executed before a point $b$ if $a < b$ (a precedes $b$ lexicographically). Thus given a polyhedron, loop bound expressions can be derived by finding the lexical minimum and maximum of the polyhedron in a given set of directions and in terms of parameters and

\[\begin{align*}
\text{for } (i=0; i<=6; i++) \\
&\text{for } (j=0; j<=1; j++) \\
&\text{...}
\end{align*}\]

\[\begin{array}{c}
j \uparrow \\
\downarrow \quad \downarrow \\
0 & 6 & i
\end{array}\]

Figure 3.1: The Spatial Interpretation of a Loop Nest
outer loop variables. A technique to do this uses the Parametric Integer Program (PIP) developed by Feautrier [20, 21]. PIP finds the lexicographic minimum of the set of integer points which lie inside a convex polyhedron which depends linearly on one or more parameters. PIP is called twice for each loop in the loop nest, once for the lower and once for the upper bound. The loop expressions must then be extracted out of the PIP output, which is a quasi-affine expression tree, where each branch is guarded by a constraint and the terminal nodes hold either an index expression or ⊥ (bottom) meaning that that branch is infeasible. The extraction of the loop bound expressions from PIP output is not an easy problem and requires post processing. Collard et al. [14] show how PIP can be used to find loop bounds and how PIP output can be simplified. Chamski [8] reviews the PIP method of finding loop bounds and gives timing comparisons between the PIP method and the Fourier-Motzkin method.

In this chapter, I present a method published by Le Verge, Van Dongen, and myself [46] to scan parameterized polyhedra using the polyhedral library [88].

3.2 The Polyhedron Scanning Problem

3.2.1 Introduction to Parameterized Polyhedra

This section quickly introduces the concept of parameterized polyhedra to help in the understanding of the polyhedron scanning problem. A polyhedron is defined to be the set of points bounded by a set of hyperplanes. Each hyperplane is associated with an inequality (\(ax \geq b\)) which divides space into two halfspaces: a closed halfspace which satisfies the inequality and an open halfspace which does not. A system of such inequalities induces a polyhedron \(D = \{ x : Ax \geq b \}\) where \(A\) and \(b\) are a constant matrix and vector respectively.

Often, one is interested in describing families of polyhedra \(D(p)\), one polyhedron per instance of the parameters \(p\). This can be done by replacing vector \(b\) above with an affine combination of a set of parameters \(p\). By so doing, one obtains a parameterized polyhedron:

\[
D(p) = \{ x : Ax \geq Bp + b \}
\]

where \(A\) and \(B\) are constant matrices and \(b\) is a constant vector. This parameterized polyhedron can be rewritten in the form of a canonical projection of a non-parameterized polyhedron \(D'\) in the combined
index and parameter space as shown by:

\[ \mathcal{D}(p) = \left\{ x : (A \leftrightarrow B) \left( \frac{x}{p} \right) \geq b \right\} \]

\[ \mathcal{D}' = \left\{ \left( \frac{x}{p} \right) : A' \left( \frac{x}{p} \right) \geq b \right\} \]

### 3.2.2 The Polyhedron Scanning Problem

To generate sequential code for operations and variables declared over polyhedra, a loop nest which scans the given polyhedral region must be generated. The *polyhedron scanning problem* is formally stated as:

Given a parameterized polyhedral domain \( \mathcal{D}(p) \) in terms of a parameter vector \( p \) and a set of \( k \) constraints:

\[ \mathcal{D}(p) = \{ x : Ax \geq Bp + b \} \]

where \( A \) and \( B \) are constant matrices of size \( k \times n \) and \( k \times m \) respectively, and \( b \) is a constant \( k \)-vector, produce the set of loop bound expressions \( L_1, U_1, \ldots, L_n, U_n \) such that loop nest:

```
DO  x_1 = L_1, U_1
    ;
    DO  x_n = L_n, U_n
        body
    END
END
```

will visit once and only once each and every integer point in the domain \( \mathcal{D}(p) \) in lexicographic order of the elements of \( x = (x_1, \ldots, x_n) \).

When talking about a particular loop variable \( x_i \), I use the terminology *outer loops* to refer to loops which enclose the \( x_i \)-loop, that is, the loops of variables \( x_j, j < i \). I use *inner loops* to refer to the loops contained in the \( x_i \)-loop, that is, the loops of variables \( x_j, j > i \).

The problem of finding loop bounds is related to the linear programming problem and shares its complexity. Fortunately, these problems tend to be relatively small (in terms of the dimension and number of constraints) due to the fact that loops are not deeply nested, and exact solutions for typical problems can be found in reasonable time.

### 3.3 Example

Given the parameterized domain defined as:

\[ \{ i, j, k \mid i \geq 0; -i+M \geq 0; j \geq 0; -j+N \geq 0; k \geq 0; i+j-k \geq 0 \} \]

and the context domain \( \{ N, M \mid N > 0; M > 0 \} \) describing what is known to be true a priori, the following four different loop nests (in *Alpha* syntax) were generated by the method described in this chapter. Each loop nest scans the domain in a different order.

<table>
<thead>
<tr>
<th>a. The loop nest in {i, j, k} scan order.</th>
<th>b. The loop nest in {j, k, i} scan order.</th>
</tr>
</thead>
<tbody>
<tr>
<td>{i \mid 0 \leq i \leq M} ::</td>
<td>{j \mid 0 \leq j \leq N} ::</td>
</tr>
<tr>
<td>{j, i \mid 0 \leq j \leq N} ::</td>
<td>{k, j, i \mid 0 \leq k \leq i+j} :: S</td>
</tr>
<tr>
<td>{k, j, i \mid 0 \leq k \leq i+j} :: S</td>
<td></td>
</tr>
<tr>
<td>c. The loop nest in {k, j, i} scan order.</td>
<td></td>
</tr>
<tr>
<td>{k \mid 0 \leq k \leq M+N} ::</td>
<td>{j, k, i \mid 0 \leq j \leq k-M} ::</td>
</tr>
<tr>
<td>{j, k \mid 0 \leq j \leq k-M; j \geq k-M} ::</td>
<td></td>
</tr>
<tr>
<td>{i, j, k \mid 0 \leq i \leq M; i \geq k-j} :: S</td>
<td></td>
</tr>
<tr>
<td>d. The loop nest in {i, k, j} scan order.</td>
<td></td>
</tr>
<tr>
<td>{i \mid 0 \leq i \leq M} ::</td>
<td>{j, k, i \mid 0 \leq j \leq k-M} ::</td>
</tr>
<tr>
<td>{k, i \mid 0 \leq k \leq i+N} ::</td>
<td>{j, k, i \mid 0 \leq j \leq k-M; j \geq k-i} :: S</td>
</tr>
<tr>
<td>{j, k, i \mid 0 \leq j \leq k-M; j \geq k-i} :: S</td>
<td></td>
</tr>
</tbody>
</table>

24
Figure 3.3: Projection using Fourier–Motzkin Elimination

```
for each constraint c1
  for each constraint c2
    if adjacent(c1,c2)
      compute_bound(c1,c2)
    end
  end
end
a. Motzkin method
```

```
for each constraint c1
  for each constraint c2
    compute_bound(c1,c2)
  end
end
remove_redundant_bounds()
```

b. FM method

Figure 3.4: Comparison of Motzkin and Fourier–Motzkin algorithms

3.4 Description of new method

The new method resembles the Fourier–Motzkin (FM) method in that it projects the polyhedron in the canonical direction of inner loop variables in order to eliminate dependencies on them. Thus bounds on a loop are found independent of inner loop indices. However, the FM method considers all pairs of constraints when finding the bound on the projection. In figure 3.3, for example, lower bounds \( lb2 \) and \( lb \) and upper bounds \( ub2 \) and \( ub \) are all considered by the FM method. Given \( n \) constraints, as many as \( \binom{n}{2} \) loop bounds could be generated in eliminating a single variable, and this number grows exponentially with the number of variables. In general, to eliminate \( i \) loop indices from \( n \) constraints, as many as \( 4 \left( \frac{n}{4} \right)^2 \) loop bounds could be generated. Most of these bounds turn out to be redundant (non-tight) and must be eliminated. The elimination of these redundant loop bounds is a significant problem when using the FM method. For a system with \( d \) loop indices and \( n \) constraints, the FM method has worst case time complexity of \( \mathcal{O}\left(\left(\frac{n}{4}\right)^2d^2\right)\).

The proposed method is based on the double description method by Motzkin [59] which only considers pairs of constraints that are adjacent, and therefore never generates any redundant bounds (like \( lb2 \) and \( ub2 \) in the example). Thus the redundant bound elimination phase of the FM method is replaced with an adjacency test on pairs of constraints in the proposed method. Figure 3.4 illustrates the basic difference between the two methods. This Motzkin adjacency test is accomplished very efficiently assuming the rays and vertices of the polyhedron are known. The algorithm maintains both the constraint and the ray–vertex representation of a polyhedron allowing the employ of the adjacency test. Keeping the dual representation is only feasible for small dimensional domains since a \( d \)-polyhedron with \( n \) constraints might have as many as \( d\binom{n}{2} \) rays/vertices, in the worst case. This is also the worst case time complexity. I rely on the fact that computational domains tend to be relatively small polyhedra (in terms of the dimension and number of constraints) due to the fact that loops are not deeply nested.
3.4.1 Add Rays to a Domain

The library function DomainAddRays joins a set of lines, rays, or points to a domain and produces the resulting domain with all redundancies eliminated. It is used in this chapter to eliminate (or project out) the inner loop indices from the bound expressions of outer loops. This is illustrated in figure 3.5 where \( x_i \) is an outer loop variable and \( x_j \) is an inner loop variable. To compute the loop bounds for the \( x_i \)-loop as a function of parameters and outer loop variables, the inner loop variables \( x_j, j > i \) must be removed from the domain. This is done by projecting the domain in the direction of the inner loop variables onto the \( x_i \)-axis, giving \( lb \) and \( ub \) as the lower and upper bounds of \( x_i \), respectively. This can be accomplished using the polyhedral library by adding lines \( \{l_{i+1}, l_{i+2}, \ldots\} \) in the direction of all of the inner loop variables \( \{x_{i+1}, x_{i+2}, \ldots\} \). The resulting polyhedron is a cylinder open in the direction of inner loop variables (as shown in the figure) which has no constraints in terms of the inner loop variables.

3.4.2 Domain Simplify

Consider the two domains \( A \) and \( B \) in figure 3.6. Let the domain \( B \) be called the context. The simplify function finds the largest (in terms of inclusion) domain (or smallest list of constraints) that, when intersected with the context \( B \) is equal to \( A \cap B \). The simplify operation is done by computing the intersection \( A \cap B \) and while doing so, recording which constraints of \( A \) are "redundant" with the result of the intersection. The result of the simplify operation is then the domain \( A \) with the "redundant" constraints removed. In the example in figure 3.6, domain \( A \) is simplified (resulting in domain \( C \)) by eliminating the two constraints that are redundant with context domain \( B \).

The function simplify in context called DomainSimplify in the library is formally defined as follows:

Given domains \( A \) and \( B \) (where \( B \) is called the context): DomainSimplify\( (A, B) = C \), when
\( C \cap B = A \cap B, \ C \supset A \) and there does not exist any other domain \( C' \supset C \) such that \( C' \cap B = A \cap B \). If \( A \cap B \) is empty, then \( C \) is defined to be empty.

3.4.3 Loop Separation

In this section, I respesify the polyhedron scanning problem in terms of polyhedral operations and show how it can be solved using the DomainSimplify and the DomainAddRays procedures described above.

Given a \( d \) dimensional polyhedron \( D \) to be scanned, with an initial context domain \( D_0 \),

Find a sequence of loop domains \( D_1, D_2, \cdots, D_d \) such that

\[
D = D_0 \cap D_1 \cap D_2 \cap \cdots \cap D_d
\]

where each loop domain \( D_i \) is only a function of outer loop variables \( x_j, 1 \leq j \leq i \).

Using the above two library functions, a function can be written which takes a specified domain \( D \) and separates \( ( \) or factors \( ) \) it into an intersection of the initial context domain \( D_0 \) and a sequence of loop domains \( D_1, D_2, \cdots, D_d \) (so that \( D = D_0 \cap D_1 \cap \cdots \cap D_d \) where each loop domain is not a function of inner loop variables.

The loop domain \( D_i, 1 \leq i \leq d \) is computed in two steps. First, the inner loop variables are projected out of the original domain \( D \) by calling DomainAddRays and adding lines \( \{ l_{i+1}, l_{i+2}, \cdots, l_d \} \) in the directions of the inner loop dimensions. The resulting domain is not a function of inner loop variables. This domain is then simplified in context of the initial context domain \( D_0 \) and all of the outer loop domains \( D_1, D_2, \cdots, D_{i-1} \).

Accordingly, the loop domain \( D_i, i \geq 1 \) can be recursively computed as:

\[
D_i = \text{DomainSimplify}(\text{DomainAddRays}(D, \{ l_{i+1}, \cdots, l_d \}), D_0 \cap \cdots \cap D_{i-1});
\]

3.5 Conclusion

I have implemented the procedure described in this chapter, and it is used as part of the ALPHA compiler. It’s difficult to make a fair comparison between the FM method and the method described here, because of the differences in implementation. The FM method programmed by Marc Le Fur [44] is based on CAML, an interpreted functional language, where as the method described here is programmed in C. Testing showed about two orders of magnitude difference in run time, however, this is no doubt due to the implementation differences.

Some examples have been found to cause problems in the polyhedral library. Two different problems have been encountered. The first is an numeric overflow problem. The polyhedral library performs exact rational computation, and numbers are stored using 32 bit integer numerators and denominators. If two rational numbers are multiplied, and there is no cancellation, then the storage requirement for the result is the sum of the storage for the two operands (measured in number of bits). The solution to this problem is to use a multi-precision arithmetic package in which storage grows to meet demand.

The second problem is a memory overflow problem. Given a \( d \) dimensional polyhedron with \( n \) constraints, as many as \( n^{\frac{d}{2}} \) vertices could be required in the dual representation. This effectively limits computation to small dimensional polyhedra. This problem is aggravated by the fact that the current implementation statically allocates a fixed amount of work space to perform a computation. A dynamic work space would be better, in light of this problem.

On the sunnier side, this method has several advantages. First of all, this method produces well minimized results in a convenient form. The implementation is very straightforward, using procedures from the polyhedral library. The time complexity is \( \mathcal{O}(n^{\frac{d}{2}}) \) compared to \( \mathcal{O}(\left( \frac{n}{2} \right)^{2^d}) \) for the Fourier-Motzkin method. The primary reason for the difference in time is that by using the polyhedral library, no redundant bounds are generated, as in the Fourier-Motzkin method.
Chapter 4

Allocation of Memory for Polyhedra

Alpha variables are declared over polyhedral index domains. The compiler must be able to declare storage for these domains. This chapter deals with the issues involved in the allocation of memory to domain variables, which is used in generating declarations for the program variables.

In the simulation or execution of systems of affine recurrence equations, there are two strategies involving variables

1. No variable storage. Variables are recomputed according to their functional definitions each time they are needed. This is not a practical approach, except for variables with infinite domains. The problem with this approach, is that the cost of recomputing can be very high. For example, the time to compute the Fibonacci function is linear when storage is used, but without storage, it takes exponential time to compute.

2. Variable storage. The first time a variable is computed, the result is stored. The single assignment property of systems of recurrence equations ensures that each variable will only ever have one value. If the value of a variable is ever needed again, its value is not recomputed, but is obtained from memory. This technique is known as applicative caching or tabulation.

This chapter deals with ways to facilitate the second strategy by statically allocating memory for variables declared over arbitrary finite convex polyhedral domains.

We may not need to allocate memory for the entire domain of a variable. Often only a projection of the domain (which is also a polyhedron) is necessary (see chapter 9. In this case, techniques developed in this chapter are applied to the allocation of the projected domain.

Definition 8 A variable $X$ of type datatye declared over domain $D$ is defined as

$$X ::= \{ X_i : X_i \in \text{datatype}, i \in D \}$$

(4.1)

where $X_i$ is the element of $X$ corresponding to the point $i$ in domain $D$.

To map an $n$-dimensional polyhedron to memory (which is 1-dimensional), two things must happen. First of all, enough memory to represent the polyhedron must be reserved or allocated. The amount of memory needed is proportional to $\text{volume}(D)$ which is the number of integer points in domain $D$. Second, a function which maps each point in the polyhedron to a memory location must be found. A memory index function, $\text{index}() : \mathbb{Z}^n \rightarrow \mathbb{Z}$, maps each point in a domain $D$ to a unique integer in the set $\{n : n \in \mathbb{Z}, 0 \leq n < \text{volume}(D)\}$. This function is usually not unique.

If a vector of length $\text{volume}(D)$ starting at some address base in memory is allocated to store variable $X$ (as defined in equation 4.1), then the amount of memory (in bytes) needed is

$$\text{size of}(X) = \text{size of (datatype)} \times \text{volume}(D)$$

(4.2)
and the memory address of an element $X_i$ of variable $X$ is:

$$\text{address}(X_i) = \text{base} + \text{size of datatype} \times \text{index}(i)$$

(4.3)

Complicated $\text{index}(i)$ functions result in complicated memory addressing functions, and since a variable’s addressing function must be evaluated each time the variable is indexed, the complexity of the entire simulation is affected. It is therefore very desirable to limit, as far as possible, the complexity of the $\text{index}(i)$ function.

We limit the $\text{index}(i)$ function to be an affine function of the indices of $D$ by allocating enough memory to store the smallest rectangular domain (which we will call the bounding box) containing the variable. Figure 4.1 shows (on the left) a domain $D$ and (on the right) $D$ inside its bounding box $D_{BB}$. The cost of having an affine index function is that more memory is allocated for a variable than may be necessary. The difference in the volume of the bounding box domain and the volume of the variable domain is the amount of unusable memory that needs to be allocated in order to have the benefits of an affine index function.

Rectangular domains have the following form:

$$D = \{ x : x \in \mathbb{Z}^n, \ x_1 \geq l_1, \ x_1 \leq u_1 \}
\vdots
\{ x_n \geq l_n, \ x_n \leq u_n \}$$

(4.4)

where $x_i$ is the $i^{th}$ index (coordinate) of the point $x$, and where $l_i$ and $u_i$ are constant scalars for $i = 1 \cdots n$, with $n$ being the dimension of the domain.

A bounding box of $D$ is defined as the smallest rectangular domain $D_{BB}$ which contains $D$. Given a finite domain $D$, the scalars $u_i$ and $l_i$ of a bounding box $D_{BB}$ can be computed. By the dual definitions of a polyhedron, $D$ is defined implicitly using a set of constraints (rows of matrix $A$) and parametrically using a set of vertices (columns of matrix $R$), as follows:

$$D = \{ x : x \in \mathbb{Z}^n, \ Ax \geq b \}
= \{ x : x \in \mathbb{Z}^n, \ x = R\mu, \ \mu \geq 0 \}$$

(4.5) (4.6)

where $A$ and $R$ are constant rational matrices and $x$ and $\mu$ are rational vectors.

The polyhedral library [88] is able to compute the $R$ matrix given the constraint matrix $A$. The columns of the $R$ matrix represent the $k$ vertices of the polyhedron. To bound a finite polyhedron, it is sufficient to bound its vertices. The rational constants $l_i$ and $u_i$ are therefore defined as the minima and maxima, respectively, of the $k$ values $r_{ij}$’s in the $i^{th}$ row of $R$:

$$l_i = \min(r_{i1}, r_{i2}, \cdots, r_{ik}), \ 1 \leq i \leq n$$
$$u_i = \max(r_{i1}, r_{i2}, \cdots, r_{ik}), \ 1 \leq i \leq n$$

(4.7)

The bounding box $D_{BB}$ is defined in terms of $l_i$ and $u_i$.

$$D_{BB} = \{ x : x \in \mathbb{Z}^n, \ l_i \leq x_i \leq u_i, \ 1 \leq i \leq n \} \supseteq D$$

(4.8)
One can define index() and volume() functions for a rectangular domain in terms of a vector \( w \) whose elements \( w_i \) are defined recursively in terms of \( l_i \) and \( u_i \) is defined as:

\[
    w_1 = \begin{cases} 
        1 & \text{when } i = 1 \\
        w_{i-1}(u_{i-1} \leftrightarrow l_{i-1} + 1) & \text{when } 1 < i \leq n + 1 
    \end{cases}
\]  

The scalar \( w_i \) is the volume of the domain in the first \( i \leftrightarrow 1 \) dimensions. The index and volume functions are then defined in terms of \( w \):

\[
    \text{index}(i) = \begin{pmatrix} 
        w_1 \\
        w_2 \\
        \vdots \\
        w_n
    \end{pmatrix} \circ (i \leftrightarrow l) 
\]

\[
    \text{volume}(D) = w_{n+1}
\]

where \( n \) is the dimension of \( D \) and the operation \( \circ \) is a vector dot product.

### 4.1 Reducing the memory allocation

As we pointed out, this scheme allocates more memory than is used for some non-rectangular domains. We asked ourselves whether we could reshape a domain with a unimodular transformation, so as to reduce the size of the bounding box, and thus the amount of memory allocated. This problem can be stated formally in the following terms:

**Given a domain** \( D \),

**Find a linear unimodular transformation** \( T : D \to D' \)

**such that the volume of the bounding box of** \( D' \) **is minimized.**

where \( T(x) = Tx \) is a linear function mapping every point \( p \) in \( D \) to a point \( p' = Tp \) in \( D' \). \( T \) is an \( n \) by \( n \) matrix.

This problem turns out to be a non-linear optimization problem. I discussed this problem and gave a heuristic to solve it in [89]. The method yielded the smallest possible memory size in all cases tried, however, the method was not proved optimal. In this method, \( T \) is written as the product of \( n \) unimodular skews, (one for each of the \( n \) dimensions), and an unimodular scale:

\[
    T = S \times T_n \times \cdots \times T_2 \times T_1
\]

where each \( T_i \) is a unimodular skew and \( S \) is a unimodular scale. A unimodular scale is a rational diagonal matrix whose determinant is one. A skew transformation \( T_i \) which minimizes the size of a bounding box in the \( i \)th dimension may be found by setting up a linear programming problem. By finding the \( n \) skews, and then computing the necessary scaling matrix \( S \) to make \( T \) an integral matrix, a matrix \( T \) may be found.

In practice, most domains are rectangular, in which case memory allocation is optimal, or else they are right triangular, in which case this memory reduction technique does not reduce the memory allocation.

### 4.2 Open Problem

I have not treated the problem of finding an allocation for a parameterized domain. Such an allocation would be a non-linear function of the parameters, where for different ranges of parameters, there might be different volume and index functions. This is left as an open problem.
4.3 Conclusion

In this chapter, I presented a method for allocating linear storage for polyhedron-based variables, where I limit the indexing function to be an \textit{affine} function of the indices. This is done by allocating enough memory to store the smallest rectangular domain containing the polyhedron, which may mean allocating more memory than needed to store the values in the polyhedron. I then showed how to compute the allocation length of the memory array and the memory index function which maps points in the polyhedron to elements stored in memory.
Chapter 5

The Naive Execution of ALPHA

5.1 Introduction

Even though ALPHA programs tend to be short and intuitive, their naive execution can often be inefficient in both speed and the use of memory. In generating the naive code to execute a set of recurrence equations, we have encountered some of the same issues that are found in the compilation of arrays in functional languages. Thus, some of the same methods which are used for compiling functional languages can be used for the compilation of ALPHA.

In this chapter, I demonstrate an automatic translation scheme which compiles ALPHA into the imperative language C. I described this translator in a recent paper [69]. The translator attempts to produce code which is as efficient as possible, given the fact that no execution schedule is available (which is a worse case scenario). Two problems needed be solved in order to implement the translator. In ALPHA, variables are arrays based on polyhedral index spaces. These arrays must be stored in linear memory. Thus, we need to know how much memory to allocate to a polyhedron and then how to map each point in the polyhedron to its unique memory location. This problem was discussed in chapter 4.

We also need to be able to scan the polyhedron, accessing the memory location corresponding to each point in the polyhedron. This can be done with a set of nested loops with piecewise affine upper and lower bounds. This problem was addressed in chapter 3.

5.2 The ALPHA Translator

In order to enable a user to debug an initial specification and to validate the operation of an ALPHA program, we investigated making ALPHA executable by means of a naive compiler. I encountered many issues that arise in the general compilation of functional languages, particularly in the efficient implementation of arrays.

In functional languages, arrays are treated as either incremental or monolithic [86]. Programmers may view incremental arrays as a conventional (and hence, persistent) data structure, manipulated explicitly with an update operation, whose semantics denote a function that returns a new array, thereby retaining referential transparency. If implemented naively, incremental arrays require unacceptably high amount of copying [34], and there has been work to ameliorate this [34, 80]. On the other hand, monolithic arrays [86] consider an array to be simply a map from indices to values. This leads to an elegant programming style which remains declarative and is easy to use. Given that every expression in ALPHA denotes a function from a polyhedral domain to a value, it is natural to consider a program as a collection of monolithic arrays (one for each variable), and its main body as a specification of the “filling functions”.

A standard approach to efficient implementation of recursive programs is tabulation [4, 28], also called applicative caching [37]. Here, recomputation is avoided by maintaining a table (or cache) of previously computed values. This makes the underlying implementation imperative, while preserving
a purely applicative interface to the user (i.e., referential transparency is retained). This technique is particularly attractive when all the possible arguments to the function, are known statically. It is thus a good method for implementing monolithic arrays: one simply allocates an empty table of the appropriate size, and fills it as needed. This enables the compiler to generate code (albeit naïve) when no scheduling information is available.

In producing C-code, the translator follows the classic demand driven evaluation strategy used in functional program languages. At the top level, the following is done:

- Allocate space for every instance of every variable.
- Create a function for each variable which evaluates the variable at specific index point. This function is computed using demand driven evaluation:
  - If the value has been previously computed and stored in the table (tag is set), then return the stored value; otherwise
  - Evaluate the arguments by (recursively) calling the appropriate evaluation function for each argument.
  - Apply the filling function, i.e., compute the RHS of the Alpha equation.
  - Store the result in the table (and set tag bit) and also return the value.
- Evaluate each output variable at each index point in its respective domain.

The code first allocates storage (arrays or tables) for each finite variable, the size of which is determined by the variable’s domain. I discussed how this is done in chapter 4. The code then fills up this storage by traversing each index point in the domain of each output variable, and at each point, computing the value using a demand-driven evaluation. This is called scanning a polyhedron [1] (the order of scanning being the lexicographic order). I discuss how to scan polyhedra in chapter 3.

This naïve code suffers from two drawbacks, namely the overhead of context switching for the demand driven evaluation, and memory inefficiency. Also, if the program has infinite domains, we need dynamic memory management, and may not be able to completely avoid reevaluation. This remains an open problem.

I illustrate the translation strategy using the forward substitution program shown in figure 5.1. The Alpha program was translated into a C-code program using the command line:

```
read_alpha < fs.alpha | write_c -p 100 > fs.c
```

During the first step, the Alpha file `fs.alpha` is parsed by the `read_alpha` program, producing an abstract syntax tree (AST). This AST is then translated by the `write_c` program into C-code, while setting the parameter `N` to 100 with the switch `-p 100`. The resulting C-program is shown in figure 5.2. This program, in fact, compiles and executes solving a lower triangular system of 100 equations.

### 5.3 Improvements over naïve code generation

In generating naïve code, we have made no assumptions about the source Alpha program, other than it should parse correctly. It is clear that naïve code generation can be viewed as a simplistic compiler for the Alpha language. Starting from this, we ask ourselves what optimizations we can make to improve the quality of the compilation. Here, I discuss some of the optimizations which can be made.

If a schedule can be statically determined, then domains can be scanned in an order consistent with this schedule, without any context switching [66]. Finding a schedule for an Alpha program is a generalization of the scheduling problem in systolic synthesis [70, 67] to multidimensional, variable-dependent schedules [24], and can be cast in terms of linear programming. A subtle point to note is that the scheduling problem is undecidable in general [79], and hence the naïve demand driven code has to be a fallback position for the compiler. When equations are executed in an order determined by a valid schedule, computations are placed (in execution order) before instructions which use the resulting values, and there is no need to do expensive context switching. Thus, much more efficient code can be generated.
-- Forward Substitution Problem (Parameterized)
--
-- [ A11 0 0 ... 0 ] [ X1 ] [ B1 ]
-- [ A21 A22 0 ... 0 ] [ X2 ] [ B2 ]
-- [ A31 A32 A33 ... 0 ] [ X3 ] = [ B3 ]
-- [ ... ] [ ... ] [ ... ]
-- [ An1 An2 An3 ... Ann ] [ Xn ] [ Bn ]
--
-- Given A, B; Solve for X.
--

system ForwardSubstitution : { N | N>0 }
  (A : {i,j | 1<=i<=N; 1<=j<=i} of integer;
   B : {i | 1<=i<=N} of integer)
  returns (X : {i | 1<=i<=N} of integer);
var
  b : {i,j | 0<=i<=N; 1<=j<=i} of integer;
let
  b[i,j] = case
    { | j=0 } : 0;
    { | j>0 } : A[i,j] * X[j] + b[i,j-1];
  esac;
X[i] = ( B[i] - b[i,i-1] )/A[i,i];
tel;

Figure 5.1: ALPHA program example: Forward Substitution

A consequence of scheduling analysis is that if a schedule exists, we can easily detect parallelism which exists when the dimension of the schedule is strictly smaller than the dimension of the system. In such a case, we may partition the domains of different variables and assign each partition to a different processor. A static analysis of the communication [73] is needed to choose the best partitioning. In effect, we aim to be able to generate code that is as efficient as parallel imperative code [65].

Naively generated code is memory inefficient, since each table entry is written exactly once. Knowing a schedule, one can perform static lifetime analysis to determine whether and when memory can be reused (see chapter 9). Once again, this analysis can be cast in terms of (linear and nonlinear) optimization problems using the polyhedral domains and affine dependencies of ALPHA. This is a generalization of the linear allocation functions used in systolic array synthesis.

5.4 Conclusion

I have shown how an ALPHA program can be naively implemented by viewing it as a set of monolithic arrays and their filling functions, implemented using applicative caching. This is the general approach which was used by the translator.

Storage is allocated for domain based variables using the techniques introduced in chapter 4. An affine indexing function is synthesized for each variable, which maps each point in its domain to an index in the memory vector.

To generate sequential code which evaluates all elements of a variable, a loop nest which scans the domain of the variable must be synthesized. In chapter 3, I described the polyhedron scanning problem which finds a set of nested loops which visit once and only once all integer points in a polyhedron in a lexicographic order.
/* system ForwardSubstitution */
typedef struct { int value; int computed; } intvar;
typedef struct { int value; int computed; } boolvar;
typedef struct { double value; int computed; } realvar;
double atof(); int atoi(); char _s[32];
#define N 100 /* --parameters */
intvar A[10000]; /* --input variables */
int compute_A(i,j)
int i,j;
{ intvar *tmp; tmp =&(A[i+100*j-101]);
if (!tmp->computed) { printf("Input A[%d,%d] =", i,j); gets(_s);
    tmp->value = atoi(_s); tmp->computed = 1; }
    return tmp->value;
}
intvar B[100];
int compute_B(i)
int i;
{ intvar *tmp; tmp =&(B[i-1]);
if (!tmp->computed) { printf("Input B[%d] =", i); gets(_s);
    tmp->value = atoi(_s); tmp->computed = 1; }
return tmp->value;
}
intvar X[100]; int compute_X(); /* --output variables */
intvar b[10000]; int compute_b(); /* --local variables */
int compute_b(i,j) /* --let equations */
int i,j;
{ intvar *tmp; tmp =&(b[i+100*j-101]);
if (!tmp->computed) {
    tmp->value = ( j==0 ) ? ( 0 ) :
                    ( j-1==0 ) ? ( compute_A(i,j) * compute_X(j) + compute_b(i,j-1) ) :
                                        ( printf("% error"), exit(-1));
    tmp->computed = 1;
return tmp->value;
}
int compute_X(i)
int i;
{ intvar *tmp; tmp =&(X[i-1]);
if (!tmp->computed) {
    tmp->value = (compute_B(i) - compute_b(i,i-1)) / compute_A(i,i);
    tmp->computed = 1;
return tmp->value;
}
int main()
{ int i;
    for (i=1; i<=N; i++) { printf("X[%d] = %d\n", i, compute_X(i)); }
}

Figure 5.2: C program example for Forward Substitution
Chapter 6

Analysis of $\text{ALPHA}$

In this chapter, I discuss the application of systolic analysis to the compilation of $\text{ALPHA}$ programs. The $\text{ALPHA}$ language restricts the class of programs it can represent to those that only use affine dependencies. This restriction gives $\text{ALPHA}$ the analyzability needed to facilitate the compilation process. In this chapter, I show that the same methods which are used for solving regular array synthesis problems can be applied to the compilation of $\text{ALPHA}$ as a functional language. In order to find timing and allocation functions, or map registers and communication links to hardware, all domains and dependencies must first be statically known. Many of the above problems can be cast in terms of doing operations on polyhedra and solving linear programming problems. $\text{ALPHA}$ was designed to facilitate these kinds of analyses. The kinds of analysis needed to generate efficient code for array based functional programs is in fact a generalization of scheduling techniques used in systolic array synthesis.

Naive code, which can be generated as shown in the previous chapter, suffers from two drawbacks, namely the overhead of context switching for the demand driven evaluation, and memory inefficiency. The compilation can be viewed as a transformation process to incrementally improve this naive code. The following analysis questions are posed:

- **Can a schedule be statically determined?** If so, the domains can be scanned in an order consistent with this schedule, without any context switching. This problem is a generalization of the scheduling problem in systolic synthesis [70, 67] to multidimensional schedules [24], and can be cast in terms of linear programming and polyhedral problems.

- **An immediate consequence of scheduling analysis is that if a schedule exists, we can immediately detect any parallelism** (the dimension of the schedule is strictly smaller than the dimension of the system). In such a case, we may partition the domains of different variables and assign each partition to a different processor. A static analysis of the communication is needed to choose the best partitioning. This can also be cast into the framework of linear programming and polyhedra [73].

- **Regardless of whether parallelism is possible or not,** the code that is naively generated is memory inefficient, since each table entry is written exactly once. Knowing a schedule, one can perform static lifetime analysis to determine whether and when the memory can be reused. Once again, this analysis can be cast in terms of (linear and nonlinear) optimization problems using the polyhedral domains and affine dependencies of $\text{ALPHA}$. This is a generalization of the linear allocation functions used in systolic array synthesis.

Using dependency analysis, an execution schedule giving relative times for computation of variables can be computed. Given this schedule, and using a transformational approach, $\text{ALPHA}$ statements are separated, regrouped and reordered according to their scheduled execution times. When program statements are executed in an order consistent with the schedule, context switching can be avoided.

In this chapter, I illustrate the above compilation strategy using the forward substitution program shown in figure 6.1.
**Figure 6.1: ALPHA program for Forward Substitution**

### 6.1 Dependency Analysis

In his PhD thesis, Kuhn [40] made use of dependence analysis and the dependence graph. Feautrier [21] explains how a simple imperative language program (consisting only of assignments, for loops with affine loop limits, and arrays with affine index expressions), can be statically analyzed to find the flow dependencies.

An important concept used in ALPHA is the notion of a *dependency*, which is an affine index function mapping the index domain of the LHS variable to the index domain of a RHS variable. Syntactically, a dependency function is written as:

\[
(\text{index}, \text{index}, \ldots \rightarrow \text{index-expr}, \text{index-expr}, \ldots)
\]

where each index-expr is an affine expression of the indices to the left of the arrow and of system parameters. Examples of affine dependency functions are \((i \rightarrow i-2)\) and \((2z, i, k \rightarrow z-1, 2z-k-1, k-1)\). An example of an equation which explicitly uses affine functions is:

\[
X = B \cdot (i \rightarrow -1) - b \cdot (i \rightarrow i-1)\; \text{or equivalently, using the so-called array notation of ALPHA: } X[i] = B[i] - b[i, i-1];\]

ALPHA restricts the kinds of programs which can be represented to those where the index domains of variables are unions of convex polyhedra, and data dependencies are affine index functions.

During the first step, ALPHA is read and parsed, producing an AST which is read into a symbolic algebra system where it is analyzed and transformed symbolically. The static analysis of the AST is simplified by employing the *dependency table*, which is extracted directly from the AST and which enumerates the flow dependencies between variables. The dependency table holds the same information as the data flow graph used in [87]. The notation for a dependency \(D: A[p] \rightarrow B[Mp]\) means that for all \(p\) in the index domain \(D\), the computation of \(A\) at \(p\) depends on \(B\) at \(Mp\) where \(M\) is an affine mapping matrix. Accordingly, the domain \(D\) must be a subset of the domain of \(A\) and Image\((D, M)\) must be a subset of the domain of \(B\). The dependency table for the forward substitution example is shown below.

\[
\begin{array}{ll}
\{i, j \mid 2 \leq j, 1 \leq i \leq N\} & : b[i,j] \rightarrow b[i,j-1] \\
\{i, j \mid 2 \leq j, 1 \leq i \leq N\} & : b[i,j] \rightarrow A[i,j] \\
\{i, j \mid 1 \leq i < N\} & : b[i,j] \rightarrow X[j] \\
\{i \mid 1 \leq i < N\} & : X[i] \rightarrow b[i, i-1] \\
\{i \mid 2 \leq i < N\} & : X[i] \rightarrow b[i, i] \\
\{i \mid 1 \leq i < N\} & : X[i] \rightarrow A[i, i] \\
\end{array}
\]
6.2 Usage Analysis

In this section, I describe the usage table and show how it is constructed. The usage table is fundamental and important in doing lifetime analysis for determining memory reuse, as well as doing communication analysis. The construction of the usage table given below is a major contribution of this thesis and it underscores how the polyhedral model can be used as a powerful analysis tool.

Whereas dependency analysis asks the question: “What values are needed to do this computation?”, usage analysis asks the opposite question: “What computations need to use this value?” Dependency analysis is thus the consumer point-of-view, while usage analysis is the producer point-of-view.

I start with the dependency table which is obtained almost directly from the Alpha program. Each entry in that table infers a dependency relation (written “→” and read “depends on”) from a variable on the LHS of an equation, qualified by a subdomain, to a variable on the RHS. Given the dependency table, one can derive the usage table. Each entry in the usage table gives a usage relation (written “⇒” and read “is used by”) that shows where a particular subdomain of a variable is used. This section describes a method to construct the usage table.

Given a dependence of $X$ on $Y$

$$
\forall p \in D : X[p] \Rightarrow Y[Mp]
$$

we seek to find the usages of $Y$ by $X$ by inverting the affine transformation $M$ in the context of domain $D$, that is find an inverse matrix $M'$ such that $\forall p \in D : M'Mp = p$.

There are two cases to consider. Le Verge [48] showed that $M'$ exists in the context of a domain with lineality space $Ap = 0$, if and only if the right Hermite normal form of the matrix $\begin{bmatrix} A \\ M \end{bmatrix}$ is $\begin{bmatrix} Id \\ 0 \end{bmatrix}$, that is to say, there exists a unimodular matrix $U$, such that $\begin{bmatrix} A \\ M \end{bmatrix} = U \begin{bmatrix} Id \\ 0 \end{bmatrix}$. The lineality space is characterized by a (non-unique) matrix $A$, such that lin.space$D = \{ p \in D : Ap = 0 \}$. $M'$ can be computed by inverting $U$ and extracting out the submatrix which when multiplied by $M$ gives $Id$.

However, if the matrix $\begin{bmatrix} A \\ M \end{bmatrix}$ is not full column rank, then the dependency table entry is a many-to-one relation. The inverse usage table entry will correspond to a usage of one value by a whole subdomain of dependent elements. To find usage table entries in this case, I extend the results of Le Verge.

6.2.1 Construction of the Usage Table

To find the usage table entry corresponding to the dependence: $\forall p \in D : X[p] \Rightarrow Y[Mp]$, there are two cases to consider.

Case 1: One-to-one

The matrix $\begin{bmatrix} A \\ M \end{bmatrix}$ is full column rank and $\begin{bmatrix} A \\ M \end{bmatrix} = U \begin{bmatrix} Id \\ 0 \end{bmatrix}$ where $U$ is unimodular and $Id$ is the identity matrix. Refer to figure 6.2-a. This is the case for most one-to-one transformations. $M'$ can be computed by the Le Verge method described above. Then, having computed $M'$ the usage can be stated as:

$$
\forall q \in D' : Y[q] \Rightarrow X[M'q]
$$

where $D' = \text{Image} (D, M)$.

Case 2: Many-to-one

The array $\begin{bmatrix} A \\ M \end{bmatrix}$ is not full column rank. This happens when many instances of $X$ depend on a single instance of $Y$. Refer to figure 6.2-b. We thus seek to find the set of points where the computation of $X$ depends on $Y[p], p \in D$. We will call this domain $E(q)$. It is constructed so that:

$$
\forall p \in D : Mp = q \iff \exists r \in E(q) : \tilde{M}p = \begin{bmatrix} q \\ r \end{bmatrix} \quad \text{and} \quad \tilde{M}' \begin{bmatrix} q \\ r \end{bmatrix} = p
$$
The function $M : p \in D \mapsto D' = \text{Image}(D,M)$ is many-to-one and does not have an inverse $M'$. So instead, we search for a corresponding one-to-one function $\tilde{M} : p \in D \mapsto \begin{bmatrix} r \\ q \end{bmatrix} \in E = \text{Image}(D,\tilde{M})$ where $\tilde{M}$ is a dimensional expansion of $M$ and there exists a canonical projection $\Pi$ such that $\Pi \tilde{M} = M$. $E$ is the non-parameterized polyhedron corresponding to $E(q)$ and is obtained as follows:

- If $E(q) = \{r \mid Cr + Dq + e \geq 0\}$ for all $q \in D' = \{q \mid Aq + b \geq 0\}$

then $E = \left\{ \begin{bmatrix} q \\ r \end{bmatrix} \mid \begin{bmatrix} C & D \\ 0 & A \end{bmatrix} \begin{bmatrix} q \\ r \end{bmatrix} + \begin{bmatrix} e \\ b \end{bmatrix} \geq 0 \right\}$.

Since $\tilde{M}$ is one-to-one, we can find the inverse matrix $\tilde{M}'$ such that $\forall p \in D : \tilde{M}' \tilde{M}p = p$. Then the dependence $\forall p \in D : X[p] \rightarrow Y[Mp]$ has the usage:

$$\forall q \in D' : Y[q] \Rightarrow \forall r \in E(q) : X[\tilde{M}' \begin{bmatrix} r \\ q \end{bmatrix}]$$

where $D' = \text{Image}(D,M)$ and $\text{Preimage}(D',\Pi) \cap E = \text{Image}(D,\tilde{M})$, which leads to $E(q) = \text{DomainSimplify}(\text{Image}(D,\tilde{M}),\text{Preimage}(D',\Pi))$. The domain functions $\text{Preimage}$ and $\text{Image}$ are defined in section 2.5 and the $\text{DomainSimplify}$ function is defined in section 3.4.2.

The function $\tilde{M}$ is constructed by augmenting the matrix $M$ with a basis of the kernel of $\begin{bmatrix} A \\ M \end{bmatrix}$ as follows:

$$B = \ker\text{basis}\left(\begin{bmatrix} A \\ M \end{bmatrix}\right)$$

$$\tilde{M} = \begin{bmatrix} B \\ M \end{bmatrix}$$

The $B$ is not necessarily unique.

Having computed $\tilde{M}$, the following are shown to be true:

1. $\tilde{M}$ is invertible in the context of $D$. This follows from the fact that the matrix $\begin{bmatrix} A \\ \tilde{M} \end{bmatrix}$ is full column rank.
2. There exists a canonical projection $\Pi$ such that $\Pi \tilde{M} = M$. This follows from:

$$\Pi \tilde{M} = \begin{bmatrix} B \\ M \end{bmatrix} = M$$

and thus $\Pi = \begin{bmatrix} 0 & Id \end{bmatrix}$.

3. Image$(\mathcal{D}, \tilde{M})$ is a higher dimensioned domain containing the domain $\mathcal{D}'$ and the former can be canonically projected onto the latter: Image$(\text{Image}(\mathcal{D}, \tilde{M}), \Pi) = \mathcal{D}'$.

**Example** Below is the usage table derived from the dependency table for the forward substitution example.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 &lt;= i &lt; N-1</td>
<td>x[i]</td>
<td>a[i]</td>
<td>b[i]</td>
</tr>
<tr>
<td>1</td>
<td>1 &lt;= i &lt; N</td>
<td>b[i]</td>
<td>a[i]</td>
<td>x[i]</td>
</tr>
<tr>
<td>1</td>
<td>i = j+1, 1 &lt;= j &lt; N-1</td>
<td>a[i, j]</td>
<td>x[i, j]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i = j, 1 &lt;= j &lt; N</td>
<td>a[i, j]</td>
<td>x[i, j]</td>
<td></td>
</tr>
</tbody>
</table>
```

As indices take on meanings of time, processor space, and memory space — entries in the usage table can be reinterpreted as data communications. Entries in the usage table become statements in the LACS language [74] and one can use the tools developed by Rajopadhye for LACS to analyze and classify each communication.

### 6.3 Schedule Analysis

One of the most important kinds of analysis that can be done on an ALPHA program is dependency analysis for finding a schedule. Methods for finding schedules have been well studied and are still the subject of current research in the systolic array community. In this thesis, I simply use these results.

The goal of scheduling is to find a *timing function* for each variable which maps every index point in the variable’s index space to a positive integer representing a relative “execution time”. The timing function $t_X(p)$ gives the relative execution time when the recurrence equation to compute the value $X[p]$ is evaluated.

All timing functions must respect two important rules. First, the *causality rule* states that if variable $A$ at index point $p$ is dependent on variable $B$ at index point $q$ (written $A[p] \leftrightarrow B[q]$) then $t_A(p) > t_B(q)$, that is, the computation of the $A[p]$ follows the computation of $B[q]$. A second rule ensures that there is a “beginning time”, and states that for any timing function, $t_X(p) \geq 0$ for all $p \in \text{domain}(X)$, that is, no value is computed before time 0.

Early work concentrated on finding *affine* timing functions which are of the form $t(p) = \lambda^T p + \alpha$ (where $\lambda$ is a vector constant and $\alpha$ is a scalar constant) for programs with uniform dependencies [63] and later for affine dependency programs [75]. Recently, Feautrier [24] showed how to schedule recurrence equations when no affine schedule can be found. He extended scheduling analysis to problems whose schedules are polynomial, rather than affine. Affine timing functions are generalized to *multidimensional timing functions* which map each index point in a variables index space to a *vector* of positive integers representing execution time. Temporal ordering is then the lexicographic order of these time vectors, and the causality rule is restated that if $A[p] \leftrightarrow B[q]$ then $t_A(p) > t_B(q)$, where $>$ means “lexicographically greater”. Feautrier showed that multidimensional affine schedules are equivalent to polynomial schedules in one dimensional time.

One of the objectives of this dissertation is to show how the results of scheduling can be used for the compilation of functional languages. The key idea is that when recurrences are executed in an order determined by a valid schedule, all computations are guaranteed to be performed before instructions which use the resulting value, and there is no need to do expensive context switching. Thus, much more efficient code can be generated.

For the forward substitution algorithm (Example 2, Figure 6.1), schedules for variables $b$ and $X$ are:

- $t_b(i, j) = 2j$ and $t_X(i) = 2i \Leftrightarrow 1$ for a minimum system latency of $2N \Leftrightarrow 1$, $(0 \leq t \leq 2N \Leftrightarrow 2)$. The inputs are all assumed to available at $t = 0$. Alternately, the schedules $t_b(i, j) = i + j$ and $t_X(i) = 2i$ also
yield the same latency. Usage analysis shows that the first schedule requires a one to many broadcast communication \( X[i] \rightarrow b[i,j], i < j < N \), where as the second schedule allows \( X[i] \) to be propagated to \( b[i,j], i < j < N \) sequentially in a systolic fashion. If the target architecture better supported row broadcasting than chained sequential communication, the first schedule would be chosen, otherwise, the second schedule would be chosen. Here, we choose the first. An intermediate local variable\(^1\) \( x \) is introduced and is placed between the variable \( X \) and its computation. Then, the change of basis \((i \rightarrow 2i)\) is performed on variable \( x \) and \((i, j \rightarrow 2j + 1, i)\) is performed on variable \( b \) to incorporate their respective schedules. The time index variable is renamed to \( t \) to remind us that it is now a temporal dimension. The resulting \( \text{ALPHA} \) code is shown in figure 6.3. Causality can be quickly hand checked by observing that every dependency is now of the form \((t, \cdots \rightarrow t \rightleftharpoons i, \cdots)\).

\[
\text{system ForwardSub} : \{N \mid N>1\}
\begin{align*}
&\text{returns} \quad (X : \{i \mid 1\leq i \leq N\} \text{ of real }) \\
&\text{var} \\
&\begin{array}{c}
x : \{Z,t \mid 2Z=t+1; 1\leq t\leq 2N-1\} \text{ of real}; \\
b : \{Z,t,i \mid 2Z=t; 0\leq t\leq 2i-2; 2\leq i\leq N\} \text{ of real};
\end{array}
\end{align*}
\begin{align*}
&\text{let} \\
&X[i] = x[i,2i-1];
&b[Z,t,i] = \text{case}
&\begin{array}{l}
\{Z=0\} : 0\square; \\
\{Z>1\} : b[Z-1,t-2,i] + A[i,Z] \times x[Z,t-1];
\end{array}
&\text{esac;}
\end{align*}
\begin{align*}
x[Z,t] = \text{case}
&\begin{array}{l}
\{Z=2\} : (B[Z] - b[Z-1,t-1,Z]) / A[Z,Z]; \\
\end{array}
&\text{esac;}
\end{align*}
\text{tel;}
\]

Figure 6.3: Forward Substitution example after scheduling

### 6.4 Alignment and Virtual Allocation

The alignment problem is to find a mapping from the domain of each variable and computation to a common virtual domain for the entire system. Thus the variables are aligned with respect to each other and are placed on a common grid. Alignment affects communication and thus, can be done with a goal of reducing communication.

Li and Chen [49] worked to find a set of suitable alignment functions to embed the index domains of variables into a common index domain so as to minimize the cost of data movement. The problem was shown to be \( \text{NP-complete} \) for an arbitrary nest of dimension \( 2 \) and a greedy heuristic based on bipartite graph matching was given. Later, Feautrier [22] addressed the problem of mapping affine loop nests onto distributed memory architectures. The main idea was to zero out edges corresponding to the greatest communication volume, using a heuristic to estimate the communication volume. A greedy strategy is used to select the edges to be zero-ed out. Most recently, Darte and Robert [15] dealt with the mapping of data and computation to a virtual processor grid, reducing non-local communication as much as possible. It was shown to be \( \text{NP-complete} \). Heuristics that work in most practical cases were derived.

\(^1\)Since \( X \) is an output, and we cannot do a change of basis on it without changing the system definition.

\(^2\)The index \( Z \) is induced because the change of base is non-unimodular. It is a redundant temporal index \((= \lfloor \frac{t}{2} \rfloor)\), but necessary to keep the system affine.
A greedy strategy for doing alignment is to enumerate all of the communications using the usage table. Then identify which communications could be eliminated by a spatial alignment of a variable. A communication is eliminated when the source and destination are the same spatially. Determine the cost of each communication, and eliminate the most costly communication. Repeat this process until no other communications can be eliminated and all variable are aligned.

For the forward substitution algorithm (Example 2, Figure 6.1), the variable $x$ is a one-dimensional variable and $b$ is a two-dimensional variable. During alignment, these two variables are placed on a common two-dimensional grid. This is illustrated in figure 6.5. We map $b$ to the virtual grid using the identity function $(i,j \rightarrow i,j)$. We place $x$ on the virtual grid with $b$. The dependence $x[Z,t] \rightarrow b[Z \equiv 0, t \equiv 1, Z]$ suggests that $(Z,t \rightarrow Z,t,Z)$ mapping $x$ to a diagonal would be a good change of basis, since it aligns $x$ with the points on the virtual grid with which it communicates (see figure 6.5). After alignment, the resulting ALPHA program is shown in figure 6.4.

```
system ForwardSub : \{N \times 1\}
  ( A : \{i,j\} 1\leq i,j\leq N\} of real;
  B : \{i\} 1\leq i\leq N\} of real )
returns ( X : \{i\} 1\leq i\leq N\} of real )

var
  x : \{Z,t,i\} Z=t+1; \{Z=1\} 1\leq i\leq N\} of real;
  b : \{Z,t,i\} Z=0; 0\leq t\leq 2i-2; 2\leq i\leq N\} of real;

let

X[i] = x[i,2i-1,i];

X[i] = case
  \{ Z=0 \} : 0;
  \{ Z=1 \} : b[Z-1,t-2,i] + A[i,Z] * x[Z,t-1,Z];
  esac;

b[Z,t,i] = case
  \{ Z=0 \} : 0;
  \{ Z=1 \} : A[Z,t,Z];
  esac;

x[Z,t,i] = case
  \{ Z=0 \} : b[Z] - b[Z-1,t-1,i]) / A[Z,Z];
  esac;
}
teil;
```

Figure 6.4: Forward Substitution example after alignment

### 6.5 Communication Analysis, Partitioning and Processor Allocation

Partitioning is the problem of mapping variables and computations to physical processors [5]. The virtual grid is partitioned or tiled into groups which are to be mapped to physical processors. This partitioning has a profound effect on communication. It is therefore done so as to minimize communication while maximizing parallelism. When a computation is mapped to the same point as the variable on the left-hand side of the equation, it is called using the owner computes rule for aligning computation. Variables and computation can also be separately mapped to the common grid by using different mapping functions for computation and for variables.

For the forward substitution algorithm (Example 2, Figure 6.1), the following new dependence table is generated:
And the corresponding usage table is:

![Diagram showing alignment of variables in Forward Substitution](image)

Each dependence represents a potential communication. With usage analysis, the amount and cost of the communication can be estimated [73]. This information can be used to choose an allocation function for the virtual array. Dependency 1 represents the writing of the result $X$. It would represent a communication with the host, or I/O system. Likewise, dependencies 3, 5, and 7 represent the inputs to the system which are also communications with the host, or I/O system. That leaves dependencies 2, 4, and 6 as possible processor to processor communications. If the processor allocation function is chosen to be $p(Z,t,i) = i$ then dependencies 2 and 6 are between processor $i$ and itself, and hence have no cost. That leaves communication 4 which is a set of $N$ one to many broadcasts: $1 <= Z <= N : (z[Z,t,Z] to (Z < j <= N : b[Z,t+1,j])$. Thus the projection $p(Z,t,i) = i$ is chosen. The transformed index is renamed $p$ to represent the physical processor dimension. The resulting ALPHA code is similar to the last result, except the $i$-index is replaced with $p$ in the equations for $x$ and $b$. 

---

And the corresponding usage table is:

<table>
<thead>
<tr>
<th>$Z$</th>
<th>$t$</th>
<th>$i$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$t$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$t$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$t$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$t$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$t$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

Each dependence represents a potential communication. With usage analysis, the amount and cost of the communication can be estimated [73]. This information can be used to choose an allocation function for the virtual array. Dependency 1 represents the writing of the result $X$. It would represent a communication with the host, or I/O system. Likewise, dependencies 3, 5, and 7 represent the inputs to the system which are also communications with the host, or I/O system. That leaves dependencies 2, 4, and 6 as possible processor to processor communications. If the processor allocation function is chosen to be $p(Z,t,i) = i$ then dependencies 2 and 6 are between processor $i$ and itself, and hence have no cost. That leaves communication 4 which is a set of $N$ one to many broadcasts: $1 <= Z <= N : (z[Z,t,Z] to (Z < j <= N : b[Z,t+1,j])$. Thus the projection $p(Z,t,i) = i$ is chosen. The transformed index is renamed $p$ to represent the physical processor dimension. The resulting ALPHA code is similar to the last result, except the $i$-index is replaced with $p$ in the equations for $x$ and $b$. 

---

The equations for $x$ and $b$ are:

1. $x[i] \rightarrow x[i,2i-1,i]$ (1)
2. $x[Z,t,i] \rightarrow b[Z-1,t-2,i]$ (2)
3. $x[Z,t,i] \rightarrow A[Z-t+1,Z]$ (3)
4. $x[Z,t,i] \rightarrow x[Z,t-1,Z]$ (4)
5. $x[Z,t,i] \rightarrow B[Z]$ (5)
6. $x[Z,t,i] \rightarrow b[Z-1,t-1,i]$ (6)
7. $x[Z,t,i] \rightarrow A[Z,Z]$ (7)

The transformed index is renamed $p$ to represent the physical processor dimension. The resulting ALPHA code is similar to the last result, except the $i$-index is replaced with $p$ in the equations for $x$ and $b$. 

---

43
6.6 Conclusion

Static analysis uses dependence and usage information extracted from an Alpha program to derive timing and processor allocation functions. The results of this analysis is encoded back into the program by doing a change of basis, and giving a global interpretation to certain indices.

Before doing any static analysis, all indices are non-descript. After performing a change of basis using the timing function, certain indices become temporal indices, and all other indices are called spatial. Before alignment, every variable has its own space, possibly of a different spatial dimension than other variables. After alignment, the spatial indices of all variables will relate to a common virtual space. After processor allocation, a part of the spatial indices will be interpreted as processor space indices and the remainder of the spatial indices will be interpreted as memory space.

Thus, the result of static analyses is an Alpha program in which the indices have been assigned a meaning above and beyond the semantics of the Alpha language. In the strict definition of Alpha, indices have only a very local scope, usually within a single domain, a single affine function, or a single equation. After analysis, indices are given system wide interpretations. In this way, the schedule and allocation functions are encoded back into the program. Subsequent transformations to the program use this information to generate code.
Chapter 7

**ALPHA Transformations**

In this chapter, I show how to make use of the ALPHA language formalism and transformation tools which have already proved useful for the synthesis of systolic arrays. Change of basis referred to in the previous chapter is one of a variety of program transformations which are available. Each transformation operates on an entire ALPHA program, producing a transformed but equivalent ALPHA program.

The ALPHA language environment has a rich set of proven program transformations which can be performed on a program. I made use of existing transformations as well as developed new transformations which are useful for deriving imperative code. The synthesizing compiler is thus based on a transformational approach. Through a series of independently provable program transformations, ALPHA is transformed into a form suitable for imperative code generation. The ALPHA environment becomes a synthesis tool through the application of the transformations described in this chapter.

### 7.1 Related Work

Chen [11] presented an approach where a problem specification in the functional language Crystal was transformed into what were called first-order recurrences (which turned out to be a subset of URE’s) and an inductive technique was applied to map these onto a space-time domain. Yang [94] presented a parallel programming methodology based on transformational program derivation with Crystal. It formalized the parallelizing process of aligning, partitioning, broadcast-removing, communication-aggregating, scheduling, and optimizing high level functional specifications onto target parallel architectures.

The ALPHA language is more restrictive than Crystal. The restrictions were designed into the language to guarantee closure under the program transformations described in this chapter. These transformations allow much flexibility in the ways ALPHA can be restructured.

### 7.2 Transformation using the ALPHA Environment

The collection of tools to manipulate an ALPHA program is called the ALPHA environment. It is built on top of the symbolic algebra system Mathematica [92] where the abstract syntax tree of an ALPHA program can be symbolically manipulated. An ALPHA transformation is implemented in Mathematica as a set of rewrite rules which transform an ALPHA program (in abstract syntax tree form) to another equivalent ALPHA program. Transformations also rely heavily on a polyhedral library [88] which provides the capability for doing basic operations on unions of polyhedra such as Image, Preimage, Intersection, Difference, and Union.
7.3 Basic Transformations

This section describes all of the basic transformations (or rewrite rules) which are elemental to the more complex transformations. Many of these transformations come from the theses of Mauras [33] and Le Verge [47] who pioneered the Alpha language. Many others having to do with new Alpha constructs are introduced in this thesis.

7.3.1 Proof of Basic Transformations

Basic transformations are proved using the definition of equivalent expressions in Alpha (definition 7) given in section 2.6. It basically states that two expression are equivalent if they have the same domain, data type, and denotational semantics. Thus if we can show that a program has the same semantics before and after a transformation, then Alpha is closed under that transformation.

To illustrate, I give a detailed proof of rule 7.31, which states

\[(\text{Dom}_1 : \text{Exp}_1) \ op \ (\text{Dom}_2 : \text{Exp}_2) \iff \text{Dom}_1 \cap \text{Dom}_2 : (\text{Exp}_1 \ op \ \text{Exp}_2)\]

**Proof:**

\begin{align*}
\text{Type} &\quad \text{Type}((\text{Dom}_1 : \text{Exp}_1) \ op \ (\text{Dom}_2 : \text{Exp}_2)) = \\
&\quad \text{Type}(\text{TypeTable}(\ \ op , \text{Type}(\text{Dom}_1 : \text{Exp}_1), \text{Type}(\text{Dom}_2 : \text{Exp}_2)) = \\
&\quad \text{Type}(\text{TypeTable}(\ \ op , \text{Type}(\text{Exp}_1), \text{Type}(\text{Exp}_2)) = \\
&\quad \text{Type}(\text{Exp}_1 \ op \ \text{Exp}_2) = \\
&\quad \text{Type}(\text{Dom}_1 \cap \text{Dom}_2 : (\text{Exp}_1 \ op \ \text{Exp}_2))
\end{align*}

\begin{align*}
\text{Domain} &\quad \text{Domain}((\text{Dom}_1 : \text{Exp}_1) \ op \ (\text{Dom}_2 : \text{Exp}_2)) = \\
&\quad \text{Domain}(\text{Dom}_1 : \text{Exp}_1) \cap \text{Domain}(\text{Dom}_2 : \text{Exp}_2) = \\
&\quad \text{Dom}_1 \cap \text{Domain}(\text{Exp}_1) \cap \text{Dom}_2 \cap \text{Domain}(\text{Exp}_2) = \\
&\quad \text{Domain}(\text{Dom}_1 \cap \text{Dom}_2 : (\text{Exp}_1 \ op \ \text{Exp}_2))
\end{align*}

\begin{align*}
\text{Eval} &\quad \text{Eval}(\text{Dom}_1 : \text{Exp}_1) \ op \ (\text{Dom}_2 : \text{Exp}_2), z) = \\
&\quad \text{Eval}(\text{Dom}_1 : \text{Exp}_1, z) \ op \ \text{Eval}(\text{Dom}_2 : \text{Exp}_2, z) = \\
&\quad (\text{Eval}(\text{Exp}_1, z) \text{ if } z \in \text{Dom}_1) \ op \ (\text{Eval}(\text{Exp}_2, z) \text{ if } z \in \text{Dom}_2) = \\
&\quad (\text{Eval}(\text{Exp}_1, z) \text{ op } \text{Eval}(\text{Exp}_2, z)) \text{ if } z \in \text{Dom}_1 \cap \text{Dom}_2 = \\
&\quad \text{Eval}(\text{Dom}_1 \cap \text{Dom}_2 : (\text{Exp}_1 \ op \ \text{Exp}_2), z)
\end{align*}

Other basic transformations are proved in an analogous fashion.

Two special symbols are introduced to accommodate rules which do reduction of empty expressions and equations.

**Definition 9** (Empty expression and empty equation notation)

An empty expression is denoted as \(\varepsilon\).

An empty equation is denoted as \(\xi\).

7.3.2 Case Rules

\begin{align*}
\text{case}(\text{Exp}) &\iff \text{Exp} & (7.1) \\
\text{case}(\cdots, \text{case}(\text{Exp}_1, \cdots, \text{Exp}_n), \cdots) &\iff \text{case}(\cdots, \text{Exp}_1, \cdots, \text{Exp}_n, \cdots) & (7.2) \\
\text{case}(\cdots, \text{Exp}_1, \cdots, \text{Exp}_n, \cdots) &\iff \text{case}(\cdots, \text{Exp}_2, \cdots, \text{Exp}_1, \cdots) & (7.3) \\
\text{Exp} \ op \ \text{case}(\text{Exp}_1, \cdots, \text{Exp}_n) &\iff \text{case}(\text{Exp} \ op \ \text{Exp}_1, \cdots) & (7.4)
\end{align*}

46
\[ \text{case}(Exp_1, \ldots, Exp_n) \text{ op } Exp \iff \text{case}(Exp_1 \text{ op } Exp, \ldots, Exp_n \text{ op } Exp) \]  
(7.5)

\[ \text{uop case}(Exp_1, \ldots, Exp_n) \iff \text{case}(\text{uop } Exp_1, \ldots, \text{uop } Exp_n) \]  
(7.6)

\[ \text{case}(Dom_1 : Exp, Dom_2 : Exp, \ldots) \iff \text{case}(Dom_1 \cup Dom_2 : Exp, \ldots) \]  
(7.7)

\[ \text{let}(\ Var = Exp_1; \ldots; \ Var = Exp_n) \iff \ Var = \text{case}(Exp_1, \ldots, Exp_n) \]  
(7.8)

\[ \text{case}(\cdots, \varepsilon, \ldots) \iff \varepsilon \]  
(7.9)

\[ \text{case}() \iff \varepsilon \]  
(7.10)

\[ \text{if Context}(Exp) \cap \text{Domain}(Exp) = \emptyset \implies \]  
\[ Exp \iff \varepsilon \]  
(7.11)

\[ Exp \text{ op } \varepsilon \iff \varepsilon \]  
(7.12)

\[ \varepsilon \text{ op } Exp \iff \varepsilon \]  
(7.13)

\[ \text{uop } \varepsilon \iff \varepsilon \]  
(7.14)

### 7.3.3 Let Rules

\[ \text{let}(Eqn) \iff Eqn \]  
(7.15)

\[ \text{let}(\cdots, \text{let}(Eqn_1, \ldots, Eqn_n), \cdots) \iff \text{let}(\cdots, Eqn_1, \ldots, Eqn_n, \ldots) \]  
(7.16)

\[ \text{let}(\cdots, Eqn_1, \ldots, Eqn_n, \ldots) \iff \text{let}(\cdots, Eqn_2, \ldots, Eqn_1, \ldots) \]  
(7.17)

\[ \text{let}(Dom_1 : Eqn, Dom_2 : Eqn, \ldots) \iff \text{let}(Dom_1 \cup Dom_2 : Eqn, \ldots) \]  
(7.18)

\[ \text{let}(Dom_1 : Eqn, Dom_2 : Eqn, \ldots) \iff \text{let}(Dom : \text{let}(Eqn_1, Eqn_2), \ldots) \]  
(7.20)

\[ \text{let}(\ Var = Exp_1; \cdots; \ Var = Exp_n) \iff \ Var = \text{case}(Exp_1, \cdots, Exp_n) \]  
(7.21)

\[ \text{let}(\cdots, \xi, \cdots) \iff \text{let}(\cdots, \cdots) \]  
(7.22)

\[ \text{let}() \iff \xi \]  
(7.23)

\[ \ Var = \varepsilon \iff \xi \]  
(7.24)

### 7.3.4 Restriction Rules

\[ \text{Dom} : Exp \iff Exp \text{ iff } \text{Dom} \supseteq \text{Domain}(Exp) \]  
(7.25)

\[ \text{UniverseDomain} : Exp \iff Exp \]  
(7.26)

\[ \text{Domain}(Exp) : Exp \iff Exp \]  
(7.27)

\[ \text{Dom}_1 : (\text{Dom}_2 : Exp) \iff (\text{Dom}_1 \cap \text{Dom}_2) : Exp \]  
(7.28)

\[ (\text{Dom} : Exp_1) \text{ op } Exp_2 \iff \text{Dom} : (\text{Exp}_1 \text{ op } Exp_2) \]  
(7.29)

\[ \text{Exp}_1 \text{ op } (\text{Dom} : Exp_2) \iff \text{Dom} : (\text{Exp}_1 \text{ op } Exp_2) \]  
(7.30)

\[ (\text{Dom}_1 : \text{Exp}_1) \text{ op } (\text{Dom}_2 : \text{Exp}_2) \iff \text{Dom}_1 \cap \text{Dom}_2 : (\text{Exp}_1 \text{ op } \text{Exp}_2) \]  
(7.31)

\[ \text{uop} (\text{Dom} : Exp) \iff \text{Dom} : (\text{uop} \text{Exp}) \]  
(7.32)

\[ \text{Dom} : \text{case}(Exp_1, \ldots, Exp_n) \iff \text{case}(\text{Dom} : Exp_1, \ldots, \text{Dom} : Exp_n) \]  
(7.33)

\[ \text{Dom} : \text{let}(\ Var_1 = Exp_1; \cdots; \ Var_n = Exp_n) \iff \text{let}(\text{Dom} : \ Var_1 = Exp_1; \cdots; \text{Dom} : \ Var_n = Exp_n) \]  
(7.34)
7.3.5 Dependence Function Rules

\[ \text{Dom} : \text{Var} = \text{Exp} \iff \text{Var} = \text{Dom} : \text{Exp} \]

(7.35)

\[ \text{EmptyDomain} : \text{Exp} \iff \varepsilon \]

(7.36)

\[ \text{Dom} : \varepsilon \iff \varepsilon \]

(7.37)

7.3.6 Reduction Rules

These rules cover the expansion, distribution, and serialization of the reduction operator, and are taken from [45, 47].

\[ g \text{ admits a left inverse } \implies \]

\[ \text{reduce}(\oplus, f, \text{Exp}) \iff \text{reduce}(\oplus, f.g, \text{Exp}.g) \]

(7.46)

\[ \text{Dom} = \{ x \mid Cx + d \geq 0 \}, \ f = (z \rightarrow Az + b), \]

\[ \forall t : (At = 0 \implies Ct = 0) \implies \]

\[ \text{reduce}(\oplus, f, \text{Dom} : \text{Exp}) \iff \text{Image} (\text{Dom}, f) : \text{reduce}(\oplus, f, \text{Exp}) \]

(7.47)

\[ \text{reduce}(\oplus, f, \text{case}(\text{Exp}_1, \text{Exp}_2)) \iff \text{case}(\text{D}_{12} : \text{reduce}(\oplus, f, \text{Exp}_1) \oplus \text{reduce}(\oplus, f, \text{Exp}_2), \]

\[ \text{D}_1 : \text{reduce}(\oplus, f, \text{Exp}_1), \]

\[ \text{D}_2 : \text{reduce}(\oplus, f, \text{Exp}_2)) \]

(7.48)

where

\[ \text{D}_{12} = \text{Image} (\text{Domain} (\text{Exp}_1), f) \cap \text{Image} (\text{Domain} (\text{Exp}_2), f) \]

\[ \text{D}_1 = \text{Image} (\text{Domain} (\text{Exp}_1), f) \backslash \text{D}_{12} \]

\[ \text{D}_2 = \text{Image} (\text{Domain} (\text{Exp}_2), f) \backslash \text{D}_{12} \]

\[ \text{reduce}(\oplus, f, \text{Exp}).g \iff \text{reduce}(\oplus, f', \text{Exp}.g') \]

where \( f, g, f', \) and \( g' \) are affine integer functions \( f \) and \( f' \) are right invertible

\[ g \circ f' = f \circ g \]

(7.49)

\[ \text{reduce}(\oplus, f, \text{Exp}) \iff \text{case} (\text{Image} (D, T) \backslash D : \text{Identity} (\oplus), \]

\[ D : YT \oplus \text{Exp}.g \]

(7.50)

where

\[ D = \text{Domain} (\text{Exp}), \]

\[ f : \mathbb{Z}^n \rightarrow \mathbb{Z}^{n-1} = (z \rightarrow Az + b), \]

\[ T = (z \rightarrow z \leftrightarrow t), \] where \( At = 0, \)

\[ \forall y \in D \backslash \text{Image} (D, T) : f \circ g(y) = y \]

if \( f = f_1 \circ f_2 \circ \cdots \circ f_{n-m}, \)

where \( f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m, n > m, \)

\[ f_i : \mathbb{Z}^{n+1-i} \rightarrow \mathbb{Z}^{n-i} \implies \]

\[ \text{reduce}(\oplus, f, \text{Exp}) \iff \text{reduce}(\oplus, f_1, \]

\[ \text{reduce}(\oplus, f_2, \]

\[ \text{reduce}(\oplus, f_{n-m}, \text{Exp}) \]

(7.51)

\[ \ldots) \]

\[ \text{reduce}(\oplus, f_{n-m}, \text{Exp}) \]

48
### 7.4 Change of Basis

Of all the transformations which can be done on an Alpha program, the change of basis [53] is the most common. It is similar to the reindexing of loop variables [90] and other loop transformations done in vectorizing and parallelizing compilers. Given a valid program and a change of basis of a variable, a new, provably equivalent program is derived which not only includes the change of basis of the variable, but also the reindexing of all array references to that variable.

In general, a change of basis of variable $A$ defined over an index domain $D$, using the affine transformation function $F$, can be performed if there exists another affine function $G$, such that $\forall y \in D : (F \circ G)(y) = y$. The change of basis is done by syntactically rewriting an Alpha program as follows:

$$
\begin{align*}
A : D & \quad \text{of integer; } \\
& \text{let } \\
& \quad A = \ldots A \ldots; \\
& \quad \ldots = \ldots A \ldots; \\
& \text{tel;}
\end{align*}
\iff
\begin{align*}
A : D.F & \quad \text{of integer;} \\
& \text{let } \\
& \quad A = (\ldots A,F \ldots), G; \\
& \quad \ldots = \ldots A,F \ldots; \\
& \text{tel;}
\end{align*}
$$

where $A.F$ refers to the Alpha dependence operator defined in section 2.8.3. Change of basis is its own inverse transformation.

If normalize is performed after a change of basis, the affine function $G$ is distributed over the expression and the domain $D.F$ is computed in the course of bringing the program into normal form.

### 7.5 Substitution

An immediate consequence of referential transparency is that equality is substitutive — equal expressions are always and everywhere interchangeable. An equation specifies an assertion on a variable which must always be true. Thus, any instance of a variable on the left hand side of any equation may be replaced with the right hand side of its definition. This transformation is done by rewriting an Alpha program as follows:

$$
\begin{align*}
A : D & \quad \text{of integer; } \\
& \text{let } \\
& \quad A = \text{expression}; \\
& \quad \ldots = \ldots A \ldots; \\
& \quad \ldots = \ldots A \ldots; \\
& \text{tel;}
\end{align*}
\iff
\begin{align*}
A : D & \quad \text{of integer; } \\
& \text{let } \\
& \quad A = \text{expression}; \\
& \quad \ldots = \ldots (D:\text{expression}) \ldots; \\
& \quad \ldots = \ldots (D:\text{expression}) \ldots; \\
& \text{tel;}
\end{align*}
$$

### 7.6 Add Local Variable

This transformation is the inverse of substitution. It involves the replacement of common subexpressions with a local variable. To add local variable $A$ defined to be expression, a declaration and definition are added for variable $A$ and all instances of expression are replaced with $A$. The search for expression is based on simple syntactic pattern matching, and thus the expressions $A + B$ and $B + A$ are not recognized as equivalent.

$$
\begin{align*}
& \text{let } \\
& \quad \ldots = \ldots \text{expression} \ldots; \\
& \quad \ldots = \ldots \text{expression} \ldots; \\
& \quad \ldots = \ldots \text{expression} \ldots; \\
& \text{tel;}
\end{align*}
\iff
\begin{align*}
& \text{let } \\
& \quad A = \text{expression}; \\
& \quad \ldots = \ldots A \ldots; \\
& \quad \ldots = \ldots A \ldots; \\
& \quad \ldots = \ldots A \ldots; \\
& \text{tel;}
\end{align*}
$$

49
7.7 Serialization

The serialization transformation [45, 47] is the implementation of rule 7.50. It effectively serializes a reduce operation and replaces it with a recurrence which computes the reduction. Serialization is described in more detail later in this thesis (section 8.4.1) in connection with an example.

7.8 Normalization

There is also an Alpha transformation that “normalizes” any Alpha program into a predefined syntactic form which fixes the nesting order of Alpha constructs to: (going from outermost to innermost) case statements, restriction operations, binary and unary operations, dependency functions, and last of all variables and constants. This transformation consists of about 30 rewrite rules1 which, among other things, change the nesting order of incorrectly nested operations, flatten nested cases, and combine restriction and affine functions where possible. The set of rewrite rules is confluent and guaranteed to terminate at a stable fix-point. This transformation is often used as a follow up to other transformations (such as the change of basis and substitution transformations presented above) and is where the computation of new domains and dependency functions is done.

\[
\begin{align*}
\text{any}
&\quad \text{ Alpha program} \\
\text{var} : \mathcal{D} \text{ of integer;}
&\quad \text{let}
&\quad \text{case}
&\quad \quad \mathcal{D}_1 : f_1(\text{var}.\text{dep}, \cdots);
\quad \mathcal{D}_2 : f_2(\text{var}.\text{dep}, \cdots);
&\quad \quad \cdots
&\quad \text{esac;}
&\quad \ldots
&\quad \text{tel;}
\end{align*}
\]

7.9 Other Normalizations

Other normal forms can be created by applying a different set of basic transformations, as long as the set of rules are confluent. Two other normal forms are introduced in this thesis. The R-normal form is described in section 8.4.4 and imperative normal form is described in section 8.4.6. The ability to restructure an Alpha program through the normalization process is one of Alpha’s most powerful features.

7.10 Conclusion

One property that sets the Alpha language apart from other similar languages is that it is closed under a rich set of transformations. The ability to restructure an Alpha program through the normalization process is another distinguishing and powerful feature. In this chapter, I have introduced all the basic, as well as some of the more complex transformations used. It is the application of these transformations which forms the basis of the synthesizing compiler for Alpha.

---

1 e.g. rules 7.1, 7.2, 7.4, 7.5, 7.6, 7.9, 7.10, 7.12, 7.13, 7.14, 7.25, 7.28, 7.29, 7.30, 7.32, 7.33, 7.36, 7.37, 7.38, 7.39, 7.40, 7.41, 7.42, 7.43, 7.44, and 7.45.
Chapter 8

Imperative Code from ALPHA

8.1 Introduction

Because of the elegance and power of functional languages, one would expect them to be naturally suited for numeric and scientific computing. However, this potential has not yet been well exploited, in spite of efforts such as SISAL and others [60, 54]. To fully exploit this potential, compiled functional programs must run as efficiently as compiled imperative programs. Two of the difficulties encountered in compiling functional languages seem to be avoiding the high overhead of context switching and dealing with arrays efficiently.

In this chapter, I show how ideas developed for doing systolic array synthesis can be extended and adapted for use in the compilation of functional languages. The kinds of analysis needed to generate efficient code for array based function programs is in fact a generalization of scheduling techniques used in systolic array synthesis. I show how a transformational strategy used in systolic array synthesis can be used as a compilation engine, and that a very large part of the the compilation process can be done using transformations. The application of systolic techniques to the compilation of functional programs can yield efficient results for scientific and numeric data-parallel programs.

In chapter 5, I described an automatic but naive translation scheme which transforms a program written in ALPHA to C-code. This translator produces C-code without knowing an execution schedule. The C-program works on a demand driven basis and suffers from high overhead due to frequent context switching.

In this chapter, I improve on that implementation by using timing information which can be deduced by doing static analysis as discussed in chapter 6. Using dependency analysis, an execution schedule giving relative times for computation of variables can be found. Given this schedule, and using a transformational approach, ALPHA statements are separated, regrouped and reordered according to their scheduled execution times. This produces a significantly transformed ALPHA program which is restructured like an imperative program, but in fact is a functional program constructively equivalent to the original specification. From this transformed ALPHA program, C-code is generated by a simple and straightforward code generator. The proposed method produces efficient structured loop programs such as those that are common to imperative languages.

The chapter is organized as follows. In section 2, I present an example used in this chapter. In section 3, I give an overview of the proposed compilation process, and in section 4, I describe the compilation of ALPHA in detail and illustrate it with the aid of an example. In section 5, I make some observations about the compilation of functional languages and summarize the chapter.
8.2 An example

For an example, I present an algorithm for solving the Yule-Walker equations which arise in conjunction with certain linear prediction problems [30].

\[
\begin{bmatrix}
1 & r_1 & r_2 & \cdots & r_{n-1} \\
r_1 & 1 & r_1 & \cdots & r_{n-2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
r_{n-1} & r_{n-2} & \cdots & 1 & 1
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix} =
\begin{bmatrix}
\leftrightarrow_1 \\
\leftrightarrow_2 \\
\vdots \\
\leftrightarrow_n
\end{bmatrix}
\]

The problem is given \( r_1 \cdots r_n \), find \( y_1 \cdots y_n \).

The algorithm below was proposed by Durbin in 1960. I have written it in \textsc{Alpha} in terms of a set of mutually recursive equations and using a reduce operator. This example illustrates the rich capability of \textsc{Alpha} to efficiently encode non-trivial mathematical algorithms. I use this program as a running example throughout this chapter to illustrate the steps in compilation.

```
system Durbin : { N | N>=1 }
  ( r : { i | 1<=i<=N } of real )
  returns ( y : { i | 1<=i<=N } of real );
var
  Y : { k,i | 1<=k<=N; 1<=i<=k } of real;
  B : { k | 1<=k<=N } of real;
let
  Y[k,i] = case
    { | k=1; i=1 } : -r[i];
    { | k>=2; i<k } : Y[k-1,i] +
      Y[k,k] * Y[k-1,k-i];
    { | k>=2; i=k } : (-r[k] - reduce(+, (k,i->k),
      r.(k,i->i) * Y.(k,i->k-1,k-i)[k]) / B[k];
  esac;
  B[k] = case
    { | k=1 } : 1 ;
    { | k>=2 } : B[k-1] *
      (1 - Y[k-1,k-1] * Y[k-1,k-1]);
  esac;
y[i] = Y[1,i];
tel;
```

8.3 Overview of the Sequentializing Translator

In this section, I describe the compilation strategy for the sequentializing translator. First of all, I assume that I can find a schedule consisting of a set of variable dependent scheduling functions. Given this set of timing functions, I show how \textsc{Alpha} recurrence equations can be ordered using \textsc{Alpha} program transformations, such that all data values are computed before they are used.

The translator begins with the specification of an algorithm stated at a high level of abstraction, in a notation similar to the set of mathematical equations describing the operation. Specifying the algorithm fixes the data dependencies of the algorithm which in turn determine the amount of parallelism available.

Then a series of transformation steps are performed on the program. These steps are summarized below and then explained in detail in the next section.

1. Serialization— the translation of a reduce operator to a set of simple recurrence equations by the successive composition of the associated binary operator.
2. Dependency analysis and derivation of a schedule— the finding of valid timing functions for each variable.
3. Change of basis using scheduling functions—domains of variables are translated, rotated, skewed, etc., so that their domains are aligned with the temporal indices. These indices are afterward interpreted as time. All other indices are interpreted as spatial indices.

4. R-normalization—rewriting the program so that all equations are of the form domain: $X = \text{simple-expression}$ in which case operators have been eliminated and all equations are qualified with the domain over which they are defined.

5. Time separation—domains are separated into their temporal and spatial components.

6. Sort by time—combining equations that are evaluated at the same time into blocks and then sorting the blocks by order of time.

7. Code generation—the generation of C-code by pretty printing the ALPHA program in C.

Of these steps, the r-normalization and sorting by time transformations are new and have never before been reported in the literature.

8.4 Compilation

In this section, I describe the compilation steps in detail, and illustrate the steps using the Durbin program as an example. I describe each transformation used and present the rewrite rules that make up the transformation.

8.4.1 Serialization

Since a typical processor does not directly support reduce operators\footnote{also known a parallel prefix}, but only the associated binary operators, it is necessary to replace reductions with an equivalent set of recurrence equations. The translation of a reduce operator to a set of simple recurrence equations is called serialization. A serialize transformation which is based on a successive composition of the associated binary operator is available in the ALPHA environment. For example, the following transformation serializes the reduction $A = \sum_{i=1}^{N} x_i$ in the positive direction of the $i$ index. The “direction” of the serialization can be obtained semi-automatically and is known to be the dual of a transformation called localization that has been well studied in the systolic synthesis community.

$$A = \text{reduce}(+, (i \rightarrow), x) ;$$

\[
\begin{align*}
    & $S[i] = \text{case}$ \\
    & \{ i = 0 \} : 0 ; \\
    & \{ 1 \leq i \leq N \} : S[i-1] + x[i] ; \\
    & \text{esac} ; \\
    & A = S[N] ;
\end{align*}
\]

A local variable $S$ is introduced in the above transformation and is called the summation variable. Note that this transformed form only executes the reduction in $O(n)$ time, whereas I know that these computations can be done in $O(\log(n))$ parallel time best case. The simpler form is used here because it only introduces uniform dependencies which are easier to handle by a scheduler. Furthermore, since our current target is sequential code, the $O(\log(n))$ serialization is unnecessary.

In the Durbin example, a local summation variable $Z$ is introduced to perform the reduction with a recurrence equation. Then, the reduce operation used in the definition of $Y$ is replaced with the term $Z[k, k \Leftarrow 1]$ which is the result of the reduction at index $k$. 

\[\text{also known a parallel prefix}\]
8.4.2 Scheduling

Using static analysis, a schedule can be derived which obeys the causality rule for all of the variable dependencies in the program. The dependencies are extracted from the Alpha program, as shown here for the Durbin example:

\[
\begin{align*}
\{k, i, N \mid i+1<k\leq N; 1<i\} & : Z[k, i, N] \rightarrow Z[k, i-1, N] \quad (1) \\
\{k, i, N \mid i+1<k\leq N; 1<i\} & : Z[k, i, N] \rightarrow r[i, N] \quad (2) \\
\{k, i, N \mid i+1<k\leq N; 1<i\} & : Y[k, i, N] \rightarrow Y[k-1, k-i, N] \quad (3) \\
\{k, i, N \mid k=1; i=1; 1<N\} & : Y[k, i, N] \rightarrow r[k, i, N-1] \quad (4) \\
\{k, i, N \mid i+1<k\leq N; 1<i\} & : Y[k, i, N] \rightarrow Y[k-1, i, N] \quad (5) \\
\{k, i, N \mid i+1<k\leq N; 1<i\} & : Y[k, i, N] \rightarrow Y[k, k-i, N] \quad (6) \\
\{k, i, N \mid k=1; 2<i\leq N\} & : Y[k, i, N] \rightarrow r[k, N] \quad (7) \\
\{k, i, N \mid k=1; 2<i\leq N\} & : Y[k, i, N] \rightarrow Z[k, i-1, N] \quad (8) \\
\{k, i, N \mid k=1; 2<i\leq N\} & : Y[k, i, N] \rightarrow B[k, N] \quad (9) \\
\{k, N \mid 2<k\leq N\} & : B[k, N] \rightarrow B[k-1, N] \quad (10) \\
\{k, N \mid 2<k\leq N\} & : B[k, N] \rightarrow Y[k-1, k-1, N] \quad (11) \\
\{k, N \mid 2<k\leq N\} & : B[k, N] \rightarrow Y[k-1, k-1, N] \quad (12) \\
\{i, N \mid 1<i\leq N\} & : y[i, N] \rightarrow Y[k, i, N] \quad (13)
\end{align*}
\]

Dependencies involving inputs (lines 2,4) and outputs (line 14) are discarded, and the remaining dependencies are used to set up a system of constraints on a set of variable dependent timing functions. For the Durbin example, this resulted in a vacuous system, meaning no set of linear timing functions could meet all of the constraints. Indeed, it is known that Durbin's algorithm has a quadratic schedule in the best case. To find this quadratic schedule, the causality rule is (temporarily) relaxed to \(X[p] \rightarrow Y[q] \Rightarrow t_X(p) \geq t_Y(q)\) and a linear set of functions are found. Then dependencies which satisfy \(X[p] \rightarrow Y[q] \Rightarrow t_X(p) > t_Y(q)\) (the true causality rule) are eliminated. For the dependencies which are left, \(t_X(p) = t_Y(q)\), and a second dimension of time is added to resolve the dependency. A new system of constraints is set up to solve for the second dimension of time and the process is repeated. Using this procedure, a multidimensional, variable-dependent schedule was found for the Durbin algorithm. It is given as:

\[
\begin{align*}
t_B \left( \begin{array}{c} k \\
\end{array} \right) & = \begin{bmatrix} k \\
0 \\
\end{bmatrix} = \begin{bmatrix} t_1 \\
t_2 \\
\end{bmatrix} \\
t_Z \left( \begin{array}{c} k \\
i \\
\end{array} \right) & = \begin{bmatrix} k \\
i + 1 \\
\end{bmatrix} = \begin{bmatrix} t_1 \\
t_2 \\
\end{bmatrix} \\
t_Y \left( \begin{array}{c} k \\
i \\
\end{array} \right) & = \begin{bmatrix} k + i + N + 1 \\
\end{bmatrix} = \begin{bmatrix} t_1 \\
t_2 \\
\end{bmatrix}
\end{align*}
\]
system Durbin (N : {N | 1<=N} parameter;
  r : {i | 1<=i<=N} of real)
  returns (y : {i | 1<=i<=N} of real);
var
  Z : {t1,t2 | (t2,2)<=t1<=N; 1<=t2} of real;
  Y : {t1,t2 | t2-N<=t1<=N; N+1<=t2} of real;
  B : {t1,t2 | 1<=t1<=N; t2=0} of real;
let
  Z[t1,t2] =
    case
      | t2=1 : 0;
      | 2<=t2 : Z[t1,t2-1] +
        r[t2-1] * Y[t1-1,t2+N-1];
    esac;
  Y[t1,t2] =
    case
      | t1=1; t2=N+1 : -r[t1];
      | N+2<=t2 : Y[t1-1,t2-1] +
        Y[t1,N+1] * Y[t1-1,t1-t2+2N+1];
      | 2<=t1; t2=N+1 : (-r[t1] - Z[t1,t1]) /
        B[t1,t2-N-1];
    esac;
  B[t1,t2] =
    case
      | t1=1 : 1;
      | 2<=t1 : B[t1-1,t2] *
        (1 - Y[t1-1,t2+N+1] * Y[t1-1,t1,t2+N+1]);
    esac;
y[i] = Y[N,-i+2N+1];
tel;

tel;

Figure 8.1: Timed ALPHA for Durbin’s Algorithm

8.4.3 Change of Basis Using the Schedule

Now that a schedule is known, the program is rewritten in terms of time and space. Using the change of basis transformation, the program can be transformed so that the variables are a function of the time indices $t_1$ and $t_2$ instead of $k$ and $i$.

In general, a change of basis of variable $A$ defined over an index domain $D$, using the affine transformation function $F$, can be performed if there exists another affine function $G$, such that $\forall y \in D : (F \circ G)(y) = y$. The change of basis is done by syntactically rewriting an ALPHA program as shown in section 7.4. The following change of basis transformations are specified by the schedule:

<table>
<thead>
<tr>
<th>Variable</th>
<th>$F$</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z$</td>
<td>$(k,i \rightarrow k,i + 1)$</td>
<td>$(k,i \rightarrow k,i \leftarrow 1)$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$(k,i \rightarrow k,k \leftarrow i + N + 1)$</td>
<td>$(k,i \rightarrow k,k \leftarrow i + N + 1)$</td>
</tr>
<tr>
<td>$B$</td>
<td>$(k \rightarrow k,0)$</td>
<td>$(k,i \rightarrow k)$</td>
</tr>
</tbody>
</table>

After performing the change of basis on variables $Z$, $Y$, and $B$, as specified by the scheduling functions given above, the Durbin algorithm is transformed into the program shown in figure 8.1 where indices named $t_1$ and $t_2$ can be interpreted as the temporal indices.
8.4.4  R–Normalization

This step explicitly tags each equation with the domain over which it is defined. This information is required later to generate loop nests and to sort the equations by time.

In this transformation, the program is rewritten in a normal form I call *r-normal* form. In this form, where all equations are restricted by their domain of computation, case statements are eliminated, and equations are rewritten to only have a simple expression on the right-hand side. The result is that computations of a variable in different computational subdomains are represented by separate domain-qualified equations.

*R-normalization* is done with a set of simple rewriting rules, the most important of which are presented here. It is assumed that the program is already in standard normal form (equation—case—restrict form) as shown in the example Alpha programs up to this point. Then a set of rules is applied until no further program modifications can be made with that rule set (i.e. a fixed point is reached). These rules are described below.

**Rule 1**  This rule eliminates case statements, replacing them with a sequence of equations.

\[
\begin{align*}
\text{Var} &= \text{case} \\
    &\quad \text{Exp}_1 \\
    &\quad \ldots \\
    &\quad \text{Exp}_n \\
    &\quad \text{esac}; \\
\end{align*}
\]
\[
\begin{align*}
\text{let} &\quad \text{Var} = \text{Exp}_1; \\
    &\quad \ldots \\
    &\quad \text{Var} = \text{Exp}_n; \\
    &\quad \text{tel}; \\
\end{align*}
\]

**Rule 2**  This rule moves a restriction from the expression on the right-hand side of an equation to outside the equation.

\[
\begin{align*}
\text{Var} &= \text{Dom} : \text{Exp}; \\
\end{align*}
\]
\[
\begin{align*}
\text{Dom} \cap \text{Domain(Var)} : \\
    &\quad \text{Var} = \text{Exp}; \\
\end{align*}
\]

*Domain(Var)* is the declared domain of variable *Var*.

**Rules to simplify let blocks**  Other rules are used to keep let blocks in simplest form. Two examples are given below. The first rule removes a surrounding let block for the trivial case of a single equation. The second rule flattens two nested let blocks wherever possible.

\[
\begin{align*}
\text{let} &\quad \text{Eqn}; \\
    &\quad \text{tel}; \\
\end{align*}
\]
\[
\begin{align*}
\text{let} &\quad \text{Eqn}; \\
\end{align*}
\]

\[
\begin{align*}
\text{let} &\quad \ldots \\
    &\quad \text{let} \\
    &\quad \text{Eqn}_1; \\
    &\quad \ldots \\
    &\quad \text{Eqn}_n; \\
    &\quad \text{tel}; \\
    &\quad \ldots \\
    &\quad \text{tel}; \\
\end{align*}
\]
\[
\begin{align*}
\text{let} &\quad \ldots \\
    &\quad \text{let} \\
    &\quad \text{Eqn}_1; \\
    &\quad \ldots \\
    &\quad \text{Eqn}_n; \\
    &\quad \ldots \\
    &\quad \text{tel}; \\
\end{align*}
\]

It is fairly obvious that these rules are confluent and that they terminate. The firing order of rules 1 and 2 are controlled by the program structure. The firing of the rules to simplify the let blocks can be done in any order and does not affect the firing of rules 1 and 2. There are no circular firings possible and the process terminates when the program is in *r-normal* form. After the *r-normalization* transformation, the Durbin program is written as shown in figure 8.2.
system Durbin (N ; {N | i<=N} parameter;
  r : {i | 1<i<=N} of real)
  returns (y : {i | 1<i<=N} of real);
var
  Z : {t1,t2 | (t2,2)<t1<=N; 1<=t2} of real;
  Y : {t1,t2 | t2-N<t1<=N; N+1<=t2} of real;
  B : {t1,t2 | 1<=t1<=N; t2=0} of real;
let
  {t1,t2 | 2<=t1<=N; t2=1} :
    Z[t1,t2] = 0;
  {t1,t2 | t2<=t1<=N; 2<=t2} :
    Z[t1,t2] = Z[t1,t2-1] +
    r[t2-1] * Y[t1-1,t2+N-1];
  {t1,t2 | t1=1; t2=N+1; 1<=N} :
    Y[t1,t2] = -r[t1];
  {t1,t2 | t2-N<t1<=N; N+2<=t2} :
    Y[t1,t2] = Y[t1-1,t2-1] +
    Y[t1,N+1] * Y[t1-1,t1-t2+2N+1];
  {t1,t2 | 2<=t1<=N; t2=N+1} :
    Y[t1,t2] = (-r[t1] - Z[t1,t1]) / B[t1,t2-N-1];
  {t1,t2 | t1=1; t2=0; 1<=N} :
    B[t1,t2] = 1;
  {t1,t2 | 2<=t1<=N; t2=0} :
    B[t1,t2] = B[t1-1,t2] *
    (1 - Y[t1-1,t2+N+1] * Y[t1-1,t2+N+1]);
  {i | 1<=i<=N} : y[i] = Y[N-i+2N+1];
tel;

Figure 8.2: R-Normalized Alpha for Durbin’s Algorithm
8.4.5 Time Separation

The next step in the translation process involves separating domains which are functions of both time and space indices, into a nest of loops, where each loop iterates on a single time index, followed by a spatial domain if necessary. Assume that \( t_1, t_2, \cdots \) are time indices and \( i, j, \cdots \) are space indices. Then a domain is factored into a product of domains as shown below.

\[
\text{Dom}(t_1, t_2, \cdots, i, j, \cdots) : \text{Eqn} ; \iff \text{Dom}(t_1) :: \text{Dom}(t_2) :: \cdots :: \text{Dom}(i, j, \cdots) : \text{Eqn} ;
\]

The double colon notation indicates a loop domain in which inner loop indices have been projected out and removed from the domain.

Loop nest synthesis is based on the polyhedral scanning problem which poses the problem of finding a set of nested do-loops which visit each index point in a polyhedron. In this chapter, I employ a method [46] to scan parameterized polyhedra based on the polyhedral library [88]. The method uses the library to project out inner loop indices in a manner similar to the Fourier-Motzkin elimination method, but without generating any redundant inequalities. The loop bound expressions derived from the projection are further reduced by considering the context domain of each expression, and eliminating loop bounds which are redundant with the bounds of outer loops.

8.4.6 Code Ordering and Imperative Normal Form

When recurrences are executed in an order determined by a valid schedule, expensive context switching can be avoided. In this section, I syntactically arrange the equations so that if executed in syntactic order, the schedule will be followed. Since this is precisely the order in which the equations would be evaluated in imperative code, the resulting form is called the imperative normal form. Code ordering is done primarily with two rules.

Rule 1 The first rule performs two functions: first, it separates time domains to be disjoint, and second, it combines equations of equal time domains into blocks.

If \( \text{Dom}_1 \cap \text{Dom}_2 \) is not empty, then:

\[
\begin{align*}
\text{let} \\
\quad \ldots \\
\quad \text{Dom}_1 :: \text{Eqn}_1; \\
\quad \text{Dom}_2 :: \text{Eqn}_2; \\
\quad \ldots \\
\text{tel};
\end{align*}
\]

Rule 2 The second rule sorts adjacent time regions. Before I can talk about sorting however, I must define the ordering function. I define \( \text{Dom}_2 <_i \text{Dom}_1 \) as follows:

\( \text{Dom}_2 <_i \text{Dom}_1 \) is defined to be true iff

i. \( \text{Dom}_1 \cap \text{Dom}_2 \) is empty, and

ii. \( \exists p : ((t_1, p) \in \text{Dom}_1 \text{ and } (t_2, p) \in \text{Dom}_2) \)

iii. \( \forall p : ((t_1, p) \in \text{Dom}_1 \text{ and } (t_2, p) \in \text{Dom}_2) \iff t_2 < t_1 \)

When condition ii. cannot be satisfied, the two domains are incomparable and thus, this definition only induces a partial order on a set of domains.

To help understand this definition, figure 8.3 illustrates the concept of having all of the points in \( \text{Dom}_2 \) less than (relative to \( t \)) all of the points in \( \text{Dom}_1 \) for any given value of the parameter \( p \). To
perform a comparison of two domains, the compare is in terms of a single dimension $t$ of time, in function of a possibly multidimensional parameter $p$.

Repetitive application of rule 2 effectively performs a bubble sort on the code segments, placing them in order of ascending time in time dimension $t$. The rule is:

If $Dom_1 \cap Dom_2$ is empty and $Dom_2 \prec Dom_1$ then:

```
let ...
Dom_1 :: Eqn_1;
Dom_2 :: Eqn_2;
...
```

This transformation swaps two equations and puts them in order according to $t$, the loop index.

The two rules work together nicely. Rule 1 combines equations which work over the same domain into a single `let-tel` block, and makes all blocks time-disjoint, and rule 2 sorts the disjoint blocks. These two rules are confluent since rule 1 and rule 2 fire for different cases: rule 2 fires on adjacent blocks that are disjoint, and rule 1 fires when they are not disjoint. The termination of the rules can be proved with the same reasoning used to prove that a nondeterministic bubble sort terminates. Once two blocks are disjoint, rule 1 will no longer fire, and once two blocks are sorted (with respect to each other) rule 2 will no longer fire.

Code ordering is done recursively for the other less significant dimensions of time. The most significant time index is sorted over the entire program. Then the second most significant time index is sorted within the scope of each loop body of the previous time index, and so on, each new time index being sorted over a deeper nested loop body.

Going back to the Durbin example, after sorting by $t_1$ and $t_2$, respectively, the resulting program is shown in figure 8.4.
system Durbin (N : {N | 1<=i<=N} parameter;
   r : {i | 1<=i<=N} of real)
   returns (y : {i | 1<=i<=N} of real);
var
Z : {t1,t2 | (t2,2)<t1<=N; 1<=t2} of real;
Y : {t1,t2 | t2-N<=t1<=N; N+1<=t2} of real;
B : {t1,t2 | 1<=t1<=N; t2=0} of real;
let
{t1 | t1=1} ::
   let
   {t2,t1 | t2=0} ::
   B[t1,t2] = 1;
   {t2,t1 | t2=N+1} :: Y[t1,t2] = -r[t1];
ten;
{t1 | 2<=t1<=N} ::
   let
   {t2,t1 | t2=0} ::
   B[t1,t2] = B[t1-1,t2] *
   (1 - Y[t1-1,t2+N-1] * Y[t1-1,t2+N-1]);
   {t2,t1 | t2=1} ::
   Z[t1,t2] = 0;
   {t2,t1 | 2<=t2<=t1} ::
   Z[t1,t2] = Z[t1,t2-1] +
   r[t2-1] * Y[t1-1,t2+N-1];
   {t2,t1 | t2=N+1} ::
   Y[t1,t2] = (-r[t1] - Z[t1,t1]) / B[t1,t2-N-1];
   {t2,t1 | N+2<=t2<=t1+N} ::
   Y[t1,t2] = Y[t1-1,t2-1] +
   Y[t1,N+1] * Y[t1-1,t1-t2+N-1];
ten;
{i | 1<=i<=N} : y[i] = Y[N,-i+2N+1];
ten;

Figure 8.4: Imperative Normal ALPHA for Durbin's Algorithm
Notice that this program has little resemblance to the original algorithmic specification, and yet it is perfectly equivalent. Furthermore, the syntactic structure of the program is now very close to a nested loop imperative program, even though it is still a functional program. Once in imperative normal form, it is fairly straightforward to generate imperative code.

8.5 Code Generation

In the context of this chapter, code generation means translating structures in an Alpha program which is in imperative normal form, as shown above, into equivalent code constructs in C.

8.5.1 Declarations

An important problem in doing code generation is the generation of declarations of variables in the target language to replace the polyhedron based declarations in the Alpha language. Certain optimizations may be done on an Alpha program to reduce the amount of memory required to represent a variable. For instance, if it can be shown that the variable element lifetimes are disjoint across a certain dimension of the variable, then that dimension can be projected out when allocating memory. This analysis is the subject of chapter 9. Once the memory domain for a variable has been established, and parameters are set to specific values, the memory allocation analysis described in chapter 4 is performed. An integer value for the volume of the memory domain is computed using equation 4.11 and an affine index mapping function is computed using equation 4.10. With this information, along with the data type of the variable, a C-declaration for a variable is generated as follows:

<data_type> _<var_name>[<volume>];
#define <var_name>({<index_list>}) _<var_name>[<index_expression>]

For instance, the declaration of a variable \( Y \) declared in Alpha as:

\[
Y : \{ t_1, t_2 | t_2 \epsilon_\cdot N \leq t_1 \leq N; N + 1 \leq t_2 \} \text{ of real;}
\]

is translated as follows:

double _Y[9];
#define Y(t1,t2) _Y[3*(t1)+(t2)-7]

where 9 is the volume of \( Y \) and \( 3t_1 + t_2 \epsilon_\cdot 7 \) is the index mapping function. Then a reference to a variable written as \( Y[i,j] \) in Alpha would be translated as \( Y(i,j) \) in the C-code.

8.5.2 Imperative Code Generation

The basic concept used to implement the code generator is very simple since it is written as a particular kind of pretty printer.

After time-ordering the code, the resulting Alpha program is in imperative normal form. This form is block structured like an imperative program and statements are in the order of an imperative program. In short, this imperative normal form is suitable for generating imperative code. The following grammar describes a program in imperative normal form:

Program ::= Block
Block ::= Domain :: Block
       ::= Domain : Block
       ::= Let Block; Block; ... Tel
       ::= Equation
Equation ::= Var = Exp
All of the imperative control is encoded into the restriction operations (Domain :: Block and Domain : Block). From the restriction operations, the code generator produces for-loops and if-statements. Before I describe how this is done, I must first describe the concept of free and fixed indices. During the course of code generation, the code generator keeps track of each index by name, and tags it as either free or fixed. At the beginning of the program, only the parameter indices are fixed, and all other indices are free. When an index is assigned a value in the code, it is marked fixed for the scope of that assignment.

The rules below describe how to translate the above grammar into C-code.

1. \{i, ... | ... \} :: Block where index \( i \) is free and all other indices in the domain are fixed. This translates into a for-loop with \( i \) as the loop-index.

   ```c
   for (i ... )
   {
   Block
   }
   ```

   Index \( i \) is marked fixed in the scope of Block. At the end of the for loop, index \( i \) becomes free again.

2. \{ ... k ... | k=f(i,j,...); ... \} :: Block where \( k \) is free and \( i,j,\ldots \) are fixed. In this case, index \( k \) can be replaced with \( f(i,j,\ldots) \) throughout the code segment Block. This can be done in one of two ways.

   (a) If \( f(i,j,\ldots) \) is a simple constant, or a simple multiple of another index, then simply substitute the index \( i \) anywhere it occurs in Block with its equivalent value. Here are some examples.

   ```c
   i = 2                (i was free). Substitute i with 2
   i = 2j               (i was free and j was fixed). Substitute i with 2j
   2i = j               (i was free and j was fixed). Insert code:
   if (j%2==0) { i=j/2; Block }
   i = 2q + r; 0 \leq r \leq 2 (i was fixed and q and r were free)
   Substitute q with i/2 and substitute r with i%2.
   ```

   (b) If \( f(i,j,\ldots) \) is any other expression, then simply generate an index assignment:

   ```c
   k=f(i,j,...);
   ```

   and mark \( k \) fixed in the scope of Block.

3. \{ ... k ... | k <rel> f(i,j,...); ... \} :: Block where \( k \) is fixed and \( i,j,\ldots \) are also fixed. \(<rel>\) is one of the set \{=,<,\leq,\geq,>\}. Then generate an if-statement as follows:

   ```c
   if (k <rel> f(i,j,...) ) { Block }
   ```

4. Let Block; Block; ... Tel; is translated into

   ```c
   { Block; Block; ... }
   ```

The code generator produced the C-code shown in figure 8.5 for the Durbin algorithm. This code was compiled, executed, and validated using a few examples.
/* system Durbin */
#define N 3

double _r[3]; /* r : { i | 1<=i<=N } of real; */
#define r(i) _r[(i)-1]

double _y[3]; /* y : { i | 1<=i<=N } of real; */
#define y(i) _y[(i)-1]

double _z[6]; /* Z : {t1,t2 | (t2,2)<=t1<=N; 1<=t2} of real; */
#define Z(t1,t2) _z[3*(t1)+(t2)-7]

double _y[3]; /* Y : {t1,t2 | t2-N<=t1<=N; N+1<=t2} of real; */
#define Y(t1,t2) _y[3*(t1)+(t2)-7]

double _b[3]; /* B : {t1,t2 | 1<=t1<=N; t2=0} of real; */
#define B(t1,t2) _b[(t1)-1]

main()
{ int i, t1, t2;
  char _s[32];
  for (i=1; i<=N; i++)
  { printf("Input r[%d] = ",i);
    gets(_s);
    r(i) = atof(_s);
  }
  B(1,0) = 1.0;
  Y(1,4) = -r(1);
  for(t1=2; t1<=N; t1++)
  { B(t1,0) = B(t1-1,0) * (1 - Y(t1-1,4) * Y(t1-1,4));
    Z(t1,1) = 0.0;
    for (t2=2; t2<=t1; t2++)
    { Z(t1,t2) = Z(t1,t2-1) + r(t2-1) * Y(t1-1,t2-2);
    }
    Y(t1,4) = (-r(t1) - Z(t1,t1)) / B(t1,0);
    for(t2=N+2; t2<=t1+N; t2++)
    { Y(t1,t2) = Y(t1-1,t2-1) + Y(t1,4) * Y(t1-1,t1-t2+7);
    }
  }
  for(i=1; i<=N; i++)
  { y(i) = Y(3,-i+7);
    printf("y[%d]= %f\n", i, y(i) );
  }
}

Figure 8.5: C-code for Durbin's Algorithm
The reader can easily see the one-to-one correspondence between this program and the time sorted ALPHA program given at the end of section 8.4.6.

## 8.6 Performance

![Figure 8.6: Performance Comparisons](image)

I compared the execution times of code compiled with the naive compiler (chapter 5) with code created using the synthesizing compiler described in this chapter as well as with code which I wrote and optimized by hand. Figure 8.6 graphs the execution time versus problem size for four examples: Durbin’s algorithm, forward substitution, matrix multiplication, and Gaussian elimination without pivoting. In summary, I found that for large problem sizes the synthesized code was 5 to 11 times faster than the naive code, and the hand optimized code was from 1.5 to 2.5 times faster than the synthesized code (see figure 8.7). The ratios were smaller for smaller sized problems, but asymptotically approached constants for larger problems.

Thus the synthesizing compiler is successful in approaching the speeds of hand optimized sequential code, as compared to the naive compiler. The remaining difference between the hand code and the code made by the synthesizing compiler was due to the fact that the synthesizing compiler does no optimization.
8.7 Open Problem

The code generator described in this section 8.5 is not yet implemented in code. The procedure that I described was executed by hand to produce the examples in this thesis.

8.8 Conclusion

I have described and demonstrated a transformational compiler for \textsc{Alpha} which produces imperative C-code starting from a functional program. To simplify the final mapping to imperative code (code generation), an \textsc{Alpha} program is first transformed into imperative normal form which is very close to a nested loop imperative program. This is done through a sequence of transformations that are facilitated by the \textsc{Alpha} environment.

I have employed static schedule analysis techniques originally developed in the context of systolic array synthesis. These ideas were extended and adapted for doing compilation of functional programs. I have shown that by using a schedule, the context switching which makes naive execution of functional programs expensive, can be eliminated.
Chapter 9

Memory reuse and optimization

Functional languages have suffered from memory inefficiency which has impaired the acceptance and use of these languages by imperative programmers. ALPHA, being a functional language, shares the same problem. In this chapter, I will take advantage of the static analyzability of ALPHA to show how an ALPHA program can be compiled to use memory efficiently. Given a schedule, we determine a function \( \Pi \) which maps points in the single assignment domain of a variable to a smaller shared memory domain. The function \( \Pi \) is a generalization of linear allocation functions used in systolic array synthesis.

9.1 Previous work

Chamski [7] investigated the problem of generating efficient memory data structures to store systolic (single assignment) variables based on a multidimensional timing function common to all variables. He used the set of dependence functions to define the lifetime of a variable, but did not give a constructive method to compute the lifetime from the dependencies. Given the maximum lifetime, he gives a method to compute what he calls a reduction function \( (\Pi) \), which is an injective mapping from a variable’s domain of definition to memory space. He then goes on to describe ways to reduce the maximum lifetime by either changing the order of evaluation (i.e. changing the schedule), or by the introduction of new temporary variables.

In this chapter, I derive a more general formula for constructing the reduction function, which is based on a set of variable dependent, multidimensional timing functions, rather than a single timing function common to all variables. I also give a constructive method for computing the lifetime of variable, based on the usage table constructed in chapter 6. And finally, I give a different and more general construction method for computing the memory projection function, \( \Pi \).

9.2 Problem Statement

I pose the problem of finding a variable dependent, many-to-one, memory mapping function \( \Pi \) which projects a variable \( Y \)’s single assignment domain \( D \) to a smaller shared memory domain \( \text{Image}(D, \Pi) \).

Given:

- a variable \( Y \) defined over domain \( \mathcal{F} \);
- a multidimensional, variable dependent affine scheduling function \( \lambda_Y \) such that \( \forall p \in \mathcal{F} : \lambda_Y p = t \); the time vector when \( Y[p] \) is computed (see section 6.3 for a discussion about multidimensional timing functions);


- a set of usage table entries of the form:

\[ \forall p \in D : \forall Y[p] \Rightarrow \{ \forall r \in \mathcal{E}(p) : X[M\left( \frac{r}{p} \right)] \} \];

\( \mathcal{E} \) is a domain parameterized by \( p \). The usage domain \( D \) is a subset of \( \mathcal{F} \), the entire domain of \( Y \).

Given the above, find a variable dependent, many-to-one, memory mapping function \( \Pi_Y \) which projects a variable \( Y \)'s domain \( \mathcal{F} \) to a smaller “memory” domain \( \text{Image}(\mathcal{F}, \Pi_Y) \) so that no value in memory is overwritten until all instructions which read and use that value have executed.

### 9.3 Problem Analysis

Given any point \( z \) in the domain of a variable \( Y \), there are two important sets of points to consider when finding a shared memory allocation for that variable. The first is the set of points that will use \( Y[z] \), called the utilization set \( [58] \), and the second is the set of points in \( Y \) which are mapped to the same location in memory, but scheduled to be computed after \( z \), called the follow set. Below, I formally define these two sets.

**Definition 10** Utilization Set

Given the entry from the usage table \( \forall p \in D : \forall Y[p] \Rightarrow \{ \forall r \in \mathcal{E}(p) : X[M\left( \frac{r}{p} \right)] \} \); and given a point \( z \in D \), the (sub)domain of \( Y \), the utilization set of \( z \) (with respect to this usage entry) is:

\( \text{usage}_X(z) = \{ y : y = M\left( \frac{r}{z} \right), \left( \frac{r}{p} \right) \in \mathcal{E} \} \), i.e., the set of points of \( X \) that use \( Y[z] \).

The parameterized polyhedron \( \{ r \in \mathcal{E}(p) \} \) can be written as a non-parameterized polyhedron \( \{ \left( \frac{r}{p} \right) \in \mathcal{E} \} \)

by explicitly showing the parameters in the space of \( \mathcal{E} \). Refer to section 3.2.1 for a discussion about parameterized domains.

**Definition 11** Follow Set

Given a point \( z \in \mathcal{F} \), the follow set of \( z \) is:

\( \text{follow}(z) = \{ y \in \mathcal{F} : \Pi z = \Pi y \wedge \lambda z < \lambda y \} \), the set of all other points in \( \mathcal{F} \) which are mapped to the same point in the memory domain \( \text{Image}(\mathcal{F}, \Pi_Y) \), but which are scheduled to be computed after \( z \).

A schedule guarantees that all data flow-dependencies (also called write before read dependencies) are honored. Output-dependencies (write before write) and anti-dependencies (read before write) are not an issue in single assignment memory. However, the moment memory is shared (two array elements are allocated to the same memory location) these other two dependencies become an issue.

Let's assume that we do share memory using a memory projection function \( \Pi \). Since memory is written as soon as an element is computed, output-dependencies are guaranteed by the schedule, as are flow-dependencies. However, anti-dependencies are not guaranteed. Thus, honoring the anti-dependencies is the only requirement for determining the validity of a memory projection. In general, \( \Pi \) is a valid, conflict-free memory mapping function for a variable \( Y \) being used by variable \( X \) iff

\[ \forall z \in D : \text{Max}_{q \in \text{usage}_X(z)} \lambda X q \triangleq \text{Min}_{p \in \text{follow}(z)} \lambda Y p \]  \hspace{1cm} (9.1)

That is, the last usage (read) of any memory location must precede the earliest subsequent write to that same location. Since time is represented by a vector, \( \text{Min} \) and \( \text{Max} \) are the lexicographical minimum and maximum, respectively, of a set of time vectors.

To reduce this equation into something that will help us to construct the projection \( \Pi \) to map the variable \( Y \) onto shared memory, we consider the computation of the \( \text{Max} \) and \( \text{Min} \) separately in the sections below.
9.3.1 Computation of the earliest following write

The function $\Pi$ is a many-to-one mapping (projection) of index points in $\mathcal{D}$ to a memory location in $\text{Image}(\mathcal{D}, \Pi_Y)$. If two points $z_1, z_2 \in \mathcal{D}$ are mapped to the same memory location, then $\Pi z_1 = \Pi z_2$ and thus $\Pi(z_1 \Leftrightarrow z_2) = 0$ which means that $(z_1 \Leftrightarrow z_2) \in \ker(\Pi)$. Let $\rho_1, \ldots, \rho_n$ be a basis for the $\ker(\Pi)$. By convention, the signs of $\rho_i$ are chosen such that $\lambda_Y \rho_i > 0$, then the following are true:

1. $z$ and $z + \rho_i$ are mapped to the same location in the memory domain. This follows from the fact that $\rho_i$ is in the kernel of $\Pi$.
2. $z$ is scheduled to be computed before $z + \rho_i$. This follows from the fact the $\lambda_Y \rho_i > 0$.
3. any point in the follow set of $z$ is $z$ plus a positive combination of the vectors $\rho_1, \ldots, \rho_n$, follow$(z) = z + \sum_{i=1}^{n} w_i \rho_i$, where $w_i$ is the $i^{th}$ element of an integral vector $w > 0$ (all $w_i > 0$). This follows from (1.) and (2.) above.
4. The earliest point in follow$(z)$ is precisely one or more of the points $z + \rho_i$, $1 \leq i \leq n$. This follows from $\lambda_Y \rho_i > 0$ and $w > 0$.

Using this formulation,

$$
\begin{align*}
\text{Min}_{p \in \text{follow}(z)} \lambda_Y p & = \text{Min}_{w > 0} \lambda_Y (z + \sum_{i=1}^{n} w_i \rho_i) \\
& = \lambda_Y z + \text{Min}_{w > 0} \lambda_Y \sum_{i=1}^{n} w_i \rho_i \\
& = \lambda_Y z + \text{Min}_{w > 0} \sum_{i=1}^{n} w_i \lambda_Y \rho_i
\end{align*}
$$

By number 4 above, we know that the earliest point in the follow set will occur at one or more of the points $z + \rho_i$, thus

$$
\begin{align*}
\text{Min}_{p \in \text{follow}(z)} \lambda_Y p & = \lambda_Y z + \text{Min}_{i=1}^{n} \lambda_Y \rho_i
\end{align*}
$$

9.3.2 Computation of the last usage

Using definition 10 of a usage set, and for convenience, letting $\bar{z} = \left( \begin{array}{c} r \\ z \end{array} \right)$, then

$$
\text{usage}_x(z) = \left\{ y : y = \bar{M} \left( \begin{array}{c} r \\ z \end{array} \right), \left( \begin{array}{c} r \\ z \end{array} \right) \in \mathcal{E} \right\}
$$

Then the last usage can be written:

$$
\text{Max}_{y \in \text{usage}_x(z)} \lambda_X y = \text{Max}_{\bar{z} \in \mathcal{E}} \lambda_X \bar{M} \bar{z}
$$

9.3.3 Derivation of a construction for $\Pi$

Given the above formulations for the earliest following write and the last usage, equation 9.1 becomes:

$$
\forall z \in \mathcal{D} : [\text{Max}_{\bar{z} \in \mathcal{E}} \lambda_X \bar{M} \bar{z} \leq \lambda_Y z + \text{Min}_{i=1}^{n} \lambda_Y \rho_i]
$$

68
∀z ∈ D : [Max_{x ∈ E} λx Mz ⇐⇒ λyz] ≤ Min_{i=1}^{n} λy ρi
Max_{x ∈ E} \bigwedge_{z ∈ D}[λx Mz ⇐⇒ λyz] ≤ Min_{i=1}^{n} λy ρi
Let z = Pz
Max_{x ∈ E} \bigcap_{P^{-1} D}[λx Mz ⇐⇒ λy Pz] ≤ Min_{i=1}^{n} λy ρi
Let \( \mathcal{E} = \mathcal{E} \bigcap P^{-1} D \)
Max_{x ∈ E}[(λx Mz ⇐⇒ λy Pz)] ≤ Min_{i=1}^{n} λy ρi
Let \( L = λx Mz ⇐⇒ λy P \)
Max_{x ∈ E} Lz ≤ Min_{i=1}^{n} λy ρi

(9.2)

Let \( m \) be the number of vertices \( v_j \) and \( q \) be the number of rays \( r_i \) of the domain \( \mathcal{E} = \mathcal{E} \bigcap P^{-1} D \). Then from the definition of a polyhedron, any point \( z \in \mathcal{E} \) can be described by the sum of a convex combination of the vertices and a positive combination of the rays, i.e., \( \forall z \in \mathcal{E} \)

\[
z = \sum_{j=1}^{m} c_j v_j + \sum_{j=1}^{q} a_j r_j
\]

where each of the \( a_j \) and \( c_j \)'s are non-negative, and \( \sum_{j=1}^{m} c_j = 1 \). Thus,

\[
\text{Max}_{z ∈ \mathcal{E}} Lz = \text{Max}_{a_j, c_j} L(\sum_{j=1}^{m} c_j v_j + \sum_{j=1}^{q} a_j r_j)
\]

A finite maximum exists only if \( Lr_j ≤ 0 \) for all rays \( r_j \). Then given that the rays are such that \( Lr_j ≤ 0 \), the maximum exists at one of the vertices, \( v_j \). And thus,

\[
d = \text{Max}_{j=1}^{m} Lv_j ≤ \text{Min}_{i=1}^{n} λy ρi
\]

(9.3)

where \( v_j \) are the extremal vertices of \( \mathcal{E} \), and \( ρ_i \) are a basis for \( \ker(\Pi) \). The left hand side of equation 9.3 is \( d \), the time between when a value is written and the last time it is read, and is called the maximum lifetime of variable \( Y \) (with respect to variable \( X \)) [?]. The overall maximum lifetime of variable \( Y \) would be the maximum lifetime over all of \( Y \)'s usage table entries. The specific lifetime of \( Y[p] \) with respect to an entry in the usage table is \( Lp \).

### 9.4 A constructive algorithm to compute \( \Pi \)

In this section, I give an algorithm to construct the memory projection \( \Pi \) for a variable \( Y \) defined in \( \texttt{ALPHAAS} \):

\[
Y : \mathcal{F} \text{ of } \texttt{datatype};
\]

where variable \( Y \) defined as a mapping of points in domain \( \mathcal{F} \) to values in \( \texttt{datatype} \). Given:

- a multidimensional, variable dependent affine scheduling function \( λ_y \) that maps all points \( p \) in \( \mathcal{F} \) to \( λ_y p \), the time vector when \( Y[p] \) is computed, as well as all of the scheduling functions \( λ_x \) of variables \( X \) which use \( Y \);
- the usage table entries for \( Y \) of the general form\(^1\):

\[
∀p ∈ D : Y[p] ⇒ \{∀r ∈ \mathcal{E}(p) : X[\hat{M} \begin{pmatrix} r \\ p \end{pmatrix}]\};
\]

where \( \mathcal{E} \) is a domain parameterized by \( p \). We also derive the projection \( P = (r, p → p) \) from the entry.

\(^1\)A general form: \( ∀p ∈ D : Y[p] ⇒ X[\hat{M}p] \); may also be used, which corresponds to the general form by letting \( \hat{M} = M \), \( \mathcal{E}(p) \) is a 0-dimensional universe domain, the vector \( r \) is also 0-dimension, and \( P = (p → p) \)
1. For each usage table entry, compute \( \tilde{E} \) as follows. If domain \( D \) is defined as \( \{ p \mid Ap + b \geq 0 \} \) and domain \( E(p) \) is \( \{ \begin{pmatrix} r \\ p \end{pmatrix} \mid Cr + Dp + e \geq 0 \} \), then \( \tilde{E} = \{ \begin{pmatrix} r \\ p \end{pmatrix} \mid \begin{pmatrix} C & D \end{pmatrix} \begin{pmatrix} r \\ p \end{pmatrix} + \begin{pmatrix} e \\ b \end{pmatrix} \geq 0 \} \).

2. Compute \( L = \lambda_Y \tilde{M} \Leftrightarrow \lambda_Y P \).

3. Compute the set of vertices \( v_1, \cdots, v_m \) and rays \( r_1, \cdots, r_q \) of domain \( \tilde{E} \). This may be done using the polyhedral library [88], for instance. Then find the maximum lifetime vector \( d \) which is the lexicographical maximum of \( L v_j \), \( 1 \leq j \leq m \).

Alternatively, solve the linear programming problem \( \text{Max} \ p \in \tilde{E} \ L p \), and call the solution \( d \). This may be done using a tool such as PIP [20].

The schedule assures us that \( d \geq 0 \). The vector \( d \) is the lifetime of variable \( Y \).

4. Repeat the above steps for all entries is \( Y \)'s usage table. Retain the lexicographically largest \( d \).

5. Find the largest set of vectors \( \rho_i, 1 \leq i \leq n \), such that the set of \( \rho_i \)'s are linearly independent, and \( \min_{i=1}^n \lambda_Y \rho_i \geq d \). If \( \lambda_Y \) is not full row rank, as is the case when certain time indices are linear functions of the others, then there exists a matrix \( A \) and vector \( b \), such that \( A \lambda_Y + b = 0 \). In such a case, every vector \( \rho_i \) must also satisfy \( A \rho_i = 0 \) and the matrix \( A \) will be a submatrix of \( \Pi \). The vectors \( \rho_1 \cdots \rho_n \) give the feasible space of \( \ker(\Pi) \).

6. Given the set of vectors \( \rho_i, 1 \leq i \leq n \), compute \( \Pi \) such that \( \ker(\Pi) \) is in the space generated by basis vectors \( \rho_1 \cdots \rho_n \) and such that

\[
\begin{bmatrix}
\lambda_Y \\
\Pi
\end{bmatrix}
\]

is full column rank. This last condition guarantees that two points scheduled at the same time will be assigned to different memory locations, and two points assigned to the same memory location will be computed at different times. A result of this is that the rank of \( \Pi \) is limited to the number of indices of \( Y \) minus the row rank of \( \lambda_Y \) (the number of linearly independent temporal indices).

### 9.5 Example

As an example, we will investigate memory sharing for the forward substitution example developed earlier in chapter 6.

#### 9.5.1 Memory reduction for variable \( x \)

The usage table entry for \( x \) is:

\[
\begin{align*}
\{ Z, t, i \mid Z = i; \ t = 2Z - 1; \ 1 \leq Z \leq N - 1 \} & : x[Z, t, i] \Rightarrow \\
\{ j, Z, t, i \mid Z + 1 \leq j \leq N \} & : b[i, t + 1, j]
\end{align*}
\]

Since the schedule has already been integrated into the equations at this point, the timing functions \( \lambda \) simply select the time component from the variables:

\[
\begin{align*}
\lambda_x &= [(Z, t, i \rightarrow Z, t), \text{ where } t = 2Z \Leftrightarrow 1] = (Z, t, i \rightarrow Z, 2Z \Leftrightarrow 1) \\
\lambda_b &= [(Z, t, i \rightarrow Z, t), \text{ where } t = 2Z] = (Z, t, i \rightarrow Z, 2Z) \\
\tilde{M} &= (j, Z, t, i \Leftrightarrow > i, t + 1, j) \\
\tilde{D} &= \{ Z, t, i \mid Z = i; \ t = 2Z \Leftrightarrow 1; \ 1 \leq Z \leq N \Leftrightarrow 1 \} \\
\tilde{E} &= \{ j, Z, t, i \mid Z + 1 \leq j \leq N \}
\end{align*}
\]
\[ P = (j, Z, t, i \rightarrow Z, t, i) \]
\[ \hat{\mathcal{E}} = \{j, Z, t, i | Z + 1 \leq j \leq N; Z = i; t = 2Z \Leftrightarrow 1; 1 \leq Z \Leftrightarrow 1 \} \]
\[ v_1 = (2, 1, 1, 1) \]
\[ v_2 = (N, N \Leftrightarrow 1, 2N \Leftrightarrow 3, N \Leftrightarrow 1) \]
\[ v_3 = (N, 1, 1, 1) \]
\[ L = \lambda_b \bar{M} \Leftrightarrow \lambda_x P = (j, Z, t, i \rightarrow Z, t, i) \]
\[ Lv_1 = (0, 1) \]
\[ Lv_2 = (0, 1) \]
\[ Lv_3 = (0, 1) \]
\[ d = (0, 1) \]
\[ \lambda_x \rho \geq d \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix} \rho \Leftrightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 1 \end{pmatrix} \]
\[ \rho_1 = (1, 2, 0) \]
\[ \Pi = (Z, t, i \Leftrightarrow > 2Z \Leftrightarrow t, i) \text{ (selecting } \rho_1) \]
\[ \begin{bmatrix} \lambda_x \\ \Pi \end{bmatrix} = \begin{bmatrix} Z \\ 2Z \Leftrightarrow 1 \\ 2Z \Leftrightarrow t \\ i \end{bmatrix} \]

### 9.5.2 Memory reduction for variable \( b \)

The usage table entries for \( b \) are shown below:

\[
\{ Z, t, i \mid t = 2Z; 0 \leq Z \leq i - 2; i \leq N \} : b[Z, t, i] \Rightarrow b[Z + 1, t + 2, i] \\
\{ Z, t, i \mid Z = i - 1; t = 2Z; 1 \leq Z \leq i - 1 \} : b[Z, t, i] \Rightarrow x[Z + 1, t + 1, i] 
\]

As above, the timing functions are:

\[
\lambda_b = [(Z, t, i \rightarrow Z, t), \text{ where } t = 2Z] = (Z, t, i \rightarrow Z, 2Z) \\
\lambda_x = [(Z, t, i \rightarrow Z, t), \text{ where } t = 2Z \Leftrightarrow 1] = (Z, t, i \rightarrow Z, 2Z \Leftrightarrow 1) 
\]

For the first entry in the usage table for \( b \):

\[
\bar{M} = (Z, t, i \Leftrightarrow > Z + 1, t + 2, i) \\
\bar{D} = \hat{\mathcal{E}} = \{Z, t, i | t = 2Z; 0 \leq Z \leq i \Leftrightarrow 2; i \leq N; N \geq 2\} \\
P = (Z, t, i \rightarrow Z, t, i) \\
v_1 = (1, 2, 3) \\
v_2 = (1, 2, N) \\
v_3 = (N \Leftrightarrow 2, 2N \Leftrightarrow 4, N) \\
L = \lambda_b \bar{M} \Leftrightarrow \lambda_b P = (j, Z, t, i \rightarrow 1, 2) \\
Lv_1 = (1, 2) \\
Lv_2 = (1, 2) \\
Lv_3 = (1, 2) \\
d = (1, 2) 
\]

For the second entry in the usage table for \( b \):

\[
\bar{M} = (Z, t, i \rightarrow Z + 1, t + 1, i) 
\]
\[ D = \tilde{E} = \{ Z,t,i | Z = i \Leftrightarrow 1; t = 2Z; 1 \leq Z \leq N \Leftrightarrow 1; N \geq 2 \} \]

\[ P = (Z,t,i \rightarrow Z,t,i) \]

\[ v_1 = (1,2,2) \]

\[ v_2 = (N \Leftrightarrow 1,2N \Leftrightarrow 2,N) \]

\[ L = \lambda_{i} \tilde{M} \Leftrightarrow \lambda_{i} P = (j,Z,t,i \rightarrow 1,1) \]

\[ L v_1 = (1,1) \]

\[ L v_2 = (1,1) \]

\[ d = (1,1) \]

Combining the result of the two entries in the usage table:

\[ d = \text{Max}((1,2),(1,1)) = (1,2) \]

\[ \lambda_{i} \rho \succeq d \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix} \rho \succeq \begin{pmatrix} 1 \\ 2 \end{pmatrix} \]

\[ \rho_1 = (1,2,0) \]

\[ \Pi = (Z,t,i \Leftrightarrow > 2Z \Leftrightarrow t,i) \quad \text{(selecting } \rho_1 \text{)} \]

\[ \begin{bmatrix} \lambda_{i} \\ \Pi \end{bmatrix} = \begin{bmatrix} Z \\ 2Z \Leftrightarrow t \\ i \end{bmatrix} \]

### 9.6 Open Problems

In this chapter, I assumed that the timing functions were given for all of the variables. However, the concepts in this chapter could be extended to constrain the derivation of timing functions using the amount of memory that each timing function would require. Thus timing functions could be derived so as to reduce or minimize the use of memory.

Also, when generating imperative code, spatial indices of variables will also need to be timed, so the code can be totally ordered sequentially. Since there are no data dependencies to constrain the timing of spatial indices, these timing functions could be chosen under the constraint of memory minimization.

### 9.7 Conclusion

Code compiled from functional languages has suffered from inefficient use of memory. In this chapter, I have showed how to analyze \textsc{Alpha} in order to construct a function \( \Pi \) to project a single assignment variable onto shared memory space in order to use the least amount of memory possible. I believe the efficient use of memory will be required to convince imperative programmers that functional languages are a viable alternative to imperative languages.
Chapter 10

Parallel code from ALPHA

10.1 Introduction

In this chapter, I show what additional capabilities are needed to extend the sequential code compiler, described in previous chapters, so that ALPHA can be compiled into imperative data-parallel code. Recurrence equations are inherently data-parallel. The ALPHA recurrence equation model directly exposes potential parallelism since it only forces sequential dependencies on variables which are functionally dependent. This makes ALPHA an ideal language to express parallel algorithms.

After scheduling, certain indices are denoted as temporal. All the sequentiality in an algorithm is encoded into those indices. The remaining indices are denoted as spatial. The set of points mapped to a given time instant, and which differ only in spatial indices, may all be computed in parallel. Thus the existence of spatial indices implies the existence of parallelism in an algorithm.

Durbin Example The ALPHA code for the Durbin algorithm (presented in chapter 8), after being scheduled, was:

```plaintext
system Durbin (N : {N | 1<N}) parameter;
  r : {i | 1<i<N} of real);
  returns (y : {i | 1<i<N} of real);
var
  Z : {t1,t2 | (t2,2)<t1=<N; 1<=t2} of real;
  Y : {t1,t2 | t2-N<=t1=<N; N+1<=t2} of real;
  B : {t1,t2 | 1<=t1=<N; t2=0} of real;
let ... tel;
```

Notice that all of the local variables Z, Y, and B employ only temporal indices t1 and t2. Thus there is no parallelism to be found in this algorithm using this schedule.

Forward Substitution Example The ALPHA code for the forward substitution example (presented in chapter 6), after being scheduled, was:

```plaintext
system ForwardSub : {N | N>1} parameter;
  returns ( X : {i | 1<i<=N} of real );
var
  x : {Z,t | ZZ=t+1; 1<=t<=2N-1} of real;
  b : {Z,t,i | ZZ=t; 0<=t<=2i-2; 2<=i<=N} of real;
let ... tel;
```
In this example, both $Z$ and $t$ were temporal indices. The local variable $x$ depends on only temporal indices and therefore cannot be computed in parallel. However, the local variable $b$ depends on spatial index $i$, and therefore can be computed in data-parallel. Since $i$ is bounded by $2 \leq i \leq N$, we can see that $N \leftrightarrow 1$ computations can be done in parallel by performing them on different processors.

10.2 The data-parallel model

A data-parallel model of parallel computation is a SIMD model with an unbounded number of processing elements and a global name space [32]. In practice, today’s parallel processors actually support the similar SPMD (single program, multiple data stream) model in which processors execute identical programs, but not necessarily in lock-step (as in SIMD). Processors only synchronize when they need to communicate with each other or with the host system.

The Alpha compiler obtains a data-parallel program by factoring out the loops which scan the processor indices as the outermost loops. What is left in the body of the processor space loop(s) is an imperative program which is parameterized by the processor indices. This program is locally sequential, executing like a normal sequential program from its own point of view, and globally parallel, since instances of this program for each distinct set of processor indices execute in parallel. The only difference is that they operate on different data sets, as determined by the processor indices, and they communicate with each other:

\[
\begin{array}{l}
\text{var } X : \{ \cdots, p_1, p_2, \cdots \} \text{ of integer;}
\{ p_1, \cdots \} ::= \\
\{ p_2, \cdots \} ::= \\
\text{let }
\begin{array}{l}
A \text{ data-parallel program} = \\
A \text{ sequential imperative program parameterized by processor indices: } p_1, p_2, \cdots
\end{array}
\end{array}
\]

tel;

A data-parallel program is imperative, which is why it is possible to directly extend results in generating imperative sequential code to generating data-parallel code. The code for each processor takes a local view of computation.

The parallelism comes from the fact that the data is distributed and each processor computes with its own data set. This is the source of the term data-parallelism. It is based on decomposition and distribution of the data to the processors. This simple technique produces high performance scalable code for a large class of applications. The problem of how data is distributed will be addressed later in this chapter.

Processors may access data in other processor’s data sets. The mere act of referencing a non-local data item invokes data communication. For example, a dependency of $a[p, \cdots] \rightarrow b[p + 1, \cdots]$ would signal a communication between processor $p$ and $p + 1$. Thus interprocessor communication is represented through simple data referencing in expressions, rather than by doing explicit communication. Since there is a single locus of control (the schedule) race conditions and deadlock are not possible, assuming the compiler correctly places the proper synchronization points in the code. This problem was addressed by Hatcher and Quinn [32] in connection with their data-parallel-C compiler.

10.3 Introducing Pseudo Linearity

In order to allocate variables to a fixed size processor space, variables are partitioned into blocks which are allocated to processors, or they can be allocated cyclically, or a combination of both. In order to
accommodate this kind of partitioning, we must introduce pseudo linearity into the ALPHA code.

This is done by adding two redundant indices (which are called $i_q$ and $i_r$ in the example below), to the domain of a variable, in such a way that $i_q = i \text{ div } 3$ and $i_r = i \text{ mod } 3$. The block size (3 in this example) must be a constant in order to make the additional domain constraints affine.

For example, to block variable $A$ in the dimension of index $i$ with blocksize 3, the following transformation would be made:

$$A: \{ i \mid \cdots \} \text{of } \cdots \quad \Rightarrow \quad A = \{ i, i_q, i_r \mid i = 3i_q + i_r; 0 \leq i_r < 3; \cdots \} \text{of } \cdots$$

$$\text{let} \quad A = \cdots \text{reduce}(+, (i, i_q, i_r \rightarrow i), A) \cdots, (i, i_q, i_r \rightarrow i);$$

$$\cdots = \cdots \text{reduce}(+, (i, i_q, i_r \rightarrow i), A) \cdots;$$

$$\text{tel;}$$

The reduce operation simply does a projection of $A$ from $(i, i_q, i_r)$ back to $(i)$. This projection is always one to one. At present, the reduction is the only ALPHA construct that allows us to do a projection. Its use here is awkward. It is an open problem of how to extend the ALPHA language to do one-to-one projections in a better way.

![Figure 10.1: Embedding of domain $A$ into a higher dimension](image)

The new domain generated for variable $A$ is illustrated in figure 10.1. A horizontal plane at $(i = \text{any integer})$ only passes through a single integer point on the domain surface.

### 10.4 Data Decomposition and Distribution

In ALPHA, the question of how to distribute data is answered by finding a change of basis for each variable which maps one or more spatial indices to a common processor space. For example, say a variable is dimensioned $(t, i, j)$ where $t$ is a temporal index, and $i$ and $j$ are spatial indices, then a mapping of $p = i$ would correspond to an allocation by row (see figure 10.2-a), whereas a mapping of $p = j$ would correspond to an allocation by column (see figure 10.2-b).
10.4.1 Block Partitioning

Given a block size of 3 (for example), then $i$ would be expanded with two redundant indices $i_q$ and $i_r$ as follows:

\[
\text{Var}: \{\cdots i \cdots | \cdots \} \iff \text{Var}: \{\cdots i, i_q, i_r \cdots | i = 3i_q + i_r; 0 \leq i_r < 3; \cdots \}
\]

If $i_q$ is mapped to a processor index $p$, then the allocation is a block partitioning. For example, figure 10.2-c. shows a row block partitioning with a block size of 3 and figure 10.2-d. shows a row block partitioning with a block size of 2.

The code generator recognizes this pattern and generates the proper code for the \textit{mod} and \textit{div} functions as shown in section 8.5.2, rule 2-a.
10.4.2 Cyclic Partitioning

If $i_r$ is mapped to a processor index $p$, then the allocation is a cyclic partitioning. For example, figure 10.2-e. shows a row cyclic partitioning with a block size of 3 and figure 10.2-f. shows a row cyclic partitioning with a block size of 2.

```
var X : {⋯, i_q, p, ⋅⋅⋅ | ⋅⋅⋅} of integer;

{p, ⋅⋅⋅ | 0 ≤ p < 3; ⋅⋅⋅} ::  processor index loop
    let {t, ⋅⋅⋅ | ⋅⋅⋅} ::  time index loop
        let {i_q, ⋅⋅⋅ | i = 3i_q + p; ⋅⋅⋅} ::
            let tel;
            tel;
```

10.4.3 Other Partitionings

By pseudo-linearizing more that one index, more complex partitioning can be created. For instance, if one index is block-partitioned like figure 10.2-c, and a second is cyclic-partitioned like figure 10.2-f, then combined they create a block-cyclic partitioning as shown in figure 10.2-g.

In a similar way, block-partitions in figure 10.2-c and figure 10.2-d combine to form a block-block partitioning as shown in figure 10.2-h. Thus, there is quite a bit of flexibility in the ways domains can be partitioned.

10.5 Creating a data-parallel program

In summary, the steps for generating a data-parallel program are:

1. Choose a processor space.
2. Partition each data variable among the processors by mapping some or all of a variable’s spatial indices to processor indices.
3. Factor out the processor loops.
4. Process the body of the loop as usual for a sequential imperative program.
5. Generate data-parallel code, using a modified code generator.

The selection of the processor space is partially set by the architecture. The choice is generally either a linear array of processor (the exact number set by the architecture) or a 2-dimensional grid of processors.

The partitioning of variables among processors fixes what data-parallelism in the algorithm will be exploited, and which indices will need to be sequentialized. The choice of partitioning functions may be done so as to reduce communication, balance the workload, and/or maximize the amount of computation done per communication on each processor. When well partitioned, data-parallel programs can often achieve nearly linear speedup.

10.6 Open Problems

The development of a code generator for parallel code is still an open problem. An important subproblem of code generation is the generation of communication between processors.
10.7 Conclusion

In this chapter, I have extended the compiler described in chapter 8 to generate data-parallel code. The data-parallel extension to the Alpha compiler is not fully implemented, and there remains much work to be done. The main idea is to factor out certain spatial indices, and map them to processor indices. The code left in the body of the processor index loops is an imperative data-parallel program. Extensions to the code generator are made to handle processor indices as parallel spatial indices and do communication. The remaining spatial indices need to be sequentialized, with the goal of minimizing the memory storage needs of each variable as discussed in chapter 9.
Chapter 11

Conclusion

This thesis has brought together the concepts of dependency and timing analysis from the field of systolic array synthesis along with loop synthesis and retiming from parallelizing compilers and applied them to the compilation of ALPHA. The ALPHA environment has provided the framework to integrate and extend research from these fields in order to derive imperative code from an ALPHA program.

The compilation of functional programs into imperative code has been a difficult problem. This thesis demonstrates that, for at least an important but restrictive class of non-trivial functional programs, efficient imperative code can be derived.

Several open problems have been identified and described. This thesis lays the groundwork for doing that and other research. The Mathematica–ALPHA environment has proved to be a friendly research environment which accommodates experimentation and allows the integration of the work of many researchers.

11.1 Review of goals

Several research goals were proposed in the introduction. To conclude, it would be appropriate to review those goals, and comment on what has been done. In doing so, I summarize most of what has been done in this thesis.

• **Demonstrate the feasibility of the synthesizing compiler methodology, which is to synthesize an imperative data parallel program from the functional language ALPHA using an analysis guided, transformation-based compiler.**

  Chapter 6 considered the dependency and usage analysis of an ALPHA program. From dependency information, timing analysis is performed. With usage information, memory allocation analysis is done as was shown in chapter 9. Communication analysis can also be derived from usage information [74].

  Based on this analysis, and employing the basic transformational capabilities of ALPHA described in chapter 7, I demonstrated the transformation of ALPHA to a sequential imperative form in chapter 8. In the same chapter, a code generator was also described. In chapter 10, I discussed how to extend this transformational compiler to produce data parallel code.

• **Demonstrate that the functional language ALPHA can be analyzed in order to optimize a program for a given target architecture. Show how this analysis can be used to formulate automatic synthesis as either an optimization or a constrained search problem. Report on the issues involved in doing such an analysis.**

  The major analyses required to compile sequential imperative code are timing analysis and memory reuse analysis. Source programs with reduction operations also need to be serialized. There is
generally only a small number (usually 2) of serialization directions that can be chosen, however the choice of direction can profoundly affect the final code. Thus, it is feasible to search the space of possible serializations. Timing analysis also gives a feasible space of timing functions. Most often, only the vertices of this space are of interest. And again, the choice of timing function can greatly affect the structure of the final code. At this time, lacking better timing analyzers, timing functions are selected by hand. Memory reuse analysis, as described in chapter 9 also allows for some degree of freedom, however, choice of one memory projection over another changes the code very little, and thus, this analysis is more amenable to automation.

When compiling to parallel imperative code, additional analysis must be done to choose processor allocation functions for each of the variables. This analysis is done so as to minimize communication while maximizing parallelism. Details of this analysis were not discussed in this thesis. However, usage analysis, on which communication is based, was described and implemented (see section 6.2). Any procedure for analyzing processor allocation functions that reduce communication will be predicated on this usage analysis.

- **Confirm that transformational program derivation is an effective tool for synthesizing compilers and show to what extent the selection of transformations can be automated.**
  
  Demonstrate this by implementing the compiler.

Chapter 8 demonstrated the power of the transformational technique for doing compilation. An ALPHA program was transformed from its original specification, through a series of steps, to a timeordered statement form of ALPHA which I called imperative normal form. The ALPHA language itself was extended so that the bulk of the compilation work could be performed from within limits of the ALPHA language syntax and semantics described in chapter 2.

The last part of the compilation process was better automated than the beginning of the process. At the beginning, the serialization of reductions and the choice of timing functions were specified by hand. And for parallel code, the processor allocation functions were also chosen by hand. However, given these functions, the transformations to implement these functions in the code is automated. A few other transformations also needed to be specified by hand, such as the introduction of local temporary variable to interface between output variable, and the code. I believe that with enough experience, heuristics could be derived to automate these transformations. Once everything was in place, the placing of code in imperative normal form and the generation of code proceeded automatically. The code generator was described, and its implementation is believed to be straightforward. However, for this thesis, the code generation procedure was performed by hand.

- **Demonstrate that an imperative data parallel language is an appropriate target when starting from the functional language ALPHA. Describe the advantages and disadvantages of using a data parallel compiler as a back end to a parallelizing compiler. Describe tradeoffs made in the work done by the parallelizing compiler versus the data parallel compiler.**

Even though ALPHA is itself a data parallel language, I found that much of the work that was done by the data parallel compiler, (such as partitioning the data) was better done in the synthesizing compiler, while I still had the results of dependency and usage analysis. This information is derived and used to guide the compilation process, and then discarded after code generation. The data parallel compiler either has to recompute this information (which is difficult to do from the imperative form), or cope without it. I found that the code I generated was at a lower level of abstraction than what was required by the data parallel compiler. I thus have reconsidered the idea of using a data parallel compiler as a backend. My current thinking is that using the information at the disposal of the synthesizing compiler, that it can do a better job of generating communication than the data parallel compiler. This is a subject for future research.
11.2 Open Problems and Future Work

Besides the open problems which have been named at the end of the chapters, I pose these additional open problems.

**Dealing better with Reductions** The strategy of always serializing reductions at the beginning is too restrictive. The ability to generate naive code for programs containing reductions would be a useful feature, and has been requested by ALPHA users.

For parallel imperative code, being able to schedule a program containing reduce operations would allow parallel programs to perform reductions in conjunction with communication. This is what is done in the data-parallel-C compiler [32].

For sequential imperative code, all reductions do need to be serialized. However, serializing entire simple expressions that contain reduce operations often leads to more efficient serializations. For instance:

\[ B[i] - \text{reduce}(+, (i,j->i), x[i]*A[i,j]) \]

could be serialized by initializing the serialization with \( B[i] \) (instead of 0) and then subtracting (instead of adding) \( x*A \) terms at each step of the serialization. This eliminates one subtraction and makes the code shorter. A second example:

\[ \text{reduce}(+, (i,j->i), A[i,j]*B[i,j]) + \text{reduce}(+, (i,j->i), A[i,j]*C[i,j]) \]

could combine the serializations of both reductions into a single serialization. Often, these more encompassing serializations lead to more compact and efficient imperative code. Thus, additional transformations involving reduction operations are needed.

**Optimizations** ALPHA does a meticulous job at keeping track of all special end cases and code is generated for all of these cases. Some of these end conditions can be generalized and computed in the course of the normal code body. This is done in ALPHA by merging variables together when possible, in order to reduce the size of the code. Other optimization techniques undoubtedly exist as well. Heuristics which try to reduce the number of special cases, and simplify the final imperative code, need to be developed.

**Automization** Continuing work is needed to integrate analysis with the transformational synthesis process to further automate the compilation processes.

**Domain Visualization** As a veteran ALPHA user myself, I can attest to the need for a good visualization tool to draw domains of 2 dimensions and higher, as well as drawing dependency and usage graphs.

**Code Generation for Index Expressions** In ALPHA, all index expressions are computed using closed form equations. There is currently no facility in ALPHA to write recurrence equations to compute indices. After code generation, index expressions can often be simplified using strength reduction, which effectively replaces a closed form index expression with an index recursion. This is another type of code optimization.

11.3 Final Thoughts

This work has been motivated by a belief that there is much to be gained by bringing together research from the fields of regular array synthesis, parallelizing compilers, and functional languages. This thesis has tried to integrate ideas from all three of these areas. Each of these fields carry tremendous bodies of research and I was only able to use a small part of it. However, I believe that I proved the point that
problems in the compilation of functional languages can be effectively attacked using tools and techniques
developed for systolic array synthesis and parallelizing compilers.

There is still much research which can be done, building on the ideas and concepts found in this thesis. I hope that others may find this work interesting and useful and perhaps a few may choose to continue where this thesis leaves off.
Bibliography


Alpha Syntax and Semantics

This appendix specifies the syntax of Alpha. It was extracted from the yacc file of the actual Alpha parser.

Comments in Alpha start with a double dash (--) and terminate with the end of line (like C++).

Meta Syntax

phrase* === zero or more repetitions of phrase
phrase1 / phrase2 === Alternation, either phrase1 or phrase2
( ... ) === syntactic grouping
[ ... ] === optional semantic phrase
**bold** === a terminal
*italic* === a non-terminal

Systems

system :: system system-name ([input-declaration-list])
returns (output-declaration-list);
[var local-declaration-list]
equation-block

system-name :: id
input-declaration-list :: var-declaration-list
output-declaration-list :: var-declaration-list
local-declaration-list :: var-declaration-list

Declarations

var-declaration :: id-list: [domain of]/
  id-list: domain parameter (integer / boolean / real)

var-declaration-list :: [var-declaration-list]; var-declaration

Domains

domain :: \{ index-list | constraint-list \} /
  domain | domain /
  domain & domain /
  domain . function /
  domain . convex /
  ~ domain /
  ( domain )

index-list :: [ index-list , ] id
constraint-list :: [constraint-list ;] constraint
constraint :: increasing-sequence / decreasing-sequence / equality-sequence
increasing-sequence :: (increasing-sequence / index-expression-list) (< / <=) index-expression-list
decreasing-sequence :: (decreasing-sequence / index-expression-list) (> / >=) index-expression-list
equality-sequence :: (equality-sequence / index-expression-list) \= index-expression-list
Equations

\[
\text{equation-block :: let equation-list tel;}
\]
\[
\text{equation-list :: [ equation-list ] equation}
\]
\[
\text{equation :: id = expression ; /
\text{id [ [ index-list ] ] = expression ; /
\text{equation-block} /
\text{domain : equation} /
\text{domain :: equation}
\]

Expressions

\[
\text{expression :: case expression-list ; esac /}
\]
\[
\text{if expression then expression else expression /}
\text{domain : expression} /
\text{expression . function /}
\text{expression [ [ index-expression-list ] ] /}
\text{expression binary-op expression /}
\text{unary-op expression /}
\text{binary-op ( expression , expression ) /}
\text{reduce ( commutative-op , function , expression ) /}
\text{( expression ) /}
\text{id /}
\text{constant}
\]
\[
\text{expression-list :: [ expression-list ; ] expression}
\]
\[
\text{binary-op :: commutative-op / relative-op / \(-\) / \(\div\) / \(\mod\)
\text{com mutative-op :: \(+\) / \(*\) / \(\text{and}\) / \(\or\) / \(\xor\) / \(\text{min}\) / \(\text{max}\)
\text{relative-op :: \(=\) / \(<\) / \(<\le\) / \(>\) / \(\ge\)
\text{unary-op :: \(-\) / \(\not\)
\text{constant :: integer-constant / real-constant / boolean-constant}
\]

Dependence Functions and Index Expressions

\[
\text{function :: ( [ index-list ] \rightarrow [ index-expression-list ] )}
\]
\[
\text{index-expression-list :: [ index-expression-list , ] index-expression / ( index-expression-list )
\text{index-expression :: index-expression ( \(+\) / \(-\) index-term / \(\[\sim\]\) index-term}
\text{index-term :: integer-constant id / integer-constant / id}
\]

Terminals

\[
\text{integer-constant :: [ \{-\} number}
\text{real-constant :: [ \{-\} number . number}
\text{boolean-constant :: true / false}
\text{number :: digit digit*}
\text{digit :: 0 / 1 / ... / 9}
\text{id :: letter ( letter / digit )*}
\text{letter :: a / ... / z / A / ... / Z / _}
\]

90
**ALPHA semantics**

In this section, the semantics of ALPHA are specified.

1. **ALPHA** is a single assignment language. The definition of a variable must be unique at each point in its domain.

2. Only single system programs are supported, no subsystems are supported at this time.

3. Variables may be declared over domains consisting of unions of polyhedra. Polyhedra are defined with both equalities and inequalities.

4. Scalar variables are declared over domains of dimension zero.

5. All data types are assumed to be infinite precision and include the special value **error** which means an uncomputable value. Some operational semantics also require a value **undefined** to represent a value which is not (yet) computed.

6. A **parameter** declaration, if it exists, must be the first declaration in the input declaration section of a system. The parameter domain defines the range of values that parameters can take in that system.

7. The precedence of expression operations is defined in the following table:

<table>
<thead>
<tr>
<th>Expression Operation</th>
<th>Associative</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce(\text{commutative-op, projection-function, Expression})</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>binary-op(\text{Expression, Expression})</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>\text{Expression.affine-function} (change of basis)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>\text{Expression[index-expression*]} (value selection)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>- (negation)</td>
<td>Y</td>
<td>9</td>
</tr>
<tr>
<td>* (multiplication)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>/ (division), div (integer division), mod (modulo)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>+ (addition)</td>
<td>Y</td>
<td>7</td>
</tr>
<tr>
<td>- (subtraction)</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>\text{&lt;, \leq, =, \geq, &gt;, \neq}</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>not</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>\text{and, min, max}</td>
<td>Y</td>
<td>4</td>
</tr>
<tr>
<td>or, xor</td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>\text{Domain: Expression} (restriction)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>if \text{Expression} then \text{Expression} else \text{Expression}</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>case (\text{Expression:}* esac</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

8. The precision of arithmetic is unspecified (assumed infinite).

9. The domains of expressions within a case statement do not intersect each other.

10. The precedence of domain operations is defined in the following table:

<table>
<thead>
<tr>
<th>Domain Operation</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (inversion)</td>
<td>4</td>
</tr>
<tr>
<td>\text{Domain.affine-function} (change of basis)</td>
<td>3</td>
</tr>
<tr>
<td>\text{Domain.convex} (convex hull)</td>
<td>3</td>
</tr>
<tr>
<td>&amp; (intersection)</td>
<td>2</td>
</tr>
<tr>
<td>1 (union)</td>
<td>1</td>
</tr>
</tbody>
</table>