FORMAL IMPLEMENTATION OF HIGH-LEVEL LANGUAGES FOR DATA-PARALLEL PROGRAMMING*

Deb Banerjee

Technical Report PCS-TR92-186

*A Thesis
Formal Implementation of High-Level Languages for Data-Parallel Programming

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
by
Deb Banerjee
DARTMOUTH COLLEGE
Hanover, New Hampshire
August, 1992

Examiner Committee:

(Chairman) Clifford J. Walinsky
Siddhartha Chatterjee
Barry S. Fagin
David F. Kotz

Dean of Graduate Studies
ABSTRACT OF THE THESIS

Formal Implementation of High-Level Languages for Data-Parallel Programming

by Deb Banerjee
Thesis Chairman: Clifford J. Walinsky

The success of parallel architectures has been limited by the lack of high-level parallel programming languages and useful programming models. The data-parallel model of programming has been demonstrated to be useful and natural on a wide variety of parallel architectures. This dissertation presents a set of formal techniques for compiling high-level languages based on data-parallelism.

These techniques have been developed in the context of the high-level language FP*. FP* is a data-parallel dialect of the functional language FP which supports nested collections of data objects, polymorphism and higher-order functions. FP* is suitable for data-parallel programming because its basic data type is an aggregate of other primitive data types, and its primitive functions operate on aggregates as a whole. The compiler translates FP* programs into low-level programs with forall loops, where parallelism arises from simultaneous execution of all loop iterations. The compiler uses an inference system to determine types of data objects and sizes of arrays in the program. Structure inference makes possible compile-time optimizations that reduce synchronization and storage requirements at run-time on parallel machines. High-level languages organized around aggregates tend to suffer due to the creation and copying of large intermediate data structures. On parallel architectures, copying of large data structures may require inter-processor communication, which can be extremely expensive compared to local computation. The FP* compiler significantly minimizes the copying of large data structures, thereby reducing inter-processor communication on parallel computers. The compiler also optimizes data layout to improve load balance of compiled programs. These techniques have been devised in a formal framework. This dissertation presents a formal description of the compiler using a syntactic function that produces low-level data-parallel programs given an FP* function as input.
The effects of the compiler optimizations are demonstrated through an implementation on the Connection Machine CM-2. Running times are presented for a set of benchmark programs written in FP* and compiled into CM Fortran using the compiler reported in this dissertation. These timings demonstrate the performance improvements obtained through the compiler optimizations that are formally specified in this thesis.
Acknowledgements

I would like to express my thanks to my committee members, friends and others who helped make this possible. My thesis advisor, Cliff Walinsky, allowed me considerable breadth and scope in exploring various research areas, and patiently listened to and helped clarify my research ideas. I would also like to thank Sid Chatterjee, Dave Kotz and Barry Fagin for agreeing to be on my thesis committee and carefully reading through my drafts. Finally, I would also like to thank my family and friends who were a persistent source of encouragement and intellectual stimulation.
Table of Contents

Abstract ...................................................... ii
Acknowledgements ........................................... iv
Table of Contents ........................................... v
List of Tables ............................................... ix
List of Figures ................................................ x
1. Introduction ............................................... 1
   1.1. Motivation ............................................. 1
   1.2. Overview .............................................. 2
   1.3. Contributions ......................................... 3
   1.4. Related Work .......................................... 4
   1.5. Organization of the Dissertation ..................... 6
2. Parallel Programming in FP* ............................... 8
   2.1. Introduction .......................................... 8
   2.2. Languages for Data-Parallel Programming ............ 8
      2.2.1. Aggregate Data Types ............................ 8
      2.2.2. Aggregate Operations ............................ 9
   2.3. FP*: An Overview .................................... 9
      2.3.1. Data Objects .................................... 10
      2.3.2. Primitive Functions ............................. 10
      2.3.3. Higher-Order Functions ......................... 11
      2.3.4. Automatic Coercion Between Aggregates .......... 13
   2.4. Comparison with FP ................................ 13
   2.5. Programming Examples ............................... 14
   2.6. Algebraic Program Transformations .................. 21
   2.7. Enhancements ........................................ 22
3. Structure Inference ..................................... 24
   3.1. Introduction ........................................ 24
   3.2. Structure ............................................ 25
      3.2.1. Ground Structures .............................. 25
      3.2.2. Non-Ground Structures ......................... 26
   3.3. Two Distinct Views of Collections .................. 27
   3.4. Structure Mappings ................................ 27
   3.5. Structure Inference System ......................... 29
4. Routing Function Definitions

4.1. Introduction
4.2. Routing Functions
4.3. Programming Examples
4.4. Routing Function Definition
   4.4.1. Index References
   4.4.2. Index Equalities
   4.4.3. Ground Routing Function Definitions
   4.4.4. Structure of Index References
   4.4.5. Syntactic Restrictions on the Form of Index Equalities
4.5. Correctness of Ground Routing Function Definitions
   4.5.1. Well-Formed Index Equalities
   4.5.2. Collision Free
   4.5.3. Complete Definition
4.6. Non-Ground Routing Function Definitions
   4.6.1. Examples of Routing Function Definitions
   4.6.2. Example: Correctness of Routing Function $apnd$
4.7. Scalar Routing Function Definitions
4.8. Routing Functions in Other Programming Languages
4.9. Enhancing the Expressiveness of Routing Function Definitions

5. Allocation Functions

5.1. Introduction
5.2. Allocation Functions
   5.2.1. Syntax and Semantics
   5.2.2. Examples
   5.2.3. Syntactic Restrictions on Index Equalities in Allocation Functions
   5.2.4. Correctness of Allocation Functions
5.3. Allocation Function Evaluation
5.4. Composing Routing Function Definitions with Allocation Functions

6. The Code Generation Function

6.1. Introduction
List of Tables

7.1. Parallel array operations described using forall and equivalent array assignment statements ........................................... 82
7.2. Timings (in secs) of forall statements versus array assignment statements ................................................................. 82
7.3. Timings (in Seconds) for sum-reduction using forall statements and the sum array intrinsic .............................................. 83
7.4. Performance of Matrix Multiply: Un-optimized vs. Optimized ................................................................. 85
7.5. Performance of Forward Elimination: Un-optimized vs. Optimized ................................................................. 85
7.6. Performance of Rank: Un-optimized vs. Optimized ................................................................. 85
7.7. Performance of Matrix Multiply: Reduction using forall vs. Intrinsic ................................................................. 86
7.8. Performance of Rank: Reduction using forall vs. Intrinsic ................................................................. 86
7.9. Effect of Optimized Load Balancing for Length-Preserving Insert ................................................................. 87
7.10. Run-times for Compiled program and CM Fortran primitive, respectively, for Matrix Multiply ................................................. 87
7.11. Performance of Maxloc: Compiled vs. CM Fortran Primitive ................................................................. 88
List of Figures

2.1. Higher-Order Functions in FP* ........................................... 12
3.1. Structure Inference Axioms ........................................... 29
3.2. Structure Inference Rules for Functionals .......................... 30
3.3. Normal form Structure Axioms ....................................... 34
3.4. Explicit Inference Rules for Insert .................................. 35
4.1. Routing Function Definition Schema ................................ 42
4.2. Examples of Routing Function Definitions ........................... 50
5.1. Allocation Function Definition Schema ............................... 53
Chapter 1
Introduction

1.1 Motivation

A driving force for parallel processing is the desire for greater performance arising from the need to solve large problems. There is currently a multitude of proposed or actual parallel computers that are organized in a variety of architectures[62], including shared-memory architectures, distributed-memory MIMD and SIMD architectures. Different classes of parallel architectures require radically different paradigms for describing and executing computations. This has led to proliferation of parallel programming models and languages that are architecture-specific. There has been considerable research on the suitability of the data-parallel model for programming different parallel architectures. Data parallelism is a style of programming where parallelism comes from applying a single operation over a potentially large set of data. In contrast, control parallelism executes distinct instructions in parallel. Proponents of data parallelism argue that the model provides application programmers with more high-level and expressive abstractions than control parallelism. The data-parallel model promises massive scalable parallelism, inherent load balancing, and low overhead. Data-parallelism has also been shown to be useful for describing algorithms and implementing large-scale applications. Sabot[49] has observed that most of the parallel algorithms found in the literature for network and P-RAM models are either data-parallel or are easily converted to this form. In addition applications involving parsing[55], DNA sequence comparison[36], object recognition[63] and VLSI design[35] have been implemented in the data-parallel programming model.

The promise of data parallelism has led researchers to investigate the design and implementation of high-level programming languages for this model. High-level languages require that the programmer describe only the algorithm, not every detail of the algorithm to hardware mapping. Low-level languages suffer from the problem of architecture-dependence that makes it impossible to port applications across distinct parallel architectures. Collection-oriented languages, defined below, and functional languages are classes of high-level languages that adequately address the issues of expressiveness and architecture-independence. Collection-oriented[10, 54] languages have monolithic primitive functions that operate on data aggregates as a whole. Examples of collection-oriented languages include APL[22], FP[5], NESL[11], SETL[52], and array extensions to Fortran 90[41]. These languages differ from conventional von-Neumann languages, which also support collections, by providing operators that
manipulate collections as a whole. Proponents of collection-oriented languages argue that high-level constructs lead to code that is clear, concise, and easy to write. Also, it is argued that high-level description allows code to be mapped into a broader set of architectures, since fewer details about the implementation are included in the code, giving more flexibility to the compiler. Functional languages are another important class of high-level languages where computation is carried out entirely through the evaluation of expressions. The property of referential transparency, defined as the lack of an updatable global store, provides functional languages with simple and clean semantics, a high-level abstraction from machine details, and implicit parallelism. Functional languages have been used in developing a variety of real-world applications such as large-scale scientific computation, expert systems, CAD packages, compilers and VLSI circuit design[27].

The use of high-level languages for programming parallel computers has been hampered by the lack of efficient implementations. Since programs in these languages are far removed from machine details, there is a large space of implementation possibilities for a compiler. The development of efficient implementations on parallel computers is complicated by additional concerns raised by parallelism: partitioning the computation among multiple processors, and mapping data structures and parallel activities onto processor nodes to make good use of machine resources. Implementations are complicated further by the greater variation of architectural features and system balance observed in parallel computers as compared with sequential computers. A formal framework for describing the implementation techniques provides a clear idea of the capabilities and limitations of the compiler, and it provides formal specifications for implementors.

1.2 Overview

The goal of this dissertation is to develop a set of formal techniques for compiling high-level languages based on data parallelism. These techniques have been developed in the context of the programming language FP* where we have incorporated a small set of extensions and modifications to FP [5]. FP* combines the advantage of functional languages and collection-oriented languages for high-level programming on the data-parallel model.

High-level features in FP* include higher-order functions, polymorphism and extensive support for describing communication. Programs in FP* are constructed using a small set of higher-order functions and a fixed set of primitive functions as building blocks. Some of these higher-order functions have data-parallel operational semantics. Polymorphism is a useful programming language feature where the same function can be evaluated on inputs of different forms, allowing reuse of code and enhancing software productivity.
Rearrangement of values within aggregates data objects is an important component of data-parallel programming. FP* provides a wide variety of monolithic primitive functions, called routing functions, that perform communication among data objects. In addition, we have extended the language by allowing the programmer to describe new routing functions.

Backus pointed out that conventional imperative languages suffer from the von-Neumann bottleneck where scalar assignment statements force the programmer to think “word-at-a-time”. In FP, all primitive and higher-order functions operate on aggregates as a whole. This not only leads to programs that are more high-level and abstract, but it also fits well with the data-parallel programming model. The basic data object in FP* is the nested aggregate. All primitive operations and higher-order functions in FP* act on aggregates as a whole. This feature is inherited from the programming language FP.

This dissertation presents techniques for efficiently implementing FP* on parallel architectures. These techniques have been developed in a formal framework allowing for a precise and clear description of the compiler. An inference system is presented that determines types and sizes of data objects operated upon by FP* programs. High-level languages organized around aggregates tend to suffer due to the creation and copying of large intermediate data structures. On parallel architectures, copying of large data structures may require inter-processor communication which can be extremely expensive compared to local computation. The FP* compiler significantly minimizes the copying of large data structures, thereby reducing inter-processor communication on parallel computers. The compiler also optimizes data layout to improve load balance of compiled programs. The FP* compiler emits an abstract intermediate language where parallelism is expressed using forall loops. All iterations in forall loops can be potentially evaluated in parallel. Code generation in the FP* compiler is formally described using a syntactic function which is defined over all well-formed FP* functions. The effects of the compiler optimizations are demonstrated through an implementation on the Connection Machine CM-2. We have devised a simple scheme for translating the abstract intermediate language to CM Fortran which is the Fortran dialect on the Connection Machine. Running times are presented for a set of benchmark programs written in FP* and compiled into CM Fortran using the compiler described here. These timings demonstrate the speed-ups obtained through the optimizations described in this thesis.

1.3 Contributions

This dissertation has focused on demonstrating that high-level languages can be used for efficient data-parallel programming. The contributions of this thesis in the areas of language design and formal parallel implementations are as follows.
• **Languages:** A small set of extensions and modifications to the programming language FP are presented to make it more suitable for data-parallel programming. These modifications integrate arrays into the language and introduce a static polymorphic *structure* system where structures denote type and sizes of data objects. A notation for describing communication functions and precise criteria for determining correctness of communication function application are presented.

• **Implementations:** A set of techniques are developed in a formal framework for analyzing high-level data-parallel programs and translating them into low-level programs where parallelism is expressed using `forall` loops. The compiler is characterized by the following novel features:

  - *Structure inference* derives the types of data objects and determines how array sizes are related. A set of equations are generated by the inference system that describe the constraints to be satisfied by all array sizes. In addition, the structure inference system performs automatic coercion between distinct isomorphic representations of nested aggregates.

  - *Allocation functions* are used by the compiler to describe allocation of program objects onto physical memory. A small set of operations and transformations on allocation functions are used to improve performance of compiled programs by reducing inter-processor communication, minimizing storage overhead, and improving load balance.

  - The *code generation function* is a formal syntactic function that completely describes the code generation process. It consists of a small set of definitions that determine the optimized code generated by the compiler for any well-formed FP* function. The code generation function precisely describes the capabilities of the compiler and constitutes a concise specification to an implementor.

### 1.4 Related Work

The programming style in FP*, where operations act upon aggregate data structures as a whole, is similar to APL[30]. APL includes a fixed set of primitive functions and permits overloading of operators. Techniques have been developed for compiling APL efficiently onto sequential computers and vector computers. Guibas and Wyatt[23] have identified routing functions (called *grid selectors*) in APL and presented techniques to minimize copying of data structures in evaluating them. However, their techniques handle only a subset of the routing functions that can be optimized by the FP* compiler. The subset consists of those routing functions in APL which can be cast into the form of universal selectors where successive elements of the output array (along any dimension) lie at equal size distances in the input array. The FP*
compiler uses a more powerful scheme to keep track of how elements in the output are allocated in elements of the input while evaluating routing functions. The compilation of APL to vector machines has been examined by Budd[16]. Budd's compiler performs an inferencing step to determine type, rank and size of expression results using dataflow algorithms[1]. Fortran 90, which is the new proposed standard for Fortran, includes a set of array extensions and a set of array intrinsics. The basic aggregates are affine sections of multi-dimensional arrays, and Fortran arithmetic and logical instructions are extended to work on those array sections. More complex operations are performed by array intrinsics, which include reductions and various forms of data rearrangements. The relationship of FP* to Fortran 90 is discussed in Section 2.2.

NESL[11] is a strongly-typed, applicative data-parallel language where parallelism is supplied through a fixed set of vector operations and an apply-to-all construct. NESL supports nested vectors in which components may have different sizes. Nested vectors are important for describing divide-and-conquer algorithms using recursion. In FP*, higher-order functions are used for describing divide-and-conquer programs. VCDE[12] has been developed as an intermediate data-parallel language to which a compiler for NESL is being targeted. The compilation of VCDE for shared memory multiprocessors has been reported by Chatterjee in [19]. A technique for determining sizes of arrays has been presented that is similar to the FP* compiler. This relationship is further explored in Chapter 3.

A variety of Lisp dialects, including CM-Lisp[56], and *Lisp[59], have been designed for data-parallel programming. CM-Lisp is an architecture-independent extension to Lisp where data parallelism is described using the operators $\alpha$ and $\beta$ on parallel data called xappings. An experimental implementation of CM-Lisp on the Connection Machine is described in [68]. In this implementation, only simple Lisp objects are stored in the back-end, and only the innermost $\alpha$ in a nested sequence of $\alpha$'s is evaluated in parallel. Some of these implementation problems arise due to the extremely expressive nature of xappings, and because CM-Lisp inherits features such as dynamic typing and first class functions from the base language. *Lisp is a low-level architecture-specific dialect of Lisp for the Connection Machine. The programmer has to explicitly declare types of data objects and whether they are stored in the back-end. Only those functions which are prefixed by !! are evaluated in parallel on the back-end.

Enhancements to the FP language have been embodied in the FL[4] and FFP[5] languages. The FL language introduces lazy evaluation and hence the ability to create "infinite" sequences. Lazy evaluation requires heterogeneous processes to evaluate suspensions when they are needed, and therefore reduces available parallelism[28]. The FFP language permits definition of new combining forms. A fixed set of combining forms is important in formulating structure inference and efficient code generation. Type inference systems have been presented for FP and FL in [32, 2, 38]. The relationship of structure inference in the FP* compiler to those inference systems is
described in Section 3.10. Mago [39, 40] describes the architecture of a cellular processor capable of directly executing FP in the framework of reduction languages. The architecture is based on a tree-structured network where each processor directly interprets the high-level language. The optimizations described in this dissertation are not performed in their parallel implementation of FP.

1.5 Organization of the Dissertation

The remainder of this dissertation is organized as follows.

- Chapter 2: The programming language FP* is introduced and a set of examples are presented that demonstrate data-parallel FP* programming. The relationship of FP* to FP is explored and possible enhancements to FP* are analyzed.

- Chapter 3: This chapter introduces the notion of structure that describes types and sizes of data objects. FP* is a declaration-free language. We introduce a static polymorphic structure system for FP* in this chapter and present an inference system that derives the most general structures of inputs and outputs of functions. The inference system consists of axioms for primitive functions and inference rules for higher-order functions. We show how the inference system performs automatic coercion between distinct isomorphic representations of aggregate data objects.

- Chapter 4: We introduce a programming language feature for defining new primitive routing functions. The denotation of such functions is precisely described. We derive run-time tests for determining the correctness of the application of programmer-defined routing functions on data objects. These tests involve run-time testing for satisfiability of linear equations in array sizes.

- Chapter 5: We introduce the concept of allocation functions that are used by the FP* compiler to track allocation of FP* objects in memory. Correctness criteria for allocation are developed in a fashion similar to that for routing functions. Two operations on allocation functions are presented that are critical to performing the optimizations described in Chapter 6.

- Chapter 6: The code generation function is precisely described using a syntactic function. The target language of the compiler, IL, is described. The code generation function emits low-level loop-parallel IL code and performs a variety of optimizations including routing function optimizations, array constant propagation, and load balancing.

- Chapter 7: We present a simple scheme for translating IL into CM Fortran, a Fortran dialect for the Connection Machine CM-2. The effect of optimizations described in Chapter 6 are evaluated on a variety of programs through timings obtained on the CM-2.
• Chapter 8: Conclusion and future work.
Chapter 2
Parallel Programming in FP*

2.1 Introduction

FP* is a high-level language that allows programmers to describe data-parallel algorithms. The functional language FP* is a dialect of FP, which was introduced by Backus [5]. FP* incorporates a small set of modifications and extensions to FP to enhance its suitability for data-parallel programming. Data-parallel computation consists of applying an operation to a number of data elements concurrently. Data-parallel programs are characterized by single thread semantics and implicit lockstep synchronization. By contrast control parallelism has multiple independent control threads engaged in concurrent computation. When data structures are shared between threads (processes), communication and explicit synchronization and exclusion mechanisms are necessary to ensure determinacy. Proponents of data-parallelism [24, 49, 25] have argued that this model is suitable for describing parallel algorithms and writing portable parallel programs. In this chapter we outline essential features of FP* and present examples demonstrating its suitability for high-level data-parallel programming.

2.2 Languages for Data-Parallel Programming

In this section we review essential features of data-parallel languages. A detailed discussion of these issues is presented in [54] and [31]. Data-parallel languages rely on the following: (1) the availability of aggregate data types in the language and (2) constructs in the language for performing computation and communication have data-parallel operational semantics. We now discuss each of these issues in detail.

2.2.1 Aggregate Data Types

Arrays, sequences, sets and lists are all examples of commonly used aggregates, also called collections in [54]. Elements in nested collections can themselves be collections. Fortran 90 and APL support only simple collections where elements can only be scalars, whereas FP, APL2, Paralation Lisp, NESL, and SETL support nested collections.
2.2.2 Aggregate Operations

Data-parallel languages support two methods of programming with aggregates. The element-wise method, relies on using selection and update operations nested within looping constructs to retrieve an element from or to place an element into an aggregate. When sequential do loops are used, the compiler must unravel dependencies among iterations to discover opportunities for parallelism. This static analysis, performed in vectorizing compilers [45, 17], has been partly successful, and programmer assistance is often required to achieve parallelism. In Paralation Lisp [49], an elementwise form is used as the looping construct ensuring that all iterations are completely independent.

The alternative to the element-wise form of programming relies on providing monolithic operations that are defined on entire aggregates. The advantage of the monolithic approach is that the semantics of monolithic operations are well understood and the access patterns of elements are known a priori. Hence much more parallelism can be extracted than in the element-wise methods. A strong argument in favor of the monolithic style of programming is made in [5]. A disadvantage of the monolithic approach is that it requires that all monolithic operations be pre-defined, as part of the built-in set of operations provided by the language. For example, the monolithic operations in Fortran 90 include communication operations such as spread, transpose, and array reduction intrinsics such as sum, any, all and maxval. The introduction of higher-order monolithic operations, which are parameterized by user-defined functions, extends the scope of the monolithic approach and limits the proliferation of operations. An important advantage of monolithic operations is that the complexity and run-time performance of programs using these primitive operations can be easily determined. Monolithic operations possess predictable performance in the same way that primitive operations in sequential languages have known and predictable run-time behavior. Performance predictability ensures that inefficient aspects of programs can be identified syntactically and allows the programmer to develop intuition for optimizing the performance of programs.

2.3 FP*: An Overview

FP* has all the advantages of functional languages [29] including referential transparency, simple semantics and useful algebraic properties. FP* is suited mostly for describing data-parallel computations that are evaluated on parallel architectures. A traditional functional language based on linked list representation of data types and recursion can be used for describing sequential computation that will be evaluated on a single processor. The sequential language manipulates atoms, lists, and data-parallel objects created by sequential functions. The programmer identifies sequential and data-parallel functions using annotations. Data-parallel functions are called from a sequential function and are evaluated in parallel. The formal implementation described in this thesis focuses entirely on implementing data-parallel languages such
as FP*.

All FP* functions are monadic and operate on data objects as a whole. FP* modifies and extends FP[5] in various ways to enhance its suitability for data-parallel programming. The relation of FP* to FP is discussed in Section 2.4. In this section the main features of FP* are described. The language consists of a set of primitive functions for performing computation and communication. Realizing the importance of communication, we have introduced syntax into the language that allows programmers to define new primitive communication functions (see Chapter 4). FP* also has a fixed set of higher-order functions that support creation of new monolithic operations. Higher-order functions are the primary control constructs in FP*, as recursion is not allowed. FP* supports polymorphism in various ways: (1) FP* communication functions can take inputs of various types and sizes, and (2) functions constructed using higher-order functions operate on inputs of various lengths.

2.3.1 Data Objects

There are three basic kinds of data objects in FP*.

1. **Scalars** are numbers and boolean constants, e.g., "1" and "true"

2. **Arrays** are sequences of objects of the same type and size, e.g., [1, 11, 15]. Higher-dimensional arrays are denoted by lists of arrays of rank one lower, e.g., [[11, 13, 15], [1, 3, 5]] denotes a $2 \times 3$ matrix.

3. **Tuples** are sequences of objects of different types (and sizes), e.g., (1, (3, 5), true).

Further, we use ⊥ to describe the error object. ⊥ may not appear with arrays and tuples, i.e.,

$$[\ldots, \bot, \ldots] = (\ldots, \bot, \ldots) = \bot$$

Empty arrays are denoted by [] and empty tuples by {}. Arrays and tuples are the aggregates of FP*. Nested arrays in FP* require each array element be of the same size, in contrast to nested arrays in collection-oriented languages such as APL2[22] and NESL [11]. This allows the development of an inference system to derive types and sizes of objects, and allows the compiler to perform other optimizations described in this dissertation.

FP* is a declaration-free language – the programmer does not declare types and sizes of program objects. This information is obtained by the compiler through an inference system that is described in Chapter 3.

2.3.2 Primitive Functions

Primitive FP* functions are partitioned into computation functions and communication functions. Computation functions correspond to the standard arithmetic, relational and logical instructions in other languages. These functions take a single input,
which usually consists of a tuple of scalars, and returns a single scalar as output.

$+, \leq,$ and are some of the computational functions in $\text{FP}^*$. Examples demonstrating these functions are presented below. The infix operation $\cdot$ denotes application of a function to the input argument.

$+ : \langle 1, 4 \rangle = 5$

$\text{and} : \langle \text{true}, \text{false} \rangle = \text{false}$

Communication functions, also called routing functions, copy elements of inputs to outputs in a manner that is completely independent of scalar values within the inputs. Examples of some of these functions are given below.

$\text{trans} : [[x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}]]$

$\quad = [[x_{11}, \ldots, x_{m1}], \ldots, [x_{1n}, \ldots, x_{mn}]]$

$\text{reverse} : [x_1, \ldots, x_n] = [x_n, \ldots, x_1]$

$\text{apndl} : \langle x, [x_1, \ldots, x_n] \rangle = [x, x_1, \ldots, x_n]$

Each of the $x_i$'s could also be aggregates.

We have found routing functions to be particularly useful in developing data-parallel programs in $\text{FP}^*$. We have therefore extended the language to allow the programmer to define new primitive routing functions. The syntax and semantics of user-defined routing functions are described in Chapter 4, and criteria are developed to determine when these definitions are correct. Some of the user-defined routing functions that do not exist in $\text{FP}^*$ are the following:

1. $\text{shuffle} : \langle [x_1, \ldots, x_n], [y_1, \ldots, y_n] \rangle = [\langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle]$

2. $\text{2d.to.1d} : [[x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}]]$

$\quad = [x_{11}, \ldots, x_{m1}, \ldots, x_{1n}, \ldots, x_{mn}]$

3. $\text{pairUp}(c) : \langle [x_{11}, \ldots, x_{1n}], \ldots, [x_{c1}, \ldots, x_{cn}] \rangle$

$\quad = \langle [x_{11}, \ldots, x_{c1}], \ldots, [x_{1n}, \ldots, x_{cn}] \rangle$

### 2.3.3 Higher-Order Functions

Programs are constructed by combining primitive functions with higher-order functions. $\text{FP}^*$'s higher-order functions are similar to FP's. Functions constructed from higher-order functions are monolithic, i.e., they can be applied to all elements of an aggregate. The set of higher-order functions in $\text{FP}^*$ along with their use is given in Figure 2.1.

From a parallel programming standpoint, the Apply-All and Insert higher-order functions are the most useful. Apply-All provides a high-level and abstract method of describing loop-level parallelism. Function $f$ in $\alpha f$ is applied to each element of an input array in parallel. Similarly, Insert provides a high-level means of describing
<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>$(o)$</td>
<td>$f \circ g : x = f : (g : x)$</td>
</tr>
<tr>
<td>Apply-All</td>
<td>$(\alpha)$</td>
<td>$\alpha f : [x_1, \ldots, x_n] = [f : x_1, \ldots, f : x_n]$</td>
</tr>
<tr>
<td>Construction</td>
<td>$\langle \ldots \rangle$</td>
<td>$(f_1, \ldots, f_n) : x = \langle f_1 : x, \ldots, f_n : x \rangle$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$(\rightarrow ;)$</td>
<td>$(f \rightarrow g; h) : x = \begin{cases} g : x &amp; f : x = \text{true} \ h : x &amp; f : x = \text{false} \ \bot &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>Insert</td>
<td>$(/)$</td>
<td>$f / [x_1, \ldots, x_n] = \begin{cases} f / [f / [x_1, \ldots, x_{2^k}], x_{2^k+1}, \ldots, x_n] &amp; 1 \leq 2^k &lt; n \leq 2^{k+1} \ x &amp; n = 1 \ \bot &amp; \text{Otherwise} \end{cases}$</td>
</tr>
<tr>
<td>While</td>
<td>$(\text{While } e \text{ do } f) : x = \begin{cases} (\text{While } e \text{ do } f) : (f : x) &amp; e : x = \text{true} \ x &amp; e : x = \text{false} \ \bot &amp; \text{Otherwise} \end{cases}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: Higher-Order Functions in FP*

reduction operations on arrays. $/f$ is evaluated through a binary tree of depth $\lceil \lg n \rceil$, where $n$ is the length of an input array.

We have identified two forms for the Insert functional. For an expression $/f$, $f$ is a binary function that takes a tuple of two objects and combines them into a single object. The first form of Insert, Insert(1), occurs when the two components of the input tuple and the output object have the same type and size. For example, $/+\,$ is an Insert(1) form because both inputs and the output to $+\,$ are of numeric type. The second form of Insert, Insert(2), occurs when the output has the same type but the size is the sum of the sizes of the input objects. Insert(2) is used by $\text{merge}$ where $\text{merge}$ merges two sorted arrays into a single sorted array. The output array has the same type as each of the two input arrays, but its size is the sum of the input sizes. The compiler automatically distinguishes the two forms of Insert through structure inference as described in Chapter 3.

The While higher-order function is a useful construct for describing sequential iterations. This is important, since recursion is not allowed in FP* programs. The Composition and Conditional higher-order functions are self-explanatory. The Construction higher-order function provides opportunities for control parallelism since each of the $f_i$'s in $\langle f_1, \ldots, f_n \rangle$ can be evaluated in parallel.
2.3.4 Automatic Coercion Between Aggregates

It is possible to represent certain aggregates both as tuples and as arrays. For example, a collection of three integers can be represented as \( (1, 2, 3) \) and \([1, 2, 3]\). In FP*, conversions between tuple and array representations are automatically performed by the compiler. A function that expects an array input can also take as input a tuple where all elements have the same type and size. Similarly, a function that expects a tuple input can also take as input an array of the same size as the tuple. The techniques used by the compiler to automatically perform transformations between tuple and array representations are described in the Chapter 3.

The following examples demonstrate the usefulness of automatic coercion between arrays and tuples.

\[
+ : (x_1, x_2) = + : [x_1, x_2] = x_1 + x_2
\]

\[
\text{trans} : ([x_{11}, \ldots, x_{1n}], [x_{21}, \ldots, x_{2n}]) =
\]

\[
\text{trans} : ([x_{11}, \ldots, x_{1n}], [x_{21}, \ldots, x_{2n}]) =
\]

\[
[[x_{11}, x_{21}], \ldots, [x_{1n}, x_{2n}]]
\]

2.4 Comparison with FP

FP* modifies and extends FP in various ways to make it more suitable for data-parallel programming and to make an efficient implementation possible. FP has a single aggregate data type – the sequence. In FP* sequences are replaced by tuples and arrays.

Functions in FP* are associated with polymorphic type and size descriptions of their inputs and outputs. These descriptions impose semantics that are different from FP's semantics. For example, \(\text{trans} \) can take as input a 2-dimensional array of objects of the same type and size; whereas \(\text{trans} \) in FP can take a nested sequence of objects of different types. Introduction of a type system to FP enables a compiler to detect type errors in programs, and also determine the types and sizes of data objects in well-formed FP* programs. This information is crucial for generating efficient code, and performing compiler optimizations.

In addition, we have introduced a small set of primitive functions into FP* that modify and enhance those already present in FP. FP has a sequence selection function \(c\) which selects the \(c^\text{th}\) element of the input sequence. FP* replaces \(c\) with \(\text{sel}(c, n)\) and \(\text{sel}(c)\), both defined as follows:

\[
\text{sel}(c, n) : (x_1, \ldots, x_n) = \begin{cases} 
x_c, & 1 \leq c \leq n; 
\perp, & \text{Otherwise.}
\end{cases}
\]

\[
\text{sel}(c) : [x_1, \ldots, x_n] = \begin{cases} 
x_c, & 1 \leq c \leq n; 
\perp, & \text{Otherwise.}
\end{cases}
\]

These selection functions permit the compiler to make certain optimizations since the index of the selected element is known. FP* also has a dynamic selection function \(\text{sel}\), not present in FP, which is defined as follows:
\[ sel : \langle i, [x_1, \ldots, x_n] \rangle = \begin{cases} x_i, & 1 \leq i \leq n; \\ \bot, & \text{Otherwise}. \end{cases} \]

We have introduced the following array operations as primitive functions in FP*.

\[ index : [x_1, \ldots, x_n] = [1, \ldots, n] \]

\[ ins : \langle x, i, [x_1, \ldots, x_n] \rangle = \begin{cases} [x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n], & 1 \leq i \leq n; \\ \bot, & \text{Otherwise}. \end{cases} \]

We have also introduced routing function declarations into FP which allow the programmer to define new routing functions. The syntax and semantics of routing function declarations are discussed in Chapter 4.

### 2.5 Programming Examples

FP* function definitions have the following syntax.

\[
\text{def } f : v = \text{expr}
\]

where \( f \) is the name of the function being defined, \( v \) is used to name objects in the tuple input to \( f \). Only components of a tuple object can be named in \( v \). It is entirely syntactic sugar, since these names cannot be referred to in \( \text{expr} \), the body of the function. \( v \) is optional, and can be omitted from function definitions. It exists only to improve readability of FP* programs. \( \text{expr} \) is any FP* function constructed using higher-order functions and primitive functions of FP*.

We next present several example programs that demonstrate data-parallel programming in FP*. The processor and parallel time complexities of these algorithms are presented for the P-RAM model where inter-processor communication costs are ignored. In addition, we present the parallel work complexity, which is the product of processor complexity and (parallel) time complexity. Finally, we present the serial work complexity for the same algorithm which is its time complexity on a serial RAM.

**Program 2.1 (Simple Array Operations)**

\[
\text{def } vec\_add : \langle v_1, v_2 \rangle = (\alpha +) \circ \text{trans}
\]

\[
\text{def } vec\_mult : \langle v_1, v_2 \rangle = (\alpha \ast) \circ \text{trans}
\]

\[
\text{def } 2d\_mat\_add : \langle m_1, m_2 \rangle = (\alpha vec\_add) \circ \text{trans}
\]

\[
\text{def } 3d\_mat\_add : \langle m_1, m_2 \rangle = (\alpha 2d\_mat\_add) \circ \text{trans}
\]

In \( vec\_add \), the effect of \( \text{trans} \) is to pair up each element of \( v_1 \) with the corresponding element of \( v_2 \). \( \alpha + \) adds all such pairs. \( vec\_mult \) is similar to \( vec\_add \); vectors \( v_1, v_2 \) are element-wise multiplied. \( vec\_add \) and \( vec\_mult \), respectively, have
$O(n)$ processor complexity and $O(1)$ time complexity where input vectors $v_1$ and $v_2$ have length $n$. These complexity measures ignore the cost of evaluating the routing function $trans$. The compiler optimizations presented in this thesis, in fact, eliminate the evaluation of routing function $trans$. Both programs have parallel work complexity $O(n)$ which is the same as their serial work complexity.

In $2d.mat.add$, $trans$ pairs each row of $m_1$ with the corresponding row of $m_2$, and $ovec.add$ adds up the row vectors. $2d.mat.add$ has $O(nm)$ processor complexity and $O(1)$ time complexity where the input matrices $m_1$ and $m_2$, respectively, have size $n \times m$. The parallel work complexity for the program is $O(nm)$ which is the same as its serial complexity.

A 3-d matrix is a list of 2-d matrices. In $3d.mat.add$, each 2-d matrix of $m_1$ is paired up with the corresponding 2-d matrix of $m_2$ and added using $2d.mat.add$. $3d.mat.add$ has $O(nmp)$ processor complexity and $O(1)$ time complexity where the input matrices $m_1$ and $m_2$, respectively, have size $n \times m \times p$. The parallel work complexity of the algorithm is $O(nmp)$ which is the same as its serial work complexity.

The above programs demonstrate the usefulness of routing functions and higher-order functions in FP*. Routing functions are used to rearrange data so that computation can be performed using the higher-order functions, such as $\alpha$, which have efficient parallel implementations. FP* supports program design by composition where useful programs are obtained by combining simpler ones. Further, these programs can operate on vectors and matrices of various sizes, thus supporting polymorphism.

**Program 2.2 (Outer Product, Inner Product)**

```python
def op : (x, y) = (αα*) o (adistl) o distr

def ip : (v1, v2) = (+) o vvec mult
```

$op$ creates a 2-d matrix from the vectors $x, y$ where element $(i, j)$ is $x_i * y_j$. This is identical to the APL operation $x \circ . * y$. $\alpha \alpha*$ denotes the application of $*$ on a 2-dimensional array of pairs of numbers. $distl$ and $distr$ are FP* primitive functions which are evaluated as shown below:

- $distr : ([x_1, \ldots, x_n], y) = [(x_1, y), \ldots, (x_n, y)]$
- $distl : (x, [y_1, \ldots, y_n]) = [(x, y_1), \ldots, (x, y_n)]$

$op$ can be evaluated using $O(nm)$ processors in $O(1)$ time where $n$ and $m$ are lengths of $x$ and $y$, respectively. The parallel work complexity of this algorithm is $O(nm)$ which is identical to its serial work complexity.

In $ip$, $vvec mult$ performs element-wise multiplication of vectors $v_1$ and $v_2$. $+/\!$ then adds up all the products returning the inner product. $ip$ has $O(n)$ processor complexity and $O(\log n)$ time complexity where input vectors $v_1$ and $v_2$ have length $n$, respectively. The parallel work complexity is $O(n \log n)$ whereas the best serial algorithm has work complexity $O(n)$. The parallel work complexity of this algorithm
can be improved to $O(n)$ by using a scheduling principle due to Brent[15]. Using this scheduling technique the reduction operation can be evaluated using $O(n/\log n)$ processors in $O(\log n)$ parallel time.

**Program 2.3 (Matrix-Vector Product, Matrix-Matrix Product)**

\[
\text{def} \; \text{mv} : (m, v) = (\alpha \alpha \; ip) \circ \text{distr}
\]

\[
\text{def} \; \text{mm} : (m_1, m_2) = (\alpha \alpha \; ip) \circ (\text{adisil}) \circ \text{distr} \circ \{\text{sel}(1, 2) , trans \circ \text{sel}(2, 2)\}
\]

$mv$ computes the product of matrix $m$ with vector $v$ by first pairing up each row of $m$ with the vector $v$. This is accomplished by the $\text{distr}$ primitive. Next inner products are taken of all row vector pairs using $\alpha$. $mv$ can be evaluated using $O(nm)$ processors in $O(\log n)$ time where $m$ has size $m \times n$ and $v$ has length $n$. The parallel work complexity of $mv$ is $O(nm \log n)$ whereas the serial work complexity is $O(nm)$.

$mm$ computes the product of matrices $m_1$ and $m_2$ by pairing up each row of $m_1$ with all columns of matrix $m_2$. This creates a 2-dimensional array of row-column pairs where element $(i, j)$ is the tuple $\langle \text{row} i \; \text{of} \; m_1 , \; \text{column} j \; \text{of} \; m_2 \rangle$. The element at index $(i, j)$ of the output matrix is obtained by taking the inner product of the vectors at element $(i, j)$ of the intermediate structure. $mm$ can be evaluated using $O(nmp)$ processors in $O(\log m)$ time where $m_1$ has size $n \times m$ and $m_2$ has size $m \times p$. $mm$ has parallel work complexity which is $O(nmp \log m)$ whereas its serial work complexity is $O(nmp)$. We would like to point out that this is not an optimal serial matrix multiplication algorithm; Strassen’s matrix multiplication algorithm [57] has improved serial work complexity of $O(n^{2.81})$.

**Program 2.4 (Vector Factorial)**

\[
\text{def} \; \text{fact} : n = \\
\quad \text{sel}(2, 2) \circ (\text{While} \; > \; o(\text{sel}(1, 2), 1) \; \text{Do} \; (\text{dec} \circ \text{sel}(1, 2), \ast)) \circ \{\text{id}, 1\}
\]

\[
\text{def} \; \text{vect.fact} : v = \alpha \; \text{fact}
\]

The $\text{fact}$ program illustrates the use of the $\text{While}$ higher-order function. It iterates accumulating products while decrementing a counter. The $\text{FP}^*$ primitive function $\text{id}$ used in the definition is evaluated as follows:

\[
\text{id} : x = x
\]

A vector version of the factorial program that computes factorials of elements in an array is $\alpha \text{fact}$. $\text{vect.fact}$ can be evaluated using $O(|v|)$ processors in $O(max(v))$ time where $max(v)$ is the value of the largest element in $v$. $\text{vect.fact}$ has parallel work complexity of $O(|v|max(v))$ whereas the serial algorithm has work complexity.
$O(\sum v_i)$ where $v_i$ is the value of the $i^{th}$ element in $v$. It is straightforward to show that $\lvert v \rvert \text{max}(v) \geq \sum v_i$.

Program 2.5 (Permute, Ranksort)

```python
def permute : \langle d, v \rangle =
\alpha(+/+) \circ \alpha(= \circ \langle \text{sel}(1, 2), \text{sel}(1, 2) \circ \text{sel}(2, 2) \rangle \to \text{sel}(2, 2) \circ \text{sel}(2, 2); 0)
\circ \alpha(\text{distl} \circ \text{distr} \circ \langle \text{index}, \text{id} \rangle) \circ \text{trans}

def rank : v =
\alpha(+/+) \circ \alpha(\geq \to 1; 0) \circ \alpha(\text{distl} \circ \text{distr} \circ \langle \text{id}, \text{id} \rangle)

def rs : v =
\text{permute} \circ \langle \text{rank}, \text{id} \rangle
```

*permute* takes a vector of destinations $d$ and a vector of values $v$, and returns a vector $p$ where $p(d(i)) = v(i)$. First, a matrix is created where element $i, j$ is $(i, (d(j), v(j)))$. In a new matrix, $v(j)$ is stored if $d(j) = i$, else a 0 is recorded. By summing up rows, using $\alpha(+/+)$, each element $v(i)$ is routed to location $d(i)$ of the output vector $p$. *permute* can be evaluated using $O(n^2)$ processors in $O(\log n)$ time where input vectors $d$ and $v$, respectively, have length $n$. *Permute* has parallel work complexity of $O(n^2 \log n)$ whereas the optimal serial algorithm for performing a vector permutation has work complexity $O(n)$.

*rank* determines the rank of each element in the vector that is to be sorted. Each input element is paired up with every other input element. For every pair $(x, y)$, a 1 is recorded in a matrix if $x > y$. As a result a row corresponding to a number $x$ contains a 1 each time there is a number less than itself. By summing rows using $\alpha(+/+) \circ \alpha(\text{id})$ each element $x$ receives its rank.

*rs* ranksorts the input vector $v$ by first obtaining ranks of elements and then permuting the input vector according to the ranks. *rs* can be evaluated using $O(n^2)$ processors in $O(\log n)$ time where input vector $v$ has length $n$. The parallel work complexity of *rs* is $O(n^2 \log n)$ compared to its serial work complexity of $O(n^2)$.

Program 2.6 (Bitonic Merge Sort)

```python
def comp_xchg : \langle x, y \rangle =
\leq \to \text{id}; \langle \text{sel}(2, 2), \text{sel}(1, 2) \rangle

def btnr_merge : \langle v_1, v_2 \rangle =
\text{merge_aux} \circ \text{apnd} \circ \langle \text{sel}(1, 2), \text{reverse} \circ \text{sel}(2, 2) \rangle

def merge_aux : v =
```

17
\[ sel(2, 2) \circ (\textbf{While }\neq a(sel(1, 2), 0) \textbf{ do} \]
\[ (\text{dec} \circ sel(1, 2), \]
\[ \text{apndl} \circ a(\text{comp}_x\text{chg}) \circ \text{shuffle} \circ sel(2, 2)) \]
\[ \circ (\text{log} \circ \text{length}, \text{id}) \]

**def** btnc_sort : \( v = \)
\[ /btnc\_merge \circ a(\text{id}) \]

Sorting is performed in Merge-Sort by pairwise merging an array of sorted sequences. In function *btnc_sort*, \( a(\text{id}) \) converts an array of keys to an array of 1-element (trivially sorted) sequences. Pairwise merging of sorted sequences is performed by function *btnc_merge*.

The Merge algorithm in this version of Merge-Sort is based on Batcher’s Bitonic-Merge [9] algorithm. In function *btnc_merge*, a bitonic sequence of length \( n \) is constructed by concatenating the first sorted sequence with the reverse of the second sorted sequence. Next, the following step is repeated \( \log n \) times. The bitonic sequence is shuffled into an array of pairs and a compare-and-exchange operation is performed in each pair. The routing function *shuffle* described in Section 2.3.2 is used for this purpose. The resulting array of pairs is appended together, to produce the input for the next iteration. In function *merge_aux*, the While functional decrements an iteration counter, initially set to \( \log n \), down to 0, and performs the shuffle and compare-and-exchange steps. The merged array is returned by selecting the second element (via *sel(2, 2)*) from the output tuple object of the While functional.

The main sources of parallelism in the program are due to the Insert functional in *btnc_sort* and the the Apply-all functional in *merge_aux*. The insert functional implies that we can perform the merge in a balanced binary tree of height \( \log n \), where \( n \) is the input array length. The leaves contain the keys and the merge operation is performed at each node. All the merge operations at the same level of the tree can be performed in parallel. Further, the Apply-all functional in *merge_aux* implies that each of the compare-and-exchange operations in each merge operation can be performed in parallel.

*btnc_sort* can be evaluated using \( O(n) \) processors in \( O(\log^2 n) \) time where input vector \( v \) has length \( n \). The compiler optimizations described in Section 6.8 generate target code that has this complexity, ignoring communication costs. The parallel work complexity of *btnc_sort* is \( O(n \log^2 n) \) and its serial work complexity is also \( O(n \log^2 n) \).

**Program 2.7 (Maxloc)**

**def** MaskValue : \( \{x_1, x_f, m\} = sel(3, 3) \rightarrow sel(1, 3); sel(2, 3) \)

**def** MaskMerge : \( \{v_1, v_2, v_m\} = a(MaskValue \circ \text{pairUp}(3) \)
\textbf{def} MaxIndex : \{ (x_1, i_1), (x_2, i_2) \} = \\
\geq o \circ sel(1, 2) \circ sel(1, 2), sel(1, 2) \circ sel(2, 2) \rightarrow sel(1, 2); sel(2, 2)

\textbf{def} MaxLoc : (v, v_m) = \\
\text{sel}(2, 2) \circ /MaxIndex \circ pairUp(2) \circ (\text{MaskMerge} \circ \\
\{\alpha \text{ abs} \circ \text{sel}(1, 2), \alpha 0 \circ \text{sel}(1, 2), \text{sel}(2, 2)\}, index \circ \text{sel}(1, 2))

\textit{MaxLoc} takes a vector of values and a vector of masks, and returns the index \(i\) such that \(v(i)\) has the largest absolute value among all unmasked elements in \(v\). \textit{MaxLoc} is identical to the \texttt{maxloc} array intrinsic in CM Fortran[61]. \textit{MaxLoc} zeros out values in \(v\) where \(v_m = \text{false}\). \textit{MaxIndex} is next used to determine the index and value of the largest element by absolute magnitude. \textit{MaxLoc} can be evaluated using \(O(n)\) processors in \(O(\log n)\) time where input vectors \(v\) and \(v_m\) have length \(n\).

\textbf{Program 2.8 (Forward Elimination)}

\textbf{def} ForwardElim : A' = \\
\text{ForwardElimAux} \circ (1, id, o\text{true}, o0)

\textbf{def} ForwardElimAux : \{j, A', M, RowPerm\} = \\
\text{While} \: \leq o \circ \text{sel}(1, 4), \text{length} \circ \text{sel}(2, 4) \: \text{Do} \\
\{\text{inc} \circ \text{sel}(1, 4), \\
\alpha \alpha - o\text{atrans} \circ \text{trans} \circ \{\text{sel}(2, 4), \text{ProdM}, \\
\text{MaskUpdate}, \\
\text{ins} \circ \{\text{PivotRowIndex}, \text{sel}(1, 4), \text{sel}(4, 4)\}\}

\textbf{def} PivotCol : \{j, A', M, RowPerm\} = \\
\text{sel} \circ \{\text{sel}(1, 4), \text{trans} \circ \text{sel}(2, 4)\}

\textbf{def} PivotRowIndex : \{j, A', M, RowPerm\} = \\
\text{MaxLoc} \circ \{\text{PivotCol}, \text{sel}(3, 4)\}

\textbf{def} PivotRow : \{j, A', M, RowPerm\} = \\
\text{sel} \circ \{\text{PivotRowIndex}, \text{sel}(2, 4)\}

\textbf{def} PivotElt : \{j, A', M, RowPerm\} = \\
\text{sel} \circ \{\text{PivotRowIndex}, \text{PivotCol}\}

\textbf{def} ProdM : \{j, A', M, RowPerm\} = \\
\alpha \alpha * o\text{distl} \circ \text{distr} \circ (\text{MaskMergeo} \\
\{\alpha \alpha \circ \text{distr} \circ \{\text{PivotCol, PivotElt}, o0.0 \circ \text{sel}(2, 4), \text{MaskUpdate}\},

19
def MaskUpdate : (j, A', M, RowPerm) =
ins o \{false, PivotRowIndex, sel(3, 4)\}

The forward elimination phase in Gaussian elimination transforms a linear system \(Ax = b\) into an equivalent upper diagonal system \(A'x = b'\), such that matrix \(A'\) is upper triangular. The input to the forward elimination phase is an \(n \times (n + 1)\) augmented matrix \(A' = [A|b]\). In iteration \(j, 1 \leq j \leq n:\)

1. The pivot column index is \(j\).

2. The pivot row index is the row-index \(l\) of the largest element (magnitude-wise) among elements in the pivot column, lying in unpivoted rows.

3. The pivot element is the \(l^{th}\) element in the pivot column, denoted as \(a'_{l,j}\).

All unpivoted rows (except the current pivot row) subtract from themselves a multiple of the pivot row, such that they are zeroed along column \(j\). The multiple for row \(i\) is \(a'_{i,j} / a'_{l,j}\).

The main tasks during iteration \(i\), in obtaining the new augmented matrix from the current augmented matrix, are as follows:

1. Find the pivot row index \(l\).

2. Construct the product matrix \(P\) that is to be subtracted from the current augmented matrix \(A'\).

   \[ P_{i,j} = A'_{i,j} \times a'_{l,j} / a'_{l,l} \]

   This requires determining \(a'_{i,j} / a'_{l,j}\) for all rows \(i\), broadcasting it to all elements in row \(i\), and finally taking products, for all elements, in parallel.

3. Perform Masked Matrix Subtraction. Only elements in unpivoted rows in the current augmented matrix subtract from themselves the corresponding element in the product matrix of step 2. The new augmented matrix is the input for the next iteration.

ForwardElim simply sets up the input for ForwardElimAux which performs the actual computation. The input to ForwardElim is the augmented matrix \([A|b]\) of rank \(n \times n + 1\). ForwardElimAux performs \(n\) iterations, implemented with the While functional. Iteration terminates when all \(n\) columns have been pivoted, i.e., \(j > n\). In each iteration, the input tuple \(T = \langle j, A'M, RowPerm \rangle\) is updated as follows:
1. The Pivot Column Index \( j \), initially set to 1, is incremented in every iteration until \( n \).

2. Initially, \( A' = [A|\bar{b}] \). The product matrix \( P \) is subtracted from \( M \) in each iteration.

3. \( M \) is vector of boolean values where \( M(i) = \text{true} \) if row \( i \) has not been selected as a pivot row uptill now. After row \( l \) has been selected for pivoting, \( M(l) \) is set to \( \text{false} \). When all rows have been pivoted, \( M = \text{false} \).

4. \( \text{RowPerm} \) is an integer vector whose \( j^{th} \) element is set to \( l \) if row \( l \) is selected as the pivot row in the \( j^{th} \) iteration. This is required as partial pivoting is performed, but rows are not interchanged in order to minimize inter-processor communication.

\( \text{MaskUpdate} \) updates \( M \), by inserting \( \text{false} \) at index \( \text{PivotRowInd} \) of \( M \). This masks out row \( \text{PivotRowInd} \) from all future computations. The function \( \text{ProdM} \) computes the product matrix \( P \), as defined in step 2 above. \( \text{distr} \) pairs up \( \text{PivotElt} \) with each element in \( \text{PivotCol} \), for the \( \alpha \div \beta \) operation. An outer product is taken with the resultant vector and the pivot row vector, which pairs up elements so that products can be taken, as described above in step 2. \( \text{ForwardElim} \) can be evaluated using \( O(n^2) \) processors in \( O(n \log n) \) time where \( A' \) is a system of \( n \) equations in \( n \) variables. The parallel work complexity of \( \text{ForwardElim} \) is \( O(n^3 \log n) \) and its serial work complexity is \( O(n^3) \).

### 2.6 Algebraic Program Transformations

Functionals obey a set of strong algebraic laws that can be applied to FP* programs as rewrite rules. The rewriting transforms simple but inefficient programs to complex, efficient programs[69].

Consider the following algebraic law [5], where \( f, g, h \) are any FP* expressions.

\[
(f_1 \circ g, f_2 \circ g, f_3) = (f_1 \circ \text{sel}(1,2), f_2 \circ \text{sel}(1,2), f_3 \circ \text{sel}(2,2)) \circ (g, \text{id}).
\]

In the right hand side, \( g \) is evaluated exactly once and the results are transmitted to each place of use. The identity follows from the definition of the construction functional. Selection functions introduced in the \( \text{rhs} \) are routing functions, whose effect is completely known at compile-time. This algebraic axiom can be applied on \( \text{ForwardElimAux} \), to significantly enhance its efficiency.

In \( \text{ForwardElim} \), \( \text{PivotRowIndex} \) is invoked three separate times — in the function \( \text{PivotRow} \), in \( \text{MaskUpdate} \) and once in the body of the \textit{while} functional. In each instance it is applied on the same object, the input tuple \( T \). In an efficient program, \( \text{PivotRowIndex} \) would be evaluated once and the value stored and re-used each time it is required.
Similarly, in ProdMatrix, the functions PivotElt and PivotRow both evaluate the function PivotRowIndex. Further, PivotCol is also evaluated by PivotElt. As before, these evaluations are on the same input object T. Efficiency is enhanced if these functions are evaluated once and their results transmitted to each place of use. This improvement can be obtained by systematically performing algebraic transformations [69, 5] on the original program.

2.7 Enhancements

FP* suffers from several limitations as a high-level parallel programming language. In this section we discuss some of those limitations and suggest possible enhancements which ameliorate some of the disadvantages.

A key drawback of FP* (and FP) is that the programmer is prevented from assigning names to objects and manipulating those names. Instead, the programmer has to use selection functions such as \( sel(i, c) \) to select objects from the current environment represented as a tuple. This problem is illustrated by the FP* definition of the function \( f \) whose object-level definition is:

\[
\text{def } f : \langle x, \langle y, z \rangle \rangle = \langle \langle x, y \rangle, \langle y, z \rangle \rangle
\]

The corresponding proper FP* definition is:

\[
\text{def } f : \langle x, \langle y, z \rangle \rangle = \\
\langle \langle sel(1, 2), sel(1, 2) \circ sel(2, 2) \rangle, \langle sel(1, 2), sel(2, 2) \circ sel(2, 2) \rangle \rangle
\]

Backus [6] has described techniques to transform object-level definitions into function-level definitions. These techniques require that variable names appearing in the body of the object-level function definition satisfy a small set of restrictions.

The absence of recursion in FP* is partially overcome by the Apply-all and Insert functionals, and routing functions. These elements of FP* are useful for expressing tail-recursive programs and they have efficient parallel implementations. However, emphasis on higher-order functions for control limits the expressiveness of FP*. This limitation can be overcome in two ways. First, new higher-order functions can be introduced into the language to provide useful programming abstractions, and retain efficient parallel implementations. Mou has presented the divacon formalism in [44] which is a higher-order function useful for describing a wide variety of divide and conquer algorithms. Mou has also outlined a scheme for implementing divacon on parallel architectures in [44]. Languages with a fixed set of higher-order functions and no recursion have been proposed by Darlington et al. [21] to be useful for parallel programming. In their methodology, program transformation techniques are used to transform general functional programs with recursion into compositions of higher-order functions. The second approach is to introduce recursion, as in NESL
Chapter 3
Structure Inference

3.1 Introduction

FP* is a declaration-free programming language like FP. The type and shape attributes of objects are not declared at compile-time as is the case in imperative languages. Type-checking and inference systems have been described in [42, 18]. These systems infer most general types of functional expressions, and determine if programs are well-formed with respect to types. Type-checking and inference provide two important benefits – a large proportion of programming errors are detected at compile-time, and run-time performance is enhanced since concern for type-checking is entirely removed from compiled code. The newer functional languages such as Hope, ML, and Haskell provide strong-typing, user-defined data-types and type-polymorphism.

In FP*, each data object is associated with a structure attribute, which is an abstraction of the form of a data structure. Structures contain information about types and sizes of data objects. The FP* compiler associates structure mappings, which describe input and output structures, with all program expressions. We present a structure inference system that determines if FP* programs are well-formed with respect to structures, and infers most general structure mappings of FP* functions. Structure inference combines type inference, structure coercion, and size inference. Type inference infers the types of various objects and ranks of arrays. Size inference infers the length of each array dimension and relates symbolic dimension lengths in different objects. Structure coercion performs transformations between structures that describe the same underlying data object. Our inference system differs from type inference systems for other functional languages such as ML[42] in the following ways:

- It uses an equational unification procedure to perform structure coercions automatically.
- It incorporates arrays and determines relations between array sizes.

We first introduce the notion of ground structures and non-ground structures and formally define their meanings. We then show how structures provide two distinct views of collections, and the usefulness of automatically transforming one structure representation of collections into another. We next present axioms and inference rules of our structure inference system. In Section 3.6 we describe our structure inference
algorithm. The algorithm uses an equational unification procedure, and generates
constraints on (symbolic) array lengths. We next provide some examples of structure
inference, discuss advantages of structure inference, and conclude with a discussion
of related work.

3.2 Structure

Each FP* object has a structure attribute associated with it, that describes its type,
rank, and size. The set of FP* data objects has been defined in Section 2.3.1. In this
section we demonstrate how structures are used to characterize sets of data objects.

3.2.1 Ground Structures

Definition 3.1 The structure of a data object belongs to the set \( S \) of ground struc-
tures. \( P \) is the set of non-negative integers.

\( S \) is inductively defined as follows:

1. Scalar Structures. \( \text{num, bool} \in S \). \text{num, bool} are scalar structures.

2. Array Structures. If \( s \in S \) and \( n \in P \), then \( s[n] \in S \). \( s[n] \) is an array structure
of length \( n \).

3. Tuple Structures. If \( s_1, s_2, \ldots, s_n \in S \) then \( \langle s_1, s_2, \ldots, s_n \rangle \in S \). \( \langle s_1, s_2, \ldots, s_n \rangle \)
is a tuple structure.

Only what is defined by the above belongs to \( S \).

\( D[s] \) is the set of values denoted by structure \( s \). \( D \) is inductively defined accord-
ing to the syntactic form of structures.

1. Scalar Structures. If \( s \) is a scalar structure, \( D[s] \) is the set of all values repre-
sentable in scalar type \( s \).

2. Tuple Structures. \( D[\{s_1, \ldots, s_n\}] = \{ (d_1, \ldots, d_n) | d_1 \in D[s_1], \ldots, d_n \in D[s_n] \} \)

3. Array Structures. \( D[s[n]] = \{ [d_1, \ldots, d_n] | d_1, \ldots, d_n \in D[s] \} \)

Notice that it is possible to construct structures denoting empty vectors and tuples,
\( i.e., n = 0 \). Also, every FP* object is characterized by a unique ground structure.
3.2.2 Non-Ground Structures

Since FP* functions are polymorphic, their inputs and outputs can have structures of various forms. Non-ground structures are used to describe collections of ground (fixed) structures by using structure and length variables.

**Definition 3.2 Non-ground structures** contain structures, length variables and structure variables. Structure variables range over the structure set $S$. Length variables range over the set $\mathcal{P}$ of non-negative integers.

Only the following are non-ground structures.

1. Any structure in $S$ is a non-ground structure.
2. Any structure variable is a non-ground structure.
3. If $s_1, \ldots, s_n$ are non-ground structures, then $(s_1, \ldots, s_n)$ is a non-ground structure.
4. If $s$ is a non-ground structure, then $s[\textit{L}expr]$ is a non-ground structure. $\textit{L}expr$ is a length expression which is a linear arithmetic expression on length variables.

A ground instance of a non-ground structure is obtained by replacing all structure variables with values from $S$, all length variables with values from $\mathcal{P}$, and then evaluating all length expressions.

The denotation of a non-ground structure is defined in terms of the denotation of all its ground instances.

$$\mathcal{D}[s] = \bigcup \{ \mathcal{D}[s'] | s' \text{ is a ground instance of } s \}$$

**Example 3.1 (Non-ground Structures)** $\text{num}[n]$ denotes all possible 1-dimensional arrays of numbers.

$$\mathcal{D}[\text{num}[n]] = \{ \text{num}[0], \text{num}[1], \ldots, \}$$


$$\mathcal{D}[s[5]] = \{ \text{num}[5], \text{bool}[5], \text{num}[2][5], \ldots \}$$

$s[n][m]$ denotes 2-dimensional arrays of all possible structures and of all possible dimension lengths.

$$\mathcal{D}[s[n][m]] = \{ \text{num}[0][0], \text{num}[1][0], \ldots, \text{bool}[0][0], \text{bool}[1][0], \ldots, \text{num}[12][25], \ldots \}$$

Henceforth, structure will be synonymous with non-ground structures.
3.3 Two Distinct Views of Collections

Arrays and tuples are two distinct representations of collections consisting of ordered sequences of elements. The array structure \( s[n] \) denotes the set of all collections of arbitrary length \( n \) in which elements have the same structure \( s \). A tuple \( (s_1, \ldots, s_n) \) denotes the set of all collections of fixed length \( n \) in which each element can be of arbitrary structure. Collections with elements of the same structure, and fixed lengths can be represented by both arrays and tuples. An array of constant length represents a fixed-length collection of elements of the same structure. Similarly, fixed-length tuples can contain elements of the same structure. This implies that fixed-length collection of elements with identical structure can be represented by both arrays and tuples.

Example 3.2 The collection of three numbers can be represented as \( \langle \text{num, num, num} \rangle \) or as \( \text{num}[3] \).

Our structure inference system uses the following structure coercion axiom which recognizes the isomorphism between homogeneous tuple structures and constant length arrays:

\[
\langle s, \ldots, s \rangle = s[c],
\]

where \( c \) is a non-negative integer.

This axiom forces the compiler to automatically coerce tuples of identical structures to constant-length arrays, and constant-length arrays to tuples. The programmer need not perform explicit coercion that transforms one representation into another. The structure inference system, presented next, uses the coercion axiom to perform transformations between these distinct representations.

The structure coercion axiom can be extended to a congruence relation \( \equiv_s \) on structures. \( [s] \) denotes the equivalence class containing structure \( s \). \([s]\) is the set of structures that can be coerced into each other by zero or more applications of the coercion axiom, i.e.,

\[
[s] = \{ t | s \equiv_s t \}
\]

Example 3.3 \([\langle s, s \rangle[2]] = \{ \langle s, s \rangle[2], s[2][2], \langle s[2], s, s \rangle, \langle (s, s), s[2] \rangle, \langle s[2], s[2] \rangle \}\)

3.4 Structure Mappings

Definition 3.3 The ground structure mapping \( s_{in} \rightarrow s_{out} \) where \( s_{in}, s_{out} \) are ground structures, denotes the following set of total functions.

\[
\mathcal{D}[s_{in} \rightarrow s_{out}] = \{ f | x \in \mathcal{D}[s_{in}], f : x \in \mathcal{D}[s_{out}] \}
\]
Non-ground structure mappings are similar in form to ground structure mappings with the exception that $s_{in}, s_{out}$ are non-ground structures.

**Definition 3.4** A ground instance $s'_{in} \rightarrow s'_{out}$ of a non-ground structure mapping $s_{in} \rightarrow s_{out}$ is obtained by replacing all structure variables and length variables in $s_{in}, s_{out}$ with ground instances.

**Definition 3.5** The non-ground structure mapping $s_{in} \rightarrow s_{out}$ where $s_{in}, s_{out}$ are non-ground structures, is a denotation of all its ground instances, i.e., $\mathcal{D}[s_{in} \rightarrow s_{out}] = \bigcup \{\mathcal{D}[s'_{in} \rightarrow s'_{out}] | s'_{in} \rightarrow s'_{out} \text{ is a ground instance of } s_{in} \rightarrow s_{out}\}$

**Example 3.4** (Structure Mappings) $\_ + \in \mathcal{D}[\langle \text{num, num} \rangle \rightarrow \text{num}]$. Equivalently, we say "$+$ has structure mapping $\langle \text{num, num} \rangle \rightarrow \text{num}$" and also "$+$ : $\langle \text{num, num} \rangle \rightarrow \text{num}$".

- $\text{apnd} \in \mathcal{D}[\langle s[n], s[m] \rangle \rightarrow s[n + m]]$
- $\text{tail} \in \mathcal{D}[s[n + 1] \rightarrow s[n]]$

**Definition 3.6** A normal form structure is represented by $s_\epsilon$ where $s$ is a structure whose length expressions are simple expressions (either variables or constants) and $\epsilon$ is a collection of linear equations in length variables appearing in $s$.

A structure with linear length expressions can be converted into a normal form structure by replacing each complex length expressions $\text{expr}$ with a new length variable $l$ and adding the equation $l = \text{expr}$ into $\epsilon$.

A ground instance of a normal form structure is obtained by replacing all structure variables with structures from the structure set $\mathcal{S}$, and all length variables with non-negative integers such that $\epsilon$ is satisfied.

$\mathcal{D}[s_\epsilon] = \bigcup \mathcal{D}[s']$ where $s'$ is a ground instance of $s_\epsilon$.

**Definition 3.7** The normal form structure mapping $s_{in} \rightarrow_\epsilon s_{out}$ where $(s_{in})_\epsilon$ and $(s_{out})_\epsilon$ are normal form structures is a denotation of all its ground instances, i.e., $\mathcal{D}[s_{in} \rightarrow_\epsilon s_{out}] = \bigcup \{\mathcal{D}[s'_{in} \rightarrow s'_{out}] | s'_{in} \rightarrow s'_{out} \text{ is a ground instance of } s_{in} \rightarrow_\epsilon s_{out}\}$

A structure mapping $s \rightarrow t$ is represented by the normal form structure mapping $s' \rightarrow_\epsilon t'$ where $s'_{\epsilon_1}, t'_{\epsilon_2}$ are normal form representations of $s$ and $t$ respectively, and $\epsilon = \epsilon_1 \cup \epsilon_2$.

**Example 3.5** (Normal Form Structure Mappings)

- $\_ + \in \mathcal{D}[\langle \text{num, num} \rangle \rightarrow \text{num}]$
- $\text{apnd} \in \mathcal{D}[\langle s[n], s[m] \rangle \rightarrow s[l]]$ where $\epsilon = \{l = n + m\}$
- $\text{tail} \in \mathcal{D}[s[m] \rightarrow s[n]]$ where $\epsilon = \{m = n + 1\}$

**Definition 3.8** A normal form structure mapping $s_{in} \rightarrow_\epsilon s_{out}$ is a most general structure mapping of an $FP^*$ function $f$ if, whenever $x \in \mathcal{D}[(s_{in})_\epsilon]$ and $f : x = y$, $y \in \mathcal{D}[(s_{out})_\epsilon]$. 

28
3.5 Structure Inference System

The inference system takes as input an FP* function \( f \) and deduces its most general normal form structure mapping \( f : s_{in} \rightarrow s_{out} \) (if one exists). The inference system consists of a collection of axioms and inference rules. The axioms correspond to structure mappings of primitive functions. Each higher-order function in FP* has an inference rule associated with it. These rules derive the structure mappings of functions constructed out of higher-order constructs from the structure mappings of constituent functions.

3.5.1 Structure Inference Axioms

Axioms of the structure inference system are listed in Figure 3.1. Occurrences of \( s \) (possibly subscripted) are structure variables and \( n, m \) are length variables. Note that some of the structure mappings, such as those for \( apndl, tail, apnd \), are not in normal form. Later we present the normal form representations of these structure mappings and in Section 3.7.2 we provide a simple procedure for performing the conversion.

<table>
<thead>
<tr>
<th>( c : s \rightarrow \text{num} )</th>
<th>( + : (\text{num, num}) \rightarrow \text{num} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( * : (\text{num, num}) \rightarrow \text{num} )</td>
<td>( = : (\text{num, num}) \rightarrow \text{bool} )</td>
</tr>
<tr>
<td>( &lt;= : (\text{num, num}) \rightarrow \text{bool} )</td>
<td>( sel(i, k) : (s_1, \ldots, s_i, \ldots, s_k) \rightarrow s_i )</td>
</tr>
<tr>
<td>( sel(i) : s[n] \rightarrow s )</td>
<td>( sel : (\text{num, s[n]}) \rightarrow s )</td>
</tr>
<tr>
<td>( id : s \rightarrow s )</td>
<td>( trans : s[n][m] \rightarrow s[m][n] )</td>
</tr>
<tr>
<td>( pairUp(k) : (s_1[n], \ldots, s_k[n]) \rightarrow (s_1, \ldots, s_k)[n] )</td>
<td>( distl : (s_1, s_2[n]) \rightarrow (s_1, s_2)[n] )</td>
</tr>
<tr>
<td>( tail : s[n + 1] \rightarrow s[n] )</td>
<td>( apndl : (s, s[n]) \rightarrow s[n + 1] )</td>
</tr>
<tr>
<td>( apnd : (s[n], s[m]) \rightarrow s[n + m] )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Structure Inference Axioms

These structure mappings impose semantics on FP* primitives that are different from those in FP. For example, the input to the \( trans \) primitive is an array structure, as denoted in the input structure description \( s[n][m] \). In FP, valid inputs to the \( trans \) primitives may not be representable as arrays. The set of valid inputs to primitive functions have been reduced in FP* to allow the development of a polymorphic structure inference system.

29
3.5.2 Inference Rules

Higher-order functions in FPL are associated with inference rules. All of the rules are listed in Figure 3.2. Every rule is conditional. The conclusion below the line is derivable if the premise above the line can be dispatched. The inference rules for Apply-all, Insert(1) and Insert(2) introduce length variables, which must be unique.

| Composition | $g : s \rightarrow t_1$  
$ f : t_1 \rightarrow t_2$ | $\frac{}{f \circ g : s \rightarrow t_2}$ |
| Apply All ($\alpha$) | $f : s \rightarrow t_1$ | $\frac{}{\alpha f : s[n] \rightarrow t_1[n]}$ |
| Construction ($\langle \ldots \rangle$) | $f_1 : s \rightarrow t_1$  
$f_2 : s \rightarrow t_2$  
$\vdots$  
$f_n : s \rightarrow t_n$ | $\frac{}{\langle f_1, f_2, \ldots, f_n \rangle : s \rightarrow \langle t_1, t_2, \ldots, t_n \rangle}$ |
| Conditional ($\rightarrow$) | $f_1 : s \rightarrow \text{bool}$  
$f_2 : s \rightarrow t$  
$f_3 : s \rightarrow t$ | $\frac{}{f_1 \rightarrow f_2 ; f_3 : s \rightarrow t}$ |
| Insert(1) ($/\cdot$) | $f : \langle s, s \rangle \rightarrow t$ | $\frac{}{/f : s[n] \rightarrow s}$ |
| Insert(2) ($/\cdot$) | $f : \langle s[n], s[n] \rangle \rightarrow s[2n]$ | $\frac{}{/f : s[n][n] \rightarrow s[2n]}$ |
| While Do | $e : s \rightarrow \text{bool}$  
$f : s \rightarrow s$ | $\frac{}{\text{While } e \text{ Do } f : s \rightarrow s}$ |
| Defined Functions | $\text{def } f = e$  
$e : s \rightarrow t$  
$f : s \rightarrow t$ | $\frac{}{f : s \rightarrow t}$ |

Figure 3.2: Structure Inference Rules for Functionals
The deduction rules for functionals $\text{Insert}(1)$, $\text{Insert}(2)$ and $\text{While}$ impose a structure preservation property whereby the output structure is always contained in the input structure. In $/f$, either $f : (s, s) \rightarrow s$ or $f : (s[n], s[n]) \rightarrow s[2n]$. $\text{Insert}(2)$ requires the output object structure to be exactly the size of the input object structure. This allows for efficient storage allocation schemes as observed in Chapter 6. Similarly, in $\text{While } e \text{ Do } f$, the structure mapping for $f$ is $f : s \rightarrow s$. Efficient storage allocation schemes exploit the fact that the input object structure is identical to the output object structure.

### 3.6 Structure Inference Algorithm

Our structure inference algorithm is similar to algorithm $\mathcal{W}$ [42]. Whenever the algorithm succeeds in producing a structure mapping for a function, then that mapping can be deduced from the inference system. The structure inference algorithm is a proof heuristic, i.e., it is a strategy for determining the order in which the inference rules should be applied. The inference rules are applied to the function expression, whose structure mapping is being determined, from left-to-right and from the inside-out.

For every higher-order function except $\text{Insert}$, there is exactly one inference rule that may be used to derive the structure mapping of the function constructed using that higher-order function. In Section 3.8 we will present conditions that disambiguate the application of the inference rules for $\text{Insert}$.

When the condition of an inference rule contains more than one occurrence of the same structure, application of the rule requires the most general unifier of a list of structures $s_1, \ldots, s_n$, e.g., $\text{Composition}$ and $\text{Construction}$. Unification requires structure coercion, where arrays are coerced to tuples and vice-versa. Unification must also resolve linear constraints on length variables. In Section 3.7 we present techniques used to determine the most general unifiers of structures.

### 3.7 Most General Instances of Structures

In the conditions of some of the inference rules in Figure 3.2, the same structure appears in more than one place. For example, in the structure inference rule for $\text{Composition}$, the structure $t_1$ appears in the output of $g$ and the input of function $f$. Application of the $\text{Composition}$ functional is well-formed on functions $f$ and $g$ if the output structure of $g$ is identical to input structure of function $f$. The structure inference algorithm derives structures for $f$ and $g$ bottom-up. In general the algorithm derives a structure mapping $g : s_1 \rightarrow t'_1$ and $f : t''_1 \rightarrow t_2$. The inference rule for $\text{Composition}$ requires that structure $t'_1$ be made identical to structure $t''_1$.

The inference algorithm achieves this equality by specializing the structure mappings, applying a substitution $\sigma$ on structure and length variables in the structures such that $\sigma(t'_1) \equiv \sigma(t''_1)$. This substitution ensures that the input structure of function $f$ is either identical to that of the output structure of function $g$ or can be coerced
to it by one or more applications of the coercion axiom. It is important that the inference system specialize the structure mappings as little as possible so that most general structure mappings will be inferred.

A substitution \( \sigma \) assigns structures to structure variables and length expressions to length variables in a structure. The structure \( \sigma(t) \) is obtained by applying the assignments in \( \sigma \) to variables in \( t \). The most general unifier (cf. [47]) of structures \( s \) and \( t \) is a substitution \( \sigma \) such that \( \sigma(s) = \sigma(t) \). Further, for any other substitution \( \tau \) where \( \tau(s) = \tau(t) \), it must be the case that \( \tau = \sigma \circ \theta \) where \( \theta \) is some other substitution. Thus \( \sigma \) imposes the least amount of constraint on variables in \( s \) and \( t \) to make them syntactically identical. Most type inference systems in the literature [42, 18] use unification to determine the most general instance of types.

In our case it not sufficient to use unification since structures that are not syntactically identical may be in the same \( \equiv_s \) equivalence class. Our inference system has to perform coercions that equate homogeneous tuples to constant-length arrays.

The structure inference system uses an equational unification procedure to determine the most general substitution \( \sigma \) on structures \( s \) and \( t \) such that \( \sigma(s) \equiv_s \sigma(t) \). In the discussion on equational unification in Section 3.7.1 we assume that all length expressions in structures are simple expressions, i.e., lengths are either variables or constants. In Section 3.7.2 we will relax this constraint and allow linear expressions as array lengths.

### 3.7.1 Equational Unification

We briefly review the standard concepts of equational unification following the presentation in [33, 58]. An equational theory is characterized by a finite set of equational axioms. The theory under consideration consists of the coercion axiom presented in Section 3.3.

Equational theories are classified as being unitary, finitary, infinitary, or nullary depending on whether the minimal complete set of unifiers for a pair of unifiable terms is in general a singleton, finite, infinite or empty set. Unification in the empty theory is unitary, under commutativity it is finitary, under associativity it is infinitary, and under associativity and idempotence it has the unpleasant feature that no complete set of general unifiers even exists.

Thatte [58] has formally characterized the class of equational theories that are unitary.

An equational theory \( E \) is finite if the following hold:

- \( E \) must be regular, i.e., each equation in \( E \) must have the same variables on both sides.

- The two sides in each equation in \( E \) must have the same depth (nesting of function symbols).
Thatte also discusses conditions under which equational theories are acyclic. We present the following sufficient condition under which a theory is acyclic.

- A theory \( E \) with exactly one equation that does not have the same function symbols on both sides is acyclic.

**Theorem 3.1** The equational theory of the coercion axiom is a finite acyclic theory.

**Proof** The coercion axiom defines an equation that satisfies both conditions for finiteness, and is trivially acyclic. \( \square \)

**Theorem 3.2** (Thatte) Every finite acyclic theory is unitary.

**Proof** [58]. \( \square \)

**Corollary 3.1** There exists a procedure for determining the unique most general unifier (if one exists) of structures \( s \) and \( t \) under the coercion axiom.

Algorithms for performing equational unification have been presented in the literature [58, 53]. Our equational unification algorithm is similar. It is a minor variant of the standard unification algorithm due to Robinson[47]. When the unification procedure detects a function clash between \( \langle \rangle \) and \( [\ ] \), it attempts to coerce the array into a tuple, and attempts to unify components of the tuple structures.

### 3.7.2 Linear Length Expressions

In the previous section we presented procedures for determining the most general substitution \( \sigma \) for structures \( s \) and \( t \) such that \( \sigma(s) \equiv_s \sigma(t) \) where all length expressions in \( s \) and \( t \) are simple expressions consisting of either variables or constants. However, axioms for \( apndl \) and \( apnd \), and the inference rule for \( Insert(2) \) contain non-simple length expressions. In this section we extend the techniques in Section 3.7.1 to deal with linear length expressions in structures. The procedure is based on representing structures in normal form. Normal form structures were described in Section 3.4. A normal form structure \( s_e \) contains only simple length expressions in \( s \), and a set \( \epsilon \) of linear equations which the length variables in \( s \) must satisfy.

The structure mapping of all primitive functions can be converted into normal form mappings as shown in Figure 3.3 using the procedure described in Section 3.4. \( s_{in} \rightarrow s_{out} \) denotes \( s_{in} \rightarrow \langle \rangle s_{out} \).

The most general instance of normal form structures \( s_{e1} \) and \( t_{e2} \) is obtained by computing the most general equational unifier \( \sigma \) of structures \( s \) and \( t \); satisfiability of \( \sigma(e1 \cup e2) \) is then determined. The most general instance is \( \sigma(s) \) with constraints \( \sigma(e1 \cup e2) \). Note that \( \sigma(s) \) contains only simple length expressions.
We illustrate unification by applying the inference rule for the Composition higher-order function as in \( f \circ g \). The inference algorithm derives the structure mappings \( f : s \rightarrow_{c1} t' \) and \( g : t'' \rightarrow_{c2} t \) in a bottom-up fashion. The most general unifier \( \sigma \) of \( t' \) and \( t'' \) is found such that \( \sigma(t') \equiv_{s} \sigma(t'') \). The derived output structure mapping is \( \sigma(s) \rightarrow_{s} \sigma(t) \) where \( \epsilon = \sigma(c \cup c2) \).

Inference rules for all higher-order functions are applied in similar fashion. The most general unifier of structures is determined, and then substitutions are applied on the structures appearing in the structure mapping derived for the functional. The set of equations is obtained by taking the union of the constraints generated for each of the constituent functions and then applying the substitution \( \sigma \). Since most general unifiers are obtained, each inference rule derives the most general structure mapping for functions constructed using the corresponding higher-order function.

Unification of structures in the general setting where length variables must satisfy sets of linear constraints requires a test of satisfiability. The solution(s) must range over the set \( \mathcal{P} \) of non-negative integers.

**Theorem 3.3** Testing satisfiability of a set of linear constraints over \( \mathcal{P} \) is NP-complete.

**Proof** Satisfiability of linear constraints reduces to the following NP-complete problem [51].

Given rational matrix \( A \), and rational vector \( b \) does \( Ax = b \) have a non-negative integral solution \( x \)?
There exists an exponential-time algorithm to determine if a solution to a set of linear equations exists[51]. In practice, however, these equations arise rarely. If all primitive functions in an FP* function have structure mappings with simple length expressions, and the inference rule for Insert(2) is not used, then all array lengths in the inferred structure mapping are simple expressions. Consequently no equations are present in the inferred structure mapping. The only equations that are present in the derived structure mapping are either equations in structure mappings of primitive functions, or added because of the second inference rule for Insert. The algorithm determines the existence of a solution to the equations generated by structure inference. Since array lengths are known, in general, only at run-time, this requires that the satisfiability of these equations be determined at run-time, for the given array lengths. This is further discussed in Section 6.4.

3.8 Disambiguating Inference Rules for Insert

There are two inference rules for deriving structure mappings for functions constructed using the Insert functional. These rules correspond to two distinct semantic interpretations of the Insert functional. In the first interpretation \( f \) in \( /f \) must be a binary function whose output has the same structure as each of the two structures in the input tuple. In the second interpretation, the output of \( f \) is an array whose length is the sum of the lengths of the two array structures in the input tuple. The two rules with equational unification made explicit are shown in Figure 3.4.

\[
\begin{align*}
\text{Insert(1)} \ (\text{f1}) & : \quad f : \langle s_1, s_2 \rangle \rightarrow_{\epsilon} s_3 \\
\sigma(s_1) & \equiv_s \sigma(s_2) \equiv_s \sigma(s_3) \\
\therefore f : \sigma(s_1)[n] \rightarrow_{\sigma(\epsilon)} \sigma(s_1)
\end{align*}
\]

\[
\begin{align*}
\text{Insert(2)} \ (\text{f2}) & : \quad f : \langle s_1, s_2 \rangle \rightarrow_{\epsilon} s_3 \\
\sigma(s_1) & \equiv_s \sigma(s_2) \equiv_s \sigma(s_3) \\
\sigma(s_1) & \equiv_s \sigma[|n_1|], \sigma(s_2) \equiv_s \sigma[|n_2|], \sigma(s_3) \equiv_s \sigma[|n_3|] \\
\therefore f : s[c][n] \rightarrow_{\epsilon_3} s[n_4], \epsilon_3 = \sigma(\epsilon_2) \cup \{ n_1 = n_2, n_2 = 2n_3, n_4 = cn \}
\end{align*}
\]

Figure 3.4: Explicit Inference Rules for Insert

Insert(2), which is the second inference rule for Insert, is applicable only if the input of \( /f \) is known to be of structure \( s[c][n] \). This information may be known either through user annotations, or by deriving the output or input structure of other functions. For example in \( (/f) \circ g \), the derived output structure of \( g \) is the input structure of \( /f \).

If the input structure is known to be \( s[c][n] \) (or is coercible to it), then both inference rules for Insert are applicable, because \( f \) in \( /f \) may have structure mapping
\((s[n], s[n]) \rightarrow \epsilon \ s[n]\). In this case the first rule should be applied. If \(f\) has structure mapping \((s[n], s[n]) \rightarrow \epsilon \ s[2n]\), the second rule should be applied. The inference algorithm needs to analyze \(\epsilon\) to deduce which rule is applicable.

The structure inference system derives the structure mapping \(f : (s_1, s_2) \rightarrow s_3\). For the first rule to be applicable, it must be the case that \(s_1 \equiv s[n_1], s_2 \equiv s[n_2], s_3 \equiv s[n_3]\), and \(s[n_1] \equiv s[n_2] \equiv s[n_3] \equiv s[n]\). The equations \(n_1 = c, n_2 = c, n_3 = c\) are added into \(\epsilon\) and the inference system determines if \(\sigma(\epsilon) \cup \{n_1 = c, n_2 = c, n_3 = c\}\) is satisfiable. If so, then the first rule is applied as the output object has the same length as the input length. If it is not satisfiable, then the second rule is applied. The correctness of this approach lies in the fact that if the set of equations were satisfiable, then the array length of the output structure of \(f\) would be equal to that of each of the input array structures. If the equations were inconsistent, then it must be the case that the array lengths were not equal, and in fact the output array length may be twice that of each of the input array lengths. The structure inference algorithm proceeds by adding these constraints into the equation set and applying the second inference rule for Insert.

### 3.9 Structure Inference Examples

In this section, we describe in detail structure inference on the FP* function \(ip\) (program 2.2) in Section 2.5. After replacing all user-defined functions with their definitions, we obtain the following definition for \(ip\).

\[
\text{def } ip : \langle v_1, v_2 \rangle = (\it{+/}) \circ \alpha(*) \circ \text{trans}
\]

The structure mappings for primitive functions \(\text{trans}\) and \(*\) (from Figure 3.3) are as follows.

\[
\begin{align*}
\text{trans} & : s[n][m] \rightarrow s[m][n] \\
\ast & : \langle \text{num, num} \rangle \rightarrow \text{num}
\end{align*}
\]

Applying the inference rule (from Figure 3.2) for \(\text{ApplyAll} (\alpha)\) on (3.2), we obtain the following.

\[
\alpha(*) : \langle \text{num, num} \rangle[n'] \rightarrow \text{num}[n']
\]

The structure mapping for \(\alpha(*) \circ \text{trans}\) is derived by applying the inference rule (from Figure 3.2) for \(\text{Composition} (\circ)\) on (3.1) and (3.3) This requires that the most general instance of structures \(s[m][n]\) and \(\langle \text{num, num} \rangle[n']\) be obtained. First, the following coercion is performed.

\[
\langle \text{num, num} \rangle[n'] \equiv \text{num}[2][n']
\]

Next, the following set \(\sigma\) of substitutions is generated through equational unification such that \(\sigma(s[m][n]) \equiv \sigma(\text{num}[2][n'])\).

\[
\sigma = \{ s := \text{num}, m := 2, n := n' \}
\]
Applying (3.5) on (3.1) and (3.3), we have the following.

\[ \alpha(*) \circ \text{trans} : \text{num}[2][n] \to \text{num}[n] \quad (3.6) \]

The structure mapping for + (from Figure 3.3) is identical to that for * as shown below.

\[ + : (\text{num}, \text{num}) \to \text{num} \quad (3.7) \]

Applying the inference rule (from Figure 3.2) for Insert(1) on (3.7) we get the following.

\[ /+ : \text{num}^{[n'']} \to \text{num} \quad (3.8) \]

Finally, applying the inference rule for Composition on (3.6) and (3.8), we obtain the following structure mapping for ip.

\[ (/+) \circ \alpha(*) \circ \text{trans} : \text{num}[2][n] \to \text{num} \quad (3.9) \]

Equivalently, since

\[ \text{num}[2][n] \equiv (\text{num}, \text{num})[n] \quad (3.10) \]

we obtain the following structure mapping for ip.

\[ (/+) \circ \alpha(*) \circ \text{trans} : (\text{num}, \text{num})[n] \to \text{num} \quad (3.11) \]

The derived structure mappings of other FP* programs described in chapter 2 are shown below:

**Example 3.6**

\[ \text{num} : (\text{num}[n][m], \text{num}[p][n]) \to \text{num}[p][m] \]
\[ \text{permute} : (\text{num}[n], \text{num}[n]) \to \text{num}[n] \]
\[ \text{rank} : \text{num}[n] \to \text{num}[n] \]
\[ \text{MaxLoc} : (\text{num}[n], \text{bool}[n]) \to \text{num} \]
\[ \text{ForwardElim} : \text{num}[n][m] \to (\text{num}, \text{num}[n][m], \text{bool}[m], \text{num}[m]) \]

### 3.10 Discussion and Related Work

Structure inference has all the benefits of type inference. It checks if programs are well-formed with respect to structures. It also determines the structures of input and output objects in FP* programs. This information is used to reduce storage allocation overhead at run-time. Structure inference also provides the benefit of automatic coercion of tuples and arrays. This reduces the programmer’s burden in
developing programs in FP*. Structure inference also analyzes array lengths and determines relations among lengths of different dimensions in the same array, and across different arrays. Array sizes appear as index bounds in forall loops, which is the target execution model of our compiler. Chatterjee has shown techniques based on loop fusion that require loop bound information for implementing forall loops efficiently onto MIMD machines[19].

Our contribution has been twofold. We have introduced a polymorphic structure system for the FP* programming language. Arrays are an important part of the structure system. We have unified type inference, type coercion, and size inference into a common framework based on a single inference system. This inference system has been implemented and used to derive most general structure mappings for FP* functions. A preliminary report of these results has appeared in [8, 66].

Type inference [18] has been extensively researched in functional programming. Milner [42] presented the earliest type inference system, which was later used for the functional language ML. Extensions to Milner’s type inference system have been proposed by others[43]. Type coercion has been studied extensively by Thatte[58]. Chatterjee[20] has performed size inference for VCODE, an applicative strongly typed data-parallel language, that is compiled for shared memory machines. Katayama[32] has constructed a type inference system for FP similar to the system presented here for inferring types of non-recursive programs. The inference system has been presented in a relational algebraic framework. Inference rules correspond to basic relations on type expressions. However, inference rules are not presented for the while and the tree insert functionals and the system handles only simple array length expressions. Details of the unification procedure are not presented. The types of a class of recursive FP programs are deduced using linear fixed point iteration. Again, it is not clear if the procedure can be fully automated. Type inference systems have also been presented for dynamically typed languages. Aiken et al [2] present a type inference system for FL where types are sets of expressions, and not sets of values.
Chapter 4
Routing Function Definitions

4.1 Introduction

Primitive functions in FP can be distinguished into two categories — (1) \textit{computational functions}, and (2) \textit{routing functions}. The computational functions in FP correspond to the arithmetic, logical and relational instructions in programming languages. Examples of computational functions are $+$, $\text{and}$, $\leq$. All computational functions in FP are included in FP*. FP also has an extensive set of routing functions which copy values from inputs to outputs. Examples of such functions are \textit{transpose}, \textit{reverse}, \textit{distribute from right}. In Section 4.2 we precisely define routing functions. Routing functions are particularly useful in data-parallel programming in FP*. In Section 4.3 we demonstrate the importance of routing functions in data-parallel programming through a set of examples. FP provides a fixed set of primitive routing functions. We have also introduced a programming language feature which allows the definition of new primitive routing functions. Routing functions are characterized by collections of \textit{index equalities}, which associate locations in an output object with locations in an input object, such that the value in the output location is always a copy of the input location. In Section 4.4 we introduce index equalities and show how routing functions can be defined using them. We present criteria in Section 4.5 that determine the correctness of routing function definitions. We also describe tests that correct routing function definitions must satisfy. Evaluation of routing functions on massively parallel architectures could potentially lead to significant inter-processor communication because elements in data structures may be distributed across distinct processor nodes. Routing function evaluation would require movement of elements among nodes through the network. On the CM-2 inter-processor communication can be as much as a two orders of magnitude slower than local computation, and may lead to severe performance degradation. Our compiler pays special attention to evaluating routing functions in order to minimize inter-processor communication. The compiler uses routing function definitions based on index equalities for performing the optimizations described in Chapter 6.

4.2 Routing Functions

We precisely define routing functions in this section. Routing functions copy a subset of values from inputs to outputs. Each location in an output object receives a value
from the input. Further, the location of the input value is fully determined by the
destination location in the output object.

Routing functions have the following properties:

1. No new values are created. All scalars in the output are a subset of those of
the input.

2. Each output location completely determines the input location containing a
value to be copied. Locations are independent of values in the input.

3. Exactly one input value is copied into each output location. However, the same
input element can be copied onto multiple output locations.

The first property eliminates computational functions such as +, and, ≤. However,
it does not eliminate functions such as sort in which the output is a copy of the
input values. The second property rules out functions like sort by stating that each
output element’s location in the input is independent of the values in the input. The
third property rules out collisions in routing functions where distinct input values
may be sent to the same output location. Each output location receives a value from
exactly one input location. Routing functions are, therefore, said to be collision free.

The second property of routing functions enable them to be completely charac-
terized by mappings of locations, called index equalities, which describe how destina-
tion locations in an output object are mapped to source locations in an input object.
In Section 4.4 we show how index equalities can be used to describe mappings of
locations.

4.3 Programming Examples

Routing functions are useful in designing data-parallel programs in FP*. The effect
of routing functions is to rearrange data so that data-parallel computation can be
specified using the Apply-all and Insert functionals.

For example, consider the routing function trans (transpose) in the function
definition of ip (inner-product) from program 2.2.

\[
def \text{ip} = (/+) \circ (\alpha \ast) \circ \text{trans}
\]

The input to ip consists of two vectors \([x_1, \ldots, x_n], [y_1, \ldots, y_n]\) which is coerced into
\([x_1, \ldots, x_n], [y_1, \ldots, y_n]\). trans re-arranges the input to \([x_1, y_1], \ldots, [x_n, y_n]\) which is
coerced into \([x_1, y_1], \ldots, (x_n, y_n)]\) where each \(x_i\) is adjacent to \(y_i\). At this point, \(\alpha \ast\)
evaluates each product \(x_i \ast y_i\) in parallel on the rearranged object. The overall effect
of trans is to re-arrange values so that data-parallel operations can be specified using
\(\alpha \ast\).
4.4 Routing Function Definition

We present a notation that allows programmers to express primitive routing functions. Programmers can use this notation to describe their own routing functions. This notation is used by the compiler to describe all primitive FP* routing functions. The notation describes routing functions in sufficient detail to enable the compiler to optimize routing function evaluation. In this section we present ground routing function definitions that operate on inputs whose array lengths are fixed. In Section 4.6 we extend this schema to define routing functions that take inputs with variable lengths.

4.4.1 Index References

Definition 4.1 An index reference $x[i_1, ..., i_n]$ where $[i_1, ..., i_n]$ is a list of positive integers, denotes an element of FP* object $x$.

An index reference can be to the entire object $x$ itself, or some sub-element in $x$, or a scalar element in $x$. $x[i_1, ..., i_n]$ is inductively defined below.

1. $x[] = x$

2. $([x_1, ..., x_m])[i_1, ..., i_n] = \begin{cases} (x_{i_1})[i_2, ..., i_n] & 1 \leq i_1 \leq m \\ \bot & \text{otherwise} \end{cases}$

3. $(x_1, ..., x_m)[i_1, ..., i_n] = \begin{cases} (x_{i_1})[i_2, ..., i_n] & 1 \leq i_1 \leq n \\ \bot & \text{otherwise} \end{cases}$

All other index references return $\bot$, e.g., $5[3] = \bot$.

Example 4.1

$x = \langle x_1, [x_{21}, ..., x_{28}], ..., x_6 \rangle$

$x[] = \langle x_1, [x_{21}, ..., x_{28}], ..., x_6 \rangle$

$x[2] = [x_{21}, ..., x_{28}]$

$x[2, 3] = x_{23}$

4.4.2 Index Equalities

Definition 4.2 A ground index equality $y[i_1, ..., i_n] = x[j_1, ..., j_n]$ denotes the equality of values in $x$ and $y$.

Definition 4.3 $y[i_1, ..., i_n] = x[j_1, ..., j_n], R$ denotes a non-ground index equality where

- $i_k, j_k$'s are expressions in variables ranging over the integers.
• $R$ describes constraints that variables must satisfy. The constraints are of the form $lb \leq i \leq ub$ where $lb, ub$ are integer constants.

A ground instance of a non-ground index equality is obtained by replacing variables with integers such that $R$ is satisfied. $\sigma_v$ denotes a binding of variables with integers that satisfies $R$. The set of ground index equalities obtained by substituting variables with integers such that $R$ is satisfied is defined by function $S_I$.

$$S_I[y[i\text{list}_1] = x[i\text{list}_2], R] = \{y[\sigma_v(i\text{list}_1)] = x[\sigma_v(i\text{list}_2)]|\sigma_v \text{satisfies } R\}$$

### 4.4.3 Ground Routing Function Definitions

A ground routing function is defined as shown in Figure 4.1.

```
def r(x : s_in, y : s_out)
    IndexEquality_1, 
    ...
    IndexEquality_n
end def
```

Figure 4.1: Routing Function Definition Schema.

In this scheme $x, y$ are variable names that denote input and output objects respectively, of the routing function $r$. $s_{in}$ and $s_{out}$ are (non-ground) structures that denote the structure of the input and output, respectively. $s_{in}$ and $s_{out}$ do not contain length variables.

Example 4.2 shows the definition of primitive routing function $apnd$ where input and output arrays have constant lengths denoted by $m, n$.

**Example 4.2**

def apnd(x : (s[m], s[n]), y : s[m + n])
    y[i] = x[1, i], 1 \leq i \leq m
    y[i] = x[2, i - m], m + 1 \leq i \leq m + n
end def

It must be the case that all structure variables occurring in $s_{out}$ also occur in $s_{in}$. This ensures that the structure of the output object, for a given input instance, can be obtained by substituting structure variables in $s_{out}$ with these values.

The definition of routing function $r$ is denoted as $r(x, y)$ where $x$ denotes the name of an input object, and $y$ the name of an output object. The set of ground
index equalities $S_T[r(x, y)]$ described by routing function definition $r(x, y)$ is defined as follows:

$$S_T[r(x, y)] = \bigcup_{i=1}^{n} S_T[IE_i]$$

The denotation of a ground routing function definition $r(x, y)$ is expressed in terms of functions from $s_{in}$ to $s_{out}$ that satisfy the input-output constraints of the index equalities. If $F = D[s_{in} \rightarrow s_{out}]$ and $IE_i$ denotes a non-ground index equality in $r(x, y)$, then $D_R[r(x, y)]$ denotes the following subset of $F$:

$$D_R[r(x, y)] = \{ f \in F \mid f(x[ilist_2]) = y[ilist_1] \text{ and } y[ilist_1] = x[ilist_2] \in S_I[r(x, y)] \}$$

4.4.4 Structure of Index References

Definition 4.4 The function $struct(s, [i_1, \ldots, i_n])$, which denotes a sub-structure of $s$, is defined as follows:

- $struct(s, []) = s$
- $struct(s[l], [i_1, \ldots, i_n]) = struct(s, [i_2, \ldots, i_n])$ where $l$ is any length expression.
- $struct((s_1, s_2, \ldots, s_c), [i_1, \ldots, i_n]) = \{ struct(s_{i_1}, [i_2, \ldots, i_n]) \mid 1 \leq i_1 \leq c \}
\begin{cases} 
\perp & \text{otherwise} 
\end{cases}$

Each index expression $i_k$ in an index reference $x[i_1, \ldots, i_m]$, where $x \in D[s]$, is either an array index or a tuple index.

- $i_k$ in $[i_1, \ldots, i_m]$ is an array index if $struct(s, [i_1, \ldots, i_{k-1}]) = s[n]$.
- $i_k$ in $[i_1, \ldots, i_m]$ is a tuple index if $struct(s, [i_1, \ldots, i_{k-1}]) = (s_1, \ldots, s_n)$.

Theorem 4.1 If $x[i_1, \ldots, i_n]$ and $x[j_1, \ldots, j_n]$, where $x \in D[s]$, differ only in their array indices then $struct(s, [i_1, \ldots, i_n]) = struct(s, [j_1, \ldots, j_n])$.

Proof By induction on $n$. \hfill \Box

4.4.5 Syntactic Restrictions on the Form of Index Equalities

The following restrictions are imposed on the form of index references in index equalities within routing function definitions. $[ilist_1]$ and $[ilist_2]$ denote index references on the output and input objects, respectively.

- Each index occurring in $[ilist_1]$ must be either a constant or a distinct variable. Indices occurring in $[ilist_2]$ may be expressions using variables occurring in $[ilist_1]$ and operators $+, -, *, \text{mod}, /_c$, where $/_c$ denotes division by constant.
• All tuple indices in \([i\text{list}_1]\) and \([i\text{list}_2]\) must be positive integers. Indexing variables are not allowed in tuple index expressions.

In addition, the set of constraints \(R\) satisfies the following:

• For each variable \(i\) in \([i\text{list}_1]\) there is exactly one constraint \(lb \leq i \leq ub \in R\).

4.5 Correctness of Ground Routing Function Definitions

In this section we present conditions that a correct ground routing function definition must satisfy.

Definition 4.5 A ground index equality instance \(y[i\text{list}_1^\prime] = x[i\text{list}_2^\prime]\), where \(x \in D[s_{in}], y \in D[s_{out}]\), is well-formed if:

• It is well-structured: \(\text{struct}(s_{out}, [i\text{list}_1^\prime]) = \text{struct}(s_{in}, [i\text{list}_2^\prime])\).

• It is well-defined: \(y[i\text{list}_1^\prime] \neq \bot\) and \(x[i\text{list}_2^\prime] \neq \bot\).

Definition 4.6 A non-ground index equality \(IE\) is well-formed if all ground index equalities in \(S(IE)\) are well-formed.

In Section 4.5.1 we present compile-time tests to determine whether non-ground index equalities in a ground routing function definition are well-formed.

Routing functions of \(FP^\star\) should be collision free: exactly one value will be copied into any output location. A ground routing function definition \(r(x, y)\) denotes a collection of ground index equality instances. In Section 4.5.2 we present compile-time tests to determine whether a routing function is collision free.

Finally, a routing function definition must assign all locations of the output object with values from the input, i.e., the output is completely defined. In Section 4.5.3 we present compile-time tests to determine whether a definition completely assigns the output object.

Definition 4.7 A ground routing function definition is correct if it satisfies the following:

• All index equalities are well-formed.

• The definition is collision free.

• The output is completely defined.
4.5.1 Well-Formed Index Equalities

Distinct ground instances $y[i\text{\small{st}}_1'] = x[i\text{\small{st}}_2']$ and $y[i\text{\small{st}}_1''] = x[i\text{\small{st}}_2'']$ of index equality $y[i\text{\small{st}}_1] = x[i\text{\small{st}}_2]$ can differ only in the value of array indices in $[i\text{\small{st}}_1'], [i\text{\small{st}}_2']$ and $[i\text{\small{st}}_1''], [i\text{\small{st}}_2'']$, respectively. Let $x \in D[s_{\text{in}}]$ and $y \in D[s_{\text{out}}]$. By Theorem 4.1, $\text{struct}(s_{\text{out}}, [i\text{\small{st}}_1]) = \text{struct}(s_{\text{in}}, [i\text{\small{st}}_2])$ if and only if $\text{struct}(s_{\text{out}}, [i\text{\small{st}}_1]) = \text{struct}(s_{\text{in}}, [i\text{\small{st}}_2])$.

A non-ground index equality $y[i\text{\small{st}}_1] = x[i\text{\small{st}}_2]$, $R$ is well-structured if $\text{struct}(s_{\text{out}}, [i\text{\small{st}}_1]) = \text{struct}(s_{\text{in}}, [i\text{\small{st}}_2])$, for all possible $i\text{\small{st}}_1$ such that $\text{struct}(s_{\text{out}}, i\text{\small{st}}_1) \neq \bot$.

Index equality $y[i\text{\small{st}}_1] = x[i\text{\small{st}}_2]$, $R$ is well-defined if all array indices in $i\text{\small{st}}_1$ and $i\text{\small{st}}_2$ lie within valid array locations in $s_{\text{out}}$ and $s_{\text{in}}$, respectively. Let $C(IE)$ be the set of constraints that need to be satisfied if all array references in index equality $IE$ are valid.

$$C(IE) = C(y[i_1, \ldots, i_n] = x[j_1, \ldots, j_m], R) =$$

$$\{1 \leq j_k \leq l | 1 \leq k \leq m \text{ and } \text{struct}(s_{\text{in}}, [j_1, \ldots, j_{k-1}]) = s[l]\} \cup$$

$$\{1 \leq i_k \leq l | 1 \leq k \leq n \text{ and } \text{struct}(s_{\text{out}}, [i_1, \ldots, i_{k-1}]) = s[l]\}$$

Determining the satisfiability of $1 \leq i_k \leq l$ requires that we determine the minimum value $\min(i_k, R)$ and the maximum value $\max(i_k, R)$ of $i_k$ that satisfies $R$.

The function $\max(i, R)$ is inductively defined below:

- $\max(i_1 + i_2, R) = \max(i_1, R) + \max(i_2, R)$
- $\max(i_1 - i_2, R) = \max(i_1, R) - \min(i_2, R)$
- $\max(i_1 \cdot i_2, R) = \max(i_1, R) \cdot \max(i_2, R)$
- $\max(i/c, R) = \max(i, R)/c$
- $\max(i \mod c, R) = c - 1$
- $\max(i) = ub$ where $lb \leq i \leq ub \in R$
- $\max(c, R) = c$ where $c$ is a constant

$\max(i, R)$ returns a rational expression. It can be easily determined if $\max(i, R) \leq l$. The relation $\min(i, R)$ is defined analogously, and similarly the validity of $1 \leq \min(i, R)$ can be determined.

If all constraints in $C(IE)$ can be satisfied then all ground instances of $IE$ are well-defined, i.e., non-ground index equality $IE$ is well-defined.

The following proposition proves that each index equality in the definition of routing function $apmd$ (example 4.2) is well-formed.

**Proposition 4.1** All index equalities in the definition of routing function $apmd$ (example 4.2) are well-formed.

**Proof:**
1. Index equality $IE_1$, which is $y[i] = x[1,i], 1 \leq i \leq m$, is well-formed. Since

$$\text{struct}(s[m+n],[i]) = s = \text{struct}((s[m],s[n]),[1,i])$$

the index equality is well-structured.

$C(IE_1) = \{1 \leq i \leq m+n, 1 \leq i \leq m\}$. Since $R = \{1 \leq i \leq m\}$, $C(IE_1)$ is satisfiable iff the following hold:

$$1 \geq 1, m \leq m+n, 1 \geq 1, m \leq m$$

These conditions can be shown at compile-time to be true for all positive integer values of $m, n$. The validity of the above algebraic inequalities imply that all array index expressions in $IE_1$ satisfy bound constraints.

2. Index equality $IE_2$, which is $y[i] = x[2,i-m], m+1 \leq i \leq m+n$, is well-formed. And since

$$\text{struct}(s[m+n],[i]) = s = \text{struct}((s[m],s[n]),[2,i-m])$$

the index equality is well-structured.

$C(IE_2) = \{1 \leq i \leq m+n, 1 \leq i-m \leq n\}$. Since $R = \{m+1 \leq i \leq m+n\}$, $C(IE_2)$ is satisfiable iff the following hold:

$$1 \leq m+1, m+n \leq m+n, 1 \leq m+1-m, m+n-m \leq n$$

These conditions can be shown at compile-time to be true for all positive integer values of $m, n$. \qed

### 4.5.2 Collision Free

The following is a precise characterization of the collision free property of index equalities in routing functions:

**Proposition 4.2** An index equality $IE \in r(x,y)$ is collision free if $y[ilist_1] = x[ilist_2]$ and $y[ilist_3] = x[ilist_4] \in S_f[IE]$ implies $[ilist_2] = [ilist_4]$.

The syntactic restrictions on the form of index equalities in Section 4.4.5 ensure that all index equalities in routing functions are collision free. Any two distinct ground instances of an index equality are obtained by two distinct assignments to variables that satisfy $R$. Such assignments result in distinct index references to the output $y$.

**Definition 4.8** Index equalities $IE_1, IE_2 \in r(x,y)$ are pairwise collision free if $y[ilist_1] = x[ilist_2] \in S_f[IE_1]$ and $y[ilist_1'] = x[ilist_2'] \in S_f[IE_2]$, where $ilist_2 \neq ilist_4$, implies $[ilist_1] \neq [ilist_1']$. 

46
Let \( r(k, IE) \) denote the range of index \( i_k \) in the output reference in index equality \( IE \) and \( R \) denote the range constraint in \( IE \). \( r(k, IE) \) is defined as follows:

\[
\begin{array}{ll}
r(k, IE) = \{ (lb_k, ub_k) \} & i_k \text{ is a variable and } lb_k \leq i_k \leq ub_k \in R \\
\{ (c, c) \} & i_k = c \text{ where } c \text{ is a constant}
\end{array}
\]

Index equalities \( IE_1, IE_2 \) are pairwise collision free if the ranges assigned by \( IE_1 \) and \( IE_2 \) respectively are disjoint. This requires the existence of an index position \( k \) where indices assigned by \( IE_1 \) differ from those assigned by \( IE_2 \). Formally, index equalities \( IE_1, IE_2 \) are pairwise collision free if:

\[
\exists k, 1 \leq k \leq \min(m, n), r(i, IE_1) \cap r(i, IE_2) = \emptyset,
\]

where \( n \) and \( m \) are the number of index expressions in the output index reference of index equalities \( IE_1 \) and \( IE_2 \), respectively.

We now show that all index equalities in the definition for \( apnd \) are collision free.

**Proposition 4.3** All index equalities in the definition for \( apnd \) (example 4.2) are collision free.

**Proof:**

1. Index equality \( IE_1 \), which is \( y[i] = x[1, i], 1 \leq i \leq m \), is collision free. \( IE_1 \) satisfies the syntactic restrictions described in Section 4.4.5. Hence \( IE_1 \) is collision free. Similarly \( IE_2 \) is collision free.

2. \( IE_1 \) and \( IE_2 \) are pairwise collision free, i.e., the set of output locations assigned by \( IE_1 \) and \( IE_2 \) are disjoint.

Since \( r(i, IE_1) = (1, m) \) and \( r(i, IE_2) = (m + 1, m + n) \), the following algebraic relation must be satisfied:

\[
(1, m) \cap (m + 1, m + n) = \emptyset
\]

This can be shown to be true, at compile-time, for all non-negative integer values of \( m \) and \( n \). \( \square \)

### 4.5.3 Complete Definition

A routing function definition completely defines an output object if the union of the set of indices in the output object assigned by each index equality is equal to the set of indices in the output object.

We need to formalize the set of locations contained in a structure. We use a tree representation of structures, where array and tuple constructors make up internal nodes and structure variables and base types are leaves. Each array node in the tree is associated with its length. Each leaf in this tree corresponds to a set of locations determined by the array nodes lying in the path from the root to the leaf.
The set of indices of an output object defined by an index equality $IE$ is obtained by replacing each array index expression $i_k$ by its range $r(k, IE)$ as shown in the previous section. For the output object to be completely defined, the union of all ranges must be equal to the length of the array at each array node. In addition, all tuple nodes must be assigned.

We demonstrate that the definition for apmd (example 4.2) completely defines the output object. The set of indices assigned by the first index equality $IE_1$ is given by $r(i, IE_1) = (1, m)$ and the set of index equalities assigned by the second index equality $r(i, IE_2)$ is given by $(m + 1, m + n)$. The union of these index sets is $(1, m + n)$ which is the set of indices in the output object.

4.6 Non-Ground Routing Function Definitions

Ground routing function definitions operate on structures with fixed lengths. This requires all arrays to have constant lengths, and hence limits support for polymorphism. Programmers would have to define multiple versions of the same routing function for different input array lengths. In this section we introduce non-ground routing function definitions where expressions in length variables can be used to denote array lengths. Non-ground definitions have the same form as ground definitions except for the use of variables to denote array lengths.

Length expressions in structure mappings $s_{in} \rightarrow s_{out}$ satisfy the following conditions:

- All length variables occurring in $s_{out}$ also occur in $s_{in}$.
- Length expressions in $s_{in}$ may be expressions in at most one length variable.

Length expressions in $s_{out}$ can be in more than one length variable. This ensures that values of length and structure variables in the input structure $s_{in}$ can be determined for the ground structure of an input instance to the routing function. The structure of the output object, for a given input instance, is obtained by substituting variables in $s_{out}$ with these values.

Length variables can appear in index equalities in various forms. They can appear in index expressions of index references as symbolic constants. They can also appear in the set of constraints $R$. Each constraint, as shown earlier in Section 4.1 is of the form $lb \leq i \leq ub$. $lb, ub$ in non-ground definitions can be expressions in length variables. Otherwise non-ground definitions are similar to ground definitions and must satisfy all the restrictions that ground definitions must satisfy.

A ground instance $r'(x, y)$ of non-ground definition $r(x, y)$ is obtained by replacing length variables in $r(x, y)$. However, all ground instances $r'(x, y)$ of $r(x, y)$ may not satisfy correctness conditions presented in Section 4.5. The denotation of a non-ground definition $r(x, y)$ is defined in terms of the denotations of all correct ground instances $r'(x, y)$.
\[ D_R[r(x, y)] = \bigcup D_R[r'(x, y)] \]

where \( r'(x, y) \) is a correct ground instance of \( r(x, y) \).

We generate a set of constraints \( C(r) \) on length variables in \( r(x, y) \) such that any substitution of length variables with values that satisfy \( C(r) \) generates a correct ground instance \( r'(x, y) \) of \( r(x, y) \). These constraints are generated in the same fashion as those for ground definitions. However, numeric ranges are represented symbolically as lower-upper bound pairs. In the case of ground definitions all array lengths are known and the satisfaction of the constraints is completely determined. Correctness requirements of non-ground definitions are analyzed using expressions with symbolic constants as array lengths. Techniques identical to those outlined in Section 4.5 are used to generate constraints on expressions in length variables. At run-time when all input lengths are completely known, the satisfaction of these constraints can be determined. This determination has to be performed for every occurrence of a routing function in an FP* program.

4.6.1 Examples of Routing Function Definitions

**Example 4.3** The definitions of the following primitive routing functions in FP* are given in Figure 4.2.

- \( \text{tail} \) returns the tail of a vector.

- \( \text{distr} \) broadcasts the second element in the input tuple to every element of the array object in the input.

- \( \text{shuffle} \) produces two output arrays by shuffling elements in the input array.

4.6.2 Example: Correctness of Routing Function apnd

The routing function \( \text{apnd} \) was defined in example 4.2.

**Theorem 4.2** The routing function definition for apnd is correct.

**Proof:**

By Proposition 4.1, all index equalities in \( \text{apnd} \) are well-formed. By Proposition 4.3, the definition is collision free, and by Section 4.5.3, the output is completely defined.

Hence the routing function for \( \text{apnd} \) is correct. \( \square \)
4.7 Scalar Routing Function Definitions

The input and output object variables $x$ and $y$ in a non-ground routing function definition $r(x,y)$ have non-ground structures with structure and length variables. For a given input instance, the values of the structure and length variables are completely known. By substituting values for such variables, an instance $r'(x,y)$ of $r(x,y)$ is obtained. Index equalities in $r'(x,y)$ equate objects in the output to objects in the input. These objects are not necessarily scalars but also arrays and tuples. A convenient representation of routing function $r'(x,y)$ is provided by $r''(x,y)$ where all index equalities equate scalars in the output to scalars in the input.

We outline a transformation whereby each index equality $IE' \in r'(x,y)$ is replaced by a collection of index equalities in $IE''_1, \ldots, IE''_n \in r''(x,y)$. An index reference $x[ilist]$ in an index equality provides access paths to a nested object. This path may not lead to scalars, but to arrays and tuples. By adding sufficient indices to $[ilist]$, we can construct paths to scalars. Further, we have to ensure that all the scalars in $x[ilist]$ are covered. This is performed by using variables for array indices whose bounds are determined from array length, and constructing multiple index references which have distinct indices for each tuple object in $x[ilist]$. This transformation is performed to the $lhs$ and $rhs$ index reference in $IE' \in r'(x,y)$ to obtain $IE''_1, \ldots \in r''(x,y)$. 

---

def tail(x : s[n + 1], y : s[n])
    y[i] = x[i + 1], 1 \leq i \leq n
endef

def distr(x : \langle s_1[n], s_2 \rangle, y : \langle s_1, s_2 \rangle[n])
    y[i, 1] = x[1, i], 1 \leq i \leq n
    y[i, 2] = x[2], 1 \leq i \leq n
endef

def shuffle(x : s[2n], y : \langle s[n], s[n] \rangle)
    y[1, i] = x[2i - 1], 1 \leq i \leq n
    y[2, i] = x[2i], 1 \leq i \leq n
endef

Figure 4.2: Examples of Routing Function Definitions
4.8 Routing Functions in Other Programming Languages

Routing functions are not specific to FP*. They also occur in APL [22] and Fortran 90 [41]. Routing functions in APL include Ravel, Reshape, Transpose, reverse, and Outer Product. Fortran 90 also has an extensive set of routing functions including transpose, shift, and spread. The programmer can define new routing functions as array section assignments as follows:

\[ A(n : m) = B(1 : n - m + 1) \]

where \( A, B \) are arrays. In CM-Fortran, which is a dialect of Fortran 90 for the Connection Machine system, forall statements can be used to describe an even more powerful set of routing functions. For example:

\[
\text{forall}(i = 1 : n, j = 1 : m) \ A(i, j) = B(i, i)
\]

The techniques we present for optimizing routing function in FP* can also be used in optimizing routing functions of Fortran 90 and APL as shown in [67].

4.9 Enhancing the Expressiveness of Routing Function Definitions

The syntax for describing the domain of variables in index equalities is restricted to intervals. An interval definition of an index variable \( i_k \) has the form \( lb \leq i_k \leq ub \) where \( lb \) and \( ub \) are expressions in length variables in the structure mapping of the routing function. Thus index variables take a contiguous set of values from an interval. Intervals restrict the kinds of routing functions that can be described. The expressiveness of a routing function definition can be enhanced by allowing equations in index variables to define the domain of indexing variables. However, correctness conditions for such routing function definitions become significantly more complex. The techniques we present for optimizing routing functions do not depend on the fact that domains of index variables are positive integer intervals. They can be applied also for routing functions defined using equations as index variable domains.
Chapter 5
Allocation Functions

5.1 Introduction

The FP* compiler described in this thesis translates FP* programs into imperative intermediate-language (IL) programs. Data objects in IL are scalars and arrays. IL arrays are in general dynamically allocated, since dimension lengths are not known at compile-time. IL is described in more detail in Section 6.2. IL objects are data objects manipulated by IL program statements and are allocated in the memory of the computer. The allocation function of an FP* object, in effect, describes how values in the FP* object are allocated in physical memory. Allocation functions are created and manipulated entirely by the compiler. The compiler uses allocation functions to describe how to access input and output objects of FP* functions that have been allocated on IL objects. Allocation functions also provide a convenient framework for performing various compile-time optimizations including optimizing routing function evaluation, array constant propagation, and optimizations based on improved data layouts. In Chapter 6, where the compiler is described in detail, we will show how the compiler performs these optimizations using allocation functions.

In this chapter we first present the notation used to describe allocation functions. This notation and its meaning is similar to that used for describing routing functions. We next discuss correctness criteria of allocation functions in a manner similar to those for routing function definitions. Examples are then presented that demonstrate the flexibility provided by allocation functions for describing allocation of FP* data objects on IL objects. The compiler makes extensive use of two important operations on allocation functions. The first operation transforms FP* object references to corresponding IL object reference where the value in the FP* object is allocated. A second operation is critical to optimizing routing functions. The compiler uses this operation to generate an allocation function for an output object from a routing function definition and an allocation function for the input object. A preliminary version of these results have been reported in [67, 8].

5.2 Allocation Functions

IL objects provide an abstraction of memory, consisting of a collection of multi-dimensional arrays, scalars, and (compile-time) scalar constants. "Index equalities"
describe how indices in an FP* object are mapped to indices in IL objects. An allocation function consists of (1) a collection of index equalities and (2) the structures of FP* and IL objects that are used in the allocation. In this section we present the notation used for describing allocation functions. Allocation functions satisfy some basic restrictions that are described next. Although allocation functions and routing functions perform entirely different tasks, they have some underlying similarities. The notation used for describing allocation functions is similar to that for describing routing functions. We will describe this relationship, and show how correctness of allocation functions can be described in a manner similar to that for routing functions.

5.2.1 Syntax and Semantics

The allocation function \( A(x) \) for FP* object \( x \) has the form shown in Fig 5.1.

\[
A(x) = \begin{bmatrix}
  x : s_{fp} \\
  m_1 : s_{il_1}, \ldots, m_n : s_{il_n} \\
  \text{IndexEquality}_1 \\
  \vdots \\
  \text{IndexEquality}_n
\end{bmatrix}
\]

Figure 5.1: Allocation Function Definition Schema

\( x \) denotes an FP* object with structure \( s_{fp} \). \( m_1, \ldots, m_n \) denote IL objects with structures \( s_{il_1}, \ldots, s_{il_n} \) respectively. The allocation function describes the allocation of an object \( x \in D[s_{fp}] \) on IL objects \( m_1, \ldots, m_n \) according to the index equalities.

The structures \( s_{fp}, s_{il_1}, \ldots, s_{il_n} \) are non-ground structures without structure variables. Only length variables are allowed in the structures. Further, all IL structures \( s_{il} \) are either array structures or base types, since IL objects are either multi-dimensional arrays or scalars.

The index equalities are of the form \( x[i_1, \ldots, i_n] = m[j_1, \ldots, j_m] \), \( R \) or \( x[i_1, \ldots, i_n] = c, R \). \( x[i_1, \ldots, i_n] \) is an index reference on FP* object \( x \). This index reference must be a scalar index reference, i.e., \( x[i_1, \ldots, i_n] \) denotes a scalar value. \( m[j_1, \ldots, j_m] \) should denote the same scalar type as \( x[i_1, \ldots, i_n] \). Index equalities can also assign (compile-time) constants to FP* objects. Range of index variables are defined in \( R \). Ranges are described by intervals \( lb \leq v \leq ub \), where \( lb, ub \) are length expressions, and by equations in index variables and length expressions.

Ground instances of allocation functions can be described in a manner similar to that of routing function definitions. \( A'(x) \) is a ground instance of \( A(x) \) if \( A'(x) \) is obtained by substituting all length variables in \( A(x) \) with lengths from a particular input instance \( x \). A ground instance of an index equality \( I' \) in \( A'(x) \) is obtained by replacing all index variables in \( I' \) with values that satisfy its range constraints. Such a ground instance \( I'' \) of \( I' \) has the form \( x[\overline{r}] = m_i[j] \) (or \( x[\overline{r}] = c \)), where \( \overline{r} \) and \( j \) are
lists of positive integers. These ground index equalities denote the fact that value \( x[i] \) is allocated in location \( m_i[j] \) (or equals \( c \)).

### 5.2.2 Examples

We present some examples that demonstrate how allocation functions are used by the compiler for allocating FP* objects.

**Example 5.1** The following allocation function allocates a 2-D FP* array onto a 1-D IL array.

\[
A(x) = \begin{bmatrix}
  x : \text{num}[n][m] \\
  m : \text{num}[n \times m] \\
  x[i_1, i_2] = m[(i_2 - 1) \times n + i_1], 1 \leq i_1 \leq n, 1 \leq i_2 \leq m
\end{bmatrix}
\]

**Example 5.2** The following allocation function allocates a 2-dimensional FP* array on a 2-Dimensional IL array in transposed fashion.

\[
A(x) = \begin{bmatrix}
  x : \text{num}[n][m] \\
  m : \text{num}[m][n] \\
  x[i_1, i_2] = m[i_2, i_1], 1 \leq i_1 \leq m, 1 \leq i_2 \leq n
\end{bmatrix}
\]

**Example 5.3** Allocation functions for tuple FP* objects must describe how each element of the tuple is allocated.

\[
A(x) = \begin{bmatrix}
  x : \langle \text{num, bool, num}[n] \rangle \\
  m_1 : \text{num}, m_2 : \text{bool, m}_3 : \text{num}[n] \\
  x[1] = m_1[] \\
  x[2] = m_2[] \\
  x[3, i] = m_3[i], 1 \leq i \leq n
\end{bmatrix}
\]

**Example 5.4** Allocation functions can also be used to allocate array elements in permuted fashion. The following allocation function lays out an FP* array onto an IL array in reverse order.

\[
A(x) = \begin{bmatrix}
  x : \text{num}[n] \\
  m : \text{num}[n] \\
  x[i] = m[n - i + 1], 1 \leq i \leq n
\end{bmatrix}
\]

**Example 5.5** Finally, allocation functions can describe constant-valued array elements. The following allocation function assigns a constant to the first half of the array; the remaining half is allocated to IL array \( m \) shifted by two positions.
\[ A(x) = \begin{bmatrix} x : \text{num}[2n] \\
            m : \text{num}[n + 2] \\
            x[i] = c, 1 \leq i \leq n \\
            x[i] = m[i - n + 2], n + 1 \leq i \leq 2n \end{bmatrix} \]

5.2.3 Syntactic Restrictions on Index Equalities in Allocation Functions

Allocation functions for FP* objects are created and manipulated by the compiler. All allocation functions satisfy the following restrictions:

- Every index expression in left-hand side index references in index equalities is either a distinct variable or a constant. More general expressions are not created, and no variable occurs twice in an index reference to the FP* object in the allocation function.

- All tuple index references are positive integer constants in all index references.

5.2.4 Correctness of Allocation Functions

The correctness of an allocation function is defined in a manner similar to that for routing function definitions. An allocation function \( A(x) \) is correct if the following conditions hold:

- Each index equality in \( A(x) \) is well-typed.

- Bounds constraints are satisfied by all array index expressions in \( A(x) \).

- Each value in an FP* object is allocated to exactly one IL element. This is analogous to the collision free requirement of routing functions.

- All values in the FP* object are allocated in locations on IL objects.

These conditions correspond to correctness conditions for routing function definitions. An index equality is well-typed if it is well-structured and the structure of index references, as defined in Chapter 4, is a scalar type.

We prove later that all allocation functions created by the compiler are correct in the above sense if definitions of all routing functions in the program satisfy correctness criteria for routing functions.
5.3 Allocation Function Evaluation

When the compiler generates IL statements for evaluating a function $f$ applied to an input $x$, it needs to transform all references to elements in $x$ to corresponding IL object references as defined in allocation function $A(x)$. In this section we present the technique used by the compiler for generating IL object references given an allocation function $A(x)$ and an index reference $x[i\text{\_ilist}]$ to an FP* object $x$.

Array indices in $[i\text{\_ilist}]$ can be (IL program) variables or constants, whereas tuple indices are constants. The procedure returns a set of scalar IL index references, one of which contains the FP* object denoted by $x[i\text{\_ilist}]$. The actual IL index reference containing the value $x[i\text{\_ilist}]$ depends on actual values of variables appearing in $x[i\text{\_ilist}]$ and values of length variables of FP* and IL arrays.

Algorithm 5.1 (Allocation Function Evaluation) $A(x): x[i\text{\_ilist}]$

*Input*. Allocation function $A(x)$, and a scalar index reference $x[i_1, \ldots, i_n]$.

*Output*. A set of pairs of IL index references and range constraints $I_{iI} = \{(m_i[i\text{\_ilist}_i], R_i), \ldots\}$. For any given valuation for variables in $i\text{\_ilist}$, and length variables in $A(x)$, exactly one $R_i$ will hold true, and it is the case that $m_i[i\text{\_ilist}_i]$ is equal to the value $x[i\text{\_ilist}]$.

*Method.* The algorithm applies the following sub-procedure to each index equality $I$ in $A(x)$. The sub-procedure determines if the set of indices of an FP* object allocated by $I$ contains the index $i\text{\_ilist}$. It fails if it can determine that the set does not contain $i\text{\_ilist}$, else it succeeds and returns an IL index reference and a range constraint to be added into $I_{iI}$.

Perform the following steps on each index equality $x[i_1, \ldots, i_m] = m_i[i\text{\_ilist}_2], R_i$ in $A(x)$.

1. If $m \neq n$ then fail.

2. Set up equations $i_k = j_k$ for all $1 \leq k \leq n$. Determine if they are inconsistent. This is achieved by splitting the set of indices $i_k, j_k$'s into equivalence classes, and determining if there are distinct constants in the same class. If consistent, associate each equivalence class with a unique label, where the label is either a member that is a constant (if one exists), or is some $i_k$.

3. If the equation set is consistent, then replace all variables $j_k$ in $m_i[i\text{\_ilist}_2], R_i$ with their equivalence class label to obtain $m_i[i\text{\_ilist}_2'], R_i'$ and return the pair. If the equation set is inconsistent then fail.

Example 5.6 Let

\[
A(x) = \begin{bmatrix}
x : \{\text{num}[n], \text{bool}[m]\}, \\
m_1 : \text{num}[n], m_2 : \text{bool}[2m] \\
x[i, i] = m_1[i], 1 \leq i \leq n \\
x[2, i] = m_2[i + m], 1 \leq i \leq m
\end{bmatrix}
\]
\[ A(x) : x[2, 3] = \{m_2[3 + m], 1 \leq 3 \leq m\} \]
\[ A(x) : x[2, j] = \{m_2[j + m], 1 \leq j \leq m\} \]

For any given \( x \), \( A'(x) \) is an instance of \( A(x) \) obtained by replacing all length variables with actual lengths from object \( x \). The substitution also gives an instance \( I'_d = \{ (m_1[\text{list}'_1], R'_1), (m_2[\text{list}'_2], R'_2), \ldots \} \) of \( I_d = A(x) : x[\text{list}] \). An instance \( x[\text{list}'] \) of \( x[\text{list}] \) is obtained by replacing index variables in \( [\text{list}] \) with values. We prove correctness of the above algorithm for any correct instance \( A'(x) \) of \( A(x) \) and any valid ground instance \( x[\text{list}'] \) of \( x[\text{list}] \). Validity of \( x[\text{list}'] \) implies that all indices in \( [\text{list}] \) satisfy bound constraints.

**Theorem 5.1** For any instance of length and index variables such that \( A'(x) \) is correct and \( x[\text{list}'] \neq \bot \), and where \( I'_d = A'(x) : x[\text{list}'] = \{m_1[\text{list}'_1], R'_1, \ldots, m_n[\text{list}'_n], R'_n\} \), the following holds:

Exactly one \( R'_i \) in \( I'_d \) is satisfiable and \( m_i[\text{list}'_i] \) is the IL index reference containing the value \( x[\text{list}] \).

**Proof**

A satisfiable \( R'_i \) exists. Since \( A'(x) \) is correct and \( x[\text{list}'] \) is a valid index, it must be the case that some index equality \( x[\text{list}_1] = m_i[\text{list}_2], Ra_i \) in \( A'(x) \) allocates the index \( x[\text{list}'] \), implying that \( Ra_i \) is true. Then there exists a ground instance \( x[\text{list}_1'] \) of \( x[\text{list}_1] \) such that \( [\text{list}_1'] = [\text{list}'] \) and \( Ra_i \) is satisfiable for \( [\text{list}_1] \). Hence \( Ra_i \) is satisfiable which implies \( R'_i \) is satisfiable.

Exactly one constraint \( R'_i \) is satisfiable. If more than one satisfiable \( R'_i \) exists then it must be the case that more than one index equality in \( A'(x) \) corresponding to each satisfiable \( R'_i \) allocates \( x[\text{list}'] \). This cannot be the case since \( A'(x) \) is correct. \( \square \)

### 5.4 Composing Routing Function Definitions with Allocation Functions

The routing function definition for \( r : x = y \), denoted as \( r(x, y) \), describes the mapping of indices of \( y \), the output object, to indices in \( x \), the input object. The allocation function \( A(x) \) for \( x \) describes how each index reference to \( x \) is allocated on some IL object. The compiler composes \( r(x, y) \) and \( A(x) \) to obtain the allocation function \( A(y) \) for \( y \).

In this section we present a procedure to construct \( A(y) \) using \( r(x, y) \) and \( A(x) \), and we prove correctness of \( A(y) \). We assume that index equalities in a routing function definition refer to only scalar elements in the input and output. We have
shown in Section 4.7 how to transform index equalities in routing function definitions to equivalent index equalities that only refer to scalar elements in input and output. The following procedure composes the routing function definition \( r(x, y) \) with the allocation function \( A(x) \) to obtain \( A(y) \).

**Algorithm 5.2** \( A(y) = r(x, y) \circ A(x) \)

*Input.* Definition \( r(x, y) \) of routing function \( r: x = y \) and allocation function \( A(x) \).

*Output.* Allocation function \( A(y) \).

*Method.* The procedure applies the following steps to each pair of index equalities \( Ir, Ix \) where \( Ir \in r(x, y), Ix \in A(x) \). Each pair contributes an index equality \( Iy \) to \( A(y) \) if it can be determined that the set of index references to \( x \) in \( Ir \) has a non-empty intersection with the set of indices of \( x \) allocated by \( Ix \). If it can be determined that the intersection is empty then no index equality is added into \( A(y) \).

Perform the following on each pair of index equalities \( Ir \) and \( Ix \) where \( Ir \) is \( y[\text{rlist}] = x[i_1, \ldots, i_n], Rr \) and \( Ix \) is \( x[j_1, \ldots, j_m] = m[\text{alist}], Ra \).

1. If \( n \neq m \) then empty intersection.

2. Set up equations \( i_k = j_k \) for all \( 1 \leq k \leq n \). The \( i_k \) are algebraic expressions in general. If there is any equation \( i_k = c_1 \neq c_2 = j_k \) for distinct constants \( c_1, c_2 \) then the equations are inconsistent and intersection is empty. Otherwise, a non-empty intersection may exist; proceed to step 3.

3. Classify equations as \( A \)-substitution equations, \( r \)-substitution equations and constraint-equations as shown. Equations where \( j_k \) is a variable are \( A \)-substitution equations. Of the remaining equations, the set in which \( i_k \) is a variable are \( r \)-substitution equations. The remaining equations are of the form \( expr = c \) where \( expr \) is neither a variable or a constant and \( c \) is a constant. The set of such constraint-equations is denoted by \( C \).

These substitution and constraint equations are applied on index equalities \( Ir \) and \( Ix \) to generate index equality \( Iy \) as follows:

- The \( A \)-substitutions are applied to variables in \( m[\text{alist}], Ra \) to obtain \( m[\text{alist'}], Ra' \). This corresponds to replacing variables in \( Ix \) with expressions from \( Ir \).

- The \( r \)-substitutions are applied to variables in \( y[\text{rlist}], Rr \) to obtain \( y[\text{alist'}], Rr' \). This corresponds to replacing variables in \( Ir \) with constants occurring in \( Ix \).

The index equality \( Iy \) is \( y[\text{rlist}] = m[\text{alist'}], Rr' \cup Ra' \cup C \).
Example 5.7 We determine the allocation function for $y$ where \textit{reverse} : $x = y$, given $A(x)$.

$A(x)$ allocates FP* array $x$ on two IL arrays $m_1, m_2$, as follows:

$$A(x) = \begin{bmatrix}
    x = \text{num}[n+m] \\
    m_1 : \text{num}[n], m_2 : \text{num}[m] \\
    x[i] = m_1[i], 1 \leq i \leq n \\
    x[i] = m_2[i-n], n+1 \leq i \leq n+m
\end{bmatrix}$$

The definition of routing function \textit{reverse} is:

```python
def reverse(x : num[n+m], y : num[n+m])
    x[j] = y[n+m - j + 1], 1 \leq i \leq n+m
endef
```

Then

$$A(y) = \begin{bmatrix}
    y = \text{num}[n+m] \\
    m_1 : \text{num}[n], m_2 : \text{num}[n+m] \\
    y[j] = m_1[n+m - j + 1], m+1 \leq j \leq n+m \\
    y[j] = m_2[m - j + 1], 1 \leq j \leq m
\end{bmatrix}$$

In the remaining portion of this section we prove correctness of the allocation function $A(y) = r(x, y) \circ A(x)$. There are two parts to proving $A(y)$ correct. Firstly, we must show that $A(y)$ satisfies all requirements of an allocation function for all valid input instances $x$ to $r$. These requirements have been enumerated in Section 5.2.4. Secondly, we must show that $A(y)$ is in fact the composition of $r(x, y)$ and $A(x)$, i.e., a value in the output $y$ of routing function $r(x, y)$ is allocated on some IL object $z$ (as described in $A(y)$) iff the same value in the input $x$ to $r(x, y)$ is allocated in $z$ (as described by $A(x)$).

Consider any index equality $I_y$ in $A(y)$ of the form $y[rlist1'] = m_i[jalist2'], Rr' \cup Ra' \cup C$. This index equality must be generated from index equality $Ir$ of the form $y[rlist1] = x[rlist2], Rr$ in $r(x, y)$ and $Ix$ of the form $x[jalist1] = m[jalist2], Ra$ in $A(x)$.

Theorem 5.2 For any instance of length variables for which $r(x, y)$ and $A(x)$ are correct, the allocation function $A(y) = r(x, y) \circ A(x)$ derived by algorithm 5.2 is correct according to criteria presented in Section 5.2.4.

Proof We show that $A(y)$ satisfies all 4 conditions enumerated in Section 5.2.4.

All Index equalities in $A(y)$ are well structured. The consistency of equations $i_k = j_k$ where $rlist2 = [i_1, \ldots, i_n]$ and $jalist1 = [j_1, \ldots, j_n]$ implies that both index references have the same set of tuple indices. Therefore $\text{struct}(s_x, rlist2) = \text{struct}(s_x, jalist1)$ where relation struct is defined in Section 4.4.4, and $s_x$ is structure of $x$. Since the structure of the right-hand side index reference in $Ix$ is the same as the left-hand side index reference in $Ir$, and $Ir$ and $Ix$ are well-structured, it follows that $I_y$ is well-structured.
All index expressions in $A(y)$ satisfy bound constraints. All index expressions in $r(x, y)$ satisfy bound constraints since $r(x, y)$ is correct. Each ground index instance $y[\text{alist1}]$ of $y[\text{alist1}]$ is constrained by both $Rr$ and $Ra$ in $Iy$. By a similar argument involving correctness of $Ia$ we can show that all indices in ground instance $[\text{alist2}]$ of $[\text{alist2}]$ satisfy bound constraints.

All index equalities in $A(y)$ are collision free. It is obvious that $Iy$ is collision free if $Ir$ and $Ia$ are collision free. We also have to show that all pairs of index equalities $A(y)$ are collision free. Let ground instances of index equalities $Iy_j, Iy_k$ allocate the same index in $y$. The algorithm generates $Iy_j$ from index equalities $Ir_j, Ia_j$, and $Iy_k$ from $Ir_k, Ia_k$. If $Ia_j \neq Ia_k$, then it must be the case that ground instances of $Ir_j, Ir_k$ assign the same index in $y$ with distinct values from $x$ implying that $r(x, y)$ is incorrect. Similarly, if $Ir_j \neq Ir_k$ then $A(x)$ is incorrect. Since $A(x)$ and $r(x, y)$ are assumed correct it follows that $Ir_j = Ir_k$ and $Ia_j = Ia_k$, and hence $Iy_j = Iy_k$.

All values in $y$ are allocated on IL objects by $A(y)$. Let some index in $A(y)$ be unallocated by $A(y)$. It must be the case that $r(x, y)$ does not assign values into that index of $y$ or $A(x)$ does not allocate the index of $y$ that is copied into $y$ by $r(x, y)$. In either case one (or both) of $r(x, y)$ and $A(x)$ is incorrect. □

A valid input $x$ to $r$ defines a ground instance $r'(x, y), A'(x)$ and $A'(y)$ by substituting all length variables in $r(x, y), A(x)$, and $A(y) = r(x, y) \circ A(x)$.

Theorem 5.3 $y[\text{alist1}] = x[\text{alist2}]$ and $x[\text{alist2}] = m[\text{alist3}]$ are ground instances of $r'(x, y), A'(x)$ respectively iff $y[\text{alist1}] = m[\text{alist3}]$ is a ground instance of $A'(y)$.

Proof

($\Rightarrow$)

Let $y[\text{alist1}] = x[\text{alist2}]$ and $x[\text{alist2}] = m[\text{alist3}]$ be ground instances of index equalities $Ir', Ia'$ in $r'(x, y)$ and $A'(x)$ respectively. Then there are values of variables for which the index references to $x$ are identical in $Ir'$ and $Ia'$ such that the range constraints in $Ir'$ and $Ia'$ are satisfied. Hence the procedure for composition (algorithm 5.2) generates an index whose ground instance is $y[\text{alist1}] = m[\text{alist3}]$ of $Iy'$.

($\Leftarrow$)

Let $y[\text{alist1}] = m[\text{alist3}]$ be a ground instance of index equality $Iy'$ in $A'(y)$. There exist index equalities $Ir'$ and $Ix'$ in $r'(x, y)$ and $A'(x)$, respectively, such that $Iy'$ is generated by algorithm 5.2. There exists a ground instance $y[\text{alist1}] = x[\text{alist2}]$ of $Ir'$, Further, there also must be an instance $x[\text{alist2}] = m[\text{alist3}]$ of index equality $Ix'$. The index equality $Ir'$ must assign into $y[\text{alist1}]$ some value from $x$, and $Ix'$ allocates the same value in $x$ to some IL object. □
Chapter 6
The Code Generation Function

6.1 Introduction

The compilation process consists of two phases: (1) structure inference and (2) code generation. Structure inference, presented in Chapter 3, determines if the structure constraints of the function \( f \) are satisfiable, and derives the most general structure mapping of \( f \). Structure inference also produces an annotated representation of \( f \) where each component function has been annotated with its inferred input and output structure. In addition, it also emits a set of linear equations in length variables that occur in the derived structures.

The code generation phase of the FP* compiler has been formalized as a simple syntactic function \( C \) which is defined inductively, based on the form of the functions to be compiled. When \( C \), the code generation function, is applied to an FP* function \( f \) annotated by structure mappings, it returns an IL program that evaluates \( f \). Parallelism in IL programs is specified using forall loops where all iterations are executed in parallel. \( C \) is defined separately for each primitive function in FP*. The definition of \( C \) for user-defined routing functions follows a simple schema. Schemas are presented for each of the higher-order functions in FP* that define \( C \) recursively in terms of each of the constituent functions. The code generation phase performs a variety of optimizations including routing function evaluation and array constant propagation, and generates efficient data layouts. By formalizing the code generation process as a syntactic function, we have been able to build a rapid prototype implementation in Prolog. We provide examples of IL code generated by \( C \) for the FP* functions. A preliminary version of the code generation function appeared in [64]; a refined version appeared in [8].

6.2 Target Language: IL

The FP* compiler described in this thesis translates FP* programs into imperative intermediate-language (IL) programs. Data objects in IL are scalars and arrays. IL arrays are in general dynamically allocated, since dimension lengths are not known at compile-time. The only construct in IL suitable for describing data parallelism is the forall statement, which has the form shown below:
$$\text{forall}(i_1 = 1 : n_1 : s_1, \ldots, i_m = 1 : n_m : s_m)$$

{ assignment statement }

The $i_k$'s are subscript variables ranging over integer values $1, 1 + s_k, \ldots, 1 + l s_k$ where $l s_k \leq n_k < (l + 1) s_k$. $s_k$ is called the stride. A forall statement is similar to a do loop except that all iterations of the forall can be executed simultaneously. The operational semantics of the forall statement in IL are identical to the forall statement in CM Fortran[61]. In addition, IL has a sequential do loop, a conditional if then else statement, and a sequential while loop. In fact, IL represents a very small subset of CM Fortran [61], and all IL programs can be translated easily into CM Fortran programs.

Example 6.1 (Matrix Multiply) The IL program for the Matrix multiply program presented in Section 2.5 is displayed in Example 6.5.

6.3 Code Generation Function $C$

The code generation function $C$ has the following inputs and outputs:

$$C[f : s_{in} \rightarrow \epsilon, s_{out}](\bar{i}, M, A_{in}) = (P, A_{out})$$

- $f$ is a function annotated with the structures of input, output and intermediate objects. Structure inference is used to determine the most general structure Mapping $f : s_{in} \rightarrow \epsilon, s_{out}$. The consistency of $\epsilon$ for given input array lengths is determined at run-time as discussed in the next section.

- $\bar{i}$ and $M$ together describe the parallel environment in which $f$ is evaluated.

  - $\bar{i}$ is a list of loop index variables with their ranges, and has the form $[i_1 = 1 : n_1 : s_1, \ldots, i_m = 1 : n_m : s_m]$. $\bar{i}$ describes an $m$-dimensional iteration space in which index variable $i_k$ ranges from 1 to $n_k$ with stride $s_k$.

  - $M$ describes a logical mask on the iteration space described by $\bar{i}$. $M$ is a boolean expression on index variables that occur in $\bar{i}$. These boolean arrays are created by evaluating guard expressions in enclosing conditional functionals. $x_1, \ldots, x_m$ is an active point in the iteration space if $M$ evaluates to true for these values of $i_1, \ldots, i_m$. $f$ is evaluated at each active point in the iteration space described by $\bar{i}$ and $M$.

- $(A_{in})$ The code generation function refers to scalar values in the input of structure $s_{in}$ to $f$. Since $f$ is evaluated at each active point in the iteration space defined by $\bar{i}$, scalar index references are made at each active point. The input allocation function $A_{in}$ describes allocation of the active portion of the iteration space to IL arrays. However, $A_{in}$ is not completely defined for all valid indices.
in the $m$-dimensional object of structure $s_{in}$, thereby violating one of the correctness properties discussed in Chapter 5. $A_{in}$ allocates only those indices that are evaluated to true by $M$. We relax the complete definition criteria to one of conditional completeness, where all indices for which $M$ evaluates to true are allocated on IL objects. In addition, we ensure that only those indices that satisfy mask $M$ in FP* objects are referenced. Hence, allocation evaluation operations will be well-defined.

- $P$ is an IL program emitted by the compilation function, which evaluates $f$ in an environment defined by $\vec{i}$ and $M$ under input allocation function $A_{in}$.

- $A_{out}$ describes the allocation of the active portion of the output object of which is an $m$-dimensional object of structure $s_{out}$.

We define a small set of relations on $\vec{i}$.

**Definition 6.1** $\text{ind}([i_1 = d_1, \ldots, i_m = d_m]) = [i_1, \ldots, i_m]$

$\text{ind}$ returns the sequence of index variables in $\vec{i}$. These will be used in IL array references in the definition of $C$.

**Definition 6.2** $\text{ub}([i_1 = 1 : n_1 ; s_1, \ldots, i_m = 1 : n_m ; s_m]) = [n_1, \ldots, n_m]$

$\text{ub}$ returns the sizes of iteration space dimensions which are used for creating array data structures of the same shape.

**Definition 6.3** $s[\text{ub}([i_1, \ldots, i_n])]$ is an $n$-dimensional structure each element of which is of structure $s$, and the $k$th dimension has size $\text{ub}(i_k)$.

$s[n_1, \ldots, n_m]$ denotes an array whose dimension sizes are given by the $n_k$'s and whose base type is $s$, a scalar type.

We describe how the code generation function is invoked for compiling FP* programs. Since every FP* function $f$ is non-recursive, all user-defined functions, excluding user-defined routing functions, can be replaced by their definitions. The input to the compiler is an FP* expression $f$ consisting only of primitive functions, user-defined routing functions, and higher-order functions, annotated with structures inferred through structure inference. Initially, $\vec{i} = []$ and $M = \text{true}$. Further, the code generation function requires an allocation function $A_{in}$ for the input that describes how the input to $f$ is allocated in memory.
6.4 Determining Consistency of $\epsilon$ for Given Inputs

The structure inference system produces a most general structure mapping of well-formed FP* functions. The structure mapping of function $f$ is denoted as $f : s_{in} \rightarrow \epsilon s_{out}$. $\epsilon$ is a set of equations in length variables occurring in derived structures. These equations impose constraints on values of length variables occurring in $s_{in}$. At runtime, when all input array lengths are known, the satisfiability of the set of equations can be determined by the result of Theorem 3.3. If $\epsilon$ is satisfiable for these values then the IL program is executed; otherwise, a size-error is signalled and program execution terminates.

6.5 Primitive Functions

In this section we describe $C$ for primitive functions in FP*. We first present $C$ for some of the primitive computation functions. Next, we present $C$ for constant functions, where the compiler is able to perform constant propagation across array assignments. Lastly we present schemas for primitive and user-defined routing functions. No code is generated while compiling constant functions, and routing functions. The compiler is able to capture their effect in the output allocation functions produced by $C$.

6.5.1 Computational Functions

The code generation for all primitive computational functions is similar to the definition presented for addition ($+$) below.

$$C[+ : \langle num, num \rangle \rightarrow num](\bar{r}, M, A_{in}) =$$

$$\left(\forall l, (\bar{r}, M) S, \right.$$  

$$A_{out}\right)$$

where $A_{out} = A(y) =$

$$\begin{bmatrix}
  y : \langle num, num \rangle [ub(\bar{r})] \\
  m : num [ub(\bar{r})] \\
  y[ind(\bar{r})] = m[ind(\bar{r})]
\end{bmatrix}$$

$m$ is a new array name of type $num[ub(\bar{r})]$.

Statement $S$ denotes a collection of IL assignment statements as shown below. Let $A_{in} : [ind(\bar{r}), 1] = \{(m_{1,1}, R_{1,1}), \ldots, (m_{1,n}, R_{1,n})\}$ and

$A_{in} : [ind(\bar{r}), 2] = \{(m_{2,1}, R_{2,1}), \ldots, (m_{2,l}, R_{2,l})\}$. Then

$$S = \{\text{if } R_{1,i} \land R_{2,j} \text{ then } m[ind(\bar{r})] = m_{1,j} + m_{2,k} \mid 1 \leq j \leq n, 1 \leq k \leq l\}$$

The emitted code consists of a set of conditional assignments, one for each $i,j$ pair.
Example 6.2

\[
A_{in} = A(x) = \begin{cases} 
  & x : (\text{num, num})[100] \\
  & m_1 : \text{num}[50], m_2 : \text{num}[50], m_3 : \text{num}[100] \\
  & x[i, 1] = m_1[i], 1 \leq i \leq 50 \\
  & x[i, 1] = m_2[i], 50 \leq i \leq 100 \\
  & x[i, 2] = m_3[i] 
\end{cases}
\]

\[A[i] = A_{in} : [i_1, 1] + A_{in} : [i_1, 2]\] denotes the following sequence of IL assignment statements:

if \(1 \leq i_1 \leq 50\) then \(m_4[i_1] = m_1[i_1] + m_3[i_1]\)

if \(51 \leq i_1 \leq 100\) then \(m_4[i_1] = m_2[i_1] + m_3[i_1]\)

The allocation function \(A_{out}\) for the output \(y\) is:

\[
A_{out} = A(y) = \begin{cases} 
  & y : \text{int}[100] \\
  & m_4 : \text{int}[100] \\
  & y[i] = m_4[i] 
\end{cases}
\]

6.5.2 Constant Functions

A class of primitive computation functions generate only constants as output, independent of input values. The code generation function does not allocate space for the constant outputs. Instead it keeps track of constants stored in an output FP* object through its allocation function. This optimization is similar to the constant propagation optimization for scalar assignment in sequential languages [1]. However, this technique performs constant propagation across array assignment statements also.

\[C[c : s \rightarrow \text{num}](\bar{i}, M, A_{in}) = (\text{nop}, A_{out})\]

where \(A_{out} = A(y) = \begin{cases} 
  & y : \text{num}[ub(\bar{i})] \\
  & y[ind(\bar{i})] = c 
\end{cases}\)

\(\text{nop}\) denotes an empty operation. The compiler does not generate any code for constant functions.

6.5.3 Routing Functions

\[C[r : s_{in} \rightarrow s_{out}](\bar{i}, M, A_{in}) = \begin{cases} 
  & \text{nop} \\
  & E(r, \bar{i}) \circ A_{in} 
\end{cases}\]

where \(E\) is a new routing function defined below.

\(E(r(x, y), \bar{i}) = r(x', y') = \)
\[
\begin{align*}
\begin{bmatrix}
x' & : & s_{in}[ub(i)] \\
y' & : & s_{out}[ub(i)] \\
\{ y'[ind(i), ilist_1] = x'[ind(i), ilist_2], R \mid y[ilist_1] = x[ilist_2], R \in r(x, y) \}
\end{bmatrix}
\end{align*}
\]

where \( x' \) and \( y' \) are new variable names.

\( r \) is a routing function evaluated in the iteration space defined by \((\vec{i}, M)\). Its effect is to copy a subset of values from the input object of structure \( s_{in}[ub(\vec{i})] \) to the output object of structure \( s_{out}[ub(\vec{i})] \). \( E(r(x, y), \vec{i}) \) is a routing function definition produced from \( r \), which applies \( r \) at each point of the iteration space. Correctness of \( E(r(x, y), \vec{i}) \) follows from the correctness of \( r(x, y) \).

The compiler does not allocate space for the output and emit copy statements to copy elements of the input to elements of the output. Instead, it uses the IL storage allocated to the input FP* object to describe the output FP* object. The compiler first constructs a new routing function definition for environment \((\vec{i}, M)\). It then derives an allocation function for the output object by composing the constructed routing function definition with the allocation function for the input object.

### 6.6 Higher-Order Functions

There exists a recursive definition for \( C \) of each higher-order function in FP* in terms of constituent functions.

#### 6.6.1 Apply-All

\[
C[\alpha f : s_1[n] \rightarrow s_2[n]](\vec{i}, M, A_{in}) = (P, A_{out})
\]

where

\[
C[f : s_1 \rightarrow s_2](\vec{i}, j = 1 : n : 1], M, A_{in}) = (P, A_{out})
\]

The code generation function for \( \alpha f \) is described in terms of the code generation function for \( f \). \( \alpha f \) is evaluated at every point of the iteration space described by \( \vec{i} \) where \( M \) is true. This is identical to evaluating \( f \) at every point of the iteration space obtained by extending \( \vec{i} \) by \( j \), where \( M \) is true.

If \( f \) is a routing function then no code is generated for evaluating \( \alpha f \). The compiler uses the storage used for allocating the input to allocate the output object, and derives the output allocation function.

#### 6.6.2 Composition

\[
C[f \circ g : s_{in} \rightarrow s_{out}](\vec{i}, M, A_{in}) = (P_g; P_f, A_{out})
\]

where

\[
C[g : s_{in} \rightarrow s_{tmp}](\vec{i}, M, A_{in}) = (P_g, A_{tmp})
\]
\( \mathcal{C}[f : s_{\text{tmp}} \rightarrow s_{\text{out}}](\overline{i}, M, A_{\text{tmp}}) = (P_f, A_{\text{out}}) \)

\( P_g; P_f \) denotes sequencing of IL program fragments \( P_g \) and \( P_f \). The transmission of values from \( g \) to \( f \) is achieved by using the output allocation function of \( g \) as the input allocation function for \( f \).

If \( f \) and \( g \) are routing functions then no code is emitted for \( f \circ g \). The compiler derives \( A_{\text{tmp}} \) by composing the routing function definition of \( g \) with \( A_{\text{in}} \), and then derives \( A_{\text{out}} \) by composing the definition for \( f \) with \( A_{\text{tmp}} \).

### 6.6.3 Construction

\( \mathcal{C}[(f_1, f_2, \ldots, f_n) : s_{\text{in}} \rightarrow (s_1, \ldots, s_n)](\overline{i}, M, A_{\text{in}}) = (P_1; P_2; \ldots; P_n, A_{\text{out}}) \)

where

\( \mathcal{C}[f_k : s_{\text{in}} \rightarrow s_k](\overline{i}, M, A_{\text{in}}) = (P_k, A_{k_{\text{out}}}) \)

In order to describe \( A_{\text{out}} \) we need the following definitions. \( IL(A) \) is the set of IL objects and their structures that are referenced in allocation function \( A \). \( IE(A) \) is the set of non-ground index equalities in \( A \).

\[
A_{\text{out}} = A(y) = \begin{cases} y : (s_1, \ldots, s_n)[ub(\overline{i})] \\ \bigcup_{k=1}^n IL(A_{k_{\text{out}}}) \\ \bigcup_{k=1}^n T(IE(A_{k_{\text{out}}}), \overline{i}, k) \end{cases}
\]

where \( T \) is a new function defined below:

\[
T(IE(A), \overline{i}, k) = \{ x[ind(\overline{i}), j, ilist1] = m[ilist2], R \mid x[ind(\overline{i}), ilist1] = m[ilist2], R \in IE(A) \}
\]

The output object allocation \( A_{\text{out}} \) of \((f_1, \ldots, f_n)\) is derived from the output object allocation \( A_{k_{\text{out}}} \) of the \( f_k \)'s. The output object structure is \((s_1, \ldots, s_n)[ub(\overline{i})]\). The set of IL objects on which it is allocated is the collection of the IL objects used to allocate outputs from each of the \( f_k \)'s.

The set of index equalities in \( A_{\text{out}} \) is obtained by transforming index equalities in the \( A_{k_{\text{out}}} \)'s. Each index \([\text{ind}(\overline{i}), j]\) in the FP* object of structure \( s_k[ub(\overline{i})] \) corresponds to index \([\text{ind}(\overline{i}), k, j]\) in the object of structure \((s_1, \ldots, s_n)[ub(\overline{i})]\). This transformation of index equalities in \( A_{k_{\text{out}}} \)'s is formally described by \( T \).

Code is emitted only for those \( f_k \)'s that are not routing functions. If the \( f_i \)'s are routing functions then no code is emitted for evaluating \((f_1, \ldots, f_n)\).

Evaluation of the Construction higher-order function has implications for control parallelism, since each of the \( f_i \)'s can be potentially evaluated in parallel. This can be achieved by evaluating all the \( P_i \)'s in parallel. The single thread of control forks into \( n \) independent threads which can be executed in distinct processors. After each thread finishes execution, they must synchronize and a single thread of control is regenerated. Since our focus is entirely on data-parallelism, only a single thread of
control is allowed to exist at all times. This is achieved by sequentially executing the code blocks.

### 6.6.4 Flatten

The definitions for Conditional, While and Insert higher-order functions require use of a flatten operation. flatten transforms an output allocation function $A_{out}$ into an allocation function $A_{id}$ where the index equalities are all identities. flatten achieves this transformation by allocating new IL objects, and copying elements from IL objects referred to by $A_{out}$ to those in $A_{id}$.

We first describe the form of the identity allocation function $A_{id}$. We next show how IL copy statements are emitted that coerce any correct allocation function $A$ of an FP* object into $A_{id}$. We also describe the IL statements generated by flatten($A_f, A_{id}, \tau, M$) so that an object allocated by $A_f$ in environment $(\tau, M)$ is allocated by $A_{id}$ in the same environment.

**Identity Allocation Function $A_{id}$**

$A_{id}$ for an FP* object $x$ of structure $s$ consists of a collection of index equalities and IL objects that are used for allocating $x$. $I(x, s)$, defined below, denotes the set of index equalities that allocate an FP* object $x$ of structure $s$.

- $I(x, \text{num}) = \{x[] = m[]\}$ where $m$ is a new IL variable name. Similarly for $I(x, \text{bool})$.
- $I(x, s[n]) = \{x[i, ilist] = m[i, ilist], (R, 1 \leq i \leq n) \mid x[ilist] = m[ilist], R \in I(x, s)\}$
- $I(x, (s_1, \ldots, s_n)) = \bigcup_{i=1}^{n} I(x, s_i)$

**Example 6.3**

$$A_{id} = A(x) = \left[\begin{array}{l} x: \text{num}[m][n], \text{num}[p][m] \\ m_1: \text{num}[m][n], m_2: \text{num}[p][m] \\ I(x, (\text{num}[m][n], \text{num}[p][m])) \end{array}\right]$$

where

$$I(x, (\text{num}[m][n], \text{num}[p][m])) = \left[\begin{array}{l} x[1, i, j] = m_1[i, j], 1 \leq i \leq n, 1 \leq j \leq m \\ x[2, i, j] = m_2[i, j], 1 \leq i \leq m, 1 \leq j \leq p \end{array}\right]$$

There is a unique identity allocation function for an FP* object of structure $s$, modulo renaming of IL objects. Using induction on $s$, it can be shown that the identity allocation function that allocates an FP* objects of structure $s$ is correct.
Copy Statements Emitted By flatten

Values of IL objects in allocation function $A$ are to be copied into IL objects in $A_{id}$ using IL statements. Pairs of index equalities $(I_j, I_k)$ where $I_j \in A_{id}$ and $I_k \in A$ are analyzed to determine if they allocate a common set of indices in an FP* object. If so, a statement is emitted that copies from IL references in $I_k$ to IL references in $I_j$.

The following procedure is used to analyze a pair of index equalities $I_j, I_k$ where $I_j$ is $x [I_{list2}] = m_j[I_{list2}^j], R_j$ and $I_k$ is $x[A_{list2}] = m_k[A_{list2}^k], R_k$.

Let $I_{list1} = [i_1, \ldots, i_m]$ and $[A_{list1}] = [j_1, \ldots, j_n]$.

If $n \neq m$ then fail.

Set up equations $i_k = j_k$ for $1 \leq k \leq n$. If there are equations of the form $c_1 = c_2$ where $c_1, c_2$ are distinct constants then fail.

Classify equation $i_k = j_k$ as an r-substitution if $j_k$ is a variable; else if $i_k$ is a variable classify it as l-substitution. Apply the r-substitutions on $m_k[A_{list2}^k], R_k$ and l-substitutions on $m_j[I_{list2}^j], R_j$ giving $m_k[\tilde{A}_{list2}^k], \tilde{R}_k$ and $m_j[I_{list2}^j], \tilde{R}_j$ respectively. The following copy statement is generated

\[ \text{if } R'_k \land R'_j \text{ then } m_j[I_{list2}^j] = m_k[\tilde{A}_{list2}^k] \]

A forall statement wraps the copy statement if there are index variables in $[I_{list2}^j]$, and $R'_k \land R'_j$ is the mask descriptor in the forall statement.

When flatten is called in environment $(\tilde{\tau}, M)$, the output object is defined only for those indices in $\tilde{\tau}$ that satisfy $M$. Therefore values of index variables in $[I_{list2}^j]$ must satisfy $M$. The forall statement that wraps the copy statement adds $M$ to its mask so that the assignment is evaluated only for instances of index variables in $I_{list2}^j$ that satisfy $M$.

$flatten(A_f, A_{id}, \tilde{\tau}, M)$ denotes the set of IL copy statements that copy objects allocated by $A_f$ in environment $\tilde{\tau}, M$ to objects allocated by $A_{id}$ in the same environment.

6.6.5 Conditional

There are two definitions of $C$ for the conditional higher-order function: one for evaluating the conditional in a scalar environment, i.e. $\tilde{\tau} = []$; another for evaluating the conditional in a parallel environment, i.e. $\tilde{\tau} \neq []$. Both definitions require the use of flatten which coerces an arbitrary allocation function into the identity allocation function.

flatten is required in compiling $f \rightarrow g; h$ because $g$ and $h$ are compiled independently, returning separate output allocation functions $A_g$ and $A_h$. Since the compiler returns a single allocation function for the output object produced by $f \rightarrow g; h$, flatten is required to copy $A_g$ and $A_h$ respectively onto an identity allocation function $A_f$. $A_f$ is returned as the output allocation function for $f \rightarrow g; h$. 

69
When \( f, g \) and \( h \) are routing functions, the only IL statements emitted while compiling \( f \circ g ; h \) are due to \textit{flatten}.

### Scalar Conditional \( \overline{\tau} = \[] \)

\[
\begin{align*}
\mathcal{C}[f \rightarrow g; h : s_{in} \rightarrow s_{out}](\[], true, A_{in}) &= \\
&\begin{cases} 
P_g; \\
\text{if } b \text{ then} \\
\text{flatten}(A_g, A_I, [], true); \\
\text{else} \\
P_h; \\
\text{flatten}(A_h, A_I, [], true)
\end{cases} \\
&\text{endif, } \\
&\text{A_I}
\end{align*}
\]

where

\[
\begin{align*}
\mathcal{C}[f : s_{in} \rightarrow \text{bool}](\[], true, A_{in}) &= (P_f, A(x)), \\
b &= A(x) : x[], \\
\mathcal{C}[g : s_{in} \rightarrow s_{out}](\[], true, A_{in}) &= (P_g, A_g), \text{ and} \\
\mathcal{C}[h : s_{in} \rightarrow s_{out}](\[], true, A_{in}) &= (P_h, A_h).
\end{align*}
\]

### 6.6.6 Vector Conditional \( \overline{\tau} \neq \[] \)

\[
\begin{align*}
\mathcal{C}[f \rightarrow g; h : s_{in} \rightarrow s_{out}](\overline{\tau}, M, A_{in}) &= \\
&\begin{cases} 
(1) P_f; \\
(2) P_g; \\
(3) \text{flatten}(A_g, A_I, \overline{\tau}, M \land b(\text{ind}(\overline{\tau}))); \\
(4) P_h; \\
(5) \text{flatten}(A_h, A_I, \overline{\tau}, M \land \lnot b(\text{ind}(\overline{\tau})))
\end{cases} \\
&\text{A_I}
\end{align*}
\]

where

\[
\begin{align*}
\mathcal{C}[f : s_{in} \rightarrow \text{bool}](\overline{\tau}, M, A_{in}) &= (P_f, A(x)), \\
b(\text{ind}(\overline{\tau})) &= A(x) : x[\text{ind}(\overline{\tau})], \\
\mathcal{C}[g : s_{in} \rightarrow s_{out}](\overline{\tau}, M \land b(\text{ind}(\overline{\tau})), A_{in}) &= (P_g, A_g), \text{ and} \\
\mathcal{C}[h : s_{in} \rightarrow s_{out}](\overline{\tau}, M \land \lnot b(\text{ind}(\overline{\tau})), A_{in}) &= (P_h, A_h).
\end{align*}
\]

The boolean value returned by the evaluation of \( f \) is combined with the current mask \( M \) to generate an environment for compiling \( g \) and \( h \) respectively.
6.6.7 Insert(1)

/f is evaluated through a binary tree of height \( \lfloor \lg n \rfloor \) where \( n \) is the length of the input array. Each level of the tree is evaluated sequentially through a for loop. However, all applications of \( f \) at any given level of the tree are evaluated in parallel.

\( flatten \) in line (1) of the compiled code copies the input array into a temporary array under an identity allocation function \( A_I \). The new array is destructively modified at each iteration. \( flatten \) in line (4) copies the output of each iteration back onto the temporary array. Parallel application of \( f \) at each node in a tree level is achieved by evaluating \( f \) in an environment augmented by index variable \( j = 1 : n : 2^k \), so that nodes at level \( k \) of the tree are separated by distance \( 2^k \) in the temporary array. The inputs at an active site \( j \) are obtained from sites \( j \) and \( j + 2^{k-1} \) of the previous iteration.

\[
C[[f : s[n] \to s]](\vec{r}, M, A_{in}) = \\
(1) flatten(A_{in}, A_I, \vec{r}, M); \\
(2) do k = 1, \lfloor \lg n \rfloor \\
(3) \quad C_{j}; \\
(4) flatten(A_{f_out}, A_I, [\vec{r}, j = 1 : n : 2^k], M); \\
(5) \text{enddo,} \\
A_{out})
\]

where

\[
C[[f : \langle s, s \rangle \to s]]([\vec{r}, j = 1 : n : 2^k], M, A_{f_{in}}) = (P_f, A_{f_{out}})
\]

\( A_{f_{in}} \), obtained from the identity allocation function \( A_I \), is the allocation function for \( f \) in environment \([\vec{r}, j = 1 : n : 2^k], M \). \( A_{f_{in}} \) is computed by having the first object in the input tuple to \( f \) evaluated at array location \( j \), and the second object at location \( j + 2^{k-1} \).

Formally, \( A_{f_{in}} = A(y) = \)

\[
\left\{ y[ind(\vec{r}), j, 1, ilist] = m[ind(\vec{r}), j, ilist] \mid \right. \\
\left. y[ind(\vec{r}), j, ilist] = m[ind(\vec{r}), j, ilist] \in A_I \right\} \\
\bigcup \\
\left\{ y[ind(\vec{r}), j, 2, ilist] = m[ind(\vec{r}), j + 2^{k-1}, ilist] \mid \\
\left. y[ind(\vec{r}), j, ilist] = m[ind(\vec{r}), j, ilist] \in A_I \right\}
\]

6.6.8 While

We present two definitions for the code generation function for the While functional. The first definition derives the code generation function in a scalar environment where \( \vec{r} = [] \). The second definition derives the code generation function in a parallel environment where \( \vec{r} \neq [] \).
flattened appears in the derived code in both definitions. flattened ensures that the input to the body $f$ of the While loop at the beginning of the first iteration is the same as the input at the beginning of the remaining iterations. Calls to flattened’s that appear at lines (1) and (5) of the derived copy the input value to a temporary, and copy the output of the loop back into the temporary, respectively.

**Scalar While $\vec{r} = []$**

\[
C[\text{While } e \text{ do } f : s \rightarrow s][[], \text{true}, A_{in}) = \\
((1) \quad \text{flattened}(A_{in}, A_I, [], \text{true}) \\
(2) \quad P_e \\
(3) \quad \text{while } (b) \\
(4) \quad P_f \\
(5) \quad \text{flattened}(A_f, A_I, [], \text{true}) \\
(6) \quad P_e \\
(7) \quad \text{endwhile}, \\
A_I)
\]

where

\[
C[e : s \rightarrow \text{bool}][[], \text{true}, A_I] = (P_e, A(x)), \\
b = A(x) : x[], \text{ and} \\
C[f : s \rightarrow s][[], \text{true}, A_I] = (P_f, A_f).
\]

**Vector While $\vec{r} \neq []$**

\[
C[\text{While } e \text{ do } f : s \rightarrow s][\vec{r}, M, A_{in}) = \\
((1) \quad \text{flattened}(A_{in}, A_I, \vec{r}, M \wedge b(\text{ind}(\vec{r}))) \\
(2) \quad P_e \\
(3) \quad \text{while } (\text{ANY}(b)) \\
(4) \quad P_f \\
(5) \quad \text{flattened}(A_f, A_I, \vec{r}, M \wedge b(\text{ind}(\vec{r}))) \\
(6) \quad P_e \\
(7) \quad \text{endwhile}, \\
A_I)
\]

where

\[
C[e : s \rightarrow \text{bool}][\vec{r}, M, A_I] = (P_e, A(x)), \\
b(\text{ind}(\vec{r})) = A(x) : x[\text{ind}(\vec{r})], \text{ and} \\
C[f : s \rightarrow s][\vec{r}, M \wedge b(\text{ind}(\vec{r})), A_I] = (P_f, A_f).
\]
ANY is an array operator that returns \textit{true} if any of the elements at active points of array \( b \) are \textit{true}. ANY can be described entirely using Insert and Apply-All functionals. The While loop executes as long as \( e \) evaluates to \textit{true} in an active site in the environment described by \( t, M \).

### 6.7 nop FP* functions

An FP* function \( f \) is a \textbf{nop} function if the code generation function \( C \) derived for \( f \) is of the form:

\[
C[f : s_{in} \rightarrow s_{out}][[], true, A_{in}] = (\text{nop}, A_{out})
\]

i.e., no IL statements are generated for evaluating \( f \).

\textbf{Theorem 6.1} FP* functions constructed using only the following are \textbf{nop} functions.

- \textit{Primitive constant functions}
- \textit{Primitive and user-defined routing functions.}
- \textit{Composition, Construction, and Apply-All higher-order functions.}

\textbf{Proof} By induction on the height \( h \) of the parse tree for \( f \).

\textit{Base Case.} \( h = 1 \). \( f \) must be a constant function or a primitive routing function or a user-defined routing function. In all cases the definition of \( C \) generates only \textbf{nops}'s as IL statements.

\textit{Inductive Case.} Consider a parse tree of height \( n > 1 \). It must be the case that the root is either the Composition, Construction or Apply-All higher-order functions. The code generation function derived using the definitions for the higher-order functions generates only \textbf{nop} statements if all the constituent functions are \textbf{nop} functions which holds from the inductive hypothesis. \( \square \)

### 6.8 Code Generation for Insert(2)(/2)

In this section we describe techniques required to efficiently compile the Insert(2) functional. The compiler is able to differentiate the two distinct forms of the Insert higher-order function, Insert(1) and Insert(2), during structure inference as described in section 3.8. Code generation for Insert(1) has been described in section 6.6.7. We first discuss problems with using the code generation function of Insert(1) for generating code for Insert(2). These problems include excessive storage overhead and load imbalance. We next present techniques and define the code generation function for Insert(2) which overcomes these problems.
6.8.1 Inadequacy of $\mathcal{C}$ for Insert(1)(/1)

The code generation function for $/1f$ assumes static storage requirements, i.e. the storage requirements across iterations are constant. Iteration $i$ corresponds to parallel evaluation of $f$ at each node at level $i$ of the reduction tree. The output of $f$ can be copied back into the input array since $f$ has structure mapping $\langle s, s \rangle \rightarrow s$. This does not hold true for $/2f$ where $f$ has structure mapping $\langle s[n], s[n] \rangle \rightarrow s[2n]$ and $/2f : s[c][n] \rightarrow s[cn]$. Since the output of $f$ is twice the size of either input, copying can not be performed. An obvious solution is to pre-allocate sufficient storage at the beginning.

There are two problems associated with this approach.

- Large storage overhead because an array of size $cn^2$ is required when the input is of size $cn$.

- Poor load balance because at no time are more than $cn$ locations of the $n \times cn$ array are active. A large fraction of the processors which contain unused elements are inactive in every iteration.

6.8.2 An Efficient Compilation Scheme for Insert(2)

In this section we present a compilation scheme that uses only a 1-dimensional array of size $n$ and statically achieves load balancing. This compilation scheme exploits the fact that only $cn$ locations of the 2-dimensional array of size $cn \times n$ contain active data. Instead of allocating a 2-dimensional array, the compiler maps it onto a 1-dimensional IL array. This is possible by making use of allocation functions to describe the mapping. In addition, the iteration space vector $\vec{r}$ is enhanced so that the compiler can keep track of the fact that a 2-dimensional iteration space due to the program has been mapped onto a 1-dimensional IL iteration space. The compiler also makes use of a simple set of index transformations to accomplish this mapping of iteration spaces. The code generation function for Insert(2) is shown below.

Insert(2) (/2)

\[
\begin{align*}
\mathcal{C}[/f, s[c][n] & \rightarrow s[cn]](A_{in}, \vec{r}, M) = \\
&(1) \text{ flatten}(A_{in}, A_f(0), \vec{r}, M) \\
&(2) \text{ do } j = 1, [\lg n] \\
&(3) \quad \mathcal{C}_f \\
&(4) \quad \text{ flatten}(A_{out}, A_f(j), \vec{r}, M) \\
&(5) \quad \text{ enddo,} \\
&\quad A_{out}
\end{align*}
\]

where
\[ C[f : \langle s[c2^{j-1}], s[c2^{j-1}] \rangle \rightarrow s[c2^j]](A_{fin}(j), [\overline{i}, i_1 = 1 : n/2^j, (i_2 = 1 : c2^{j-1})], M) = (C_f, A_{fout}) \]

\( A_f(j) \) is the allocation function of the input array that is being reduced. \( A_f(j) \) describes the allocation of the input at the beginning of iteration \( j + 1 \). The 2-dimensional input object of structure \( s[c2^j][n/2^j] \) is allocated on a 1-dimensional IL array of structure \( s[nc] \). The set of index equalities in \( A_f(j) = A(x)(j) \) is given as follows:

\[ A(x)(j) = \{ x[\overline{i}, i_1, i_2, \overline{v}] = y_i[\overline{v}, (i_1 - 1)c2^j + i_2, \overline{v}'] \mid x[\overline{v}] = y_i[\overline{v}'] \in I(x, s) \} \]

\( A_{fin} \) describes the allocation of the input object to \( f \) in environment \( ([\overline{i}, i_1 = 1 : n/2^j, (i_2 = 1 : c2^{j-1})], M) \). In the evaluation of binary function \( f \) at each site \( i_1 \) during iteration \( j \), the two operands are at adjacent locations of the IL array.

\[ A_{fin}(j) = A_f(x)(j) = \]

\[
\left\{ x[\overline{i}, i_1, 1, i_2, \overline{v}] = y_i[\overline{v}, (i_1 - 1)c2^j + i_2, \overline{v}'] \mid x[\overline{v}] = y_i[\overline{v}'] \in I(x, s) \right\} \cup \left\{ x[\overline{i}, i_1, 2, i_2, \overline{v}] = y_i[\overline{v}, (i_1 - 1)c2^j + c2^{j-1} + i_2, \overline{v}'] \mid x[\overline{v}] = y_i[\overline{v}'] \in I(x, s) \right\}
\]

The iteration space vector \( [\overline{i}, i_1 = 1 : n/2^j, (i_2 = 1 : c2^{j-1})] \) keeps track of the fact that a 2-dimensional iteration space has been mapped onto a single dimension of the IL iteration space. Therefore, generated for all loops will have a single index variable \( i = 1 : n \) instead of index variables \( i_1 = 1 : n/2^j, i_2 = 1 : c2^{j-1} \). All references to \( i_1, i_2 \) in \( M \) and IL object references are transformed into references to \( i \) as follows. Index \( i_1 \) is mapped to index \((i - 1) \mod 2^j + 1 \) and index \( i_2 \) is mapped to \((i - 1)/2^j + 1 \). This index transformation is automatically performed by the compiler.

### 6.9 Correctness of Allocation Functions

All allocation functions generated by the compiler satisfy criteria for correctness, ensuring that their evaluation and composition with routing functions is well-defined. Allocation functions are generated during the following steps:

1. Compiling primitive computation functions.
2. Composing allocation functions with routing function definitions.
3. flatten.

The allocation function created for the output object of a primitive computation function is obviously correct. flatten generates identity allocation functions which can be shown to be correct based on their simple form. Finally the allocation functions generated for the output object of a routing function are correct if the allocation functions for the input object is correct and the routing function definition is correct.

75
It is straightforward to construct an inductive proof that all allocation functions generated while compiling an FP* function $f$ is correct if the allocation function for the input to $f$ is correct.

6.10 Storage Allocation

Structures of FP* objects are inferred by structure inference. The code generator requires that all inferred structures contain no structure variables, only length variables. However, structures generated by structure inference may not satisfy this requirement. In that case, compilation fails. The programmer must provide type declarations in the program to remove structure variables. Hence only array lengths are unknown at compile-time. Arrays are allocated dynamically at run-time when their lengths are completely known. Optimizing storage by re-use and destructive update for functional languages has been extensively studied. In [7] an inference based approach is presented for efficient allocation and deallocation of aggregate data-structures. An approach based on abstract interpretation for determining when arrays can be safely destructively updated has been presented in [26, 14]. These ideas can be incorporated in the FP* compiler to minimize storage overhead of run-time programs.

6.11 Additional Optimizations

`flatten` in the code generation functions for Insert, While and Conditional requires evaluation of copy statements resulting in additional storage overhead, and may possibly cause inter-processor communication. It is possible to eliminate `flatten's in some cases. In the case of Insert, if we can determine that the input is allocated by an identity allocation function, and the values in it are not used by any other function in the program, the `flatten in line (1) can be omitted. The same analysis can be used to eliminate the `flatten in line (2) of the IL program for the While functional. Elimination of `flatten's for the Conditional higher-order function requires more careful analysis. Potentially, both flatten's can be eliminated. The output allocation function is constructed by incorporating the boolean value in the range constraints of index equalities in allocation functions produced by both branches, and taking their union. However, this results in allocation functions with a large number of index equalities, possibly affecting performance by sequentializing computation.

Another optimization minimizes masks in `forall statements that are Boolean functions on index variables. If $i_k = 1 : n_k : s_k$ and if there exists conditions in mask $M$ of the form $1 \leq i_k \leq n_k$, the condition may be eliminated. It is often the case that constraints in $M$ are due to such redundant range conditions. Our compiler eliminates these constraints, thus enhancing performance by minimizing run-time mask computation.
6.12 Additional Remarks

6.12.1 Separate Compilation

Separate compilation, where functions are compiled separately and the resulting object files are linked together, is useful for minimizing the complexity of developing large software systems. When a function that is compiled separately is called from another file, its input and output structures must be known in the calling file. This is required in the FP* compiler for structure inference of the function that makes the call. In addition, code generated for the called function is used for evaluating it in an environment defined by $(\mathcal{I}, M)$ and the input allocation function $A_{in}$. It must be the case that the function is called in an identical environment. The input allocation function, however, can be different. The compiler must flatten the input allocation function (supplied by the calling function) into the one with which the (called) FP* function has been compiled for. The procedure for performing the coercion is similar to the one for coercing arbitrary allocation function to an identity allocation function as described in Section 6.6.4.

6.12.2 Complexity of the Compiler

The compiler consists of two phases: (1) structure inference and (2) optimized code generation. The structure inference phase requires a traversal of the parse tree where each node is visited exactly once. In addition, a visit to a node requires a constant number of unifications which has been shown to be in linear time. However, structure inference also requires determining whether a set of linear equations have any positive integer solutions, which requires exponential time in the worst case. However, such equations are generated rarely in practice. Code generation also requires a traversal of the parse tree with each node being visited exactly once. A constant number of operations on allocation functions are performed at each node. The complexity of these operations depend linearly on the number of index equalities in the allocation function.

6.13 Examples

We present examples of IL code generated by the code generation function described in this chapter.

Example 6.4 3D_mat_add (program 2.1)

\[
C[3D\text{\_mat\_add}: (\text{num}[p][m][n], \text{num}[p][m][n]) \rightarrow \text{num}[p][m][n]](\text{[]}, \text{true}, A(x)) = (P, A(y))
\]
where
\[ A(x) = \begin{bmatrix}
  x : (\text{num}[p][m][n], \text{num}[p][m][n]) \\
  m_1 : \text{num}[p][m][n], m_2 : \text{num}[p][m][n] \\
  x[1, i, j, k] = m_1[i, j, k], 1 \leq i \leq n, 1 \leq j \leq m, 1 \leq k \leq p \\
  x[2, i, j, k] = m_2[i, j, k], 1 \leq i \leq n, 1 \leq j \leq m, 1 \leq k \leq p
\end{bmatrix} \]

and \( P = \)
\[
\text{forall } (i = 1 : n, j = 1 : m, k = 1 : p) \\
 m_3[i, j, k] = m_1[i, j, k] + m_2[i, j, k]
\]

and
\[ A(y) = \begin{bmatrix}
  y : \text{num}[p][m][n] \\
  m_3 : \text{num}[p][m][n] \\
  y[i, j, k] = m_3[i, j, k], 1 \leq i \leq n, 1 \leq j \leq m, 1 \leq k \leq p
\end{bmatrix} \]

Example 6.5 Matrix Multiply (program 2.3)

\[ C[\text{matmult} : (\text{num}[m][n], \text{num}[p][m]) \rightarrow \text{num}[p][n]]([], \text{true}, A(x)) = (P, A(y)) \]

\[ A(x) = \begin{bmatrix}
  x : (\text{num}[m][n], \text{num}[p][m]) \\
  m_1 : \text{num}[m][n], m_2 : \text{num}[p][m] \\
  x[1, i, j] = m_1[i, j], 1 \leq i \leq n, 1 \leq j \leq m \\
  x[2, i, j] = m_2[i, j], 1 \leq i \leq m, 1 \leq j \leq p
\end{bmatrix} \]

and \( P = \)
\[
\text{forall}(i = 1 : n, j = 1 : p, k = 1 : m) \\
 m_3[i, j, k] = m_1[i, k] \ast m_2[k, j]
\]
do \( x = 1, \lceil \log m \rceil \)
\[
\text{forall}(i = 1 : n, j = 1 : p, k = 1 : m : 2^x) \\
 m_4[i, j, k] = m_3[i, j, k] + m_3[i, j, k + 2^{x-1}]
\]
\[
\text{forall}(i = 1 : n, j = 1 : p, k = 1 : m : 2^x) \\
 m_3[i, j, k] = m_4[i, j, k]
\]
enddo

and
\[ A(y) = \begin{bmatrix}
  y : \text{num}[p][n] \\
  m_3 : \text{num}[m][p][n] \\
  y[i, j] = x[i, j, 1], 1 \leq i \leq n, 1 \leq j \leq p
\end{bmatrix} \]
Chapter 7
Implementation and Performance on the CM-2

7.1 Introduction

The code generation function $C$ described in Chapter 6 emits IL programs which are very similar to CM Fortran programs. Beyond minor syntactic differences due to column alignment requirements, CM Fortran mainly differs from IL in requiring explicit data declarations. We describe modifications to IL programs to produce CM Fortran subroutines. CM Fortran has a fixed set of array intrinsics that have efficient low-level implementations. To enhance performance, we have extended IL to include these array intrinsics and introduced modifications to the code generation function of Chapter 6 to generate these IL array intrinsics. In the absence of those modifications, the code generation function emits equivalent forall statements. We will describe the modified code generation function for the Insert functional which emits these array intrinsics.

We present timings on the CM-2 for some of the programs described in Chapter 2 in this thesis. These timings are presented for compiled programs with and without the optimizations described in Chapter 6. The effect of routing function optimization on run-time and storage overhead is shown to be significant. We have also presented an optimizing compilation scheme for Insert(2) that provides significant improvements in run-time and storage overhead compared to the naive version. A preliminary version of these results have been reported in [65, 66].

Timings of compiled programs (with optimizations) are compared with library routines provided by Thinking Machines Corporation for the CM-2. These routines are highly optimized hand-coded implementations and demonstrate higher performance than compiled FP* programs. We have identified data layout inefficiencies resulting in non-local communication as an important cause of the performance differences.

7.2 The Connection Machine CM-2 System

The Connection Machine system [24, 60] supports the data-parallel programming style by providing thousands of hardware processors that can operate on as many data elements simultaneously. A full Connection Machine CM-2 includes 64K 1-bit physical processors, each with its own memory of up to 64Kbits, organized in groups
of 16. Each group of 16 processors forms a node in the 12 dimensional hypercube that serves as the main interprocessor communication resource. There is a floating point processor for groups of 32 1-bit processors. The CM-2 is organized into sections consisting of 8K 1-bit processors having 256 floating point processors. The Connection Machine is accessed through a front end that provides the programming environment. The front end holds scalar data, and also controls execution of the data parallel program. Each program instruction is broadcast for simultaneous execution by all processors. Most instructions are executed conditionally, i.e., each processor has a context flag whose state determines if the processor will execute the instruction.

Inter-processor communication on the CM-2 is implemented by a high-speed routing network. The router executes send instructions where each processor contains data and a pointer to a processor that is to receive the data. The CM-2 singles out certain patterns of communication for special hardware and microcode support. Cartesian grids of any number of dimensions can be embedded within the boolean hypercube structure used by the router. Array elements that are neighbors along any dimension are allocated on processors that are neighbors within the hypercube structure. As a result, a single-position shift along any axis of such a grid can be performed much faster than the general case of the send instruction. Such communication operations are called NEWS operations.

The following figures have been presented in [34] as a rough measure of communication cost on the CM-2.

If a 32-bit addition (to be performed by all 64K processors) takes 1 time unit, then

- a 32-bit move within processor memory takes 1 time unit.
- a 32-bit NEWS transfer takes 4 units.
- a general 32-bit SEND takes 80 units.

One unit is approximately 16µsec.

CM Fortran consists of Fortran 77 supplemented with array-processing features proposed in the Fortran 90 standard. Fortran 90 features included in CM Fortran are array assignments involving entire arrays or array sections and intrinsic functions for array reduction, permutation and transformation. In addition, CM Fortran provides a forall statement for performing elemental array assignment. It is used to specify array assignments in terms of individual elements or sections. A forall statement specifies some number of subscript names and values over which they range in terms of lower and upper bounds and strides, as \( i = 1 : N : 2 \); it also may optionally specify a mask expression. The body of a forall statement must be a single assignment statement whose left-hand side specifies an array element or array section. The semantics of a forall statement is assignment to each of these elements or sections (one for every possible combination of subscript values for which the mask expression is true)
with all right-hand sides being evaluated before any left-hand sides are assigned. The semantics of CM Fortran forall statements is identical to that of forall statements in IL.

CM Fortran arrays are either allocated on the back-end or the front-end. When the CM Fortran compiler allocates an array on the CM, it does so in a canonical layout. In addition, CM Fortran provides compiler directives which enable the programmer to describe array layouts and communication patterns. For example, the compiler directive cmf$ layout A(:news, :news) allocates the 2-d array A on the back-end in a grid configuration. Many virtual processors may be multiplexed onto a single physical processing node by system software. Hence array elements assigned to distinct virtual processors may reside on the same physical processor. Operations on back-end arrays using forall statements, array intrinsics, and array assignment statements are evaluated in parallel.

When compiling a CM Fortran program, the programmer chooses Paris (field-wise) or slicewise[61, 50] execution models. Programs compiled for the Paris model use the bit-serial processors as the basic physical processing units whereas those compiled for the slicewise model use the floating point processors as the basic physical units. The slicewise compiler generates significantly faster code than the fieldwise compiler as it views the hardware at a much lower level of detail where the floating point units, the memory hierarchy and the registers are directly exposed[50]. The timings presented in this thesis are obtained by compiling generated CM Fortran programs with the slicewise compiler.

In addition to forall loops, CM Fortran provides another method of describing parallel array operations which makes use of array assignment statements that involve entire arrays and/or array sections[61]. For example, forall(i = 1 : n : 2) a(i) = b(i) can alternatively be described in CM Fortran as a(1 : n : 2) = b(1 : n : 2). The array section a(1 : n : 2) denotes the following sequence of indices: 1, 3, ..., 2l + 1, where 2l + 3 > n, in array a. Array assignment statements are of the form var = expr, where var is an array section, and expr evaluates into an array of the same shape, i.e., same rank and identical extent along each dimension. forall statements are more expressive than array assignment statements. We provide some examples which demonstrate the superior expressiveness of forall statements.

forall statements can be used to describe higher-dimensional broadcasts which cannot be expressed using array assignment statements. For example, forall(i = 1 : n, j = 1 : m) a(i, j) = b(i) cannot be described using array assignment statements. Similarly, the statement forall(i = 1 : n, j = 1 : m) a(i, j) = b(i + j) cannot be expressed using array assignment statements. forall statements with masks can be written as array assignment statements using the where construct. For example, forall(i = 1 : n, a) b(i) = c(i) can be written as where (a) b = c. However, in array assignment statements, the mask expression must be of the same shape as var and expr. Such a restriction does not apply to masks in forall statements. For example, forall(i = 1 : n, j = 1 : m, k = 1 : p, A(j, k)) b(i, j, k) = c(i, j, k) cannot be
described using where and an array assignment statement.

It is this powerful expressiveness of forall statements that has led us to choose it as the main data-parallel construct in CM Fortran programs generated by the FP* compiler. The FP* compiler makes use of the fact that masks in forall statements may be defined on a subset of the subscripts defined in the forall statement. This is useful for compiling the Conditional higher-order function. The guard of the condition is evaluated into a boolean object which is then used as the mask expression in forall statements emitted by the compiler for evaluating the true and false branches. This property of forall statements is also made use of in compiling the While and Insert higher-order functions. In addition, the FP* compiler also makes use of the more general form of array assignments that can be described using forall statements. This is because rhs operands in array assignments that are obtained by evaluating allocation functions, as shown in example 6.2, have linear expressions in forall subscript variables as array indices and distinct array indices may share subscript variables. Such forms of array assignments cannot, in general, be described using array sections.

However, the CM Fortran compiler currently implements array assignment statements more efficiently than forall statements. Techniques for efficiently implementing forall statements on data-parallel computers have been described in [3]. The timings for parallel array operations using forall statements and array assignment statements, which are described in Table 7.1, are given in Table 7.2. Timings were taken with vector length \( n = 8192 \) on an 8K section of a CM-2.

<table>
<thead>
<tr>
<th>Case</th>
<th>forall Statement</th>
<th>Array Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>forall((i = 1 : n)) (a(i) = b(i) + c(i))</td>
<td>(a(1 : n) = b(1 : n) + c(1 : n))</td>
</tr>
<tr>
<td>(2)</td>
<td>forall((i = 1 : n : 2)) (a(i) = b(i) + c(i))</td>
<td>(a(1 : n : 2) = b(1 : n : 2) + c(1 : n : 2))</td>
</tr>
<tr>
<td>(3)</td>
<td>forall((i = 1 : n : 2, m((i + 1)/2))) (a(i) = b((i + 1)/2))</td>
<td>(a(1 : n : 2) = b(1 : n/2))</td>
</tr>
</tbody>
</table>

Table 7.1: Parallel array operations described using forall and equivalent array assignment statements

<table>
<thead>
<tr>
<th>Case</th>
<th>(A) forall</th>
<th>(B) Array Assignment</th>
<th>Ratio ((A) \div (B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>8.5E-05</td>
<td>8.52857E-05</td>
<td>.996</td>
</tr>
<tr>
<td>(2)</td>
<td>0.00053</td>
<td>0.00038</td>
<td>1.37</td>
</tr>
<tr>
<td>(3)</td>
<td>0.0059</td>
<td>0.0021</td>
<td>2.79</td>
</tr>
</tbody>
</table>

Table 7.2: Timings (in secs) of forall statements versus array assignment statements
7.3 Generating CM Fortran Programs

In this section we describe enhancements to IL whereby a small set of array operations are introduced into IL. These operations perform array reductions. We also describe modifications to the code generation function \( C \) presented in Chapter 6 which allow it to emit these array intrinsics. IL consists of a small subset of Fortran 77 and has \texttt{forall} statements as the only parallel construct. The Fortran 77 subset of IL has sequential assignment statements, \texttt{do} loops, \texttt{while} loops and \texttt{if} statements. These IL statements have the same syntax as corresponding CM Fortran statements.

CM Fortran has a wide variety of array intrinsics [61] that are part of the Fortran 90 standard. An important category of intrinsic functions are array reduction functions, which summarize information in an array by applying some combining operation over its values. Each of these reduction functions can be applied to the whole array or to a single dimension in it. The following reduction function are provided in CM Fortran: \texttt{maxval}, \texttt{minval}, \texttt{sum}, \texttt{any}, \texttt{all}, \texttt{count}.

Array reduction can also be evaluated using \texttt{forall} statements within a sequential \texttt{do} loop based on recursive doubling [25]. For example, the following program fragment sums up all elements in array \( a \) of type \texttt{num[n]} as follows:

\[
\begin{align*}
\text{do } & k = 1, \lfloor \log n \rfloor \\
& \text{forall}(i = 1 : n : 2^k) \\
& \quad a(i) = a(i) + a(i + 2^{k-1}) \\
\text{end do}
\end{align*}
\]

The same effect is achieved by \texttt{sum(a)}. However, since array intrinsics have highly-optimized low-level implementation, they execute much faster than the equivalent CM Fortran code. This is demonstrated in the following timings in Table 7.3 which were taken on an 8K section of a CM-2.

<table>
<thead>
<tr>
<th>Array Size</th>
<th>\begin{tabular}{c} \texttt{do+} \end{tabular}</th>
<th>\begin{tabular}{c} \texttt{sum} \end{tabular}</th>
<th>\begin{tabular}{c} Ratio \end{tabular}</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>0.0164</td>
<td>0.00037</td>
<td>44.3</td>
</tr>
<tr>
<td>16384</td>
<td>0.0300</td>
<td>0.00038</td>
<td>78.9</td>
</tr>
<tr>
<td>32768</td>
<td>0.0542</td>
<td>0.00039</td>
<td>138.9</td>
</tr>
<tr>
<td>65536</td>
<td>0.1082</td>
<td>0.00041</td>
<td>263.9</td>
</tr>
</tbody>
</table>

Table 7.3: Timings (in Seconds) for sum-reduction using \texttt{forall} statements and the \texttt{sum} array intrinsic.

These array reduction intrinsics are introduced into IL along with modifications to the code generation function for the \texttt{Insert} functional to enable it to emit reduction intrinsics instead of \texttt{forall} statements. A new definition for \( C \) for \(/f\) is added to the existing definitions, where \( f \) is one of +, *, or, \texttt{and}, \texttt{max} and \texttt{min}.

CM Fortran declarations for data objects were not emitted by the code generation function. The only data objects manipulated by emitted IL programs are the
following: (1) input to program, (2) objects that store results of primitive computation functions, and (3) objects created during flatten operations. The types of all these objects are known from structure inference. Each object is declared with its type in the beginning of the CM Fortran program. The dimension list in IL array structures are reversed for generating CM Fortran array declarations since the outermost IL array dimension is mapped to the first CM Fortran array dimension and so on for the remaining dimensions. Thus, IL array \( a \) with structure num\( [m][n][p] \) has the CM Fortran declaration real, array(p,n,m): a. In addition, the code generation function emits cmfs$ layout directives that allocate all arrays that are manipulated in parallel by forall statements or by array intrinsics on the back-end of the CM-2.

### 7.4 Performance

In this section we present the performance enhancements achieved due to the optimizations performed by the FP* compiler. The effects of three distinct optimizations are presented. The first optimization is to eliminate run-time evaluation of routing functions. The second optimization takes advantage of efficient implementations of array reduction intrinsics on the Connection Machine CM-2. The last optimization is useful for divide-and-conquer programs which use the Insert(2) higher order function where the use of improved data layouts lead to better load balance. We present timings for programs with and without these optimizations which were taken on an 8K section of a CM-2 using the slicewise CM-Fortran compiler.

#### 7.4.1 Effect of Routing Function Optimization

This technique avoids allocating storage for output objects of routing functions and avoids execution of copy statements for evaluating routing functions. The code generation function uses allocation functions to keep track of storage for input and output objects, and derives output allocation functions by composing routing function definitions with input allocation functions. Because interprocessor communication can be two orders of magnitude slower than computation, it achieves significant reductions in run-time. Further, since storage is not allocated for outputs of routing functions, the optimized programs have lower storage overhead. These effects are demonstrated in the timings presented for un-optimized versus optimized versions of Matrix Multiply (Table 7.4), Forward Elimination phase of Gaussian elimination (Table 7.5) and Rank (Table 7.6). These programs have been described in Chapter 2. Both versions make use of array reduction intrinsics; the only difference is that routing functions are explicitly evaluated using forall statements in the un-optimized version.

For Matrix Multiply, we have obtained improvements of as high as 35, and the optimized program runs on larger data sets.

For Forward Elimination, the effect of storage overhead is severe; the naive implementation runs out of storage for inputs of more than 32 equations, whereas the
<table>
<thead>
<tr>
<th>Matrix 1 Dim.</th>
<th>Matrix 2 Dim.</th>
<th>(A) Naive (Secs)</th>
<th>(B) Opt. (Secs)</th>
<th>Ratio $(A) \div (B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 $\times$ 16</td>
<td>16 $\times$ 16</td>
<td>0.1131</td>
<td>0.0096</td>
<td>12.5</td>
</tr>
<tr>
<td>32 $\times$ 32</td>
<td>32 $\times$ 32</td>
<td>0.2779</td>
<td>0.0159</td>
<td>17.5</td>
</tr>
<tr>
<td>64 $\times$ 64</td>
<td>64 $\times$ 64</td>
<td>2.04543</td>
<td>0.0574</td>
<td>35.632</td>
</tr>
<tr>
<td>64 $\times$ 128</td>
<td>128 $\times$ 64</td>
<td>---</td>
<td>0.0925</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 7.4: Performance of Matrix Multiply: Un-optimized vs. Optimized

<table>
<thead>
<tr>
<th># Eqns</th>
<th>(A) Naive (Secs)</th>
<th>(B) Opt. (Secs)</th>
<th>Ratio $(A) \div (B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2.88</td>
<td>0.705</td>
<td>4.08</td>
</tr>
<tr>
<td>64</td>
<td>---</td>
<td>1.84</td>
<td>---</td>
</tr>
<tr>
<td>128</td>
<td>---</td>
<td>4.08</td>
<td>---</td>
</tr>
<tr>
<td>256</td>
<td>---</td>
<td>12.57</td>
<td>---</td>
</tr>
<tr>
<td>512</td>
<td>---</td>
<td>54.93</td>
<td>---</td>
</tr>
<tr>
<td>1024</td>
<td>---</td>
<td>288.74</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 7.5: Performance of Forward Elimination: Un-optimized vs. Optimized

optimized version can solve inputs data sets having up to 1024 equations.

<table>
<thead>
<tr>
<th># Elts</th>
<th>(A) Naive (Secs)</th>
<th>(B) Opt. (Secs)</th>
<th>Ratio $(A) \div (B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.0137</td>
<td>0.0102</td>
<td>1.343</td>
</tr>
<tr>
<td>256</td>
<td>0.0283</td>
<td>0.01895</td>
<td>1.493</td>
</tr>
<tr>
<td>512</td>
<td>0.0794</td>
<td>0.0499</td>
<td>1.591</td>
</tr>
<tr>
<td>1024</td>
<td>---</td>
<td>0.163</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 7.6: Performance of Rank: Un-optimized vs. Optimized

The Rank program shows smaller improvements compared to the other programs since it provides fewer opportunities for optimizations as few routing functions are used in the program.

### 7.4.2 Effect of Reduction Intrinsic

We present timings for Matrix Multiply and Rank where reduction is executed explicitly using forall statements and where evaluation of routing functions have been optimized. These timings are compared with those for versions where both reduction intrinsics are emitted and routing functions are optimized.
<table>
<thead>
<tr>
<th>Matrix 1 Dim.</th>
<th>Matrix 2 Dim.</th>
<th>(A) Naive (Secs)</th>
<th>(B) Opt. (Secs)</th>
<th>Ratio (A) \div (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 \times 16</td>
<td>16 \times 16</td>
<td>0.0164</td>
<td>0.0096</td>
<td>1.708</td>
</tr>
<tr>
<td>32 \times 32</td>
<td>32 \times 32</td>
<td>0.0546</td>
<td>0.0159</td>
<td>3.433</td>
</tr>
<tr>
<td>64 \times 64</td>
<td>64 \times 64</td>
<td>0.390</td>
<td>0.0574</td>
<td>6.794</td>
</tr>
<tr>
<td>64 \times 128</td>
<td>128 \times 64</td>
<td>0.859</td>
<td>0.0925</td>
<td>9.286</td>
</tr>
</tbody>
</table>

Table 7.7: Performance of Matrix Multiply: Reduction using forall vs. Intrinsic

<table>
<thead>
<tr>
<th># Elts.</th>
<th>(A) Naive (Secs)</th>
<th>(B) Opt. (Secs)</th>
<th>Ratio (A) \div (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.0366</td>
<td>0.0102</td>
<td>3.588</td>
</tr>
<tr>
<td>256</td>
<td>0.1173</td>
<td>0.01895</td>
<td>6.189</td>
</tr>
<tr>
<td>512</td>
<td>0.4683</td>
<td>0.0499</td>
<td>9.384</td>
</tr>
<tr>
<td>1024</td>
<td>1.955</td>
<td>0.163</td>
<td>11.994</td>
</tr>
</tbody>
</table>

Table 7.8: Performance of Rank: Reduction using forall vs. Intrinsic

For the Rank program, the effect of using intrinsics is more significant than optimizing routing functions. However, the optimization of routing functions allows the compiled program to work on larger input sizes.

### 7.4.3 Compiling Insert(2)

The compiler generates efficient data allocation schemes for certain forms of the insert functional. These allocation schemes improve load balancing, by allocating 2-dimensional FP array objects onto 1-dimensional CM Fortran arrays, such that all CM Fortran array elements are active in all iterations. In Table 7.9, we present run-times for bitonic-sort with this data-allocation optimization, and without this optimization. The optimized bitonic sort is as much as a factor of 10 faster than the un-optimized one. Further, the un-optimized version requires larger storage, which restricts it to smaller data sets. For larger data sets, the un-optimized version runs out of memory space.

### 7.5 Absolute Performance

In this section the performance of compiled code for matrix multiply is compared with the CM Fortran primitive matmult provided by Thinking Machines Corporation. Run-times are presented for both programs on matrices of varying sizes. The performance of the compiled program is between a factor of two and six of the CM
<table>
<thead>
<tr>
<th>#Elts</th>
<th>(A) Un-Optimized (Secs)</th>
<th>(B) Optimized (Secs)</th>
<th>Ratio (A ÷ (B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>3.401</td>
<td>0.443</td>
<td>7.677</td>
</tr>
<tr>
<td>512</td>
<td>12.901</td>
<td>0.588</td>
<td>21.940</td>
</tr>
<tr>
<td>8192</td>
<td>---</td>
<td>2.445</td>
<td>---</td>
</tr>
<tr>
<td>16384</td>
<td>---</td>
<td>3.711</td>
<td>---</td>
</tr>
<tr>
<td>32768</td>
<td>---</td>
<td>8.323</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 7.9: Effect of Optimized Load Balancing for Length-Preserving Insert

Fortran primitive for matrix multiply as shown in Table 7.10. These timings demonstrate that the performance of the FP* matrix multiply program is within a small factor of the hand-coded CM Fortran routine for matrix multiplication.

<table>
<thead>
<tr>
<th>Matrix 1</th>
<th>Matrix 2</th>
<th>(A) Compiled FP* Program (secs)</th>
<th>(B) CM-F Primitive (secs)</th>
<th>Ratio (A ÷ (B))</th>
</tr>
</thead>
<tbody>
<tr>
<td>n x 16</td>
<td>m x 16</td>
<td>0.00960343</td>
<td>0.00362343</td>
<td>2.65</td>
</tr>
<tr>
<td>32 x 32</td>
<td>32 x 32</td>
<td>0.015981</td>
<td>0.00488386</td>
<td>3.27</td>
</tr>
<tr>
<td>64 x 64</td>
<td>64 x 64</td>
<td>0.0574081</td>
<td>0.0108767</td>
<td>5.28</td>
</tr>
<tr>
<td>64 x 128</td>
<td>128 x 64</td>
<td>0.0925419</td>
<td>0.0170383</td>
<td>5.43</td>
</tr>
</tbody>
</table>

Table 7.10: Run-times for Compiled program and CM Fortran primitive, respectively, for Matrix Multiply

7.6 Inefficiencies

The code generation function suffers from limitations regarding data layout and network usage in the CM-2 implementation described in this chapter. This limitation is illustrated in the code generation function for Insert(1). The code emitted performs binary operations where the first operand is at index \( i \) and the second operand at index \( i + 2^{k-1} \) in the array which is being reduced. Before the operation can be executed, each of the operands must be present on the same processor which necessitates an inter-processor communication operation. The CM Fortran compiler emits a **send** instruction for performing this interprocessor communication. As pointed out in §7.2, **send** is significantly more expensive than **new** communication instructions. The timings presented in Table 7.3 demonstrate the advantage of using **new** instruction over **send** instructions. The code generation function overcomes this limitation by analyzing source programs and emitting reduction intrinsics whenever possible. Since CM Fortran has a small, fixed set of such intrinsics this is not always possible and the code generation function emits **forall** statements whose evaluation require
send instructions. This is the case for the MaxLoc function shown in Chapter 2 in which the primary computation is reduction. We have provided timings for the Maxloc function, and the MaxLoc primitive provided in CM-Fortran which performs the same function. The timings shows the significant impact data layout has on performance.

<table>
<thead>
<tr>
<th>Size</th>
<th>(A) Compiled (Secs)</th>
<th>(B) Primitive (Secs)</th>
<th>Ratio $(A) \div (B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.057</td>
<td>0.0045</td>
<td>12.66</td>
</tr>
<tr>
<td>8192</td>
<td>0.084</td>
<td>0.0048</td>
<td>17.5</td>
</tr>
<tr>
<td>16384</td>
<td>0.152</td>
<td>0.0049</td>
<td>31.02</td>
</tr>
</tbody>
</table>

Table 7.11: Performance of Maxloc : Compiled vs. CM Fortran Primitive

All communication operations in evaluating the Insert functional can be performed using the news instruction entirely if the underlying hypercube structure of the network is exploited. This is achieved by using a hypercube data structure instead of an array and embedding the data object onto the hypercube network such that nearest neighbors on the data object are either on the same processing node or are nearest neighbors on the network. Such layout and compilation techniques that use only news moves for operand alignment in compiled programs will provide significant improvements. Most parallel computers provide a small set of operations whose implementations have been highly optimized for that specific architecture. Efficient implementations of high-level languages must analyze source programs and generate those operations, if possible, in the compiled program. Using a code generation function that is inductively defined makes the task of analyzing source language programs and emitting such operations relatively simple, as shown earlier with regard to array reduction intrinsics.
Chapter 8
Conclusions and Future Work

We have described a set of formal techniques for compiling high-level languages based on the data-parallel programming model. We conclude by summarizing our contributions that we have described in earlier chapters of this dissertation and outline areas that require further investigation.

8.1 Contributions of this dissertation

The focus of this thesis has been on compiling high-level languages based on the data-parallel programming model for parallel architectures. We have focused on functional languages that are organized around arrays and array operators as a means of data-parallel programming. This differs from the traditional approach to parallel functional programming in two ways: (1) we use a fixed set of higher-order functions instead of recursion for control, and (2) we use arrays instead of linked lists as the primary aggregate data object. These new approaches have allowed us to explore new trade-offs in language design and implementation. We have developed a set of compilation techniques suitable for analyzing data-parallel programs written in a functional language. These techniques infer types and array sizes, perform program transformations that minimize inter-processor communication and statically improve load balancing. The analysis phase of the compiler feeds information to the conversion phase which produces low-level parallel programs. Parallelism is described exclusively with forall loops.

We have devised these compilation techniques in a formal framework that includes inference systems and inductive function definitions. An inference system has been used to derive most general types and array sizes of data. Code generation is completely defined using a syntactic function $C$ that, when evaluated on an input consisting of an FP* expression and an execution environment, returns a target language program. This formal specification provides a clear and concise description of the compiler including program transformations, program optimizations, and code generation. This formal approach eliminates ambiguity regarding the abilities and limitations of the compiler. In addition, since formal specifications can also be regarded as software specifications, they are useful to implementors. We have implemented the compiler in Prolog in which program statements are inference rules and recursive function definitions. We have compiled a variety of programs, including matrix algorithms, sorting algorithms and linear system solvers for efficient execution.
on the Connection Machine CM-2. These benchmarks have helped us in examining trade-offs in expressiveness and implementation efficiency in the design of high-level languages.

Finally, we have explored issues in the design, programmability, and implementation of functional languages organized around arrays and array operators for data-parallel programming. We have also explored the use of formal methods in specifying compilers for parallel architectures. Although thorough validation involving large-scale applications is required, our research provides evidence that such high-level languages are useful for writing parallel programs and formal specifications can be used to describe a variety of compiler optimizations.

8.2 Limitations of FP*

Our research has been performed in the context of the programming language FP*. FP* incorporates a small set of modifications and extensions to FP to make it more suitable for data-parallel programming. These include the introduction of a static polymorphic type system and programmer-defined routing functions. In the course of writing and compiling programs in FP*, we came across limitations in the language that either made it difficult to express certain operations or significantly increased the complexity of the implementation. Overcoming these limitations would aid the development of large application programs in FP*.

Like FP, FP* does not allow programmers to freely define and refer to variables. Instead, the programmer must create and pass entire environments across functions and use selection functions to extract values from the environment. Another constraint in FP* is that values obtained by evaluating true and false branches of conditional must be of the same type and size, limiting the expressiveness of the conditional higher-order function. Removal of this limitation requires modification of the structure inference system to handle union types [19].

FP* has the same set of primitive functions as FP. However, for purposes of data-parallel programming, it would be useful to introduce a wider variety of primitives into the language to improve efficiency. One such primitive is permute which takes as input an array of values and a vector of destination indices. The output is obtained by performing the permutation on the input. An FP* function that performs permute is described in example 2.5. It requires $\log n$ time using $n^2$ processors to permute $n$ elements. Some parallel computers provide more efficient implementations of permute. The introduction of a permute primitive function in FP* allows the compiler to take advantage of efficient permute implementations provided in parallel computers.

FP* also provides the same fixed set of higher-order functions as FP. FP* uses the Insert higher-order function to describe divide-and-conquer programs such as Bitonic Sort. Although Insert has its advantages, it still requires that the programmer unfold the divide phase till the base case is reached after which the Insert can be used for
the combine phase. A higher-order function which integrates the divide phase with the combine phase would lead to more elegant divide and conquer programs[44].

8.3 Future Work

A number of issues remain to be addressed. These can be broadly categorized into two areas: language enhancements and compiler optimizations.

8.3.1 Language Enhancements

- **Recursion.** Recursion has been shown to be a very elegant and powerful means for high-level programming. Our research has resulted in optimizing compilation schemes for translating higher-order functions for efficient execution on parallel architectures. An interesting research problem is to analyze programs described using recursion and automatically transform them into programs constructed using higher-order functions. In fact, techniques for translating high-level languages organized around recursion into languages organized around a fixed set of higher-order functions are being investigated [46, 21].

- **Segmented Arrays.** We have focussed entirely on regular problems, such as matrix computations and linear system solvers that have highly regular computation structures. Segmented arrays have been shown to be useful for describing irregular computations such as sparse matrix computations and graph algorithms [10]. Integrating segmented arrays and operations on them into FP* would be useful for enhancing its domain of applications to irregular problems.

- **Input/Output.** The issue of parallel I/O primitives at the language level has not been addressed in this work. Parallel I/O is critical to sustained performance in real-world applications. In functional languages, I/O is typically modelled by the abstraction of a stream of input and output. By abstracting input and output as operations on a data-structure, side-effect free semantics can be preserved. To perform parallel I/O, streams must allow parallel access to input and output elements.

8.3.2 Compiler Optimizations

- **Communication Optimizations.** Communication is the most expensive operation on parallel architectures. Minimizing communication has two distinct aspects: (1) data allocation and (2) communication synthesis. There has been considerable research on generating efficient array layouts for distributed memory machines where by inter-processor communication is minimized [34, 48]. Secondly, most parallel computers provide a non-uniform view of inter-processor
communication. Some patterns of communication are more expensive than others. There is ongoing research on communication synthesis [37] whereby communication due to array reference patterns in the program is translated into low-cost communication instructions.

- **Control parallelism.** We have entirely focused on data parallelism in this work. For some applications and machines, control parallelism is more appropriate than data parallelism. Compilation techniques appropriate for this model need further development.

By no means is this research complete, as evidenced by the list of challenging problems that await. Nonetheless, the prototype compiler demonstrates that high-performance functional languages are quite feasible. We look forward to continued dramatic improvements in this technology.
References


