UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

Synthesizing Imperative Distributed-Memory Implementations from Functional Data-Parallel Programs

by

Tristan Aubrey-Jones

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Physical Sciences and Engineering

Department of Electronics and Computer Science

May 2015

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Tristan Aubrey-Jones

Distributed memory architectures such as Linux clusters have become increasingly common but remain difficult to program. We target this problem and present a novel technique to automatically generate data distribution plans, and subsequently MPI implementations in C++, from programs written in a functional core language. This framework encodes distributed data layouts as types, which are then used both to search (via type inference) for optimal data distribution plans and to generate the MPI implementations. The main novelty of our approach is that it supports multiple collections, distributed arrays, maps, and lists, rather than just arrays.

We introduce the core language and explain our formalization of distributed data layouts. We describe how to search for data distribution plans using a type inference algorithm, and how we generate MPI implementations in C++ from such plans. We then show how our types can be extended to support local data layouts and improved array distributions. We also show how a theorem prover and suitable equational theories can be used to yield a better (i.e., more complete) type inference algorithm. We then describe the design of our implementation, and explain how we use a runtime performance-feedback directed search algorithm to find the best data distribution plans for different input programs. Finally, we present some conceptual and experimental evaluation which analyzes the capabilities of our approach, and shows that our implementation can find distributed memory implementations of several example programs, and that the performance of generated programs is similar to that of hand-coded versions.

Contents

List of Figures

List of Tables

Listings

List of Publications

[13] T. Aubrey-Jones and B. Fischer. Synthesizing MPI Implementations from Functional Data-Parallel Programs. In *Proc. 7th International Symposium on High-level Parallel Programming and Applications (HLPP'14)*, 2014.

[14] T. Aubrey-Jones and B. Fischer. Synthesizing MPI Implementations from Functional Data-Parallel Programs. *International Journal of Parallel Programming*, to be published., 2015.

Nomenclature

AI ..... Artificial Intelligence
API ..... Application Programming Interface
ARMCI ..... Aggregate Remote Memory Copy Interface
ATLAS ..... Automatically Tuned Linear Algebra Software
BLAS ..... Basic Linear Algebra Subprograms
CAF ..... Co-Array Fortran
CUDA ..... Compute Unified Device Architecture
DAG ..... Directed Acyclic Graph
DDL ..... Distributed Data Layout
DFG ..... Data-Flow Graph
DFT ..... Discrete-Fourier Transform
DPH ..... Data-Parallel Haskell
DryadLINQ ..... Dryad Language-Integrated Query
FFT ..... Fast-Fourier Transform
FFTW ..... Fastest Fourier Transform in the West
FLOCC ..... Functional Language on Compute Clusters
FPU ..... Floating-Point Unit
GPGPU ..... General Processing on Graphics Processing Unit
GPU ..... Graphics Processing Unit
GUI ..... Graphical User Interface
HPC ..... High-Performance Computing
HPF ..... High-Performance Fortran
IO ..... Input/Output
JDBC ..... Java Database Connectivity
JIT ..... Just-in-time
JRMI ..... Java Remote-Method Invocation
JVM ..... Java Virtual Machine
LINQ ..... Language-Integrated Query
MIMD ..... Multiple Instruction Multiple Data
MPI ..... Message Passing Interface

MPMD ..... Multiple Program Multiple Data
NUMA ..... Non-Uniform Memory Access
PLINQ ..... Parallel Language Integrated Query
PVM ..... Parallel Virtual Machine
RDBMS ..... Relational Database Management System
SAC ..... Single Assignment C
SALSA ..... Simple Actor Language System and Architecture
SETL ..... Sets Language
SHMEM ..... Symmetric Hierarchical Memory access
SIMD ..... Single Instruction Multiple Data
SISAL ..... Streams and Iteration in a Single Assignment Language
SPMD ..... Single Program Multiple Data
SQL ..... Structured Query Language
TAG ..... Threshold Ascend on Graph
UPC ..... Unified Parallel C

Acknowledgements

Firstly, I thank my whole family, and particularly my parents, siblings, and grandparents, for their love, support, and friendship, especially in hard times, and throughout far too many years of education!

Secondly, I thank my supervisor, Bernd Fischer, for mentoring me and for going far beyond the call of duty. This has included continuing to supervise me even after moving 9000 miles away. I could not have asked for a better supervisor. I also thank my supervisor and his partner, Sylvia Diekmann, for their amazing hospitality when visiting them in South Africa, and Michael Butler for his support, and for facilitating this remote supervision.

Thank you to all my University friends and colleagues for your friendship and the fun we have had together. In particular, Nawfal F Fadhel and Shre Chatterjee for great food and great films, Adisak Intana and Boonyarat Phadermrod for countless lunchtime chats, and Neville Grech, Meng Tian, Teresa Binks, and Raied Al-Lashi as well. Thanks also go to Jeremy Morse for helping me deciper many a segmentation fault, and Owen Stephens for his patient explanations of all things Haskell.

From my Southampton friends, a special mention goes to Alex Bailey, James Pritchard, Ann Hutchinson, and Jon Paul Jude, whose friendships I treasure, as well as all those at Highfield church.

Finally, I thank and acknowledge the help of God: the Father, his Son Jesus Christ, and the Holy Spirit. Everything good in this thesis comes from Him. To Him be the glory.

"About five thousand men were there. But he said to his disciples, 'Have them sit down in groups of about fifty each.' The disciples did so, and everyone sat down. Taking the five loaves and the two fish and looking up to heaven, he gave thanks and broke them. Then he gave them to the disciples to distribute to the people. They all ate and were satisfied, and the disciples picked up twelve basketfuls of broken pieces that were left over." Luke 9:14-17

Chapter 1

Introduction

Distributed memory computer architectures have become increasingly common [138, 149, 172]. Linux clusters are now used both for high-performance computing (HPC) and web-scale "Big Data" analysis. Graphics processors (GPUs) with non-uniform memory architectures (NUMA) are now used for general purpose data-parallel computation. However, despite their prevalence, distributed memory architectures remain difficult to program manually. For example, for clusters using the message passing interface (MPI) [189], the implementation of a simple matrix multiplication already requires about 100 lines of C++ code—compared to the five lines of its sequential counterpart (see Listings A.1 and A.2). This is due to the fact that MPI requires developers to explicitly handle: partitioning, marshaling and communication of the data, as well as synchronization, which all also introduce a much greater potential for bugs.

The usual way to address this sort of problem in computer science is to raise the level of abstraction, typically by providing a higher-level language and an associated compiler. However, the automatic translation of high-level languages to efficient low-level code for distributed memory architectures remains a difficult problem, due to the many possible data distributions for any given program. Many techniques only support a fixed model, such as map-reduce [66], that is not necessarily suitable for all problems [74], or do not support distributed memory at all [159]. For example, map-reduce only supports a single collective operation applied to disk-backed key-value maps (associative arrays), and high-performance Fortran (HPF) only supports numerical computations involving memory-resident multidimensional arrays.

In this thesis we present a flexible type-based technique to search through the space of possible data distributions, and synthesize MPI implementations in C++, from a high-level language called Flocc (Functional language on compute clusters). Flocc is a simple functional language which relies on data-parallel combinators (i.e., higher-order functions) to abstract away from global state, iteration, recursion, and individual element accesses. Our key insight is that we can use rich types to formalize knowledge about

the data distribution characteristics of these combinators and type inference to derive valid data distribution plans for Flocc programs. The key advantages of our approach are that it is fully automatic, and that it supports operations involving maps, sets, lists and arrays, in the same programming model, and is therefore more flexible than existing approaches.

1.1 Problem Statement

The key problem that this thesis addresses is that programming distributed-memory architectures is hard, since most approaches force the user to manually choose data distributions and only work for very restricted programming models. We are lacking general purpose approaches that work for multiple collection types, whilst abstracting away from the concrete data layouts, and delivering good performance. This section elaborates on these difficulties and the limitations we have observed with existing approaches.

Distributed Memory Architectures Distributed memory architectures are parallel computer architectures whose memory is decentralized. This means that different processing units may experience different latencies and speeds accessing the same area of memory, since it may be local to one and remote from another. In fact, some processing units may not be able to access some areas of memory at all, or perhaps only indirectly i.e., via messages sent over a network.

Computational clusters In this thesis we restrict our attention to (Linux) clusters [172], and leave architectures like GPUs to future work. Clusters consist of hundreds (often thousands) of standalone PCs called *nodes*, each with its own processors and local memory, connected via some high-speed network like Gigabit-Ethernet or Infiniband. They are used for nearly all modern HPC tasks like numerical analysis and weather forecasting, as well as web-scale "Big Data" analyses like log processing and search-engine indexing. They are also a very good example of distributed memory architectures, since the difference in latency between accessing a node's local memory, and another node's memory (via message passing), can be 10,000-times for cache hits and 100-times for misses, even for a supercomputer with a state of the art Infiniband network.[1] It is therefore essential for performance that programs running on clusters minimize the amount of network traffic they generate, by identifying what data needs to be accessed frequently, and storing it as locally as possible. Getting this right can lead to huge performance gains, getting it wrong to huge penalties.

[1] Measurements taken on Iridis 4, a fourth generation cluster with 770-nodes, each with 16-cores at 2.6GHz, 4GB of memory per core, and an Infiniband network for interprocess communication. For more information see http://cmg.soton.ac.uk/iridis.

Manually programmed communication The most flexible way to program a cluster is to manually program all communication code using a general-purpose programming language like C or C++, and a library like MPI [189] or SHMEM [16]. MPI (Message Passing Interface) [189] is a library specification for passing messages between processes on a cluster. It relies on the "single program multiple data" (SPMD) model, where the same program is spawned multiple times (usually once per processor) with each process given a different process identifier (or rank). There is always one distinguished root process, and programs branch depending on the rank of their process.

For example, Figure A.2 shows an MPI implementation of a simple dense matrix multiplication. The root node generates the random input matrices, and then partitions them and broadcasts them to other processes. Each process computes a subset of the result matrix, and sends these result partitions back to the root. As even this simple example demonstrates, although MPI is very versatile, it requires manual implementation of all data layout and distribution, and very verbose, hard-to-debug implementations. For this reason a number of high-level data parallel languages have been developed. However, these approaches have a number of problems.

Distributed memory not supported A number of data-parallel languages exist that only target shared-memory architectures. NESL [23] specializes in nested data-parallel vector operations on vector machines. Data Parallel Haskell (DPH) [159] is an extension to Haskell based on NESL, but for modern multi-cores. Single Assignment C (SAC) [98] supports n-dimensional array computations, with an impressive implementation that has outperformed Fortran in some cases. However, none of these currently support distributed memory data-parallelism, or suggest how such support could be implemented.

No automated data distribution Other languages support clusters, but programmers must still manually specify the data layout. Co-Array Fortran [151] and Unified Parallel C (UPC) [37] are SPMD-oriented languages that abstract away from explicit message passing, but still force the programmer to specify how to distribute arrays and when to read/write remote data. Chapel is higher-level than these, and has a more flexible programming model than other languages, but requires explicit data distribution selection. High-performance Fortran (HPF) [135] is traditionally the most popular language for numerical HPC, but it also requires arrays to be annotated with data distribution directives, although an experimental tool has been developed to automatically optimize them [115].

Restricted programming models Finally, some approaches free the programmer from the burden of manual data layout and distribution, but usually only for a restricted programming model. MapReduce [66] and Hadoop [210] are frameworks for

performing aggregations on huge datasets, hosted on large-scale clusters. They handle all communication, scheduling, and failure recovery, and so greatly simplify data-parallel programming. However, they have a single restricted programming model and distributed implementation, involving a *map* function that projects key-value pairs from a dataset, and a *reduce* function that aggregates a sorted list of values for each key.

Some approaches automatically optimize data distributions for imperative code involving affine loop nests [9, 21, 27] and for languages based on array section operations [49], but these only support array-based computations and so are unsuitable for applications which require other collections like sets, maps, lists, and disk-backed collections. Parallel databases can also be used for some distributed data-parallel tasks. Like Flocc programs, parallel SQL query plans [45] are synthesized by enumerating different combinations of plan operators to minimize the overall cost [185]. Like Flocc's map combinators, SQL queries are also based on relational algebra, although they have a weak type system, no support for array-based computation, and cannot be extended with new operators. Furthermore, parallel databases typically do not generate standalone code, and the distributed schemas must be designed manually, though a tool to assist with this has been proposed [157]. All of these approaches primarily support a single collection type and associated operations.

Lacking extensibility Finally, none of the approaches that automate data distribution have shown that they can be extended to work with more collection types, operators, and their associated data layouts and distributions.

In summary, the existing approaches for data-parallel programming clusters are either too low-level and therefore laborious and error prone (MPI), or are high-level and involve some automation but have other deficiencies. Some of the most high-level approaches do not support distributed memory architectures (NESL, DPH, SAC), and those that do often still require the programmer to manually select or implement the distributed data layout, which is hard to optimize, restricts portability, and is inaccessible for non-experts (Co-Array Fortran, UPC, HPF, Chapel). Those that do provide some automatic data distribution typically have restrictive programming models that only support one collection type, and have not yet been shown to extendable to support others (MapReduce, SQL, DryadLINQ, Loop parallelization).

This thesis targets these problems, by proposing a new approach to fully automatically synthesize correct MPI implementations from a data parallel programming language. This approach will address the deficiencies identified above; in particular, it will

- abstract away from manual communication and data distribution;
- target distributed memory architectures, in particular clusters;

- automatically select appropriate data layouts or distributions;
- support multiple collection types and operators; and
- show potential for extensibility.

The key point is to automate the distributed data layout selection and implementation, for a more expressive language (i.e., one that supports more collections and operators than just arrays, or just key-value maps) while maintaining good runtime performance.

1.2 Research Objectives

The goal of this work is to develop a technique to automatically synthesize cluster implementations of data-parallel programs that support multiple collection types and their associated operators. This must include some suitable input language, a mechanism to define and automatically identify suitable data distributions, and some output code synthesis technique, which support multiple collection types and operators. To achieve this overall goal we have a number of specific objectives:

1. We define Flocc, a core functional language which supports arrays, maps, and lists, and data-parallel combinators (i.e., higher order functions) to manipulate them (cf. Chapter 3).
2. We define a type system that extends core types with information about the distributed data layouts of collections. We use these types to characterize the data distribution behavior of different functionally equivalent implementations of combinators (cf. Chapter 4 and Chapter 6).
3. We develop a type inference algorithm to automatically recover distributed data layout information for collections, given a program with combinators replaced by specific combinator implementations, if a valid typing exists (cf. Chapter 5).
4. We present an algorithm to automatically insert type casts (i.e., data redistributions) in appropriate locations in invalid programs to make them type-check (cf. Chapter 5).
5. We develop a code generation mechanism to translate programs involving concrete combinator implementations and their inferred types, which together we call *plans*, into MPI implementations in C++ (cf. Chapter 7).
6. We implement a search-based compiler which uses performance-feedback to search through different data distribution plans, by generating, compiling, and executing candidate solutions on some suitable test data (cf. Chapter 7 and Chapter 8).

1.3 Overview of the approach

Data-parallel language Before developing our automatic data layout technique, we define a suitable source language for our approach. Instead of using an existing programming language, we define a simple functional language called Flocc (cf. Chapter 3) to minimize the accidental complexity that a full language would bring. Flocc is based on the polyadic lambda calculus (i.e., lambda-calculus with tuples) and is therefore not a contribution per se, but an apt and necessary substrate for the rest of the work.

Despite its syntactic simplicity Flocc is very expressive, supporting multiple collection types (arrays, maps, and lists), and associated data-parallel operations. These operations are related via combinators (i.e., higher-order functions) which abstract away from data layout and individual element access. This makes Flocc extensible, allowing it to support multiple collection types in a single framework. Flocc is also strongly typed and supports type inference, a key prerequisite for our approach. We present the syntax, semantics, and type system for Flocc in Chapter 3, and implement a front-end for it (cf. Figure 1.1).

Data layouts as types The first step towards automatically implementing distributed data layouts for Flocc programs is to provide different functionally equivalent implementations of the high-level combinators that distribute their input and output collections in different ways. These combinator implementations are internal to the compiler (i.e., hidden from the user), and are used to explore possible data layouts by trying different implementations for the combinator applications in the input program. This is similar to how logical operators in SQL queries can be implemented by different concrete plan operators.

To characterize the data distribution behaviors of these combinator implementations, we extend our collection types with extra parameters to characterize how they distribute their inputs and outputs (see Chapter 4). For example, the type `DMap k v pmd pf pdim mdim lm lf` extends `Map k v` with six extra parameters, four of which (`pmd`, `pf`, `pdim`, and `mdim`) define which partitions elements should reside in, and two of which (`lm` and `lf`) define how the partitions should be stored locally. The primary extensions are the addition of embedded functions `pf` and `lf`— actual functions (full lambda-terms, not function types) that define the keys along which maps should be partitioned, and indexed or ordered by locally. These embedded functions can be defined in terms of type variables, concrete lambda terms, and function composition, and can also be lifted into the types from function parameters using *dependent type schemes* (cf. Section 4.3.2). Thus types like,

```
foo : DMap Int Int Hash (sqr · f) d1 d2 HashMap id
   -> DMap Int Int f d1 d2 List id
```

can be used to characterize how inputs and outputs of combinator implementations are distributed in terms of each other (e.g., partition function `f`).

We present this extended type system and prove it sound (i.e., guaranteeing that well typed programs can progress, and that evaluation does not break the types). Again, the key benefit of this approach is that it is extensible (i.e., we can add more collection types and type parameters) and works for multiple collection types, rather than just arrays (like HPF [135]) or just maps (like MapReduce [66]).

Data layout inference We call a Flocc program with combinator function applications replaced with specific combinator implementations a "plan". Characterizing data distributions as polymorphic types allows us to use type inference to automatically find data layouts for plans that satisfy the data distributions of the combinator implementations used. This is necessary to automatically handle data distributions, since otherwise the user would have to specify them by hand. In Chapter 5 we present this type inference algorithm and prove it sound (although it is not complete since we use a decidable under-approximation to unify functions).

The best plans are often those that redistribute collections, changing their data distributions and layouts so that more efficient combinator implementations can be used. These redistribution operations correspond to type-casts in our system. To find plans that involve redistributions, we develop an algorithm that automatically inserts redistribution function applications, at appropriate places in plans to mend broken type constraints, and make them type-check. We present this algorithm in Chapter 5, and implement it (cf. Figure 1.1).

Extended data layout types Chapter 6 demonstrates the extensibility of our approach, by presenting several extensions to our data layout types, to encode local layout information, implement more complex array distributions, and make the types more flexible. These types require a more nuanced type inference algorithm, that can solve equations between functions. We present an extension to our core type inference algorithm that does just this, by using an automatic theorem prover, and equational theories of projection functions, permutation functions, and integer index functions. We prove these theories sound, and explain how they could be integrated into our core type inference algorithm.

Code generation We have developed a code generator for Flocc plans that takes a plan annotated with DDL types, and generates an imperative MPI program in C++ that implements the program on a cluster (cf. Chapter 7). This generator converts the plan into a data flow graph, and then traverses the DFG to generate code, instantiating code templates for combinator implementations as it goes. We decided to generate

C++ rather than code in a functional language so that we can use MPI and so yield the high-performance necessary for many data-parallel applications.

Data layout optimization Finally, to optimize the parallel performance of Flocc programs, we have implemented a prototype compiler that uses the code generator in a feedback loop, generating, compiling, and running implementations of candidate plans, to yield performance-feedback to guide the plan search (see Chapter 7). The architecture of this compiler is shown in Figure 1.1. After parsing and type-checking a Flocc program, the performance-feedback-based implementation search starts. This search uses rule sets, which list the different possible implementations of each data-parallel combinator, and distributed data layout (DDL) type definitions for each of these definitions. These DDL types extend the high-level functional types with extra parameters, which declare how each combinator implementation distributes its input and output collections. The search explores different plans, which use different combinator implementations for the combinator function applications in the high-level program. Each plan's abstract syntax tree (AST) is converted into a data flow graph (DFG), from which C++ code is generated using templates for the different combinator implementations. These C++ implementations are compiled, executed, and the performance (i.e., total runtime) of each is measured. This performance data is then fed back into the search, to help guide which plans to consider next. Finally, in Chapter 8 we present some conceptual and experimental evaluation of our technique, that demonstrates that the performance of generated programs can come close to hand-coded equivalents, investigates the suitability of different search heuristics, and compares the core capabilities of our approach to other programming languages for data-parallelism.

Overall then, this approach takes programs written in a high-level data-parallel language, which supports multiple collection types, and yields MPI implementations in C++ from data distribution plans that have been optimized empirically through a performance-feedback-based plan search.

1.4 Original contributions

Our main contribution is a technique for automatic synthesis of distributed memory implementations of high-level data-parallel programs that supports multiple collection types, arrays, maps, and lists. Our key insight is that distributed data layout information can be embedded in types, and recovered using a type inference algorithm, in a way that works for multiple collection types, and is thus much more general than existing approaches. This overall contribution involves several subsidiary contributions.

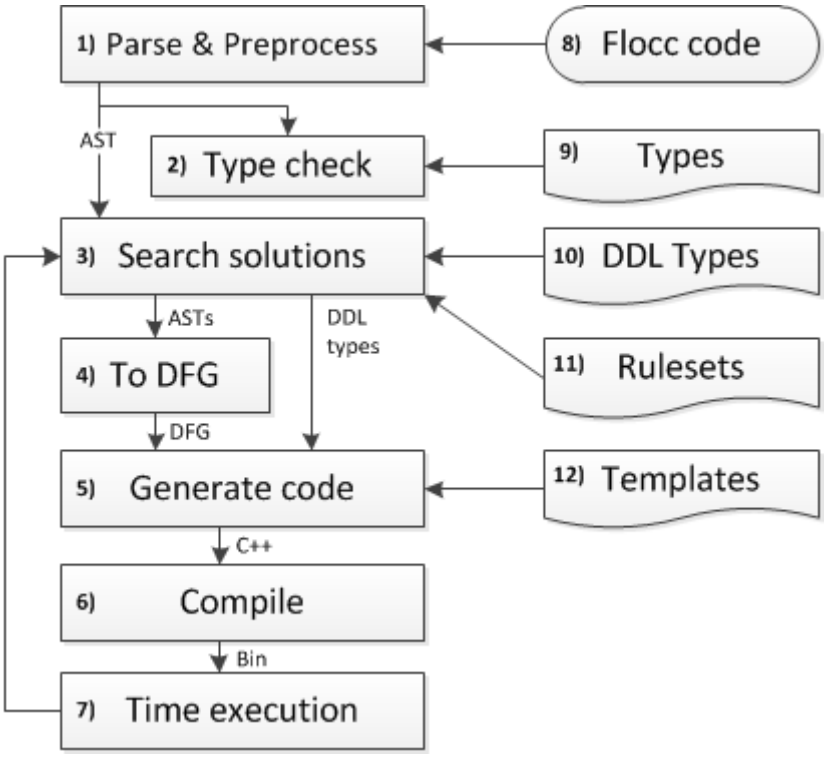


FIGURE 1.1: Feedback-directed code generation for Flocc

Data distributions as types We believe that this is the first work to specify distributed-memory data layouts as *types*, and specifically a form of restricted dependent types we call *dependent type schemes* (cf. Section 4.3.2). In fact, ours is one of the first approaches to formalize any data layout information using a type system— another recent one being a type-based technique for local auto-vectorization [188]. Furthermore, ours is the first work to recover (distributed-memory) data distribution information for programs automatically and statically via type inference (cf. Chapter 5). The approach works by characterizing the distributed and local data layout behaviors of operators and collections through polymorphic types, extended with additional parameters to carry embedded functions, which define how collections should be partitioned and stored locally. We are unaware of any existing work that characterizes data layouts using functions in this way (rather than sets of columns for example).

Data layout inference for non-arrays To our knowledge, ours is the first system to automatically reason about data distributions for multiple collection types (i.e., arrays, maps, and lists) using a single language in a unified framework. We define the distributed data layout characteristics of data-parallel operations (for all these collection types, cf. Chapter 4), and then automatically infer suitable data layouts for different combinations of them (cf. Chapter 5). Our approach is also extensible, in that more collection types and operations can be added, and data layout types extended (by adding additional parameters), without modifying the input language or core implementation

(cf. Chapter 6).

Extended type inference and automatic type cast insertion In Chapter 6 we show how our original DDL types can be extended to carry local data layout information and support more nuanced array distributions involving offsets and ghosting (i.e., overlapping fringes around array partitions). To support this our type inference algorithm needs to perform some equational reasoning (e.g., to know that addition is associative). We therefore also present an extension to our type inference algorithm, which uses equational theories of projection functions, permutation functions, and integer index functions, and a theorem prover [183] to unify constraints between functions embedded in the types (cf. Section 6.4). This extension is sufficient to support our extended DDL types, and also improves the algorithm's completeness, such that it finds solutions that the original algorithm cannot. It can also solve more complex constraints, and in particular those that involve multiple unknown functions via type variables (cf. Section 6.2), and addition and multiplication of array indices (cf. Section 6.3).

We also present a technique to automatically inserts type casts to make programs type-check (cf. Section 5.2). We use this technique to automatically insert redistribution and re-layout functions into possible distributed-memory implementations of programs, to make them type-check (i.e., to make the distributed data layouts unify). This includes an algorithm that has better computational complexity than our initial approach, which is important due to the large number of combinations of redistribution functions possible.

Innotative implementation There are several innovations in our implementation. First, to our knowledge we are one of the first to use meta-data inferred by type inference to parameterize code generation templates (cf. Section 7.4.6). Secondly, although we do not consider Flocc to be a constribution per se, it is novel, and we believe that the combination of features that allow us to generate efficient standalone imperative code from a (semi)-functional language is unique (cf. Chapter 3 and Section 7.4). For example, Flocc supports the definition of higher-order functions with polymorphic types, but requires that function parameters to such functions are statically resolvable. This constraint allows function parameters to be lifted into the types during type inference, and allows much faster implementations to be generated, since it means that implementations can use basic data types and inlined operations rather than pointers and reduction engines.

Our implementation also uses a performance-feedback-based code synthesis search in a new context (cf. Section 7.5). Although the benefits of performance-feedback-based auto-tuning for certain classes of algorithms are well known [20, 87, 166, 209], we are the first to apply such techniques to programs designed for clusters. Furthermore, we apply it to a much more expressive input language than most current work. We also include a limited comparison of different search algorithms and their comparative convergence speeds for some example programs (Section 8.3).

1.5 Outline

We present background on data-parallel programming, and a survey of existing programming languages for data-parallelism in Chapter 2. Chapter 3 defines Flocc, our data-parallel language as per Objective 1, with its type system, and some basic library functions.

In Chapter 4 we define the concept of *distributed data layout* (DDL) types for distributing collections. We use these types to specify the distributed behaviors of some distributed implementations of the library combinator functions as per Objective 2. We then present worked examples that demonstrate how we can use our DDL types to infer the standard distributed-memory implementations of several common data-parallel algorithms. The majority of Chapter 3, Chapter 4, and Section 8.2, were presented at HLPP 2014 [13], and are to appear in the International Journal of Parallel Programming 2015 [14].

In Chapter 5 we define our DDL type inference algorithm for Objective 3. We also present an algorithm to automatically insert redistribution and re-layout functions (which act as DDL type casts) into programs, to make the DDL types unify. This corresponds to Objective 4.

In Chapter 6 we present some extensions to our DDL type system that support more flexible expressions for partition and local layout functions, and extended distributed-array types with ghosting (which are also known as *fringes*), that also contribute to Objective 2. Along with these extensions we present a more sophisticated inference mechanism to derive DDL types using equational theories of projection, permutation, and index transformer functions, which also contributes to Objective 3. These extensions allow us to derive solutions that could not be found using the system in Chapter 4 (e.g., one for the MinBucket triangle enumeration algorithm).

In Chapter 7 we explain the implementation of our code generator. It starts by describing how the front-end and DDL types are implemented, and then explains how we generate MPI programs in C++ from Flocc programs and data distribution plans. This corresponds to Objective 5. It then shows how we use a performance-feedback-based search to find optimal implementations of input programs, and describes the different search algorithms available. This corresponds to Objective 6.

Then, in Chapter 8 we present some conceptual and experimental evaluation, which demonstrates that we have fulfilled Objective 6 (i.e., that our implementation works in practice). This includes an analysis of our approach's capabilities, and compares the performance of automatically generated implementations of some data-parallel programs with manually implemented PLINQ and text-book C++/MPI versions. It then compares how well different performance-feedback-based search heuristics find optimal implementations of some larger programs.

Finally, in Chapter 9 we conclude this thesis, summarize our contributions, and suggest some directions for further work.

Chapter 2

Background and Literature survey

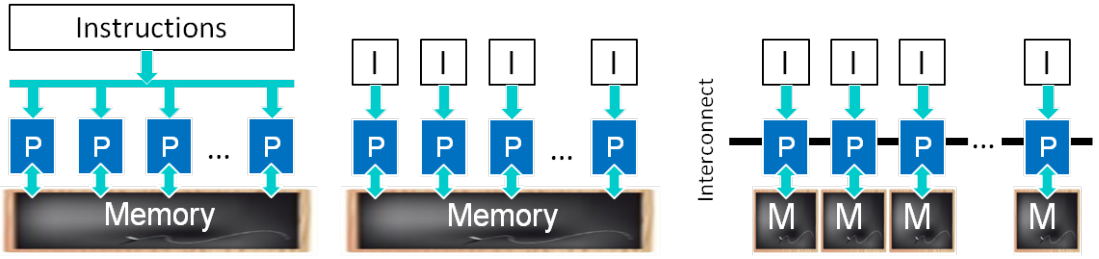
This chapter surveys programming paradigms and languages for programming data-parallel algorithms on shared-nothing clusters, and presents other relevant background for this thesis. We first introduce different parallel programming models and paradigms. We then review different languages for data-parallelism and distributed memory architectures. Finally, we present some background on type inference and auto-tuning.

2.1 Parallel Programming

Parallel architectures and programming models can be categorized using Flynn's Taxonomy [83] into single instruction multiple data (SIMD) and Multiple Instruction Multiple Data (MIMD) systems. SIMD architectures like the Cray 1 vector machine [174] perform single instructions on arrays of values simultaneously. Processing units execute in lockstep applying the same instruction to different parts of the data. SIMD architectures are no longer common for supercomputers but are still used in floating point units (FPUs), for general purpose computing on graphics processing units (GPGPU) [152], and other special-purpose hardware [154]. MIMD architectures like the Connection Machine [107] and modern supercomputers apply multiple instructions to multiple pieces of data simultaneously. They can be further divided into those with shared memory, and those with distributed memory. Figure 2.1 illustrates the topologies of these three models.

Current multicore PCs are shared memory MIMD. The majority of large MIMD systems have distributed memory with non-uniform memory access (NUMA). These are often clusters [172]: thousands of nodes each with their own CPU and local memory communicating by some interconnect, as illustrated in Figure 2.2a. They are termed

shared-nothing, since non-local memory is accessed indirectly by message passing. They are commonly used for scientific and commercial applications, performing massive simulations, processing petabytes of data, and providing cloud computing services. MIMD architectures can also be subdivided into those that use a single program multiple data (SPMD) model and those which use a multiple program multiple data (MPMD) model. In SPMD the same program is executed on all nodes, although not in lockstep, whereas in MPMD different programs execute on different nodes. Both multicores and clusters support MPMD, although they may also be programmed in an SPMD style.



(A) SIMD (B) MIMD Shared Memory (C) MIMD Distributed Memory

FIGURE 2.1: Parallel architectures

These different architectures have led to different programming models.

2.1.1 SIMD models

SIMD architectures have special instruction sets that work on vectors, and can be programmed in a variety of ways. Vectorizing compilers parallelize sequential programs, by analyzing loops to identify those that can become vector operations (e.g., map, fold, or scan) [125, 181, 199]. High-level languages like NESL directly support vector operations in their syntax [23]. GPUs have hierarchical memory architectures (see Figure 2.2b), and an SPMD/SIMD hybrid execution model. NVIDIAs compute unified device architecture (CUDA) programs them using *kernel functions* which work on individual data points, applied to arrays [152].

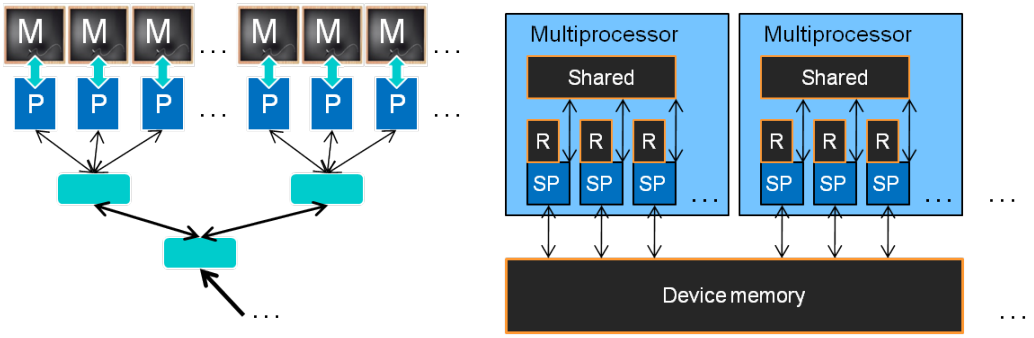
2.1.2 MIMD models

MIMD programming models fall into three categories: those that split computation into processes which communicate via *message passing*, those which split it into processes that synchronize via *shared variables*, and those which give a *high-level* abstraction of the entire algorithm. The first two most closely reflect the architectural models, but are less convenient since they fragment the algorithm into processes, and so high-level approaches can be more attractive.

Shared memory Multi-cores are MIMD with shared memory, and are often programmed using threads [122], with a shared address space, and synchronization constructs to police access to shared resources. Threads can manage separate concerns like GUI (graphical user interface) events and file IO (input output), perform different stages of a pipeline, serve different users, or process different parts of some data. They are available in C [11] via the Pthreads library [148], and in Object Oriented languages like Java [95] and C# [105] with built-in libraries. However, threads can be hard to reason about, due to the arbitrary possible interleavings between them, which can cause race conditions and deadlocks [122]

Shared memory MIMD architectures can also be programmed via assisted parallelization of sequential programs. OpenMP (Open Multi-Processing) provides directives to control parallelism and data sharing in C, C++, and Fortran [61]. Sieve is a compiler and parallel runtime which runs C++ programs that contain additional *sieve* constructs which mark sections to be parallelized [68]. Intel's Array Building Blocks provides an API (application programming interface), JIT (Just in time) compiler, and multithreaded runtime which allows C++ programs to incorporate various data-parallel array operations [93].

Finally, shared memory can be simulated on distributed memory architectures. Partitioned global address space (PGAS) languages do this on clusters. For example, Co-array Fortran [151], Unified Parallel C (UPC) [37], and Titanium [216, 106], provide distributed arrays that abstract away from explicit communication primitives.



(A) Shared-nothing cluster (B) GPU

FIGURE 2.2: Distributed memory architectures

Message passing Clusters are MIMD with distributed memory, and so require some sort of non-local memory access. This is often via message passing. MPI (Message Passing Interface) [189] is a message passing library specification that is commonly used for HPC (high-performance computing) on clusters. It includes bindings for C [11], C++ [77], and Fortran [3] (among other languages [189]) and has been implemented for many different architectures. It relies on the SPMD model, where processes branch on the rank (ID) of their current node, and provides two-sided (i.e., *send* and *receive*) message

passing primitives, and some collective ones like *broadcast* and *scatter*. SHMEM (Symmetric Hierarchical Memory access) [16] is a similar one-sided interface, that directly puts data into remote memory.

There are also language-integrated message passing approaches, often based on the *actor* programming model [4]. Scala [153] and Erlang [12] both support actors with call-by-value message passing. This can be inefficient due to uneccesary message copying on shared memory. Kilim [191, 190] is similar but implements isolation types (where references cannot *leak*) so that messages can be passed by reference on shared memory. These approaches are not normally used for data-parallelism on clusters. SALSA (Simple Actor Language System and Architecture), however, is a Java-based actor language targeting clusters and Internet wide distributed systems [204].

High-level approaches Finally, some languages abstract away for explicit shared memory access and message passing. We call these *high-level*. Many of these are based on functional programming. For example, MapReduce [66] is a framework for large-scale data aggregations based on a *map* function that projects key-value pairs from a dataset, and a *reduce* function that aggregates a sorted list of values for each key. It abstracts away from explicit communication and load balancing, etc., albeit for a restricted programming model. Languages like SISAL (Streams and Iteration in a Single Assignment Language) [82], NESL [23], Data Parallel Haskell (DPH) [39], and Single Assignment C (SAC) [182] are actually functional programming languages with data-parallel constructs and expressions. Other high-level models are based on query languages. PQL [170] embeds first-order logic queries in Java, and DryadLINQ performs SQL (Structured Query Language) queries on large distributed data sets. Pig Latin [156] translates SQL queries into MapReduce jobs. Parallel databases [117] can also be used to perform large data-parallel tasks. Finally, some imperative languages like Fortran [91] include some imperative data-parallel operations. Fortran and ZPL [132] for example include parallel array operations, and Chapel [40] includes parallel *forall* loops that can manipulate distributed arrays and other collections.

This research focuses on programming clusters. Modern cluster nodes are typically multicores and many include a GPU [202]. Thus programs for clusters may need to combine threads, message passing, and GPGPU programming models. The complexity caused by three nested programming models is a motivation for high-level programming models for such systems. However, current models like MapReduce can be ill suited to many applications. For example, one MapReduce K-means clustering benchmark was 20× slower than an MPI implementation [74]. The former may have been simpler, particularly to support huge (disk-backed) datasets, but its programming model was too restrictive to allow the algorithm to be expressed in its optimal form. It is situations like this, that our work seeks to address— to automatically generate good implementations for a high-level language that can express a wider class of applications than is currently

possible.

2.2 Data-parallel programming on clusters

This section reviews the existing approaches to data-parallel programming clusters, grouped by their main characteristics, limitations, and constraints.

2.2.1 Manually programming communication

The most flexible way to program a cluster is to manually program all communication code using a general-purpose programming language like C or Fortran, and a message passing library like MPI [189] or SHMEM [16].

Message passing interface (MPI) [189] is a library specification for passing messages between processes on a cluster, and is one of the most common ways to program distributed-memory clusters (along with High Performance Fortran [135]). It relies on the "single program, multiple data" (SPMD) model. There are MPI implementations for many different architectures, and bindings for C [11], C++ [77], and Fortran [3] (amongst other languages [189]). There are two main versions of MPI: MPI-1 [189] which is a *two-sided* interface with send calls that must be matched by corresponding receive calls; and MPI-2 [92] which extends MPI-1 with some single sided operations like *put* which directly access foreign nodes' memory without corresponding receives. (MPI-2 also adds support for dynamic process groups, so that additional processes can be spawned at runtime.) We use MPI-1 and C++ as the target language in our work. PVM (Parallel Virtual Machine) [193] is a similar two-sided interface but is much less popular than MPI.

SHMEM [16] is a single-sided message passing library where data is directly *got* from and *put* into, a remote process' memory. This can lead to more efficient implementations, but needs great care to read and write the correct regions of memory and more synchronization to ensure that data is only read and written when ready. It is not used directly, but has been generalized in ARMCI [150] and GASNet [26], which have been used to implement Co-Array Fortran [151], Titanium [216], and Unified Parallel C [37].

Although these approaches are very versatile, they require manual implementation of all data layout, partitioning, and distribution, and lead to very verbose, hard-to-debug implementations (especially with MPI) [137]. For example, Figure A.1 and Figure A.2 in Appendix A compare a single-threaded C implementation of a simple dense matrix multiplication with an MPI implementation. The MPI implementation is 148 lines compared to 8 lines (i.e., 18× longer than the original). The fragmentation of the data space makes dealing with edge cases messy, and choosing the right message passing primitives to maximize performance and avoid deadlock is non-trivial. For example,

subtle issues like the size of MPI's internal buffer can cause deadlock or sequentialization of the computation [99].

Some programming languages directly support remote message passing and spawning remote processes e.g., Erlang [12], Scala [153], SALSA [204], and Java [95] via Java RMI (Remote Method Invocation) [71]. These languages may be slightly simpler and more concise than MPI and SHMEM, but they still force the programmer to manually design all data partitioning, distribution, synchronization, and edge-case handling and so are not a great improvement.

2.2.2 High-level shared memory

To simplify the task of data-parallel programming a number of high-level languages have been developed. Many of these abstract away from explicit parallelism by providing implicitly parallel operations on data structures. However, they all require shared-memory, and are therefore limited to the size and speed of a single multi-core node.

NESL [23, 24, 25] is an applicative language developed in the 90's that specialized in nested data-parallel operations on vector machines. Its main novelty was its support for nested vectors on SIMD architectures, by flattening nested parallelism into segmented vector operations. Data parallelism was expressed using a set-comprehension like syntax similar to set-formers in SETL (Set Language) [184], and list-comprehensions in Miranda [201] and Haskell [158]. For example, {`a*b : a in [3,-4,5]; b in [1,2,3] | a < 4`} reads "in parallel, for each pair (`a`, `b`) drawn from sequence `[3,-4,5]` zipped with `[1,2,3]`, where `a` is less than `4`, return the product of the pair".

Data Parallel Haskell (DPH) [39, 160] is an extension to Haskell [158] based on NESL, but for modern multi-cores. Haskell improves on NESL in several ways. Its implementation is compiled rather than interpreted, it is a higher-order language (i.e., it supports functions as first-class objects), and has an extremely rich type system. DPH is implemented using an extension to the Haskell language, a non-parametric array representation to efficiently store algebraic data-types, and various code transformations including vectorization to flatten nested arrays, fusion to eliminate intermediates, and static scheduling. Both NESL and DPH have the disadvantage that nested data structures are arguably less intuitive for some applications (e.g., 2D stencil convolutions), but have the advantage of good support for variable length data structures (i.e., vectors/lists) and therefore sparse matrices.

Single assignment C (SAC) [182, 98, 97] is a functional language for array-based data-parallelism on multi-cores. The language is a single-assignment subset of C, without pointers or global variables, and supports n-dimensional array operations that are "shape invariant" i.e., polymorphic in the number of dimensions, via a novel type system. Its

implementation features powerful optimizations, including *with-loop folding*— a generalization of the classic map $f \cdot \text{map } g = \text{map } (f \cdot g)$ equation to n-dimensional arrays, which eliminates intermediate arrays in many cases. It therefore generates very high-performance code. For example, one benchmark showed that a SAC implementation of the PDE1 algorithm executed between 2.5 and 4 times faster, with half the memory requirements, of an HPF implementation on a Sun Ultra2 Enterprise 450 architecture [182]. This makes it suitable for high-performance numerical applications, but it is unclear how it could be extended to support other collection types and applications.

Parallel Query Language (PQL) [170] is an extension of Java that supports data-parallel operations based on first-order logic, on shared memory modern multi-cores. Here data-parallel operations are queries on Java collections, which may be optimized and parallelized by the runtime system. It has a lot in common with SQL [143], since SQL was first developed as a language for expressing a subset of first-order logic queries [45], and also depends on a runtime query optimization and planning system, provided by an RDBMS (relational database management system). The current prototype implementation is a front-end compiler integrated with Oracle's `javac`. Initial experiments show parallel performance that scales well with the number of threads, and competes with hand-coded parallel versions, for a number of example programs. PQL's advantages include runtime integration and a more expressive language than SQL, but it not yet clear how intuitive this language is, or whether it supports multi-dimensional arrays in addition to collections like lists and sets.

The key shortfall of these languages, from our perspective, is that none of them support distributed-memory architectures, or (to the best of our knowledge) suggest how such support could be implemented. NESL was designed for SIMD vector machines, and DPH, SAC, and PQL support modern multi-cores, not distributed memory-clusters. They are therefore interesting as high-level languages for data-parallelism, with interesting language constructs, but cannot directly help with automatic (compilation) techniques to target distributed-memory architectures. Furthermore, even as languages, they are all quite restricted in their programming models; a theme that we will elaborate in Section 2.2.4.

2.2.3 Manual data distribution

In this section we review some data-parallel languages that do support distributed-memory architectures, and clusters in particular. Some of these languages are termed "partitioned global address space" or PGAS languages. In these languages the data space of a program is partitioned into different areas that are owned by different processes. Some (early) languages are SPMD-based, specifying what individual processes should do, while others are implicitly data-parallel, specifying the global behavior of all the processes in the system.

Early PGAS Languages Co-Array Fortran [151], Unified Parallel C (UPC) [37], and Titanium [216] are all PGAS languages (or rather language extensions) that were developed in the 90's. They extend Fortran, C, and Java respectively. They are all SPMD-based, meaning that the behaviors of different parallel processes are all specified in a single program. This is normally achieved by branching on the current node's ID to perform actions peculiar to the current process, but this fragments and obfuscates the overall control flow, and can therefore make programs hard to understand. All these languages support some kind of distributed or shared array data structures that are accessed to replace explicit message passing.

Co-Array Fortran extends Fortran 95 [91] with *Co-arrays*, i.e., distributed arrays where each node holds a partition that is referenced via a trailing index to specify the node rank of the partition. These arrays may be regular sized—stored at each node, or irregular sized—allocated dynamically at each node. This relieves the programmer from explicit message passing, but still forces them to handle all data layout and partitioning explicitly, making it non-trivial to implement anything other than a simple 1D blocked partitioning.

Unified Parallel C (UPC) is more nuanced. Scalars and arrays may be declared *shared*, and thus replicated or distributed according to some fixed distribution, respectively. The default distribution is cyclic (i.e., successive elements on successive nodes), but blocked, and blocked-cyclic distributions can also be defined (i.e., blocks of elements stored on successive nodes). This is a great improvement, since it better encapsulates the distribution information such that the programmer does not need to explicitly reference elements by their partition's rank (i.e., node ID). There is also a `upc_forall` loop construct which resembles a parallel `for` loop with a fourth expression which defines where to execute each iteration of the loop. It provides a more global way to program, which avoids branching on node ranks, and is therefore much neater when handling edge cases. However, despite this, UPC still only supports 1D distributions (although an extension to multidimensional blocks has been proposed [17]), and forces the programmer to manually choose what data distributions to use, which requires careful thought to gain good performance, by, for example, ensuring that array distributions are aligned properly.

Titanium is quite different [216]. It does not directly support distributed arrays. References are either local or global, and so (1D blocked partitioned) distributed arrays (like Co-arrays) can be implemented as global arrays, of global references to locally allocated arrays, one per process [106]. However, it does support some collective communication methods— `exchange` and `broadcast`, which perform all-to-all and one-to-many communication, respectively. It also benefits from Java's object orientation, but omits thread support, and is translated into C rather than executing on the JVM, for performance reasons.

DARPA-funded PGAS languages During the 2000's DARPA funded three projects to develop new PGAS languages [206]. These projects produced three implicitly parallel (i.e., non-SPMD) languages: Chapel [40], X10 [47], and Fortress [6], developed by Cray, IBM, and SUN/Oracle respectively. Instead of SPMD, here programs take a global view of the computation—specifying how the whole computation should proceed, instead of explicitly splitting it up into separate tasks.

Chapel [40, 44, 41, 176, 42] evolved from ZPL (Z-level Programming Language) [132, 43], an imperative data-parallel array sublanguage for the Orca MIMD language [15]. ZPL's main novelty was the concept of named multidimensional index sets called *regions*, where parallel array operations are restricted to a given region, and arrays that share a region are automatically aligned, and therefore distributed, in the same way. This improved readability and reliability, where duplicating indices would be error prone. Benchmarks have shown that some programs show competitive performance with MPI equivalents [43].

Chapel improves on ZPL by supporting data-parallel operations on maps and graphs as well as arrays. It does this by extending *regions* (which it calls *domains*) to include *indefinite domains* with indices of arbitrary type for sets and maps, and *opaque domains* with anonymous indices to support graphs. This makes Chapel the most flexible data-parallel language of those we survey, in that it supports all these different collection types. However, to our knowledge these collections must still all be memory-resident (not disk-backed), distributed lists are not supported, and although Chapel includes some built-in and programmable distributions for these collections, they must still be chosen manually by the programmer; no automatic technique to infer good distributions for a given program seems to exist. Furthermore, since Chapel's parallel loop construct (i.e., the `forall` loop) can cause data races, programmers must also manually reason about and choose appropriate synchronizations and lock constructs.

X10 [47, 177] is similar to Chapel, although it only supports distributed arrays and runs on the Java Virtual Machine (JVM) [133]. Its main novelty is a PGAS task-parallel programming model via *activities*, which can be spawned at *places* using `async` statements. It also supports quite a powerful algebra for its *regions* (integer index sets) which includes union, intersection, set-difference, translation, and restriction operations. However, like Chapel, the distributions of regions must still be manually specified.

It is unclear exactly what data-parallel features Fortress [6] had. It appeared to focus on a very concise mathematical syntax, support for different character sets, and combination of features from different languages. However, it was never fully implemented and is now dormant.

MapReduce In 2004 Google published a paper on their propriety large-scale data processing framework called MapReduce [66]. MapReduce and Hadoop [210] (the open-

source Java implementation of MapReduce), are frameworks for performing aggregations on huge datasets, hosted on large-scale clusters. They primarily rely on a *map* function that projects key-value pairs from a dataset, and a *reduce* function that aggregates a sorted list of values for each key. They handle all communication, scheduling, and failure recovery, and so greatly simplify data-parallel programming.

However, their programming model (implementing these two functions) and single distributed implementation thereof, is restrictive and not suitable for all applications. In particular vanilla MapReduce's distributed implementation involves reading input data from unstructured files, storing all inputs, outputs and intermediates on disk, and exchanging all intermediate data over the network. There is no way to omit redundant steps, storing intermediates on disks can stifle the performance of iterative algorithms, and join algorithms must be programmed manually for computations that take multiple inputs [22]. For example, one investigation showed a Hadoop K-means clustering program performed 20× slower than an MPI version [76]. For this reason numerous alternates have been suggested to allow, e.g., iteration [33, 75], different file types [69, 34, 79], accepting multiple inputs [215], removal of intermediate files [76], and supporting different architectures [113, 168, 53]. However, each of these also has a (different) single programming model, and implementation, specialized for one particular task, and so still suffer from the same inflexibility as the original MapReduce. Furthermore, customizing the data distributions must be done manually in all of these, by specifying a non-default partition function for the exchange, and the framework has no knowledge about data layouts and so cannot co-locate *map* and *reduce* jobs to reduce network traffic where possible.

Pig Latin [156, 90] is an SQL-like scripting language for large-scale data processing on clusters that code generates Hadoop jobs. It is widely used by Yahoo!, where users have commented that they find it easier to use than Hadoop [90]. It is a single-assignment language with no loops, recursion, or conditionals, but can be embedded into Java programs in a similar way to JDBC (Java Database Connectivity) external data providers. It does extend SQL's data model slightly, to support nested relations created via the `COGROUP` operation, and could theoretically represent arrays as relations, but it is still in practice unsuitable for array-based and in-memory applications. Its similarity to SQL allows a suite of logical optimizations from relational databases to be applied. However, it does not seem to do any data-layout optimization, and in this way suffers from the same limitations as MapReduce.

Skeletons Algorithmic skeletons [59, 63] are generic and reusable algorithmic patterns for different kinds of parallel computation. They are similar to higher-order functions (HOFs) or *combinators* in functional programming. In fact, a parallel compiler for Standard ML has been developed that extracts parallelism automatically by converting recursive functions into instances of *map* and *fold* combinators, which are then imple-

mented via *processor farms* and *processor trees* respectively [180]. Skeletons have been implemented in different ways, and for different architectures, including using C++ templates for clusters [80, 118], GPUs [78] and the CELL processor [175], and functional programming languages for clusters [2, 134], multicores [126], and various archaic parallel architectures [192]. However, although the choice of skeleton partially determines the data distributions, the programmer must still manually design the distributed data layouts for the program, and try to choose appropriate layouts for the data-parallel task and architecture.

Overall, these languages all abstract away from explicit message passing, and support clusters, but they all still force the programmer to explicitly manage the distributed data layouts. Co-Array Fortran and Titanium still require explicit assignments to and from remote arrays, and the others all require the distribution of data structures to be explicitly selected by the programmer—a difficult task for non-experts. What is needed is some automatic technique to help design and optimize the data distributions of different data-parallel programs for the architectures that they will execute on.

Note that High-performance Fortran (HPF) [135] is similar in some ways to Chapel and X10, but we have chosen to discuss it in Section 2.2.4 since, unlike these languages, a tool to optimize the data distribution directives for a particular program has been developed [115].

2.2.4 Restricted programming models

Finally, some approaches do free the programmer from the burden of manual data layout and distribution, but only for restricted programming models.[1]

High-performance Fortran Traditionally, most high-performance computing applications were programmed with High-performance Fortran (HPF) [135, 114] (or MPI). In 1990 the Fortran 90 [3] specification extended Fortran 77 [32] with element-wise operations on flat multi-dimensional arrays, including array-sections, masked assignment, permutation, and some reductions and scans like `SUM` and `SUM_PREFIX`. These operations are concise, parallelizable, and paved the way for similar syntax in languages like Chapel and ZPL. For example, the statement `a(1:n) = (a(0:n-1) + a(2:n+1)) / 2` assigns the sum of each element's left-hand and right-hand neighbors over 2, for every element in the array `a` between `1` and `n` inclusive.

[1] Note that all of the high-level languages above have restricted programming models as well e.g., nested 1D vectors for NESL and DPH, and multi-dimensional arrays for SAC, UPC, and ZPL etc, but only those in this section are high-level, support clusters, and provide some automatic help with distributed data layouts.

HPF extended Fortran 95 [91] with more data-parallel operations, and directives to specify how to distribute arrays on MIMD supercomputers [135]. These directives were designed for computers with rectilinear processor arrangements, and so the *processors* and *view* directives have limited applicability to modern clusters, although those pertaining to array-alignment are still relevant. Distribution directives were mostly specified manually, but research was done to automatically optimize them for different programs [89, 115, 116]. In particular, two approaches to optimize data distributions based on linear 0-1 integer programming, have been proposed [89, 115], but do not seem to have been widely adopted.

HPF's major limitation is its programming model. It is only really suitable for numerical applications involving arrays, its data-parallel array operations cannot be extended or customized, and scans, reductions, and scatters can only be performed for a few basic numerical and boolean operations like *sum*, and *or*. Applications requiring lists, sets, and maps, disk-backed collections, or permutations based on non-integer keys, like computing an inverted index on a corpus of web pages, are not supported, and it is unclear how such support could be added.

Automatic loop parallelization Much research has been done on auto-parallelizing sequential C/Fortran (or other imperative) programs that consist of nested loops of array accesses, where the array references must often be affine functions of the loop indices [123, 29]. This is generally based on the *polyhedral model* where the loop nests are modeled as polytopes (i.e., polygons generalized to n-dimensions) which can be dissected and thus parallelized. It was originally developed for systolic arrays [169] and shared-memory multiprocessors [9, 8] (i.e., the Stanford DASH multiprocessor [124]), but has since been applied to distributed-memory clusters [96, 57, 27], and modern multi-cores [29]. We are concerned here with its application to clusters, i.e., synthesizing distributed memory implementations from these imperative input programs.

Many of the early papers only addressed one aspect of this problem, like finding a good data-decomposition [128, 167, 100, 9, 49, 48, 121], transforming the computation to minimize synchronization and maximize locality of reference [131, 130, 81], or generating efficient communication code that omits redundant communication [7, 195]. However, since then a number of fully functional end-to-end auto-parallelizing compilers that target distribution-memory clusters have been developed [28, 29, 27]. Automatic data decomposition was usually done either by trying to choose array alignments and partitionings to minimize some execution time/communication cost estimates [127, 128, 49, 100, 9], or by applying affine transformations to the loop nests to produce tilings that minimizes communication between tiles [131, 130, 28, 29]. The former is also how the HPF automatic data layout approaches worked [89, 115]. Both demonstrate very good performance in the benchmarks mentioned, but both techniques rely on the polyhedral model and integer programming, which are only applicable to arrays and loop nests, and often

only a restricted class of these. We are not aware of any work that suggests how this could be generalized to other collection types and data-parallel operations, and so like HPF this work is only applicable to a restricted programming model.

Parallel databases Parallel databases can also be used for some distributed data-parallel tasks [211, 117, 146]. Like Flocc programs, parallel SQL query plans [45] are synthesized by enumerating different combinations of plan operators to minimize the overall cost [185]. This means that they can consider different ways to co-locate a query's input relations, and different data distributions for temporary/intermediate relations. Like Flocc's map combinators, SQL queries are also based on relational algebra, though they have a weak type system, no support for array-based computation, and cannot be extended with new operators. Furthermore, parallel databases typically do not generate standalone code, and the distributed schemas must be designed manually, though a tool to assist with this has been proposed [157].

DryadLINQ DryadLINQ (Language Integrated Query) [112, 74, 217] is a framework similar to PigLatin for cluster computation in .NET languages that also takes SQL-like LINQ queries [142], and optimizes them at runtime to query large distributed datasets. It too has been used to implement scientific analyses like DNA sequencing and high-energy physics simulations, giving significantly better performance than Hadoop equivalents in some situations [74].

Instead of targeting Hadoop jobs, DryadLINQ queries are transformed into a directed-acyclic operator graph, which is then directly optimized and executed on the cluster. This means that DryadLINQ queries can avoid redundant steps, directly implement joins, etc., and optimize queries to improve data co-location of intermediates, e.g., performing rack-level partial aggregation. However, these optimizations are only for a single query; no means is provided to specify or automatically optimize data distributions for query results to take into account later queries and future processing. Furthermore, like parallel databases, DryadLINQ is restricted to the SQL query model, and does not support array-based computations.

SISAL The only functional language reviewed that targets distributed memory architectures is SISAL (Streams and Iterations in a Single Assignment Language) [82]. SISAL is strongly typed, has Pascal-like [212] syntax, and pure semantics (i.e., no side effects) which is ideal for parallelization as it renders data-races impossible. It was implemented via transformation into a dataflow graph (DFG) form (called IF1), which was then optimized to remove intermediates, pre-allocate arrays, and then used to generate code. One implementation synthesized distributed memory implementations and optimized them to minimize non-local data access [179, 178]. It did this by computing

execution-time estimates for IF1 nodes, iteratively expanding them until enough parallelism was revealed, partitioning this expanded graph into tasks, and finally scheduling these to try and minimize non-local data access. This technique successfully targeted distributed-memory architectures, but was very limited since minimizing network traffic was only considered when scheduling tasks, rather than when choosing how to partition the DFG into tasks to start with. Furthermore, SISAL did not support structured data partitionings, alignments, or data replication etc., and so was very limited in its ability to optimize data layout. Finally, its programming model was particularly limited, in that it only supported 1D arrays, and a single data-parallel *for*-expression to range over index spaces access array elements, generate intermediate values, and aggregate them.

These languages and frameworks follow three main approaches to automatic data distribution. For numerical array-based applications, HPF and polyhedral auto-parallelization, techniques either search explicitly for array alignments and partitionings for variables in the program [128, 9, 49, 116, 100], or transform the computation to find a loop-tiling [130, 28, 29], that minimizes the estimated runtime/communication cost. Then parallel databases [211, 117] and DryadLINQ [112, 74, 217] both optimize data layouts and access patterns by trying different combinations of plan operators to implement the high-level operators in the query graphs. This approach not only enumerates different data layouts, but also different local and distributed algorithms to answer the overall query. Finally, SISAL [179, 178] also generates a data flow operator graph, but instead of enumerating different implementations and distributions for each operator, partitions the graph into tasks to try and balance the estimated computational costs, without considering data distribution, and the static scheduler then tries to co-locate tasks to minimize non-local data access. This is the most simplistic of the three approaches.

While all of these approaches perform some automatic distributed data layout selection, all of them are only for restricted data and programming models. For example, none of them work for both array-based (HPF), map-based (SQL), and list-based (SISAL) data-parallel tasks, let alone for other data structures and applications. Furthermore, none of the approaches surveyed show they can be extended with more collection types and data distributions, and are therefore lacking in extensibility. We seek a common approach that works for all of these models, including the collections, operations, and data distributions they require, and that can be extended to support more in the future.

2.3 Basics of type systems

A type system is a means of classifying terms in computer programs, to permit valid behaviors, and exclude some invalid ones. Many type systems exist, for functional and imperative languages, where types are checked statically or at runtime, and where users

manually specify types, or where they may be inferred automatically. This section provides some background on polymorphic functional types, and automatic type inference. The language we define in Chapter 3 uses a polymorphic type system, and a type inference algorithm to reconstruct its types. Furthermore, we use types to encode the distributed data layouts of expressions in this thesis, and variants of standard Hindley-Milner type inference [62] in our technique for automatic data distribution generation.

2.3.1 Simple function types

The core calculus of a functional programming language is the lambda-calculus [56, 119]. Figure 2.3 shows the syntax of the lambda-calculus.

$$t ::= x \mid \lambda x \cdot t \mid t\ t$$

FIGURE 2.3: Syntax of lambda-calculus

A term in the lambda-calculus may either be a variable x, a function abstraction $\lambda x \cdot t$ where t is some term that may reference variable x, or a function application $t\ t$. A function application of the form $(\lambda x \cdot t_1)\ t_2$ evaluates to a term where all instances of x in t_1 have been replaced with t_2, which we denote as $[x \mapsto t_2]t_1$.

A simple type system for the lambda-calculus can be constructed by extending the syntax of function abstractions to include the expected type of their bound variables, as per Figure 2.4, and where the set of types are generated by the grammar in Figure 2.5.

$$t ::= \texttt{True} \mid x \mid \lambda x : T \cdot t \mid t\ t$$

FIGURE 2.4: Simply typed λ-calculus

$$T ::= \text{Bool} \mid T \to T$$

FIGURE 2.5: Simple types

Here a term can either have the type Bool (for a boolean constant `True`), or a function type of the form $T_1 \to T_2$, where T_1 is the type of the function's bound variable x, and T_2 is the type of the terms that the function evaluates to when it is applied, i.e., its result. In order to check whether a term in our simply typed lambda calculus is well typed we need to check that for every function application the type of the function's input variable x, is the same as the type of the term t_2 the function is being applied to. To do this we first construct typing rules that define the typing relation $\Gamma \vdash t : T$, giving terms t types T under the typing environment Γ, so that we can determine the types of terms in our language. Note that Γ is a mapping from variables to types that records the types assigned to the bound variables in the current context. Figure 2.6 shows the typing rules for the typing relation.

$$\frac{}{\Gamma \vdash \texttt{True} : \text{Bool}}\ \text{T-True} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \cdot t_2 : T_1 \to T_2}\ \text{T-Abs} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{T-App}$$

FIGURE 2.6: Typing rules for simply typed lambda-calculus

The axiom T-TRUE, simply says that the term `True` is always of type Bool. T-VAR says that if a binding for the variable x with type T exists in Γ, then x has type T when referenced. T-ABS says if, assuming Γ and additionally that x is of type T_1, we can deduce that $\texttt{t}_2$ if of type T_2, then the abstraction with x marked as type T_1 and expression $\texttt{t}_2$, has type $T_1 \to T_2$. Finally, T-APP says that given term $\texttt{t}_1$ of type $T_1 \to T_2$, and term $\texttt{t}_2$ of type T_1, then the term $\texttt{t}_1$ applied to $\texttt{t}_2$ has type T_2.

This system allows us so type-check functional programs by building the types of terms, using the types of their sub-terms. Then at any function application, if the type of $\texttt{t}_1$ is not an arrow type, or if the type of $\texttt{t}_2$ is not T_1, the function's parameter type, that term is not well typed.

2.3.2 Polymorphic types

A useful extension to the type system presented in the previous subsection, is the introduction of type variables. Damas and Milner [144, 62] presented just such an extension for *parametric polymorphism* and implemented it for Standard ML [104]. In this system types are held abstract during type-checking, and all possible substitutions of concrete types for type variables are held to be well typed.

A naïve implementation of this system would be of little use. For example, say we had a function with type $\alpha \to \alpha$. That function could be used with different concrete types substituted for α, however all uses of the function within a given program would have to use the same substitution for α. To avoid this problem we use *type schemes*. Type schemes extend types with a top level universal quantifier and list of bound variables. Schemes are written $\forall v_1, v_2, \ldots \cdot T$ with bound variables $v_1, \ldots, v_n$ which may be instantiated to different concrete types at different places in a program.

Java generics [31] and C++ templates [203] express similar polymorphism in object-oriented programming languages. Both generics and templates allow classes to be defined that are parameterized by type-variables, which can be instantiated with different classes. In C++ different implementations can be defined for different (partial) specializations of a template-class, so for example a special implementation of the

`std::vector<`T`>` class could be defined, based on bit-vectors, when we know that $T =$ `bool`. This ability, to partially specialize template classes, and pattern-match on them at compile time, actually forms a Turing-complete language [205], which is the basis of template libraries like Boost [1]. We exploit a similar idea in this thesis, where high-level combinators are replaced with different distributed-memory implementations, that are specialized for particular distributed data layout types. However, in our our approach these DDL types are inferred automatically, rather than being specified manually by the programmer, and so many different specializations can be tried and evaluated, without user intervention. This automatic specialization technique forms the basis of our distributed-memory implementation synthesis approach.

2.3.3 Type inference

In addition to type-checking programs with type annotations, it is also possible to automatically reconstruct the types for a particular program by analyzing how values are used in the program. This makes programs more concise whilst retaining the benefits of automatic type-checking. One such type inference system was developed by Damas and Milner for Standard ML called *Algorithm W* [62]. They implemented a kind of parametric polymorphism called let-polymorphism, where type schemes are created at let-bindings, and are instantiated when variables are referenced, by generating a fresh type variable for every bound variable. Type inference proceeds by building up a set of typing constraints for an input program, and then solving these constraints using a unification algorithm. Their system automatically finds the principle types of terms, that is types with the most general instantiations of type variables, provided that the program is well typed.

We use this approach to infer data types for programs in our language in Chapter 3, and extend it to check the validity of different data distributions in Chapter 4.

2.3.4 Dependent types

Dependent types are types where terms or values in the language can enter the types [162]. One such type is written by generalizing the function type $T_1 \to T_2$, to $\Pi x : T_1 \cdot T_2$, where $T_1 \to T_2$ can be thought of as an abbreviation of this latter form, where x does not appear in T_2. This construction is called the dependent product type or "Pi" type, and it allows the function parameter of type T_1 to be referenced using the identifier x in the type T_2. For example, given the type of n-element byte vectors BVector n, the value `empty` has type BVector 0, and the function `cons` has type $\Pi n : \text{Nat} \cdot \text{Byte} \to \text{BVector } n \to \text{BVector}(n+1)$. That is, `cons` takes a natural number, a byte, and a BVector of length n and returns a BVector of length $n+1$.

Type-checking for dependent types is undecidable in general, since deciding whether two types are equal may require deciding whether two programs produce the same result, which is undecidable in general [200]. We use a restricted form of dependent types in Chapter 4, to create distribution types that are parameterized by functions that appear in the input programs.

2.3.5 Unification

A key part of Damas and Milner's Algorithm W [62] is syntactic unification. Unification seeks a set of substitutions that replace (type) variables with other type terms, to unify a set of syntactic constraints, i.e., make the terms on both sides of the constraints syntactically equal. One suitable unification algorithm for Algorithm W is Robinsons' [173]. It proceeds one constraint at a time, discarding the constraint if both sides are equal, creating a substitution replacing a type variable with a value, if one side of the constraint is a type variable, and breaking down composite constraints into new constraints between corresponding parameters, when the number of parameters on both sides matches. This algorithm is sufficient for ML and Algorithm W, but more powerful unification algorithms are required for more expressive type systems.

For example, if types can involve functions (not just function types), a higher-order unification algorithm is required. In Chapter 5 we use a simple approximation for higher-order unification that tries to syntactically unify functions, and if this fails normalizes and then re-compares them. However, in Chapter 6 we present a better approximation of second-order unification that works for projection functions. It uses a first-order equational theorem prover (i.e., the E-prover [183]) with an equational theory for projection functions to find solutions for equations between projection functions. This allows us to solve non-trivial constraints between functions embedded in the type system, and find function implementations for type variables to make terms *semantically* equivalent.

2.4 Auto-tuning by code generation

Auto-tuning is an area of search-based software engineering [103, 101] that uses runtime performance-feedback and code generation to search for and synthesize optimized implementations of different applications. It has been developed as a method to specialize high-performance computations to different target environments. We use it to search for good MPI implementations of data-parallel programs, in Chapter 7 and Chapter 8 of this thesis.

There are currently three classes of auto-tuners. First, self-tuning library generators like ATLAS [208], PhiPAC [20], FFTW [86, 87], and SPIRAL [166, 165] are domain-specific auto-tuners for generating high-performance kernels for BLAS (Basic Linear Algebra

Subprograms), DFTs, and signal processing. Second, compiler-based auto-tuners like CHiLL [51] and PoCC [164] generate and search through alternative implementations of a computation, e.g., different loop tilings and unrollings etc. Third, application-level tuners like Active Harmony [60] and PetaBricks [46, 10] automatically search through different parameter values and code variants proposed by the application programmer. In AI, auto-tuning is also known as algorithm portfolio optimization. It has been applied to many computationally hard problems (e.g., satisfiability solving [214]), but the different optimization methods treat the algorithms as black boxes and only learn optimal schedules and parameter settings for a given set of implementation alternatives. To the best of our knowledge ours is the first work to apply these techniques to data distribution on clusters.

In this section we review the some of the ways that auto-tuners represent their search spaces, and the search algorithms they use to find solutions.

2.4.1 Search space representations

Operator graphs/trees Spiral [166, 165] generates optimized implementations of signal processing algorithms. These were originally just FFT (Fast Fourier transform) and trigonmetric transforms [5], but now include other linear transforms and non-transform kernels, like matrix multiply, circular convolution, and Virterbi decoding [84]. All of these algorithms can be implemented by recursively composing other transforms, often for smaller vectors. Spiral's search space therefore consists of the possible operator trees that can be derived from a formal grammar, defined by concrete transform implementations as terminals, and rewrite rules which replace a transform with an expression composing other transforms, as non-terminals. This work has also been extended to generate simple MPI implementations of discrete Fourier transforms (DFTs) by including *parallelized transforms* [30], and SIMD/multi-core implementations of transforms for the Cell BE processor [50]. However, these only supports 1D blocked distributions, and all permutations perform expensive all-to-all communication.

FFTW generates optimized Fast Fourier Transform (FFT) implementations [86, 87]. It produces *plans* by composing *codelets*, which are highly optimized composable blocks of C that compute part of the transform. Like Spiral, solutions are found by recursively decomposing the FFT into combinations of codelets that perform DFTs of fixed size, and codelets that combine the results of other codelets. The planner uses dynamic programming and actual performance-feedback to search for the fastest plans.

Code transformations Some auto-tuners directly transform code. ATLAS [209, 207] generates optimized implementations of the BLAS linear algebra library. It initially probes for information like FPU pipeline depth, number of registers, and cache size, and

then optimizes loops to try and find the best blocking factor and loop unrolling etc. PhiPAC applies similar techniques to general ANSI C programs [20].

Tunable parameters Active Harmony [60] and Gunther [129] tune numerical and discrete parameters respectively. Gunther optimizes configuration parameters for MapReduce installations. Active Harmony tunes parameters in input programs that have been specified by the programmer. It has also been combined with the CHiLL compiler transformation framework [52], to select and optimize loop transformations (e.g., permutations, tilings, unrollings) like the code transformation approaches above [197].

2.4.2 Search algorithms

Exhaustive search Exhaustive search is the simplest search algorithm. It uses a traversal, like depth-first or breadth-first to enumerate all candidate solutions in the search space. It is only feasible where the search space is small enough to be completely enumerated in reasonable time. Shin et al. [186] use an exhaustive search with the CHiLL loop transformation framework [54] to optimize the Nek5000 spectral element code, although they use heuristics to prune the search space for operations involving larger matrices.

Dynamic programming Dynamic programming [18, 19] optimizes sub-problems locally, independent of the larger context. It can only be used when problems can be broken down into sub-problems (e.g., trees), and it is not guaranteed to find the fastest plan, since it is based of the assumption that "the best solution is found by composing the best solutions of its sub-problems", which does not always hold. FFTW uses dynamic programming as its search algorithm [87], and Spiral implements it, finding that it gives "reasonably fast" solutions, though not as fast as those returned by STEER—their genetic search algorithm. Since this algorithm tends to evaluate the same sub-problem multiple times, implementations tend to cache feedback, so that performance evaluations are not repeated [166, 87].

Random search Random search algorithms evaluate candidate solutions at random from the search space. This often gives poor results. It can also be non-trivial to draw candidates uniformly from the search space. For example, Spiral just selects a random rule at each expansion step, which does not lead to a uniform distribution [166].

Evolutionary / Genetic algorithm Genetic algorithms [85, 64, 109] are very good at searching for solutions to unstructured problems. They iteratively improve a population of candidates, starting with a random population, from which the best n solutions are

used to generate a new population of solutions, from which a new population is created, and so on. Each candidate solution is stored as a list of integers, where each integer models a gene. A new population is created by applying a *crossover* operation to random pairs of parents, which creates a child by randomly taking each gene from either the first or second parent, respectively. A *mutation* operation is then often applied to children, which randomly replaces part of the child with a random value, with a given probability. Spiral [166] and GUNTHER [129] use genetic searches to auto-tune linear transforms and optimize MapReduce configurations respectively. Spiral uses STEER [187], an unusual tree-based genetic search. STEER's crossover operation swaps different implementations of the same node to crossover, and its mutation either replaces a subtree with a random one, or swaps, or copies, two different implementations of the same node within the tree. Apart from exhaustive search, this algorithm finds Spiral's fastest solutions.

Genetic searches have also been used to synthesize code, by generating different programs from a grammar, and evaluating how well they solve the target problem [102], i.e., evolving a CUDA kernel implementing part of gzip [120].

Integer parameter tuning A class of search algorithms called *direct search algorithms* [110] can be used where tunable parameters are numerical values. Active Harmony [60] originally used a direct search algorithm called the Nelder-Mead simplex method [155]. This method represents a solution as a simplex—an n-dimensional polytope with $n+1$ points. The first $n+1$ iterations use performance-feedback to get values for the points in the initial simplex. Then each subsequent iteration discards the *worst* point, and replaces it with a new one that has a lower value. This algorithm is designed for a continuous space, and so had been modified in Active Harmony, to take the nearest integer to each point.

More recently, Active Harmony has been updated to use another kind of direct search algorithms, called *rank ordering algorithms* [194, 198, 197]. This is because the simplex method became unpredictable when the number of variables increased, and because it cannot be parallelized [198]. Rank ordering algorithms also start with an initial simplex, but after this, the simplex is either reflected, expanded, or shrunk around its best point. These algorithms have better convergence properties than Nelder-Mead, and can be parallelized.

Bandit-based graph optimization De Mesmay et al [65] find fast implementations of recursive algorithms (e.g., Fourier transforms like in Spiral) represented by directed-acyclic graphs (DAGs) drawn from a formal grammar, using an online search algorithm called Threshold Ascend on Graph (TAG). This algorithm grows subgraphs (derivations of the grammar) by considering local bandit problems and evaluating nodes using Monte-Carlo simulations when real performance feedback is not available (i.e., when the node

is not a vertex). When a node is evaluated via performance-feedback, a corresponding reward is back-propagated to all its ancestors. This has been shown to find good solutions much faster, and sometimes find a better solution, than dynamic programming in some cases [65].

Application specific search algorithms Some auto-tuners use application specific searches that progressively try to optimize different features. For example, ATLAS tries all options for a given parameter, choosing the best, whilst keeping all other parameters fixed [209]. PhiPAC [20] uses a similar approach.

2.5 Conclusions

In this chapter we have surveyed the major technologies for data-parallel programming and provided some background on type systems and auto-tuning.

Data-parallel programming languages The data-parallel programming languages surveyed all balance three criteria: abstraction, flexibility, and performance. Raising the level of abstraction can be hugely beneficial, making code much more concise, intuitive, portable, and accessible for non-expert programmers. Abstracting away from concerns like data layout, synchronization, and communication make languages much easier to use and port to different architectures, but can limit their applicability and make generating efficient code difficult. The main languages surveyed all raise the level of abstraction, but do so in different ways which usually reduce flexibility and performance.

MPI needs a high level of skill, but is flexible and can yield high performance programs. However, it requires long complicated code and is not suitable as a general purpose paradigm as it is too tied to flat, shared-nothing parallelism.

The SPMD PGAS model (Co-Array Fortran, UPC, Titanium) still requires a high level of skill, manual data decomposition, and synchronization, but abstracts away from explicit message passing. It is less flexible than MPI and languages like Chapel, due to its restricted data structures, but can give high performance. It is a more portable model than MPI, supporting shared and distributed memory, but still cannot exploit architectures with nested parallelism (i.e., clusters of multicores with GPUs).

The imperative data-parallel model (HPF, ZPL, Chapel, X10) is more concise and intuitive as it takes a global view, but requires specialist architectural knowledge to design data layouts. It is mainly limited to numerical HPC, but can yield high performance. It could target diverse architectures, in some cases with nested parallelism. However, in its current form it is too inflexible to be suitable as a general purpose paradigm and designing data layouts makes it inaccessible and hard to optimize and port.

The applicative data-parallel languages (SISAL, NESL, DPH, SAC) are concise and intuitive, taking a similar global perspective, though the functional paradigm may be unfamiliar to some. There are no distributed memory implementations apart from SISAL, but the paradigm is very convenient for data dependency analysis and SISAL demonstrates that it can be automatically partitioned to target different architectures without modification. In some ways it is more flexible than flat models, supporting nested data structures and parallelism, but still cannot easily express operations like grouped aggregations. It is also harder to achieve good performance (except perhaps with SAC).

Query-based models (MapReduce, SQL, DryadLINQ) are very concise and accessible to non-experts. Programs can be very portable, targeting shared nothing clusters and multi-core PCs, and yielding good performance. Operations are generally limited to grouped aggregations and other data flow operations, but theoretically computations involving multidimensional arrays could also be expressed. These models arguably have had the most success in making distributed memory parallelism accessible to non-experts, but cannot currently be used to efficiently implement numerical high performance computations.

The only examples of automating data layout decisions found, are with HPF [115], polyhedral loop parallelization [27], query languages [117, 112], and SISAL [179]. They all yielded promising results, but HPF and SISAL used archaic architectures, and all four are restricted by their base languages. The best attempt to unify the different data models is Chapel [40], but its imperative semantics may make powerful data dependency analyses of the kind needed for automatic data distribution optimization difficult. Furthermore, it is still unsuitable for large data processing tasks, since unlike MapReduce, it does not support processing huge disk-backed collections.

In general portable distributed memory programming is a hard task, and is becoming increasingly necessary. There is a need for tools to assist with this process, for a wider class of data-parallel algorithms, and for heterogeneous architectures. In particular the task of assisting the programmer with the data partitioning and layout decisions required for modern distributed memory architectures, for a wider variety of data-parallel operations and collection types, seems to have been somewhat neglected, and is thus the focus of this thesis.

Basics of type systems Our brief introduction to type systems has shown that types are a convenient way to annotate programs with meta-data, that can characterize some properties about program terms, to exclude "bad" behaviors. Such systems can be proved correct, or at least consistent, formally, and then implemented such that the types of program terms can be checked mechanically by a compiler. The types for some languages are polymorphic (i.e., involve variables), involve program terms (i.e., dependent types), and can be automatically recovered using type inference. These three

features are especially useful in our context, and allow us to express generic distributed data layouts, parameterized by program terms, that can be automatically inferred for candidate implementations of data-parallel programs, in Chapters 4 to 6 of this thesis.

Auto-tuning by code generation The auto-tuning work reviewed in this chapter, demonstrates how performance-feedback-based search algorithms can be used to automatically synthesize efficient implementations of various programs. The tools mentioned all yield implementations in reasonable time, that give impressive performance, and which are portable, since they can be tuned to accurately reflect a new architecture's performance characteristics. The tools are able to yield better performing implementations than traditional compilers, since they empirically measure how the architectures perform, rather than relying on simplified models of their performance characteristics. Most of the tools work for a fixed set of applications, typically numerical transforms [86, 209, 166], though some are more versatile [20]. However, to our knowledge, no work currently exists that auto-tunes the data layouts and communication patterns of distributed-memory MPI programs, or supports a full functional programming language, and so our implementation in Chapter 7 (which is evaluated in Chapter 8), is the first to apply auto-tuning in this context.

Chapter 3

A Functional DSL for Data Parallelism

This chapter presents Flocc (Functional language on compute clusters), a functional domain specific language (DSL) for data-parallelism. Flocc is a simple core functional language with data-parallel combinator functions (i.e., higher-order functions) for array, map, and list collections. We use Flocc as an input language and substrate for our automatic distributed memory implementation synthesis technique.

3.1 Introduction

The Flocc language is not a contribution per se, but a small core language that we use as the input for our distributed memory implementation synthesis approach. We decided to use a functional core language with data-parallelism expressed via combinators for a number of reasons.

Firstly, using a small core language that omits unnecessary features, is much more manageable for a research project, than using a fully fledged existing language. It makes it more feasible to implement a toolset, and makes the type system(s) small enough to verify formally. On the other hand, this did not limit the usefulness of the language, since we have been able to express many common data-parallel algorithms in Flocc without difficulty.

Using combinators for data-parallel operations allows us to support operations for multiple collection types within a small language, without needing multiple application-specific data-parallel constructs. This keeps the language clean and simple, since it allows us to support these different operations in a unified way. In addition to supporting combinators, using a functional language supports polymorphic types, and type inference to infer them automatically, so we can omit type declarations. Not only does

this make programs more concise, it also allows us to automatically infer distributed data layouts by treating them as types as shown in Chapter 4. Furthermore, using a functional language allows us to convert programs into a data flow form, that makes it easier to analyze them, and code generate from them, as shown in Chapter 7. In fact, using a functional language is quite natural for any kind of data distribution synthesis technique, since they are particularly suitable for the kind of data flow analyses that are needed.

One possible limitation of our implementation of Flocc is that we require that lambda abstractions are statically resolvable at combinator function applications, so functions are not fully first-class. This allows us to inline the implementations of these abstractions, so that we can generate fast imperative code to run on the cluster. In practice this does not seem to be a limitation, as we have written many common algorithms without the need for functions to be fully first-class. It makes sense that HPC applications would not use this feature, due to the performance overhead it causes. Another possible difficulty is performing copy-elimination during code generation, such that updates that can be, are performed in-place. We have been able to avoid this issue for the most part by supporting value-streams and iterators as local storage modes for collections. Furthermore, SISAL uses a technique to address this for data-flow graphs, which could be applied in our situation [82].

3.2 Syntax

Flocc is essentially a polyadic lambda calculus [145], i.e., a lambda calculus with tuples. However, unlike in the lambda calculus, in Flocc functions are not fully first class. Instead, at every function application the compiler must be able to statically determine what function abstraction is being applied. This allows such lambda abstractions to be inlined, and lifted into the data distribution type language during data distribution planning (see Section 4.3.2). It also simplifies the compiler's implementation.

Flocc's expression and type syntax are shown in Figure 3.1. Throughout this thesis we use T_α to denote the syntactic domain produced by the non-terminal α. Expressions e can be identifiers, scalar literals l, tuples, collection literals, function abstractions, function applications, let bindings, or if-then-else expressions. Function abstraction arguments and let expressions bind values to tuples of identifiers x. Values v are expressions in weak head normal form. That is, they can be scalar literals l, tuples of values, lists of values, and lambda-abstractions with bodies which may be redexs (i.e., reducible expressions) or values. We only support list-literals in the syntax, since arrays and maps can be generated from lists using `listToArr` and `listToMap` (cf. Appendix B).

$$
\begin{aligned}
e &::= Id \mid l \mid \texttt{(}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\ \texttt{)} \mid \texttt{[}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\ \texttt{]} \\
&\mid \texttt{\textbackslash}x\ \texttt{[::}\ t\ \texttt{] ->}\ e \mid e_1\ e_2 \\
&\mid \texttt{let}\ x\ \texttt{[::}\ t\ \texttt{] =}\ e_1\ \texttt{in}\ e_2 \\
&\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
\\
v &::= l \mid \texttt{(}\ v_1\texttt{,}\ \ldots\texttt{,}\ v_n\ \texttt{)} \mid \texttt{[}\ v_1\texttt{,}\ \ldots\texttt{,}\ v_n\ \texttt{]} \\
&\mid \texttt{\textbackslash}x\ \texttt{[::}\ t\ \texttt{] ->}\ e \\
\\
x &::= Id \mid \texttt{\_} \mid \texttt{(}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{)} \\
\\
l &::= Int \mid Float \mid String \mid \texttt{True} \mid \texttt{False} \mid \texttt{()} \\
\\
s &::= \forall Id{\cdot}s \mid t \\
\\
t &::= Id \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{String} \mid \texttt{Bool} \mid \texttt{Null} \\
&\mid (t_1,\ \ldots,\ t_n) \mid t_1 \rightarrow t_2 \\
&\mid \texttt{Map}\ t_1\ t_2 \mid \texttt{Arr}\ i\ t \mid \texttt{List}\ t \\
\\
i &::= \texttt{Int} \mid (i_1,\ \ldots,\ i_n)
\end{aligned}
$$

FIGURE 3.1: Flocc expression and type syntax

3.3 Semantics

At the high-level (i.e., executed on a single processor with a single address space), we have given Flocc a call-by-value reduction semantics, since this corresponds to the eager evaluation strategy of our generated imperative code. The reduction rules for this are shown in Figure 3.2. These rules reduce expressions $e \in T_e$ into values $v \in T_v$, which are in weak head normal form.

The first six rules reduce expressions applied to values. They take values rather than expressions, in order to implement the call-by-value evaluation order, by forcing arguments to be fully evaluated before their parent expressions. E-APP performs standard beta-reduction [55], reducing lambda abstraction applications by substituting every occurrence of their bound variable x with the argument value v_1. E-LET does the same for `let`-expressions. E-IFTRUE and E-IFFALSE reduce `if`-expressions with constant predicate values `True` and `False` to just their `then`-clause or `else`-clause respectively. E-LETTUP and E-APPTUP reduce `let`-expressions and lambda applications that bind tuples of variables, to nested `let`s and abstraction applications respectively, which bind one variable at a time. These allow expressions that bind tuples of variables to be broken down so that they can be reduced by E-LET and E-APP respectively.

The other six rules allow subexpressions within expressions to be reduced, to transform them into values, which in turn allows the other rules to be applied. E-APP1 and E-APP2 allow functions and argument expressions respectively, to be reduced in function application expressions. E-LET1 allows `let`-bound expressions to be reduced, E-IF

$$\frac{}{\texttt{(\textbackslash}x\ \texttt{->}\ e_0\texttt{)}v_1 \rightsquigarrow [x \mapsto v_1]e_0}\ \text{E-App}$$

$$\frac{}{\texttt{let}\ x = v_0\ \texttt{in}\ e_1 \rightsquigarrow [x \mapsto v_0]e_1}\ \text{E-Let}$$

$$\frac{}{\texttt{if True then}\ e_0\ \texttt{else}\ e_1 \rightsquigarrow e_0}\ \text{E-IfTrue}$$

$$\frac{}{\texttt{if False then}\ e_0\ \texttt{else}\ e_1 \rightsquigarrow e_1}\ \text{E-IfFalse}$$

$$\frac{}{\texttt{let}\ (x_0, \ldots, x_n) = (v_0, \ldots, v_n)\texttt{in}\ e \rightsquigarrow \texttt{let}\ x_0 = v_0\ \texttt{in} \ldots (\texttt{let}\ x_n = v_n\ \texttt{in}\ e)}\ \text{E-LetTup}$$

$$\frac{}{\texttt{(\textbackslash}(x_0, \ldots, x_n)\ \texttt{->}\ e\texttt{)(}v_0, \ldots, v_n\texttt{)} \rightsquigarrow \texttt{(\textbackslash}x_0\ \texttt{->}\ \ldots\texttt{(\textbackslash}x_n\ \texttt{->}\ e\texttt{)}v_n \ldots\texttt{)}v_0}\ \text{E-AppTup}$$

$$\text{E-App1}\ \frac{e \rightsquigarrow e'}{e\ e_0 \rightsquigarrow e'\ e_0} \qquad \frac{e \rightsquigarrow e'}{e_0\ e \rightsquigarrow e_0\ e'}\ \text{E-App2}$$

$$\frac{e \rightsquigarrow e'}{\texttt{let}\ x = e\ \texttt{in}\ e_1 \rightsquigarrow \texttt{let}\ x = e'\ \texttt{in}\ e_1}\ \text{E-Let1}$$

$$\frac{e \rightsquigarrow e'}{\texttt{if}\ e\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \rightsquigarrow \texttt{if}\ e'\ \texttt{then}\ e_0\ \texttt{else}\ e_1}\ \text{E-If}$$

$$\frac{e \rightsquigarrow e'}{(e_0,\ \ldots, e,\ \ldots, e_n) \rightsquigarrow (e_0,\ \ldots, e',\ \ldots, e_n)}\ \text{E-Tup}$$

$$\frac{e \rightsquigarrow e'}{[e_0,\ \ldots, e,\ \ldots, e_n] \rightsquigarrow [e_0,\ \ldots, e',\ \ldots, e_n]}\ \text{E-List}$$

FIGURE 3.2: Flocc interpreter reduction rules

allows `if` predicates to be reduced, and E-TUP and E-LIST allow subexpressions of tuples and list literals to be reduced respectively.

All parallelism in Flocc is expressed via data-parallel operations applied to the collections. These operations include predefined combinators for arrays, maps, and lists shown in Figure 3.6; there are many further combinators not shown here for brevity (cf. Appendix B for a full list). Flocc therefore effectively has two language levels or sublanguages: a lambda calculus, and a set of primitive combinators. The semantics for the former is defined in via the reduction rules in Figure 3.2, but we do not define any semantics for the latter, although this difference is not reflected in the syntax (both rely on function applications). This distinction is deliberate, since we want to explicitly abstract away from sequential execution and individual element accesses, or some sort of super-combinator, so that there is some flexibility in the way that these data-parallel combinators are defined, so that we can choose different concrete implementations of

them at compile time. Another reason is, that although we could define the high-level semantics for these combinators, we want them to be extensible, eventually allowing power-users to define their own combinators perhaps by means of a scripting language, and so any semantics given would not be exhaustive.

We have deliberately omitted `letrec` recursive `let` expressions from the current version of Flocc, and instead provide built-in combinators for iteration (i.e., `while` and `loop`). This is for a number of reasons. Firstly, these iteration combinators are sufficient for all the example programs in this thesis, and so primitive recursion is not currently needed. Secondly, it simplifies the implementation of the code generator, so that we can focus on the core data distribution synthesis problem, for example, by allowing us to ignore detection of tail-recursion, since all loops are explicit in the code. It also encourages users to use iteration rather than recursion, and means that we can explore different implementations of the iteration combinators, for example, with different loop unrollings, in the same way that we explore different implementations of the other combinators. Thirdly, it aids analysis, since if we ignore evaluation of the built-in combinators, all expressions can be reduced to a normal form in finite time, since without primitive recursion, evaluation always terminates. We use this feature when testing for function equality in our DDL type inference system in Chapter 5.

Having said this it would be quite straightforward to add `letrec` to Flocc, if desired. The standard `letrec` typing rule used in ML-like languages [104] could easily be added to the type systems in Section 3.4 and Chapter 5, and the standard evaluation rule could be added to the semantics in Figure 3.2. The only slightly harder task would be to extend the code generator to support it, since it is currently based on traversing an acyclic data flow graph. However, this could be done by either allowing cycles in the graphs, or by using a special node for recursive calls. The generated code could then reside in a C++ function, which could be invoked recursively as required.

Finally, primitive recursion is technically possible syntactically in Flocc, by means of a fixed-point combinator like the Y-combinator [36]. However, such combinators are ruled out by the type system (cf. Section 3.4), since either recursive types [161] or a polymorphic calculus like System F [161] are required to infer types for such expressions.

3.4 Types

Flocc's type system is based on standard Hindley-Milner let-polymorphic types [108, 144]. Scalar types are integers, floating point numbers, Boolean values, strings, the null type, and compositions of these as n-ary tuples (cf. Figure 3.1). Function types (denoted by $t_1 \to t_2$) have domain t_1 and codomain t_2 type parameters, and the `Map` t_k t_v, `Arr` t_i t_v, and `List` t_v collection types, have type parameters t_k for map keys, t_i for array indices, and t_v for values. As per standard let-polymorphic types, type schemes are types

$$
\begin{array}{lcl}
ftv(X) & = & \{X\} \\
ftv(\forall X.T) & = & ftv(T) \setminus \{X\} \\
ftv(T_1 \to T_2) & = & ftv(T_1) \cup ftv(T_2) \\
ftv((T_1,\ \ldots,\ T_n)) & = & \bigcup_{k=n}^{k=1}\ ftv(T_k) \\
ftv(\texttt{Map}\ T_1\ T_2) & = & ftv(T_1) \cup ftv(T_2) \\
ftv(\texttt{Arr}\ T_1\ T_2) & = & ftv(T_1) \cup ftv(T_2) \\
ftv(\texttt{List}\ T) & = & ftv(T)
\end{array}
$$

FIGURE 3.3: Definition of ftv (Free type variables)

with a top level for-all quantified list of type variables. Type schemes allow polymorphic functions, i.e., functions that can be instantiated with many different concrete types. The types that follow frequently omit the variable lists for brevity, in which case all lower case identifiers are taken to be implicitly for-all quantified type variables. Note that for reasons of performance and simplicity, we restrict type variables to range over non-function types only. This leads to a clear separation in the types between functions and values, where parameters of type $\alpha \to \beta$ (and similar) must be passed functions, and parameters of type α must be passed scalars, tuples, or collections. This does not prevent users defining functions that take other functions as parameters, but means that the code generation templates for our built in combinators do not need to support storing functions in collections, or transmitting functions as objects over the network. The only limitation this places on the flexibility of Flocc, is that separate combinators would be required to store and transmit functions in this way, with explicit arrow types for their arguments.

Flocc uses Damas-Milner's type inference algorithm W [62] to infer types for all expressions, though function abstractions and let-bindings support optional type declarations for the compiler to check. Figure 3.4 shows the inference rules for this system. Here Γ is a mapping between variables and types, that stands for the current typing environment, where $\Gamma \oplus x : t$ (which we abbreviate to $\Gamma; x : t$) overwrites the mapping for x in Γ (if one exists) with a new mapping to type t. T ranges over types, X over type variables, e over expressions, f over functions, and x over identifier patterns. n, b, d, and s also range over integer, boolean, floating point, and string values respectively. $\bar{X}$ stands for a possibly empty set of type variables, and ftv returns the set of all free type variables in a type (or typing context), as shown in Figure 3.3.

As in standard let-polymorphism, type schemes are created at let-bindings, by quantifying over all free type variables in the type that do not appear free in the context. TY-LET implements this by getting all free type variables $\bar{X}$ in T (that do not appear free in Γ), and using these to create a type scheme for variable x. Type schemes are then instantiated at variables by creating fresh type variables for each variable bound by the forall-quantifier of the scheme. TY-VAR implements this by replacing each occurrence of

$$\frac{}{\Gamma \vdash () : \texttt{Null}}\ \textsc{Ty-Null}$$

$$\frac{}{\Gamma \vdash n : \texttt{Int}}\ \textsc{Ty-Int} \qquad \frac{}{\Gamma \vdash b : \texttt{Bool}}\ \textsc{Ty-Bool}$$

$$\frac{}{\Gamma \vdash d : \texttt{Float}}\ \textsc{Ty-Float} \qquad \frac{}{\Gamma \vdash s : \texttt{String}}\ \textsc{Ty-String}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash (e_0,\ \ldots,\ e_n) : (T_0,\ \ldots,\ T_n)}\ \textsc{Ty-Tup}$$

$$\frac{\Gamma \vdash e_0 : T \qquad \ldots \qquad \Gamma \vdash e_n : T}{\Gamma \vdash [e_0,\ \ldots,\ e_n] : \texttt{List}\ T}\ \textsc{Ty-List}$$

$$\frac{\Gamma; x : T_0 \vdash e : T_1}{\Gamma \vdash \backslash x \to e : T_0 \to T_1}\ \textsc{Ty-Abs}$$

$$\frac{\Gamma \vdash f : T_0 \to T_1 \qquad \Gamma \vdash e : T_0}{\Gamma \vdash f\ e : T_1}\ \textsc{Ty-App}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T}\ \textsc{Ty-If}$$

$$\frac{\begin{array}{c} Y_1,\ \ldots,\ Y_n \text{ are fresh vars} \\ \Gamma(x) = \forall X_1,\ \ldots,\ X_n \cdot T \end{array}}{\Gamma \vdash x : [X_1 \mapsto Y_1\ \ldots,\ X_n \mapsto Y_n]T}\ \textsc{Ty-Var}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 : T \\ \bar{X} = ftv(T) \setminus dom(\Gamma) \qquad \Gamma; x : \forall \bar{X} \cdot T \vdash e_1 : T' \end{array}}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : T'}\ \textsc{Ty-Let}$$

FIGURE 3.4: Flocc typing rules

```
eq   :: (a, a) -> Bool

not  :: Bool -> Bool                 divf  :: (Float, Float) -> Float
and  :: (Bool, Bool) -> Bool         mulf  :: (Float, Float) -> Float
or   :: (Bool, Bool) -> Bool         addf  :: (Float, Float) -> Float
                                     subf  :: (Float, Float) -> Float
addi :: (Int, Int) -> Int            minf  :: (Float, Float) -> Float
subi :: (Int, Int) -> Int            maxf  :: (Float, Float) -> Float
muli :: (Int, Int) -> Int            negf  :: Float -> Float
divi :: (Int, Int) -> Int            randf :: Null -> Float
modi :: (Int, Int) -> Int            sqrt  :: Float -> Float
mini :: (Int, Int) -> Int
maxi :: (Int, Int) -> Int            eqf   :: (Float, Float) -> Bool
negi :: Int -> Int                   gtf   :: (Float, Float) -> Bool
                                     gtef  :: (Float, Float) -> Bool
eqi  :: (Int, Int) -> Bool           ltf   :: (Float, Float) -> Bool
gti  :: (Int, Int) -> Bool           ltef  :: (Float, Float) -> Bool
gtei :: (Int, Int) -> Bool
lti  :: (Int, Int) -> Bool           toFloat :: Int -> Float
ltei :: (Int, Int) -> Bool           toInt   :: Float -> Int
```

FIGURE 3.5: Scalar library function types.

a quantified type variable X_k with a unique fresh variable Y_k. Here $[X \mapsto Y]T$ replaces all occurrences of X in T with Y.

These rules are a straightforward application of the ML/Hindley-Milner system. We therefore inherit the type safety results of *progress* (i.e., well-typed programs can progress) and *preservation* (i.e., evaluations of well-typed programs are well-typed) from [161]. Similarly, since our type inference algorithm is a straightforward application of *algorithm W*, extended to deal with new constructs (i.e., tuples, `let`-expressions, `if`-expressions and list expressions), we inherit the soundness and completeness properties from [62] (and of a constraint-based variant from [161]).

3.5 Library functions

This section introduces some of Flocc's built-in library functions. This list is not exhaustive, and more functions can be added to the language by declaring their types and implementing code generation templates for them (cf. Chapter 7). Figure 3.5 shows

```
subArr          :: (i, i, i, Arr i v) -> Arr i v
shiftArrR       :: (i, Arr i v) -> Arr i v
shiftArrL       :: (i, Arr i v) -> Arr i v
scaleArr        :: (i, Arr i v) -> Arr i v
mapArrInv       :: (i->j, j->i, (i,v)->w, Arr i v) -> Arr j w
eqJoinArr       :: (i->k, j->k, Arr i v, Arr j w) -> Arr (i,j) (v,w)
groupReduceArr  :: (i->j, (i,v)->w, (w,w)->w, w, Arr i v) -> Arr j w
unionArrWith    :: ((v,v)->v, i->v, Arr i v, Arr i v) -> Arr i v

map             :: ((i,v)->(j,w), Map i v) -> Map j w
mapInv          :: ((i,v)->(j,w), (j,w)->(i,v), Map i v) -> Map j w
eqJoin          :: ((i,v)->k, (j,w)->k, Map i v, Map j w) -> Map (i,j) (v,w)
allPairs        :: ((i,v)->k, Map i v) -> Map (i,i) (v,v)
reduce          :: ((i,v)->s, (s,s)->s, s, Map i v) -> s
groupReduce     :: ((i,v)->j, (i,v)->w, (w,w)->w, Map i v) -> Map j w
filter          :: ((i,v)->Bool, Map i v) -> Map i v
union           :: (Map i v, Map i v) -> Map i v
intersect       :: (Map i v, Map i w) -> Map i v
diff            :: (Map i v, Map i w) -> Map i v
countMap        :: Map i v -> Int
intRangeMap     :: (Int,Int,Int) -> Map Int ()

zip             :: (List v, List w) -> List (v,w)
take            :: (Int, List v) -> List v
mapList         :: (v->w, List v) -> List w
filterList      :: (v->Bool, List v) -> List v
reduceList      :: ((v,v)->v, v, List v) -> v
listToMap       :: List (i,v) -> Map i v
length          :: List v -> Int

while           :: (s -> (s, Bool), s) -> s
loop            :: (s -> s, s -> Bool, s) -> s
```

FIGURE 3.6: Predefined data-parallel combinators for arrays, maps, and lists.

the types for some of Flocc's scalar functions and Figure 3.6 list the types of some of the data-parallel combinators. We also use the standard symbolic numeric and Boolean operators as sugars for their respective scalar functions.

The first block of combinators are for combining and transforming arrays. `subArr`(i_1, i_2, i_3, a) returns the array section of a between indices i_1 and i_2 (stride i_3), and `shiftArrR`(i, a) and `shiftArrL`(i, a) transpose the indices of array a by the offsets in i, in the positive and negative directions respectively. `scaleArr`(s, a) scales/multiplies the indices of array a by the integer factors in s. `mapArrInv`(f, f_{inv}, g, a) applies f and g to the indices and values of a, and `eqJoinArr`(f, g, a, b) returns the Cartesian product of a and b restricted to where the indices returned by functions f and g respectively are equal. `groupReduceArr`(f, g, h, v_0, a) returns a new array formed by projecting new indices and values from a using f and g respectively, and aggregating values using the binary operator function h (which must be associative), where v_0 is the starting value. `unionArrWith`(f, f_0, a, b) returns a new array with indices that range from the minimum index of a and b to the maximum, where overlapping values are combined using f and undefined values are initialized using f_0.

The next block lists the types of some map combinators. The `map`, `mapInv`, `eqJoin` and `groupReduce` combinators are similar to their array equivalents. `allPairs`(f, m) groups values of m using the keys returned by f, returning all 2-combinations of values for each group. `reduce`(f, g, v_0, m) aggregates values returned by applying f to the values in m using the function g, and `filter`(f, m) returns a map that contains all key-values from m for which f returns true. Then `union`, `intersect`, and `diff` perform standard left-biased set union, intersection, and set-difference, and `countMap` m returns the number of elements in m.

The final types are list combinators. `zip`(a, b) combines lists a and b elementwise until the end of the shorter list, and `take`(n, l) returns the first n elements of the list l. `mapList`, `filterList`, and `reduceList` work like their map and array equivalents. Finally, `listToMap` l returns a map of the elements in l where duplicates replace former values, and `length` l returns the length of l.

Finally, the `while` and `loop` combinators perform iteration. `while`(f, v) is like a `do-while` loop, since it always applies f to v once and then recursively applies f until it returns `False`. `loop`(f, p, v) is like a standard `while` loop, as it only applies f to the first v if the corresponding call to p returned true `True`, returning the original v otherwise.

For a complete list of combinators with their types and descriptions please see Appendix B.

```
let mmul=(\(A,B) :: (Arr (Int,Int) Float, Arr (Int,Int) Float) ->

  -- zip all combinations of rows from A and cols from B
  let R1 = eqJoinArr (snd, fst A, B) in

  -- multiply values from A and B
  let R2 = mapArrInv (id, id, Float.*, R1) in

  -- group by destination & sum-reduce
  let C = groupReduceArr (
    \((ai,aj),(bi,bj)) -> (ai,bj),
    snd, Float.+, 0.0, R2) in C)

in ...
```

FIGURE 3.7: Matrix-matrix multiplication program

3.6 Example programs

This section presents some example programs to illustrate how to use arrays, maps, and lists in Flocc.

Matrix Multiplication In Flocc, the matrix multiplication (cf. Figure 3.7) closely follows a relational algebra version (cf. Figure A.1b). Here, `A` and `B` are arrays with pairs of integers as indices, and floating point values. The array join `eqJoinArr` computes the Cartesian product of both arrays, restricted to entries where the `snd` index from `A` is equal to the `fst` index from `B`. It thus returns an array with four indices that contains all pairs of Floats that contribute to the result. This is equivalent to zipping together all combinations of rows from matrix `A` and columns from matrix `B`. `mapArrInv` multiplies each of these pairs (like the renaming), and the aggregation `groupReduceArr` then groups these values using new keys `(ai,bj)` (i.e., the row from `A` and column from `B`), and sums up all the values in each group using `Float.+`.

Jacobi 1D The function `jac` (cf. Figure 3.8) is a cut down version of the Jacobi method, which is a numerical method for solving diagonally dominant systems of linear equations. For example, given a square system of n linear equations $A\mathbf{x} = \mathbf{b}$ where A can be decomposed into a diagonal component D and remainder R such that $A = D+R$, the solution can be obtained by iteratively computing $\mathbf{x}^{k+1} = D^{-1}(\mathbf{b} - R\mathbf{x}^k)$ until some convergence condition is met. The equivalent element-based formula is $x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}x_j^k), i = 1, \ldots, n$ which we cut down to $x_i^{k+1} = \frac{1}{2}(a_{ij-1}x_j^k + a_{ij+1}x_j^k), i = 1, \ldots, n$.

The function `jac` applies the Jacobi stencil to a 1D array `X`, `N` times. It iterates using the `while` combinator to repeatedly apply `next` to the array. Every application of `next` applies the function `shiftArrR` to shift array `A` 2-elements to the right and then

```
let N = 100 :: Int in
let jac = \X :: DArr Int Float ->

  -- next applies the 1D jacobi stencil once
  let next = \A ->
    -- coalesce with element 2-indices to right
    let A' = eqJoinArr (id, id, A, shiftArrR (A, 2)) in
    -- sum pairs
    let A'' = mapArrInv (fst, dup, \(i,v) -> addf v,  A') in
    -- divide by 2.0
      mapArrInv (id, id, \(_,x) -> divf (x, 2.0), A'') in

  -- loop performs the stencil N times
  while (\(V,k) -> ((next V, addi (k,1)), lti (k,N)),
        (X, 0))

in ...
```

FIGURE 3.8: Jacobi 1D stencil

```
let hist = (\(N,D) :: (Int, Map k Float) ->

  -- use min/max vals as x-axis bounds
  let (minV, maxV) = reduce (\(_,v) -> (v,v),
    \((x1,y1),(x2,y2)) -> (Float.min (x1,x2),
                           Float.max (y1,y2)),
          (Float.MAX_VAL, Float.MIN_VAL), D) in

  -- scaling coefficient to get bucket ids
  let i = Float./ (toFloat (Int.- (N,1)),
                   Float.- (maxV, minV)) in
  let D' = map (\(k,v) -> (k, toInt (Float.* (v,i))), D) in

  -- group by bucket & count group sizes
  groupReduce (snd, \_ -> 1, Int.+, D'))

in ...
```

FIGURE 3.9: N-bucket histogram

combines it element-wise with the original `A` using `eqJoinArr`, and sums together the corresponding values using `mapArrInv` and `addf`. `next` therefore sums together each element's left-hand (`-1`) and right-hand (`+1`) neighbors and then uses `mapArrInv` and `divf` to divide each value by `2.0`.

Histograms The function `hist` (cf. Figure 3.9) shows a use of maps in Flocc. It takes a pair of arguments `N` and `D`, where `D` is a map from keys of arbitrary type `k` to floating point values, and computes a histogram of these values. This histogram has `N` equally spaced buckets such that bucket `0` contains the minimum value in `D` and bucket `N-1` contains the maximum. The `reduce` combinator projects the values from the map

```
let dotp = (\(A,B) :: (List Float, List Float) ->

  -- zip together lists and multiply pairs
  let AB = mapList (Float.*, zip (A,B)) in

  -- sum reduce multipied pairs
  reduceList (Float.+, 0.0, AB))

in ...
```

FIGURE 3.10: Dot product

`D` into pairs and finds the minimum and maximum values. These values are used to calculate the scaling coefficient `i`, which in turn is used to calculate each value's bucket index with the `map` combinator. `groupReduce` then uses these bucket indices as the keys for the result map, where `snd` projects them out of the original key-value pairs. For each key-value pair a `1` is projected out (using `\_-> 1`), and then each group of ones is aggregated using `Float.+`, thus counting the entries in each bucket.

Dot product The function `dotp` (cf. Figure 3.10) shows a use of lists in Flocc. It takes a pair of lists of floats, and returns their dot product, computed by zipping together the lists, multiplying the pairs, and then sum reducing them.

Chapter 4

Distributed Data Layout Types

In the previous chapter we presented Flocc, a functional language for data-parallel programming. Flocc is a *high-level* language. It does not depend on any particular architecture or memory model. Given implementations for the core library functions, Flocc programs could be evaluated using a simple interpreter implementing the reduction rules in Figure 3.2. However, to execute programs on a cluster we want to synthesize efficient low level implementations of Flocc programs that can be compiled and executed directly without an interpreter. To run efficiently on clusters, these implementations must distribute their data, and process it in parallel. We are therefore faced with the challenge of automatically synthesizing distributed algorithms, and associated distributed data layouts, for these high-level Flocc programs.

In this chapter we present the first step of our type-based approach to automatically synthesize distributed data-parallel implementations of Flocc programs. In Section 4.1 we give some of the advantages to encoding distributed data layouts (DDLs) as types, and in Section 4.2 we introduce some of the data distributions that are possible on clusters. Then we present our treatment of distributed data layouts (DDLs) as types (cf. Section 4.3), and in particular, a restricted form of dependent types (cf. Section 4.3.2). This formulation allows distributed data layouts for maps, lists, and arrays to all be represented in a common form. Then, we show how these types can be used to characterize the data distribution behaviors of the distributed implementations of our high-level combinators (cf. Section 4.3.1). In Chapter 5 we present a type inference algorithm that can infer DDLs for different distributed algorithms involving these implementations, and these DDLs can then be passed to the code generator (cf. Chapter 7).

4.1 Introduction

The idea of specifying distributed data layouts by program annotations is not new– HPF uses them for its distributed arrays. However, to the best of our knowledge, ours is the

first work to formalize data distributions as *types*.[1] This is surprising since there are many features of type systems that are very desirable for data layouts. These include:

Type safety Milner characterized type safety as "Well-typed programs can't go wrong" [144]. This generally means that expressions with a given type really do return values of that type, and operations that expect members of a certain type, only ever receive values of that type. This can often be proved for a language, in which case we say that the type system is *sound*, and then checked for individual programs using a *type-checker*. For data layouts we also want to ensure that expressions with a given data layout really do return values with that layout, and that operations that require collections with a given layout only receive values with that layout. Representing DDLs as types therefore allows us to prove that our *DDL type system* is sound (cf. Section 5.1.1; i.e., evaluation preserves types, and well-typed programs can progress), and automatically check that the DDLs for a program are valid. These two features allow us to guarantee that values with incorrect data layouts never occur (assuming that the code generator is correct).

Polymorphism There are different kinds of type polymorphism. *Ad hoc* polymorphism allows us to define multiple implementations of the same operation with different types (e.g., function overloading). *Parametric* polymorphism allows the same code to be used with different concrete types (e.g., generics in object oriented languages). *Subtyping* allows different types to be used as long as they are descendants of some common super-type (e.g., inheritance or record types), and *coercion* polymorphism allows explicit and implicit type conversions to be used in programs. We use ad hoc, parametric, and coercion polymorphism for our DDLs in this thesis. We use ad hoc polymorphism to define multiple implementations of the same combinator, with different data layouts. Parametric polymorphism is particularly powerful, allowing us to define complex relationships and constraints between DDLs. It lets us range over all instances of a given DDL parameter using type variables, for example to describe that an operation's DDL permits layouts with any partition function, or permits layouts mirrored over any dimension of the node topology. Finally, coercion corresponds neatly to redistributing data and thus changing from one data layout to another.

Type inference Some type systems allow type annotations to be omitted, and provide an inference algorithm to automatically assign valid types to the expressions in a program, if a valid typing exists. One such system is ML's type system [104], which supports a kind of parametric polymorphism called *let-polymorphism* that allows types with universally quantified type variables called type schemes to be created at let-expressions

[1] Data Parallel Haskell [39] uses types to tag "distributed lists", but these are not actually "distributed" since they are for multicores, and do not contain any information about *how* to partition and distribute the lists.

[35]. Damas-Milner's Algorithm W is a type inference algorithm for this system [62]. Representing DDLs as such type schemes (generated at let-expressions) allows us to mechanically infer valid data layouts involving type variables using a similar algorithm, so that the user does not have to manually specify them.

Optimization Type information can be used by compilers to perform type-specific optimizations. Similarly, data layout information allows the compiler to generate optimized data-layout specific code. Both kinds of information are even more useful when they have been automatically inferred without user intervention.

Extensibility In most languages new types can be defined in programs, making types a very extensible way to represent program meta-data. Our compiler does not allow new DDL types to be declared in programs, but does support adding more combinators and DDL types via a configuration file and additional code generation templates. For example, currently new data-parallel combinators can be added by declaring their functional types, and the DDL types of their distributed-memory implementations, in a type declaration file, implementing code generation templates for these implementations as Haskell functions, and adding replacement rules to a configuration file, that declare which implementations implement which high-level combinators. So currently the only additional code required are the code generation templates, and these could be implemented via a scripting language in future versions. Alternatively, in future versions Flocc's main syntax could be extended to support defining new combinators, DDL types, and templates in programs, rather than in separate configuration files.

These features make types well suited for representing distributed data layouts, though there are some difficulties that need to be overcome. Firstly, not all type systems allow arbitrary program terms to be embedded in the types, which is a requirement for our DDL type system. Then, not all type systems support type inference, which restricts which type system can be used, and how DDL types can be encoded. Furthermore, Damas-Milner type inference returns the most general typing for each expression, and such a type may not exist for distributed data layouts. Finally, in some cases there maybe multiple valid data layouts for a given expression, in which case parametric polymorphism may not be sufficient to represent the complete range of possible data layouts for program terms. We address these challenges in the chapters that follow.

4.2 Distributing Collections on Clusters

To run a data-parallel algorithm on a distributed memory computer, the data structures must be distributed across the computer's nodes. The simplest way to do this is to repli-

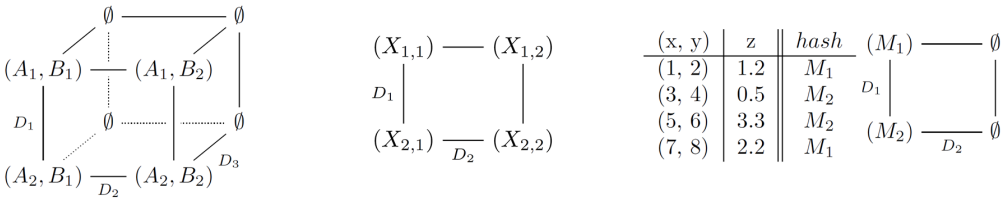


FIGURE 4.1: Array distributions (left, center). Map distribution (right).

cate (or "mirror") them such that they exist on every node. However, this is usually inefficient as it requires communicating updates to all nodes, and may be impossible due to lack of available memory. Instead data-parallel algorithms rely on partitioning their data such that different nodes work on different parts of the data set. It is therefore common for at least one data structure to be partitioned, so that different nodes possess different parts of it. This section briefly introduces various ways of distributing collections on a cluster.

Arbitrary partitions The simplest way to partition a collection is to number the nodes in the system with indices 0 to N, and to allocate approximately equal numbers of arbitrary elements to each index. Structure preserving functions like `map` can be applied to these partitions in parallel, without any need for communication between nodes.

Co-location by key Another useful way to partition a collection is to define a *key emitter function* $kf : \alpha \to \kappa$, taking an element of type α and returning some key of type κ, and to group elements that have the same key value together on the same node. This can be achieved via a *hash* or *partition function* $pf : \kappa \to \texttt{Int}$ to map keys to node indices (which we assume to be integers). The `groupReduce` function defined in Figure 3.6 benefits from this sort of partitioning. Given the function application $\texttt{groupReduce}(f_1, f_2, f_3, x)$, if the input collection x is partitioned using f_1 as its key emitter function (i.e., the function that projects the new keys from the input elements), then all the elements in each group will be co-located on the same node, and the reductions can proceed without inter-node communication.

If collections are partitioned with key emitter functions which share the same key type κ then they can be aligned by using the same partition function pf for both collections, so that all values for a given key are co-located. `eqJoin` can utilize this alignment (cf. Figure 3.6). For example, $\texttt{eqJoin}(f_1, f_2, x_1, x_2)$, can partition x_1 and x_2 using their key emitter functions f_1 and f_2, such that pairs of elements which satisfy the equality predicate $(f_1(x) = f_2(y))$ are co-located, and no inter-process communication is needed.

Cartesian topologies MPI allows the definition of virtual node topologies for clusters, where nodes are addressable via Cartesian coordinates. The MPI implementation

$$dts ::= \forall Id \cdot dts \mid \Pi x : dt \to dt \mid dt$$

$$\begin{aligned} dt ::= {} & Id \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid \texttt{Null} \mid (dt_1, \ \ldots, \ dt_n) \mid dt_1 \to dt_2 \\ \mid {} & \texttt{DArr}\ i\ t\ f\ d_1\ d_2 \\ \mid {} & \texttt{DMap}\ t_1\ t_2\ [\texttt{Hash} \mid \texttt{DynDiscrete} \mid \texttt{DynRange}]\ f\ d_1\ d_2 \\ \mid {} & \texttt{DList}\ t\ [\texttt{Blk} \mid \texttt{Cyc}]\ d_1\ d_2 \mid \ldots \end{aligned}$$

$$f ::= \texttt{\textbackslash}x\ [\ \texttt{::}\ t\ ]\ \texttt{->}\ e \mid g \mid f_1 \cdot f_2 \mid f_1 \otimes f_2$$

$$g ::= \texttt{fstFun}\ f \mid \texttt{sndFun}\ f \mid \texttt{lftFun}\ f \mid \texttt{rhtFun}\ f$$

$$i ::= \texttt{Int} \mid (i_1, \ \ldots, \ i_n)$$

$$d ::= Id \mid (d_1, \ d_2)$$

$$x ::= Id \mid \texttt{\_} \mid \texttt{(}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{)}$$

FIGURE 4.2: Distributed data layout (DDL) type syntax

then decides how best to map these onto physical nodes. This abstraction is useful, since it allows us to describe where collections are stored and replicated relative to each other, without considering the physical interconnect. We therefore identify nodes using n-dimensional grids with dimensions D_1 to D_n.

Collections can be split into partitions and distributed over the nodes in some of the dimensions, replicated across any other dimensions, and are stored at the nodes on the axis of any remaining dimensions. Figure 4.1 illustrates on the left an input distribution for a matrix multiply on an 8-node cluster organized as a 3D grid. Matrix `A` is split into two partitions $\texttt{A}_1$ and $\texttt{A}_2$ partitioned along D_1, and mirrored across D_2, but only at the axis of D_3. `B` is partitioned along D_2, and mirrored across D_1, also only at the axis of D_3. Hence, the node $(0, 1, 0)$ contains the partitions $\texttt{A}_2$ and $\texttt{B}_2$ while $(1, 1, 1)$ remains empty. Figure 4.1 (center) illustrates a 2D partitioning of an array X, and shows (on the right) a map M partitioned along D_1 and only at the axis of D_2. This system allows suitable node topologies to be expressed for all the combinators in Figure 3.6. We use such node arrangements to describe data distributions in the sections that follow.

4.3 Distributed Data Layout (DDL) Types

In our system, every high-level collection has a corresponding distributed collection type which, in addition to describing the data type, has extra parameters which specify how it should be distributed on the cluster. It is important to note that the user does not (need to) see these types, but the compiler uses them to plan the data distribution. The syntax for these *distributed data layout (DDL) types* is given in Figure 4.2. In addition to standard type schemes these include a polyadic version of dependent Π-types that

$\beta \cdot \alpha$	$\stackrel{def}{=}$	`\x -> β (α x)`
$\alpha \otimes \beta$	$\stackrel{def}{=}$	`\(x, y) -> (α x, β y)`
`id`	$\stackrel{def}{=}$	`\x -> x`
Δ	$\stackrel{def}{=}$	`\x -> (x, x)`
`dup`	$\stackrel{def}{=}$	`\x -> (x, x)`
`swap`	$\stackrel{def}{=}$	`\(x, y) -> (y, x)`
`nullF`	$\stackrel{def}{=}$	`\_ -> ()`
`fst`	$\stackrel{def}{=}$	`\(x, y) -> x`
`snd`	$\stackrel{def}{=}$	`\(x, y) -> y`
`lft`	$\stackrel{def}{=}$	`fst · fst` $\otimes$ `fst · snd`
`rht`	$\stackrel{def}{=}$	`snd · fst` $\otimes$ `snd · snd`
`one`	$\stackrel{def}{=}$	`\_ -> 1`

FIGURE 4.3: Flocc syntactic sugars

we call *dependent type schemes* and explain in Section 4.3.2. The DDL types dt extend types t with distributed arrays, maps, and lists (`DArr`, `DMap`, and `DList`). Additional type parameters include distribution modes (e.g., `Hash`, `Blk`, etc.), partition functions f, and dimension identifiers d, which are described below. Partition functions can contain function generators g which are described in detail in Section 4.3.3.

Distributed arrays The `DArr` type describes how to store an array on a cluster. `DArr` takes a partition function f and two tuples of dimension identifiers d_1 and d_2. f is an actual (projection) function that is made from lambda-abstractions from the input program, the operators listed under f in Figure 4.2, and the sugars defined in Figure 4.3. It identifies the dimensions of the array along which it should be partitioned. The array is partitioned along these dimensions using a block-cyclic distribution. Dimension identifiers d_1 and d_2 are just type variables, and pairs of dimension identifiers. d_1 has the same arity as f's co-domain, and specifies over which dimensions of the cluster the partition dimensions should be distributed. d_2 specifies a set of dimensions over which to mirror. Partitions are only stored at the zero-nodes of any dimensions that remain. For example, the matrices in Figure 4.1 (left) have DDL types,

```
A :: DArr (Int,Int) Float fst d1 d2
B :: DArr (Int,Int) Float snd d2 d1
```

which means that `A` is partitioned by row, since `fst` projects out the left-hand (i.e., row) array index, and `B` is partitioned by column (i.e., the right-hand array index). The `d1` and `d2` parameters are type variables that identify two *distinct* node topology dimensions at runtime. Therefore, `A`'s partitions are partitioned along dimension `d1` at runtime, and mirrored along `d2`, whilst `B`'s partitions are partitioned along `d2` and mirrored along `d1`.

Distributed maps Similarly, `DMaps` describe how to store `Maps` on clusters. d_1 and d_2 work in the same way as `DArr`, but f takes key-value pairs rather than indices, mapping them onto keys that are translated into specific node indices in d_1. This translation is either via a hash function (i.e., `Hash`), or a dynamic lookup table which is populated at runtime (i.e., `DynDiscrete` and `DynRange`). These tables can either record mappings between discrete values and node indices (i.e., `DynDiscrete`), or map ranges of values onto node indices (i.e., `DynRange`).

For example, the value `M` in Figure 4.1 (right) is partitioned by `z` and so has type,

```
DMap (Int,Int) Float Hash snd d1 d_null
```

meaning that `M` is partitioned by its value, since `snd` projects the value out of the key-value pairs, along topology dimension `d1`. The $\texttt{d}_{\texttt{null}}$ means that `M`'s partitions are not mirrored over any dimensions. Allowing f to be any function is much more flexible than approaches that only allow partitioning to be defined via the map's key or lists of column names like in Chapel or SQL.

Distributed lists `DList`'s parameters d_1 and d_2 work in the same way, but instead of a partition function, `DLists` just have a partition mode which is either `Blk` for a block partitioned list, or `Cyc` for a list that uses a cyclic distribution. The former splits the list into n equally sized blocks, where n is the number of nodes in d_1, and the latter stores every i-th element mod n on node i.

Top-level scalars and lambda terms are always mirrored on all nodes in the cluster. For `DArrs` and `DMaps`, if the partition function is `nullF` the collection is not partitioned. The usual $\cdot$-operator sequentially composes two functions, and the $\otimes$-operator composes pairwise, as defined in Figure 4.3. We use T_α to refer to the set of all terms defined under syntactic domain α, e.g., T_f denotes all embedded functions.

4.3.1 Distributed Function Types

For each high-level combinator (cf. Figure 3.6), the compiler internally provides different functionally equivalent implementations that work for different (often polymorphic) data distributions. We use the DDL types to characterize how these different implementations store their inputs and outputs. These implementations and their types are hidden from the user; they only see the high-level combinators.

The DDL type schemes for the combinator implementations used in this section are shown in Figure 4.4. For example, `groupReduce1` locally groups and reduces the values stored at each node, exchanges intermediates between nodes to co-locate by key, and then group-reduces again at each node. This implementation works no matter how

```
mapArrInv1 :: Π(f,_,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v (g · f) d m) -> DArr j w g d m
mapArrInv2 :: Π(_,f_inv,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v g d m) -> DArr j w (g · f_inv) d m

eqJoin1A    :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
  DMap j w p g d m) -> DMap (i,j) (v,w) p (f · lft) d m
eqJoin1B    :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
  DMap j w p g d m) -> DMap (i,j) (v,w) p (g · rht) d m
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (f · fst) d m
eqJoinArr1B :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (g · snd) d m
eqJoinArr2  :: (i->k, j->k, DArr i v fstFun(f) d1 m,
  DArr j w nullF d2 (d1,m)) -> DArr (i,j) (v,w) f d1 m
eqJoinArr3  :: (i->k, j->k, DArr i v fstFun(f) d1 (d2, m),
  DArr j w sndFun(f) d2 (d1, m)) -> DArr (i,j) (v,w) f (d1,d2) m

groupReduce1    :: ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d1 m1) -> DMap j w p fst d2 m2
groupReduce2    :: Π(f,_,_,_) : ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d m) -> DMap j w p fst d m
groupReduceArr1 :: (i->j, (i,v)->w, (w,w)->w, w,
  DArr i v f d1 m1) -> DArr j w id d2 m2
groupReduceArr2 :: Π(pf,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr i v pf d m) -> DArr j w id d m
```

FIGURE 4.4: DDL types for the main combinator implementations.

the input is partitioned. In the DDL type we therefore use the universally quantified type variable `f` to specify that the input can be partitioned by any partition function. The output is always partitioned by key, which we specify by using `fst` as the partition function in the return type.

4.3.2 Dependent Type Schemes

In addition to classic type schemes, we also have a polyadic version of dependent Π types *dts*, similar to those used in dependent ML [213]. They are similar to type schemes but the type variables are now rigidly bound to the members of the argument tuples at function applications. Hence, rather than representing *any* value, such type variables are bound to the *actual* values of parameters at runtime, or more precisely, the AST objects that represent them. These variables can then be used in the input and output types. Here we require that functions that are referenced by Π-types must be statically resolvable to specific lambda-abstractions, to increase the likelihood that we will be able to decide their equality at compile time, and so that we can inline them during code generation. This allows us to place context-dependent constraints on data distributions, to specify when different combinator implementations can be used.

For example, in contrast to `groupReduce1`, `groupReduce2` group-reduces just once, locally at each node. To yield a valid result all the input values for a given group must be co-located on the same node. We specify this constraint using a Π-type. `groupReduce2`'s type

```
groupReduce2 :: Π(f,_,_,_) :
  ((k1,v1) -> k2, (k1,v1) -> v2, (v2,v2) -> v2, DMap k1 v1 Hash f d m)
    -> DMap k2 v2 Hash fst d m
```

thus binds the value of its first parameter, the function that generates the result keys, to `f`. This specific value of `f` is then used to define the input map's partition function. All values for a given key produce the same hash, and will therefore be stored on the same node. `groupReduceArr2` uses the same technique.

`eqJoin1A` and `eqJoin1B`, and `eqJoinArr1A` and `eqJoinArr1B` (cf. Figure 4.4), work in a similar way. Here, if we know that the values that yield a given key are co-located on the same node, then no inter-node communication is necessary and local joins will suffice. To specify this we partition both inputs by their respective join-key projection functions `f` and `g`. The output is therefore partitioned by both `f·fst` and `g·snd` in the types with the suffixes `A` and `B` respectively. This is a situation where there are two valid types for the same combinator implementation, which allow different distributions to be inferred. In our approach, to simplify type inference, we treat each typing as a different combinator implementation, and let the data distribution search (cf. Section 7.5 and Section 8.3) explore different combinations of these typings.

We also use Π-types to specify partitioning information for structure-preserving transformations, like `mapArrInv1`. Here, to ensure the output is partitioned by `g`, the input must be partitioned by `g` applied after the index transformer function `f`. In the other direction, if the inverse transformer function $\texttt{f}_{inv}$ is known, and the input is partitioned by `g`, then the output of `mapArrInv2` will be partitioned by `g` applied after $\texttt{f}_{inv}$. Both these implementations work the same way (via local maps), but they enable distribution information to be inferred in different directions, from the output type to the input, and from the input type to the output, respectively.

4.3.3 Function Generators

By expressing output partition functions as compositions of input ones, our type schemes allow us to infer DDL information *forwards* (from inputs to outputs) through the programs. However, the inference also needs needs to work *backwards* in some cases, in order to automatically find input partition functions which combine to yield a given output partitioning. For unary combinators like `mapArrInv1` we can use the existing DDL information, but for binary combinators like `eqJoinArr` we need to *decompose* output partition functions. For this we use *function generators* (cf. Figure 4.2, `fstFun` to `rhtFun`).

Function generators analyze at compile-time the abstract syntax trees (ASTs) of their arguments (which are partition functions), and derive new partition functions that depend only on a subset of the inputs. If no such AST terms exist then the `nullF` function `\_->()` is generated. For instance, `fstFun` takes a function f with a domain `(x,y)`, and generates a function g with domain `x` by trying to split f's body into a pair and retaining all the parts of the AST that depend only on `x`, and throwing away those terms that also depend on `y`. `sndFun` works accordingly on the `y` domain. Recomposing the two using the $\otimes$-operator yields a partition function that equals the original. Hence, given a partition function `f = \(a,(b,c)) -> (a,c)`, `fstFun(f)` equals `\a -> a` and `sndFun(f)` equals `\(b,c) -> c` and their combination `fstFun(f)`$\otimes$`sndFun(f)` equals the original f. However, if the function cannot be deconstructed in a way that allows it to be re-composed, the function generator returns `\_->()`. For example, `\(x, y) -> (y, x)` cannot be split into two functions whose pairwise composition will equal the original. `eqJoinArr3` uses function generators so that it can partition its output by any `f`. To ensure that the output is partitioned by `f`, the inputs must be partitioned by `fstFun(f)` and `sndFun(f)` along dimensions `d1` and `d2`.

4.4 Combinator implementations

In this section we briefly explain how the combinator implementations used in the example derivations in Section 4.5 work, and how their DDL types reflect their data

distribution behaviors. For a more complete list of combinator implementations, along with their DDL types and explanations, please refer to Appendix C.

```
intRangeMap1    :: (Int,Int,Int) -> DMap Int () Hash fst d m
intRangeMap3    :: (Int,Int,Int) -> DMap Int () DynRange fst d m
intRangeMapMirr :: (Int,Int,Int) -> DMap Int () p nullF () m
```

`intRangeMap` creates a map with integer keys in the range (and with the stride) specified. `intRangeMap1` and `intRangeMap3`, both generate these keys for each partition in parallel, and use hash, and dynamic-range partitioning modes, respectively. `intRangeMapMirr` returns a mirrored map, where the whole map is replicated across the dimensions in `m`.

```
countMap     :: DMap k v p f d m -> Int
countMapMirr :: DMap k v p nullF () m -> Int
```

`countMap` returns the cardinality of a partitioned map, by summing the cardinalities of each partition, and `countMapMirr` calculates the cardinality of a mirrored map, in-place.

```
mapArrInv1 :: Π(f,_,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v (g · f) d m) -> DArr j w g d m
mapArrInv2 :: Π(_,f_inv,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v g d m) -> DArr j w (g · f_inv) d m
mapInv1 :: Π(f,f_inv,_) : ((i,v)->(j,w), (j,w)->(i,v),
  DMap i v p (g · f) d m) -> DMap j w p g d m
mapInv2 :: Π(f,f_inv,_) : ((i,v)->(j,w), (j,w)->(i,v),
  DMap i v p g d m) -> DMap j w p (g · f_inv) d m
```

`mapArrInv1` and `mapArrInv2` are structure preserving combinators that apply their first two parameter functions to the indices and values, respectively, of a distributed array. The index transformer function f is constrained to be a permutation function, since its inverse f_{inv} must be provided. Both `mapArrInv1` and `mapArrInv2` have the same implementation, which acts in-place (i.e., without communication), but they infer DDL information in different directions. In `mapArrInv1` the output partition function g is known, so we construct an input partition function that will provide this output partitioning by applying g after the index transformer function f. `mapArrInv2` does the opposite. If the input is partitioned by g then the output is partitioned by g applied after the inverse index transformer function f_{inv}, since f_{inv} returns the indices from the output elements, that were returned for the same elements in the input array. `mapInv1` and `mapInv2` are map-based combinators similar to `mapArrInv1` and `mapArrInv2`.

```
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (f · fst) d m
eqJoinArr1B :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (g · snd) d m
eqJoin1A :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
```

```
  DMap j w p g d m) -> DMap (i,j) (v,w) p (f · lft) d m
eqJoin1B :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
  DMap j w p g d m) -> DMap (i,j) (v,w) p (g · rht) d m
```

`eqJoinArr1A` to `eqJoinArr3` perform equi-joins on pairs of distributed arrays (i.e., Cartesian products restricted to where the join-keys are equal), where the two function parameters emit join-key indices from the elements in the first and second arrays respectively. `eqJoinArr1A` and `eqJoinArr1B` can do this in-place without communication because their input arrays are partitioned by their join-key emitters f and g along the same nodes in the topology d, and are therefore aligned by join-key index, such that all elements for a given key in both arrays is on the same node. The output combines the indices and elements from the inputs pairwise, and so the outputs are partitioned by the join-keys from the first array $f \cdot$ `fst` in `eqJoinArr1A`, and those from the second array $g \cdot$ `snd` in `eqJoinArr1B`. `eqJoin1A` to `eqJoin4` are map equivalents of the `eqJoinArr` functions, with equivalent distributions. The difference here is the join-key emitter functions apply to the map key and value, and so the types use `lft` and `rht` instead of `fst` and `snd` to project from the pairs of key-pair and value-pairs.

```
eqJoinArr2 :: (i->k, j->k, DArr i v fstFun(f) d1 m,
  DArr j w nullF d2 (d1,m)) -> DArr (i,j) (v,w) f d1 m
eqJoinArr3 :: (i->k, j->k, DArr i v fstFun(f) d1 (d2, m),
  DArr j w sndFun(f) d2 (d1, m)) -> DArr (i,j) (v,w) f (d1,d2) m
```

`eqJoinArr2` accepts any output partition function f, and therefore requires all combinations of elements (the Cartesian product) to be enumerable. Thus we partition the first array by `fstFun` f along `d1`, and mirror the second (hence the `nullF`) along `d1`. `fstFun` f ensures that the output is partitioned by f, and all combinations are enumerable without communication. `eqJoinArr3` is similar, but it partitions and mirrors both input arrays— the first by `fstFun` f along `d1` mirrored along `d2` and `m`, and the second by `sndFun` f along `d2` mirrored along `d1` and `m`. Here `d1` and `d2` are orthogonal dimensions of a Cartesian node toplology, and so the Cartesian product of partitions will be enumerated across `d1` $\times$ `d2`.

```
crossMaps1 :: (DMap i v p f d1 m, DMap j w p nullF d2 (d1,m)) ->
  DMap (i,j) (v,w) p f·lft d1 m
```

`crossMaps` returns the Cartesian-product of its two input maps. `crossMaps1` does this by requiring its second input to be mirrored along the dimension that its first input is partitioned along, such that it can compute the Cartesian-product of each partition in-place. The output is therefore partitioned by the first argument's partition function f, applied to the keys and values from the first argument (which are obtained by applying `lft`).

```
groupReduceArr1 :: (i->j, (i,v)->w, (w,w)->w, w,
```

```
  DArr i v f d1 m1) -> DArr j w id d2 m2
groupReduceArr2 :: Π(pf,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr i v pf d m) -> DArr j w id d m
groupReduce1 :: ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d1 m1) -> DMap j w p fst d2 m2
groupReduce2 :: Π(f,_,_,_) : ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d m) -> DMap j w p fst d m
```

`groupReduceArr1` and `groupRedueArr2` group array elements by the index returned by the first function parameter— an index projection function that returns a subset of the array's indices, applies the second function parameter to their indices and values, and then aggregates them using the third function parameter. `groupReduceArr2` can do this in-place since its input is partitioned by the index projection function parameter *pf*, where as `groupReduceArr1` has to exchange intermediate values between nodes to co-locate groups before aggregation. `groupReduce1` and `groupReduce2` are map equivalents of `groupReduceArr1` and `groupReduceArr2`.

```
union      :: (DMap k v p fst d m, DMap k v p fst d m) ->
  DMap k v p fst d m
```

`union` performs a left-biased set union. It acts in-place since its inputs and output are partitioned by the map key via `fst`, and so equivalent keys are co-located.

```
mirrMap :: DMap k v p f d m -> DMap k v p f d (m,m')
repartMap :: DMap k v p1 f1 d1 m -> DMap k v p2 f2 d2 m
redistList :: DList v p1 d1 m1 -> DList v p2 d2 m2
```

`mirrMap` to `distListLit` are redistribution functions. These functions do not modify the data in their argument collections, but change how they are distributed. They are therefore implementations of `id` (i.e., the identity function) on the high level. They can also be thought of as type-casts. `mirrMap` takes any distributed map and mirrors its partitions along another dimension of the global topology. `redistList` does the same for lists, and optionally mirrors over different dimensions too.

4.5 Example Derivations

In this section we show that our DDL type system can infer the standard distributed-memory data layouts for some common data-parallel algorithms, including some of those in Chapter 3. For each program, we show which combinator implementations it uses for each combinator application, and the DDLs that result. Note that to find these layouts automatically we need a type inference algorithm and a redistribution insertion algorithm, which we present in Chapter 5, and we need a search algorithm that considers

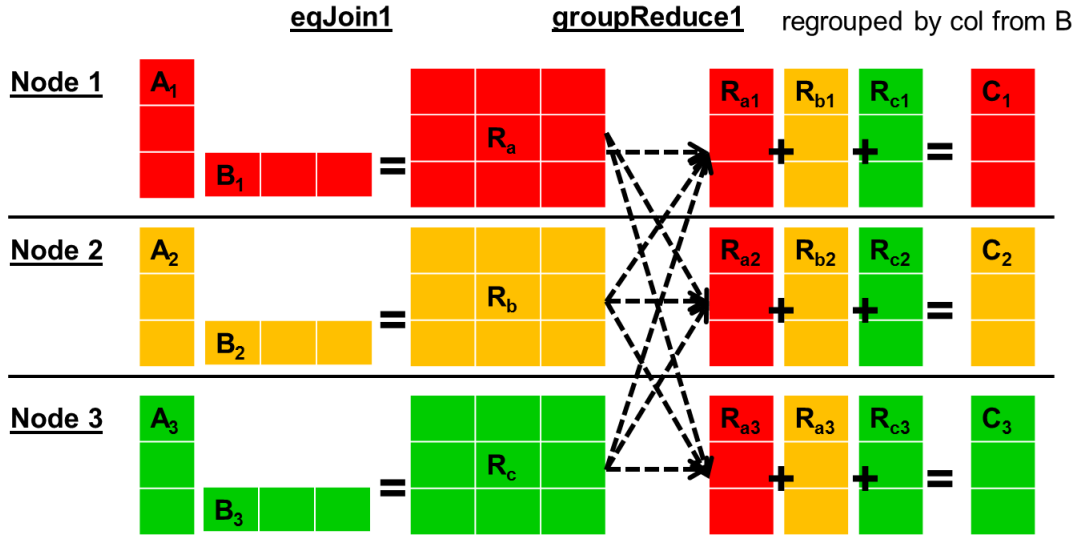


FIGURE 4.5: Matrix-multiply—partition for `eqJoinArr` communication illustration.

different choices of combinator implementations for combinator applications in programs, which we present in Section 7.5. We use `IP` as shorthand for `(Int,Int)`.

Matrix multiplication—partition for `groupReduceArr` The first matrix multiply solution is driven by an optimized partition for the group-reduce operation, which yields the usual implementation of dense matrix multiplication. It uses `groupReduceArr2` to avoid inter-node communication by requiring `R2` to be partitioned using its index projection function, which projects `A`'s row and `B`'s column from the array's indices. `R1`, `mapArrInv2`'s input, must thus be partitioned using this function too. `eqJoinArr3` satisfies this constraint, by partitioning `A` by row (using `fst`) along one dimension, and `B` by column (using `snd`) along an orthogonal one, and then mirroring both along their respective orthogonal dimensions. This yields a 2D grid, enumerating all combinations of partitions, i.e., the Cartesian product.

```
C ::  DArr IP Float id (d1,d2) m
R2 :: DArr (IP,IP) Float \((ai,aj),(bi,bj))->(ai,bj) (d1,d2) m
R1 :: DArr (IP,IP) (Float,Float) (\((ai,aj),(bi,bj))->(ai,bj))·id (d1,d2) m
A  :: DArr IP Float (fstFun (\((ai,aj),(bi,bj))->(ai,bj))·id) d1 (d2,m)
   =  DArr IP Float fst d1 (d2,m)
B  :: DArr IP Float (sndFun (\((ai,aj),(bi,bj))->(ai,bj))·id) d2 (d1,m)
   =  DArr IP Float snd d2 (d1,m)
```

Matrix multiplication—partition for `eqJoinArr` The next solution, which is illustrated in Figure 4.5 and Figure 4.6, is more unusual. It uses `eqJoinArr1A` to avoid mirroring `A` and `B`, by aligning them to co-locate partitions with common key values. `A` and `B` are partitioned by column (`snd`) and row (`fst`) respectively, and thus the join

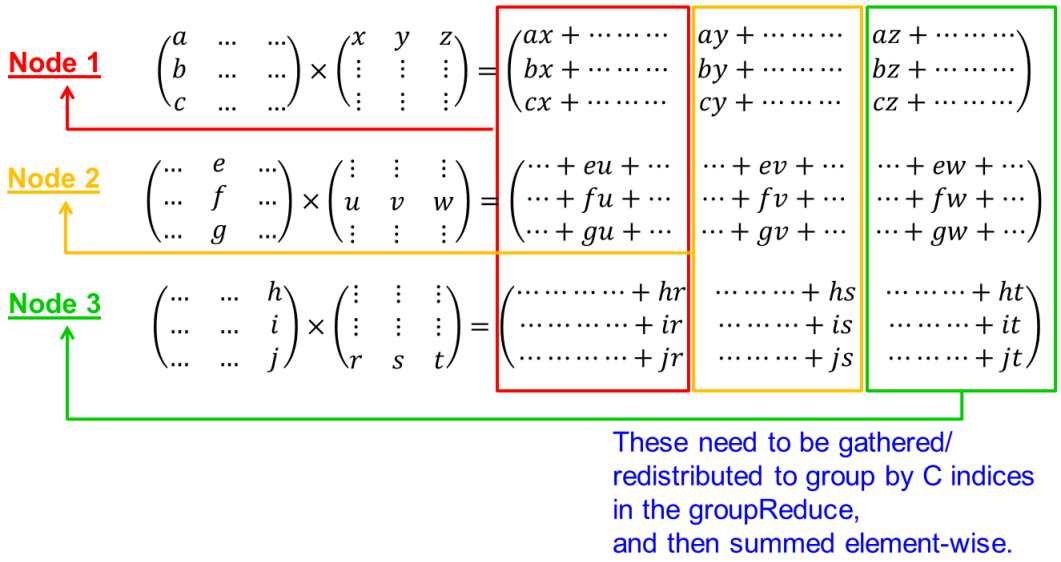


FIGURE 4.6: Matrix-multiply—partition for `eqJoinArr` algebraic illustration.

result `R1` is partitioned by the column of `A` (`snd·fst`), or if `eqJoinArr1B` were used, the row of `B` (`fst·snd`). `mapArrInv2` then constrains `R2` to have this partitioning as well, and so `groupReduceArr2` cannot be used without inserting a redistribution. Instead `groupReduceArr1` is used, as it accepts any input partitioning, at the expense of having to exchange intermediates between nodes. With dense matrices `R1` will be much larger than `A` and `B`, and so this solution will perform poorly, but if `A` and `B` are large and sufficiently sparse, exchanging the intermediates could outperform mirroring. Here, R_a, R_b, and R_c are not actually materialized as arrays, but are streams of values.

```
A  :: DArr IP Float snd d m
B  :: DArr IP Float fst d m
R1 :: DArr (IP,IP) (Float, Float) (snd·fst) d m
R2 :: DArr (IP,IP) Float (snd·fst)·id d m
C  :: DArr IP Float id d m
```

Matrix multiplication—mirror one matrix This implementation is illustrated in Figure 4.7. This implementation also uses `groupReduceArr2`, but unlike the first solution, it uses `eqJoinArr2` to give the required data distribution. This partitions `A` across all the nodes in `d`, and mirrors `B` on all of them. This solution is better than the first solution if `B` is much smaller than `A` so that it is less expensive to replicate all of `B` than partitions of `A`.

```
A  :: DArr IP Float (fstFun (\((ai,aj),(bi,bj))->(ai,bj))·id) d m = fst d m
B  :: DArr IP Float nullF d_null (d,m)
R1 :: DArr (IP,IP) (Float, Float) (\((ai,aj),(bi,bj))->(ai,bj))·id d m
R2 :: DArr (IP,IP) Float \((ai,aj),(bi,bj))->(ai,bj) d m
C  :: DArr IP Float id d m
```

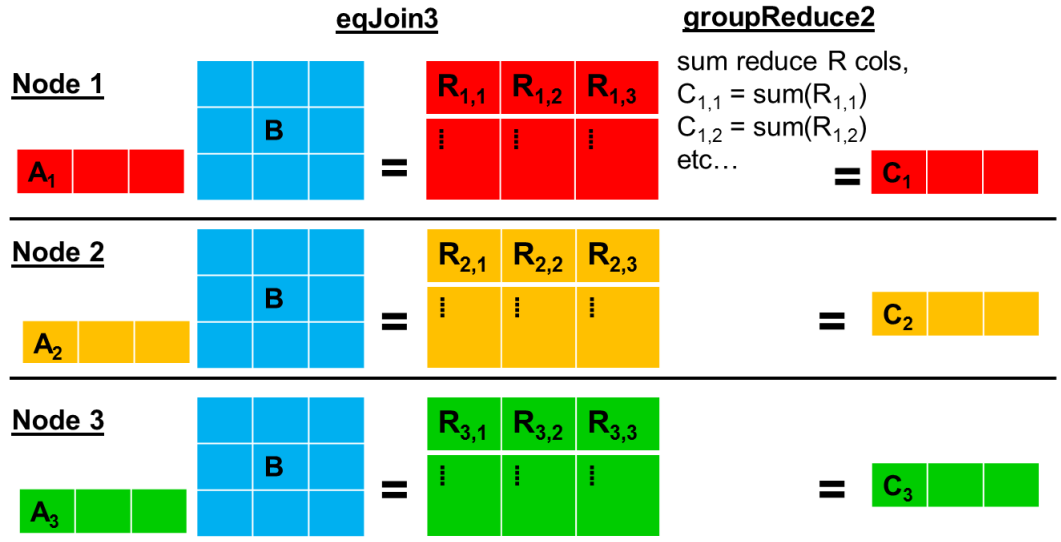


FIGURE 4.7: Matrix-multiply—mirror one matrix illustration.

```
let floyd = \(N,I0) :: (Int, Arr (Int,Int) Float) ->
  -- for each pair of vertices, see if going via
  -- current vertex k, is shorter than current path.
  let next = (\(I,k) ->
    let Iik = subArr ((0, k), (Int.- (N,1), k), (1,1), I) in
    let Ikj = subArr ((k, 0), (k, Int-.(N, 1)), (1,1), I) in
    let I1 = eqJoinArr (fst, fst, I, Iik) in
    let I2 = eqJoinArr (snd . fst, snd, I1, Ikj) in
    let I' = mapArrInv (fst . fst,
      \(i,j) -> (((i,j),(i,k)),(k,j)),
      \(_,((ij,ik),kj)) -> (Float.min (ij, Float.+ (ik,kj))),
      I2) in
    -- return new I, incremented k, and predicate (k < N-1)
    ((I', Int.+ (k,1)), lti (k, Int.- (N,1)))) in
  -- iteratively apply for all vertices: k in [0..(N-1)]
  while (next, (I0, 0)) in ...
```

FIGURE 4.8: Floyd's all pairs shortest path algorithm

Floyd All-Pairs Shortest Path—partition edges, mirror current vertex Floyd's all pairs shortest path algorithm (cf. Figure 4.8) finds the shortest paths between all pairs of vertices in a graph. It works by maintaining a 2D array of distances between vertices, which are initialized to the distance between the vertices if there exists an edge between them, and infinity (or a very large constant) otherwise. It then iterates for each vertex k, and checks for all vertex pairs (edges) whether going via the current vertex is a shorter distance than the current best distance between those vertices. If it is, then the best distance is updated to go via the current vertex k.

This implementation keeps the edges (i.e., the adjacency matrix) partitioned, and uses `mirrArr·subArr` to mirror the current row of distances from every other vertex to vertex k, and column of distances from vertex k to every other vertex. So, for each iteration, this

row and column are all-gathered to mirror them on every node (i.e., collected together and broadcast to all nodes), and each node then computes new distances for its array partition using `eqJoinArr2` and `mapArrInv2`, locally in parallel.

```
I   :: DArr IP Float f d d_null
Iik :: DArr IP Float nullF d_null d
Ikj :: DArr IP Float nullF d_null d
I1  :: DArr (IP,IP) (Float,Float) f·fst d d_null
I2  :: DArr ((IP,IP),IP) ((Float,Float),Float) f·fst·fst d d_null
I'  :: DArr IP Float f d d_null
```

Here `next` is called inside a loop, and so must have type `s -> (s, Bool)`. Therefore, its output DDL type (for `I'`) must be the same as its input one (for `I`), which it is in this example.

A better data layout for this application would partition either `Iik` or `Ikj` in the same way as `I`, and mirror the other one, rather than mirroring both. This layout cannot be derived using the DDL types in this chapter, but can be derived using the extended types and inference algorithm presented in Chapter 6.

Histogram—group locally before exchange The first Histogram (cf. Figure 3.9) solution uses `groupReduce1` so that the input `D` does not have to be partitioned by its `Float` value. The output `R` is partitioned by bucket id (`fst`), and a concrete function must still be chosen for `f`. Since in this example the type `k` is still abstract, the two possibilities for `f` are `fst` and `snd`. In a concrete program, `k` is a concrete type and so `f` could have more possible values. For `f=fst`, `D` is partitioned by `Hash` applied to `fst`, which is a valid solution. However, for `f=snd`, `D` is partitioned by `Hash` applied to `\v->toInt(Float.* (v,i))`, which is not valid, as it references `i` before it has been declared, and so this solution is discarded by the Flocc compiler.

```
D :: DMap k Float Hash (f·(id · fst ⊗ \(_,v)->toInt(Float.* (v,i))) · Δ) d m
D':: DMap k Int Hash f d m
R :: DMap Int Int Hash fst d m
```

Histogram—exchange before group The second DDL plan uses `groupReduce2` by repartitioning `D'` by `Hash` applied to `snd`. However, this plan will be sub-optimal unless the number of buckets is close to the number of data points, since the partitions of `D'` that `redistMap` communicates will be larger than the results of the local group-reduces that `groupReduce1` communicates.

```
D :: DMap k Float Hash (f·(id · fst ⊗ \(_,v)->toInt(Float.* (v,i))) · Δ) d m
D':: DMap k Int Hash snd d m
R :: DMap Int Int Hash fst d m
```

```
let mandel = (\(xs, ys) ->
 let x = intRangeMap (toInt(-2.5*xs),toInt(1*xs),1) in
 let y = intRangeMap (toInt(-1*ys),toInt(1*ys),1) in
 let axes = crossMaps (x,y) in -- make axes
 -- evaluate at each point
 mapInv (\((x,y),_) -> ((x,y),
   escape_time ((toFloat x) / (toFloat xs),
                (toFloat y) / (toFloat ys))),
         \(xy,_)->(xy,((),())), axes)) in ...
```

FIGURE 4.9: Mandelbrot set.

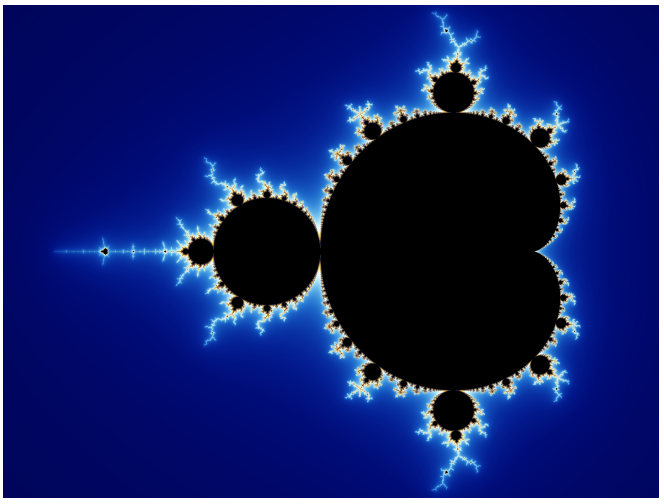


FIGURE 4.10: Un-equal load of the Mandelbrot set showing the need for non-blocked partitioning.

Mandelbrot set—block partition x, mirror y The Mandelbrot set [139] (cf. Figure 4.9) is a well known recursive fractal that can be visualized by plotting a different color depending on whether a given point on the complex plane is a member of the set. Formally the set is defined as the set of values c in the complex plane for which the iteration of $z_{n+1} = z_n^2 + c$ starting with $z_0 = 0$ remains bounded. That is, the complex number c is in the set if z_n remains bounded, and is not in the set if z_n tends to infinity. We compute this information in parallel by applying the sequential escape-time algorithm (which tests whether a point is in the set) to multiple points on the complex plane, in parallel.

This implementation uses `intRangeMap3` to generate the x coordinates partitioned into integer ranges, and `intRangeMapMirr` to mirror the y values on all nodes. We then use `crossMaps1` to compute the Cartesian product of the x and y sets, to get all the coordinates, and `mapInv2` to apply the escape time algorithm at each point. This implementation could be improved since the escape time algorithm takes a lot longer for points outside the set (see Figure 4.10[2]), and so partitioning x into integer ranges may mean some partitions (where most points are outside the set) take a lot longer to com-

[2] http://commons.wikimedia.org/wiki/File:Mandel_zoom_00_mandelbrot_set.jpg.

```
let rmat = (\N ->
 loop (\E-> (
  -- calc more edges
  let size = countMap E in
  let ints = intRangeMap (1,N-size,1) in
  let E' = mapInv (\(k,_) ->
    (k, genE (a0,b0,c0,d0,delta0)), \(k,_)->(k,()), ints) in
  -- remove duplicates
  let E'' = groupReduceMap
    (snd, \_->(), fst, (), E') in
  -- combine with existing
  let E''' = union (E'', E) in
      E'''),
  (\E -> (Int.< (countMap E, N), emptyMap())) in ...
```

FIGURE 4.11: R-MAT random graph generation

pute then others. The next implementation (i.e., hash partition x, mirror y) addresses this.

```
x :: DMap Int () DynRange fst d d_null
y :: DMap Int () p nullF d_null d
axes :: DMap IP () DynRange (fst·lft) d d_null
result :: DMap IP Bool DynRange (fst·lft) d d_null
```

Mandelbrot set—hash partition x, mirror y The previous implementation could perform poorly due to the distribution's bad load balancing. This implementation uses `intRangeMap1` instead of `intRangeMap3` to hash partition x, such that each partition contains a better balance of points that are in the set (which compute quickly) and those outside the set (which compute slowly).

```
x :: DMap Int () Hash fst d d_null
y :: DMap Int () p nullF d_null d
axes :: DMap IP () Hash (fst·lft) d d_null
result :: DMap IP Bool Hash (fst·lft) d d_null
```

Other distributions are possible here include: partitioning the y and mirroring the x, and hash partitioning and mirroring both x and y along orthogonal node dimensions. These can all be derived using the DDL types presented in this chapter.

R-MAT-generation—partition edges The R-MAT algorithm (cf. Figure 4.11) generates large random realistic graphs [38] (stored as sparse matrices), according to some probability distribution and other parameters. It is used to generate graphs to simulate social networks and other similar phenomena. The algorithm works by generating edges according to some probability distribution, using an iterative algorithm, removing duplicates, and then generating more, until some target number of edges is reached.

```
let kmkernel = (\(points, clusters) ->
  -- distances between points and clusters &
  -- new points with their closest clusters
  let pxc = crossMaps (points, clusters) in
  let points' = groupReduceMap (
    \((pid,_),_) -> pid,
    \((pid,cid),((ncid,ocid,ppos,d),cpos)) ->
      (cid, ocid, ppos, distPoints (cpos, ppos)),
    \((nc1,oc1,p1,d1),(nc2,oc2,p2,d2)) ->
      (if (d1 < d2)
       then (nc1,oc1,p1,d1)
       else (nc2,oc2,p2,d2)),
    (-1,-1,(-1.0,-1.0),9000000.0), pxc) in
  -- new cluster centres (avg member pos)
  let clusters' = groupReduceMap (
    \(pid,(ncid,ocid,ppos,d)) -> ncid,
    \(pid,(ncid,ocid,ppos,d)) -> (ppos, 1),
    \((sum1,tot1),(sum2,tot2)) ->
      (addPoints(sum1,sum2), (tot1 + tot2)),
    ((0.0,0.0),0), points') in
  let clusters'' = map (
    \(cid,(psum, ptot)) ->
      (cid, divPoint (psum, toFloat ptot)),
       clusters') in
  -- count how many memberships changed
  let nChanged = reduce (
    \(pid,(ncid,ocid,ppos,d)) ->
      (if (ncid == ocid) then 0 else 1),
      +, 0, points') in
  (points', clusters', totalChanged)) in ...
```

FIGURE 4.12: K-means clustering kernel

This implementation uses `countMap` to count how many edges there are in the current graph/matrix, and `intRangeMap1` and `mapInv2` to generate the `N-size` edges still required. `union` requires both inputs to be partitioned by map-key, which in this case is the generated edge (pair of integers). `groupReduce1` is used to remove duplicates, which returns the edges partitioned by map-key, as required by `union`. Edges are randomly generated using `genE`, so `E'` cannot be partitioned by edge (so that we could use `groupReduce2` without inserting a `repartMap` call, so we use `groupReduce1` instead.

```
E :: DMap IP () Hash fst d d_null
ints :: DMap Int () Hash fst d d_null
E' :: DMap Int IP Hash fst d d_null
E'' :: DMap IP () Hash fst d d_null
E''' :: DMap IP () Hash fst d d_null
```

This data distribution is the same as the one used in the implementation in [163].

K-means clustering— partition points, mirror cluster centers The k-means clustering algorithm (cf. Figure 4.12) takes a set of data points `points` and tries to partition them into k clusters in which each point belongs to the cluster with the nearest mean. It does this by starting with some k initial cluster positions `clusters` and iterating, recomputing the mean center of each cluster, and then repartitioning the points to belong to the cluster with the nearest mean.

This implementation is the standard distributed implementation. There are usually many more points than clusters, so `points` is partitioned and `clusters` mirrored on the compute cluster. `crossMaps1` computes the Cartesian product of these, partitioned by point, and `groupReduce2` calculates the distances between every point and every cluster center, in-place (since `pxc` is already partitioned by point). `groupReduce1` and `mapInv2` then computes the new mean for each cluster, by exchanging intermediates over the network to group by cluster, and then calculate the average position.

```
points :: DMap Int (Int,Int,(Float,Float),Float) Hash fst d dnull
clusters :: DMap Int (Float,Float) p nullF d dnull
pxc :: DMap Int (Float,Float) Hash (fst·lft) d dnull
     = DMap Int (Float,Float) Hash \((pid,_),_)->pid d dnull
points' :: DMap Int (Int,Int,(Float,Float),Float) Hash fst d dnull
clusters' :: DMap Int ((Float,Float),Float) Hash fst d dnull
clusters'' :: DMap Int (Float,Float) p nullF d dnull
```

Here, like `floyd`, `kmkernel` is called inside a loop, and so its output DDL type (for `(points',clusters'')`) must match its input one (for `(points, clusters)`). Therefore, `clusters''` must be mirrored by applying `mirrMap` to it, as it is here.

Dot product—cyclic distribution In this dot product (cf. Figure 3.10) plan `A` and `B` are aligned since they both have cyclic distributions over the same dimension `d`, so `zip` can be used without any communication. However, if `dotp` was used in a context where `A` or `B` had a different distribution, `redistList` would be automatically inserted to convert it into the required distribution.

```
A  :: DList Float Cyc d dnull
B  :: DList Float Cyc d dnull
AB :: DList Float Cyc d dnull
```

4.6 Concluding remarks

In this chapter we have shown how DDLs for maps, arrays, and lists, can be specified as types, and how the DDLs of distributed-memory combinator implementations can be written in such types. These DDL types are a restricted form of dependent Π-types

that can carry program terms, and specifically lambda-abstractions, in certain type parameters. We then explained the data distribution behaviors of some key combinator implementations, and have demonstrated how valid DDL typings can be used to specify the standard distributed-memory implementations of some example programs.

The next chapter (cf. Chapter 5) presents a DDL type inference algorithm that can automatically find valid typings for such programs, and an automatic type-cast (i.e. redistribution function) insertion algorithm that can make programs type-check by inserting casts to mend broken typing constraints. In Chapter 6 we then demonstrate the extensibility of our approach by showing how DDL types can be extended to encode local data layouts, more sophisticated array distributions, and more flexible variants of our current DDL types.

Chapter 5

Inferring Distributed Data Layout Types

In this chapter we present a type inference algorithm for DDL types. We define this algorithm, and prove that it is sound, and that it terminates for all inputs. We then also present an algorithm that automatically inserts type casts (i.e., redistribution function applications) into programs to make them well typed. Together, these two algorithms enable us to synthesize suitable distributed data layout information for any program and choice of combinator implementations.

5.1 Type Inference

In the previous chapter we characterized the DDLs of several combinator implementations as types. We would like to be able to automatically find a set of valid DDL types for a given program. This would be *desirable* if we were intending to make the user manually choose suitable combinator implementations, to save them having to design and declare correct data distributions themselves, but since we want to automatically select these, automatic inference is *essential*.

For a given input program we search for distributed implementations by exploring different choices of combinator implementations. Here each application of a combinator function in a program can use a different implementation. We then use the type inference system presented in this section, to find a valid assignment of distributed data layouts for a given choice of combinator implementations, if one exists. In other words, we are automatically inferring suitable distributed (and local) data layouts for distributed implementations of input programs.

Our type inference algorithm $\mathrm{T}_{\mathrm{infer}}$ extends a constraint-based version of Damas and Milner's Algorithm W [62] from [161]. This version traverses the AST implementing

$$
\begin{array}{lcl}
gdc(\Pi X : T_1 \to T_2, \mathtt{e}) & = & gdc(X, \mathtt{e}) \\
gdc((X_1,\ \ldots,\ X_n), (\mathtt{e_1},\ \ldots,\ \mathtt{e_n})) & = & \bigcup_{k=n}^{k=1}\ gdc(X_k, \mathtt{e_k}) \\
gdc(x, \mathtt{e}) & = & \{x = \mathtt{e}\} \\
gdc(\_, \mathtt{e}) & = & \{\} \\
otherwise & = & fail
\end{array}
$$

FIGURE 5.1: Definition of *gdc* (Generate dependent constraints)

$$
\begin{array}{lcl}
fresh((X_1,\ \ldots,\ X_n)) & = & \bigcup_{k=n}^{k=1}\ fresh(X_k) \\
fresh(x) & = & \text{fresh var} \\
fresh(\_) & = & \text{fresh var}
\end{array}
$$

FIGURE 5.2: Definition of *fresh* (Fresh type variables)

$$
\begin{array}{lcl}
bind((X_1,\ \ldots,\ X_n), (T_1,\ \ldots,\ T_n)) & = & \bigcup_{k=n}^{k=1}\ bind(X_k, T_k) \\
bind(x, T) & = & \{x : T\} \\
bind(\_, T) & = & \{\} \\
\text{otherwise} & = & fail
\end{array}
$$

FIGURE 5.3: Definition of *bind* (Bind types to tuple of variables)

the constraint-based typing rules to construct initial types and constraints, and then uses a unification algorithm [173] to find substitutions that unify the constraints. Our version extends Damas-Milner's (from [161]) to support tuples, deal with dependent type schemes, and unify functions embedded in the types. To handle dependent type schemes, we extend the function application case of the initial AST pass to instantiate any Π-bound variables using the function *gdc* (generate dependent contraints). To unify functions in the types, we use the algorithm in Figure 5.7, which extends standard syntactic unification to deal with constraints involving functions. This allows us to lift concrete functions into the types and solve equations involving them via unification.

5.1.1 Inference rules

Figure 5.4 shows the typing rules for our DDL type system. Here Γ is a mapping between variables and types, that stands for the current typing environment, where $\Gamma \oplus x : t$ (which we abbreviate to $\Gamma, x : t$) overwrites the mapping for x in Γ (if one exists) with a new mapping to type t. The rules derive typing judgments for programs; the judgments also include a set of constraints (denoted by "$\hookrightarrow C$") which must be satisfiable for the derived judgment to be valid. The rules closely mirror the inference rules for the standard polymorphic lambda calculus with conditionals [161], apart from D-LET and D-ABS which deal with identifier patterns, and D-APP which deals with dependent type schemes.

$$\frac{}{\Gamma \vdash () : \texttt{Null} \hookrightarrow \{\}} \text{(D-Null)}$$

$$\frac{}{\Gamma \vdash b : \texttt{Bool} \hookrightarrow \{\}} \text{(D-Bool)} \qquad \frac{}{\Gamma \vdash f : \texttt{Float} \hookrightarrow \{\}} \text{(D-Float)}$$

$$\frac{}{\Gamma \vdash s : \texttt{String} \hookrightarrow \{\}} \text{(D-String)} \qquad \frac{}{\Gamma \vdash n : \texttt{Int} \hookrightarrow \{\}} \text{(D-Int)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad \ldots \quad \Gamma \vdash e_n : T_n \hookrightarrow C_n \qquad C' = C_1 \cup \ldots \cup C_n \cup \{T_2 = T_1, \ \ldots, \ T_n = T_1\}}{\Gamma \vdash [e_1, \ \ldots, \ e_n] : \texttt{List}\ T_1 \hookrightarrow C'} \text{(D-List)}$$

$$\frac{Y_1, \ \ldots, \ Y_n \text{ are fresh vars} \qquad \Gamma(x) = \forall X_1, \ \ldots, \ X_n \cdot T}{\Gamma \vdash x : [X_1 \mapsto Y_1 \ \ldots, \ X_n \mapsto Y_n]T} \text{(D-Var)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad \Gamma \vdash e_2 : T_2 \hookrightarrow C_2 \quad \Gamma \vdash e_3 : T_3 \hookrightarrow C_3 \qquad C' = C_{1,2,3} \cup \{T_1 = \texttt{Bool}, T_2 = T_3\}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T_2 \hookrightarrow C'} \text{(D-If)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \ \ldots \ \Gamma \vdash e_n : T_n \hookrightarrow C_n \qquad C' = C_1 \cup \ldots \cup C_n}{\Gamma \vdash (e_1, \ \ldots, \ e_n) : (T_1, \ \ldots, \ T_n) \hookrightarrow C'} \text{(D-Tup)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad T_2 = \mathit{fresh}(x) \quad x_1 : t_1, \ \ldots, \ x_n : t_n = \mathit{bind}(x, \ T_2) \quad \Gamma, x_1 : t_1, \ \ldots, \ x_n : t_n \vdash e_2 : T_3 \hookrightarrow C_2 \qquad C' = C_1 \cup C_2 \cup \{T_1 = T_2\}}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : T_3 \hookrightarrow C'} \text{(D-Let)}$$

$$\frac{T_1 = \mathit{fresh}(x) \quad x_1 : t_1, \ \ldots, \ x_n : t_n = \mathit{bind}(x, \ T_1) \quad \Gamma, x_1 : t_1, \ \ldots, \ x_n : t_n \vdash e_1 : T_2 \hookrightarrow C}{\Gamma \vdash \backslash x\ \texttt{->}\ e_1 : T_1 \to T_2 \hookrightarrow C} \text{(D-Abs)}$$

$$\frac{\text{X fresh var} \quad \Gamma \vdash e_1 : T_1 \hookrightarrow C_1 \quad \Gamma \vdash e_2 : T_2 \hookrightarrow C_2 \qquad C' = C_{1,2} \cup \{T_1 = T_2 \to \mathrm{X}\} \cup \mathit{gdc}(T_1, e_2)}{\Gamma \vdash e_1\ e_2 : \mathrm{X} \hookrightarrow C'} \text{(D-App)}$$

Figure 5.4: DDL type rules $\mathrm{T}_{\mathrm{infer}}$

Instead of just binding to a single variable x, lambda abstractions and `let`-expressions in Flocc can bind expressions to identifier patterns, which are tuples of variables. For example, the expression `let (x,y) = (1,2) in e` is equivalent to `let x = 1 in (let y = 2 in e)`. To infer types for these expressions we use two functions: *fresh* (cf. Figure 5.2) and *bind* (cf. Figure 5.3). *fresh* creates a tuple of fresh type variables that matches the shape of an identifier pattern x, and *bind* binds the leaf types of a compatible (tuple) type T to the variables in the identifier pattern x. The D-Let and D-Abs rules use these functions to create a tuple of fresh type variables, and extend the type environment Γ with a binding between each variable and its corresponding fresh type variable.

D-App applies the *gdc* (generate dependent constraints) function to return additional constraints which bind any Π-bound type variables (in T_1) to their respective AST terms at function applications (e_2), so that uses of these variables must match the AST terms specified. *gdc* is able to inspect T_1 before the constraints in C_1 have been solved because only library functions have Π-types, and the constant propagation phase ensures that library function variables will be directly accessible at their applications. This means that functions with Π-types will always be handled by D-Var, which returns T' with an empty set of constraints. Therefore, C_1 will always be $\{\}$ whenever T_1 is a Π-type.

There are no typing rules required for maps and arrays, because both of these can be defined in terms of list literals which are handled by D-List, and then converted into maps and arrays using the `listToMap` and `listToArr` conversion functions respectively (cf. Figure B.4). Furthermore, maps with sequences of integers as keys can be generated by calling the `intRangeMap` function. `Map` and `Arr` types therefore enter the type system via these functions.

We now show that our DDL type system is sound. That is, that well typed programs can always *progress* (cf. Theorem 3), and evaluations of well typed programs are still well typed (cf. Theorem 6). These proofs are based closely on those found in [161]. The inversion lemma (cf. Lemma 1) that follows shows for a term of each syntactic form, how to calculate its type (if it has one) from the types of its sub-terms.

Lemma 1 (Inversion of the typing relation)**.**
1) If `True` : R, then $R =$ `Bool`.
2) If `False` : R, then $R =$ `Bool`.
3) If `()` : R, then $R =$ `Null`.
4) If $i : R$, where i is an integer literal, then $R =$ `Int`.
5) If $f : R$, where f is a floating point literal, then $R =$ `Float`.
6) If $s : R$, where s is a string literal, then $R =$ `String`.
7) If $(e_0, \ldots, e_n) : R$, then $R = (T_0, \ldots, T_n)$ and $e_0 : T_0, \ldots, e_n : T_n$.
8) If $[e_1, \ldots, e_n] : R$, then $R =$ `List` T and $e_1 : T, \ldots, e_n : T$.
9) If `\`x `->` $e : R$, then $R = T_1 \to T_2$ and $x : T_1 \vdash e : T_2$.
10) If $e_1\ e_2 : R$, then $R = T_2$, $e_1 : T_1 \to T_2$ and $e_2 : T_1$.

11) If `let` x = e_1 `in` $e_2 : R$, then $R = T_2$, $e_1 : T_1$ and $x : T_1 \vdash e_2 : T_2$.
12) If `if` e_0 `then` e_1 `else` $e_2 : R$, then e_0 : `Bool`, $e_1 : R$ and $e_2 : R$.

Proof. Immediate from the definition of the typing relation. □

The canonical forms lemma (cf. Lemma 2) defines the canonical expressions of each type. These expressions are program terms in the syntactic domain T_v (cf. Figure 3.1), i.e., expressions in weak head normal form.

Lemma 2 (Canonical forms)**.** For all $v \in T_v$ we have

1. If v : `Bool`, then v is either `True` or `False`.
2. If v : `Int`, then v is an integer numeric literal.
3. If v : `Float`, then v is a floating point numeric literal.
4. If v : `String`, then v is a string literal.
5. If v : `Null`, then v is `()`.
6. If $v : (T_0, \ldots, T_n)$ then v is an expression of the form $(v_0, \ldots, v_n)$ where $v_0 : T_0, \ldots$, and $v_n : T_n$.
7. If v : `List` T then v is an expression of the form $[v_0, \ldots, v_n]$ where for all v_k in $\{v_0, \ldots, v_n\}$, $v_k : T$.
8. If $v : T_1 \to T_2$ then $v = \backslash x$ `->` e.

Proof. Proceeds by case analysis on the syntax of DDL types defined by $dt \in T_{dt}$ (cf. Figure 4.2). For each alternative, we take the syntax of values defined by $v \in T_v$ (cf. Figure 3.1) and exclude those terms whose type cannot be of the current form, according to the inversion lemma. For example, for $t =$ `Bool` none of the values $v \in T_v$ can have type `Bool`, except for `True` and `False`. □

Now we have defined the canonical forms (i.e., values) of each type, we can prove that all well-typed expressions that are not values can take an evaluation step, i.e., all well typed programs can progress (c.f Theorem 3). This is the first requirement for soundness. Here, a *closed* expression is one that does not contain any unbound variables. All complete programs are *closed*.

Theorem 3 (Progress)**.** Suppose e is a closed, well-typed expression (i.e., $\vdash e : T$ for some T). Then either e is a value or else there is some e' with $e \rightsquigarrow e'$.

Proof. Proof by induction on a derivation of $e : T$, proceeding by case analysis on the last rule applied. The D-Int, D-Bool, D-Float, D-String, D-Null and D-Abs cases are immediate, since in these cases e is a value. For the other cases we have:

Case D-If: By induction hypothesis, either e_1 is a value, or else there is some e_1' s.t. $e_1 \rightsquigarrow e_1'$. If e_1 is a value then the canonical forms lemma tells us that it must be either `True` or `False`, in which case either E-IfThen or E-IfElse applies to e. Otherwise $e_1 \rightsquigarrow e_1'$, and then $e \rightsquigarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ by E-If.

Case D-Tup: By induction hypothesis, for all e_k in $(e_1, \ldots, e_n)$ we have that e_k is a value or else there is some e_k' s.t. $e_k \rightsquigarrow e_k'$. If for all k e_k is a value then so is e by the canonical forms lemma. Otherwise if $e_k \rightsquigarrow e_k'$ then by E-Tup $e \rightsquigarrow (e_1, \ldots, e_k', \ldots, e_n)$.

Case D-List: Similar to D-Tup.

Case D-App: By induction hypothesis, the function e_1 is either a value or $e_1 \rightsquigarrow e_1'$, and the expression e_2 to which it is applied, is either a value or $e_2 \rightsquigarrow e_2'$. If e_1 is not a value then $e \rightsquigarrow e_1'\ e_2$ by E-App1. If e_2 is not a value then $e \rightsquigarrow e_1\ e_2'$ by E-App2. Otherwise, if e_1 is a value then e_1 must be a lambda abstraction by the canonical forms lemma. e_1 may therefore either be of the form $\backslash(x_0, \ldots, x_n)\ \texttt{->}\ e_3$, in which case e_2 must be a tuple value of the form $(e_{2_1}, \ldots, e_{2_n})$ by induction hypothesis, and E-AppTup applies, or of the form $\backslash x\ \texttt{->}\ e_3$ in which case we get $e \rightsquigarrow [x \mapsto e_2]e_3$ by E-App.

Case D-Var: This case cannot apply since e is closed, and so does not contain any free variables.

Case D-Let: Similar to D-App. By induction hypothesis either e_1 is a value, or $e_1 \rightsquigarrow e_1'$. If e_1 is not a value then $e \rightsquigarrow \texttt{let}\ i\ =\ e_1'\ \texttt{in}\ e_2$ by E-Let1. Otherwise if e_1 is a value then it may either be of the form $(v_1, \ldots, v_n)$ or not. If it is of this form then x may either be of the form $(x_1, \ldots\ x_n)$ in which case we have $e \rightsquigarrow \texttt{let}\ x_1\ =\ v_1\ \texttt{in}\ \ldots\ (\texttt{let}\ x_n\ =\ v_n\ \texttt{in}\ e_2)$ by E-LetTup, or just of the form x, in which case we have $e \rightsquigarrow [x \mapsto e_1]e_2$ by E-AppLet. If e_1 is not of this form, but is rather of the form v, then x must also be of the form x (or else it would not be well typed by D-Let), and we have $e \rightsquigarrow [x \mapsto v]e_2$ by E-AppLet.

□

The second requirement for soundness is that evaluating an expression yields an expression with the same type (cf. Theorem 6). To prove this we must first show that weakening (i.e., adding a variable binding, cf. Lemma 4) typing environments does not affect the typing judgments that can be made under them, and that substituting a variable with an expression of the same type preserves the type of the original expression (cf. Lemma 5). Lemma 4 says that adding a new variable to an environment does not affect a typing judgment made under it.

Lemma 4 (Weakening)**.** If $\Gamma \vdash e : T$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, x : S \vdash e : T$. Moreover, the latter derivation has the same depth as the former.

Proof. Straightforward induction on typing derivations. □

In Lemma 5 we show that substituting a variable x in an expression e for an expression of the same type S does not change the type of e. In fact, that type can now be derived without the need for x in the typing environment.

Lemma 5 (Preservation of types under substitution)**.** If $\Gamma, x : S \vdash e : T$, and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]e : T$.

Proof. By induction on a derivation of the statement $\Gamma, x : S \vdash e : T$. For a given derivation, we proceed by cases on the final typing rule used in the derivation. Cases D-Bool, D-Null, D-Int, D-Float, and D-Str are only applicable if e is a literal. Hence the substitution $[x \mapsto s]e$ will not change the value for these cases.

Case D-Var: Let $e = z$ with $z : T$ in $(\Gamma, x : S)$. z must either be x or another variable. If $z = x$ then $T = S$ and $[x \mapsto s]x = s$, and since $\Gamma \vdash s : S$, $\Gamma \vdash [x \mapsto s]x : S$. Otherwise $[x \mapsto s]z = z$, i.e., z is unchanged, and so its type is still T.

Case D-Abs: Let $e = \backslash y \texttt{ -> } e_1$, $T = T_2 \to T_1$, and $\Gamma, x : S, y : T_2 \vdash e_1 : T_1$. We know $x \neq y$ and y not in $\text{ftv}(s)$ (since all abstraction variables have unique names). Since Γ is a map we can permute the bindings to give $\Gamma, y : T_2, x : S \vdash e_1 : T_1$. Using weakening on $\Gamma \vdash s : S$ we get $\Gamma, y : T_2 \vdash s : S$. This allows us to apply the induction hypothesis to yield $\Gamma, y : T_2 \vdash [x \mapsto s]e_1 : T_1$. Then by D-Abs, we get $\Gamma \vdash \backslash y \texttt{ -> } [x \mapsto s]e_1 : T_2 \to T_1$ which is what is required because $[x \mapsto s]e = \backslash y \texttt{ -> } [x \mapsto s]e_1$.

Case D-App: Let $e = e_1\ e_2$, $\Gamma, x : S \vdash e_1 : T_2 \to T_1$, $\Gamma, x : S \vdash e_2 : T_2$, and $T = T_1$. By induction hypothesis, $\Gamma \vdash [x \mapsto s]e_1 : T_2 \to T_1$ and $\Gamma \vdash [x \mapsto s]e_2 : T_2$. By D-App, $\Gamma \vdash [x \mapsto s]e_1\ [x \mapsto s]e_2 : T$ which is equivalent to $\Gamma \vdash [x \mapsto s]\texttt{(}e_1\ e_2\texttt{)} : T$.

Case D-Tup: Let $e = \texttt{(}e_0, \ \ldots, \ e_n\texttt{)}$, $T = \texttt{(}T_0, \ \ldots, \ T_n\texttt{)}$, and $\Gamma, x : S \vdash e_k : T_k$ for all $k \in \{0, \ \ldots, \ n\}$. Applying the induction hypothesis for all k gives $\Gamma \vdash [x \mapsto s]e_k : T_k$, from which we get $\Gamma \vdash \texttt{(}[x \mapsto s]e_0, \ \ldots, \ [x \mapsto s]e_n\texttt{)} : T$ by D-Tup, and therefore $\Gamma \vdash [x \mapsto s]\texttt{(}e_0, \ \ldots, \ e_n\texttt{)} : T$ by definition of substitution.

Case D-List: Let $e = \texttt{[}e_0, \ \ldots, \ e_n\texttt{]}$, $T = \texttt{List}\ T_1$, and $\Gamma, x : S \vdash e_k : T_1$ for all $k \in \{0, \ \ldots\ n\}$. Applying the induction hypothesis for all k gives $\Gamma \vdash [x \mapsto s]e_k : T_1$, which gives $\Gamma \vdash \texttt{[}[x \mapsto s]e_0, \ \ldots, \ [x \mapsto s]e_1\texttt{]} : T$ by D-List, and therefore $\Gamma \vdash [x \mapsto s]\texttt{[}e_0, \ \ldots, \ e_n\texttt{]} : T$.

Case D-If: Let $e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, $\Gamma, x : S \vdash e_0 : \texttt{Bool}$, $\Gamma, x : S \vdash e_1 : T$, and $\Gamma, x : S \vdash e_2 : T$. Applying the induction hypothesis to each subexpression yields $\Gamma \vdash [x \mapsto s]e_0 : \texttt{Bool}$, $\Gamma \vdash [x \mapsto s]e_1 : T$, and $\Gamma \vdash [x \mapsto s]e_2 : T$. This gives $\Gamma \vdash [x \mapsto s]e : T$ by D-If and the definition of substitution.

Case D-Let: Let $e = \texttt{let}\ y\ =\ e_0\ \texttt{in}\ e_1$, $\Gamma, x : S \vdash e_0 : T_0$, $\Gamma, x : S, y : T_1 \vdash e_1 : T_1$, and $T = T_1$. We proceed as in case D-Abs. We know $x \neq y$ and y not in $\text{ftv}(s)$ because all let-bound variables have unique names. We can apply the induction hypothesis directly to e_0 to give $\Gamma \vdash [x \mapsto s]e_0 : T_0$. Then for e_1 permuting the sub-derivation gives $\Gamma, y : T_0, x : S \vdash e_1 : T_1$, and weakening of $\Gamma \vdash s : S$ gives $\Gamma, y : T_0 \vdash s : S$. We can now apply the induction hypothesis to give $\Gamma, y : T_0 \vdash [x \mapsto s]e_1 : T_1$, and so $\Gamma \vdash \texttt{(let}\ y\ =\ [x \mapsto s]e_0\ \texttt{in}\ [x \mapsto s]e_1\texttt{)} : T$ by D-Let i.e., $\Gamma \vdash [x \mapsto s]e : T$.

□

Now we can show that typing judgments are preserved under evaluation, i.e., if e has type T then all evaluations of e will also have type T.

Theorem 6 (Preservation)**.** If $\Gamma \vdash e : T$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : T$.

Proof. Proof by induction on a derivation of $e : T$, proceeding by case analysis on the final rule applied. *Cases* D-Bool, D-Null, D-Int, D-Float, D-String and D-Abs: We know for these rules that e is always a value, so no e' exists s.t. $e \rightsquigarrow e'$, and the requirements of the theorem are satisfied vacuously.

Case D-If: Let $e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, $e_0\ :\ \texttt{Bool}$, $e_1 : T$, $e_2 : T$. There are three evaluation rules by which $e \rightsquigarrow e'$ could be derived: E-IfThen, E-IfElse and E-If.

Subcase E-IfTrue: $e_0 = \texttt{True}$ and $e' = e_1$. We know $e_1 : T$ from the assumptions of D-If, and so $e' : T$.

Subcase E-IfFalse: Similar.

Subcase E-If: $e_0 \rightsquigarrow e_0'$ and $e' = \texttt{if}\ e_0'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. From the assumptions of D-If we know $e_0 : \texttt{Bool}$. Applying the inductive hypothesis gives $e_0' : \texttt{Bool}$ and we know $e_1 : T$ and $e_2 : T$ by the assumptions of D-If. So $\texttt{if}\ e_0'\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ e_2 : T$ by D-If, i.e., $e' : T$.

Case D-Tup: Let $e = \texttt{(}e_0, \ \ldots, \ e_n\texttt{)}$, $T = (T_1, \ \ldots, \ T_n)$, and for all k $e_k : T_k$. The only evaluation rule that can be applied is E-Tup with $e_k \rightsquigarrow e_k'$ and $e' = \texttt{(}e_0, \ \ldots, \ e_k', \ \ldots, \ e_n\texttt{)}$. For any k, we know $e_k : T_k$, and so by the inductive hypothesis $e_k' : T_k$. This gives $\texttt{(}e_0, \ \ldots, \ e_k', \ \ldots, \ e_n\texttt{)} : (T_0, \ldots, \ T_k, \ \ldots, \ T_n)$ by D-Tup and thus $e' : T$.

Case D-List: Let $e = [e_0, \ \ldots, \ e_n]$, $T = \texttt{List}\ T_1$, and for all k $e_k : T_1$. The only evaluation rule that can be applied is E-List with $e_k \rightsquigarrow e_k'$ and $e' = [e_0, \ \ldots, \ e_k', \ \ldots, \ e_n]$. For all k, we know $e_k : T_1$, and so by the inductive hypothesis $e_k' : T_1$. Thus $[e_0, \ \ldots, \ e_k', \ \ldots, \ e_n] : \texttt{List}\ T_1$ by D-List, which is equivalent to $e' : T$.

Case D-VAR: Let $e = x$ and $x : T$. This case can never occur i.e., there is no e' s.t. $e \rightsquigarrow e'$ since full programs are always closed (i.e., contain no free variables), and so variables will always be substituted for other expressions before they need evaluation.

Case D-APP: Let $e = e_0\ e_1$, $e_0 : T_1 \to T_2$, $e_1 : T_1$, and $T = T_2$. There are four evaluation rules that could be applied: E-APP, E-APPTUP, E-APP1, and E-APP2.

Subcase E-APP: Let $e_0 = \texttt{\textbackslash}x\ \texttt{->}\ e_2$ and $e' = [x \mapsto e_1]e_2$. We know $\Gamma \vdash e_1 : T_1$ through the assumptions of D-APP, and $\Gamma, x : T_1 \vdash e_2 : T_2$ through the assumptions of D-ABS (the final rule applied in the subderivation of e_0). Therefore, $\Gamma \vdash [x \mapsto e_1]e_2 : T$ by the substitution lemma, which is $e' : T$, and we are done.

Subcase E-APPTUP: Let $e_0 = \texttt{\textbackslash(}x_0, \ldots, x_n\texttt{)}\ \texttt{->}\ e_2$, $e_1 = \texttt{(}a_0, \ldots, a_n\texttt{)}$, $e' = \texttt{(\textbackslash}x_0\ \texttt{->}\ \ldots \texttt{(\textbackslash}x_n\ \texttt{->}\ e_2\texttt{)}a_n \ldots\texttt{)}a_0$, $T_1 = (t_0, \ldots, t_n)$, and for all k $a_k : t_k$. By the assumptions of D-ABS (the rule needed to derive e_0) we know that $\Gamma, x_0 : t_0, \ldots, x_n : t_n \vdash e_2 : T_2$. Then to derive $e' : T$, for all $k = n \ldots 0$ we apply D-ABS to get $\Gamma, x_0 : t_0, \ldots, x_k : t_k \vdash \texttt{(\textbackslash}x_k\ \texttt{->}\ \ldots\texttt{)} : t_k \to T_2$ and then D-APP to get $\Gamma \vdash \texttt{(\textbackslash}x_k\ \texttt{->}\ \ldots\texttt{)}a_k : T_2$. Here the final D-APP gives $\Gamma \vdash \texttt{(\textbackslash}x_0\ \texttt{->}\ \ldots\texttt{)}a_0 : T_2$ which is equivalent to $e' : T$.

Subcase E-APP1: Let $e_0 \rightsquigarrow e_0'$ and $e' = e_0'\ e_1$. From the assumptions of D-APP we know $e_0 : T_1 \to T_2$ and $e_1 : T_1$. Applying the inductive hypothesis gives $e_0' : T_1 \to T_2$ which gives $e_0'\ e_1 : T$ by D-APP.

Subcase E-APP2: Let $e_1 \rightsquigarrow e_1'$ and $e' = e_0\ e_1'$. From the assumptions of D-APP we know $e_0 : T_1 \to T_2$ and $e_1 : T_1$. Applying the inductive hypothesis gives $e_1' : T_1$, which gives $e_0\ e_1' : T$ by D-APP.

Case D-LET: Let $e = \texttt{let}\ x\ =\ e_0\ \texttt{in}\ e_1$, $e_0 : T_1$, $\Gamma, x : T_1 \vdash e_1 : T_2$, $T = T_2$. There are three evaluation rules that could be applied: E-LET, E-LETTUP, and E-LET1.

Subcase E-LET: Let $e' = [x \mapsto e_0]e_1$. We know $\Gamma, x : T_1 \vdash e_1 : T_2$ and $\Gamma \vdash x : T_1$ by the assumptions of D-LET, and so by the substitution lemma we have $\Gamma \vdash [x \mapsto e_0]e_1 : T_2$ i.e., $e' : T$ and we are done.

Subcase E-LETTUP: Let $e_0 = \texttt{(}a_0, \ldots, a_n\texttt{)}$, $x = \texttt{(}x_0, \ldots, x_n\texttt{)}$, $e' = \texttt{let}\ x_0\ =\ a_0\ \texttt{in}\ \ldots\ \texttt{let}\ x_n\ =\ a_n\ \texttt{in}\ e_n$, $T_1 = \texttt{(}t_0, \ldots, t_n\texttt{)}$, and for all k $a_k : t_k$. We know from the assumptions of D-LET that $\Gamma, x_0 : t_0, \ldots, x_n : t_n \vdash e_1 : T_2$. This means for all $k = n \ldots 0$ we can apply D-LET to get $\Gamma, x_0 : t_0, \ldots, x_{k-1} : t_{k-1} \vdash \texttt{(let}\ x_k\ =\ a_k\ \texttt{in}\ \ldots\texttt{)} : T_2$, which finally gives $\Gamma \vdash \texttt{(let}\ x_0\ =\ a_0\ \texttt{in}\ \ldots\texttt{)} : T_2$ and thus $\Gamma \vdash e' : T$.

Subcase E-LET1: Let $e_0 \rightsquigarrow e_0'$ and $e' = \texttt{let}\ x\ =\ e_0'\ \texttt{in}\ e_1$. From the assumptions of D-LET we know $e_0 : T_1$, which gives $e_0' : T_1$ by the inductive hypothesis. Combining this with $e_1 : T_2$ (assumption of D-LET) gives $\texttt{let}\ x\ =\ e_0'\ \texttt{in}\ e_1 : T_2$ by D-LET i.e., $e' : T$.

□

Since well-typed programs can always progress, and evaluation preserves well-typedness, we say that our DDL type system is *sound*.

5.1.2 Testing for function equality

Type checking (and type inference) require testing whether type terms are equal. This is just a straightforward syntactic test for normal types, but our DDL types contain *functions*. This poses a problem, since testing whether two functions f and g are extensionally equal (i.e., $f \equiv g$) is undecidable in general [200, 171], and as such no algorithm exists that can perform this test for all cases. We are therefore forced to adopt a sound but incomplete under-approximation, which means that our type-checker (or inference algorithm) will be incomplete (i.e., some valid typings will be rejected by the checker/will not be found by the inference algorithm).

The first possible approximation is just to check for syntactic equality up to variable renaming. This is valid since syntactic equality implies semantic equality, i.e., $f = g \implies f \equiv g$. Here, $f \equiv g$ means that the type of f equals the type of g, and for all x, $(f\ x) = (g\ x)$. This will work in some cases, but cases where semantically equivalent functions are syntactically different frequently occur, especially since we build up functions using function compositions in our DDL types.

The second possibility is to convert functions to some common form and then compare them syntactically (up to renaming). This is the approach that we adopt in the current chapter. We use a normalization function $\downarrow$ (cf. Figure 5.5) to convert functions to a common simplified form (i.e., head normal form) and then compare these syntactically. This is valid as long as the $\downarrow$ is sound, i.e., $\downarrow f \equiv f$ (which we prove in Lemma 7). This approach is complete for valid projection/permutation functions (i.e., functions that accept tuples of variables, and return some tuples of those same input variables and perhaps some constants, cf. Lemma 10). `DArr` partition functions can always be reduced to such projection functions, since they are composed from index projection functions, which just map indices to new tuples of indices. For example, `(id·\(`x, y`) -> fst x)` $\doteq$ `(fst·fst)` can be simplified by desugaring `fst`, `id`, and the ·-operator, and then reducing the bodies of the lambda-abstractions using beta-reduction, to give `(\(((`a,b`),`c`) ->` a`)` $\doteq$ `(\(((`j,k`),`h`) ->` j`)`, which are syntactically equal up to renaming. Here, $\alpha \doteq \beta$ denotes a constraint equating α and β, which may or may not hold. Furthermore, `DList` types do not use functions, and so syntactic equality up to variable renaming of normalized function terms is sufficient for `DArr`s and `DList`s.

However, `DMap` partition and local layout functions can be more complex, and may include arbitrary function applications and `let`-expressions etc. That said, the examples

```
1 ↓f =
2   let f0 = desugar f in
3   let normalize f1 =
4     let f2 = applyFunGens f1 in
5     let f3 = eval f2 in
6     if f3=f2 then return f3
7     else return (normalize f3)
8   in recur f0
```

FIGURE 5.5: Normalizes embedded functions.

used in Section 4.5 show that these functions are quite simple in many practical cases. For example, none of the partition functions in the examples are recursive or iterative, and most are projection functions, in which case the current approximation suffices. Furthermore, types are not compared in all the typing rules (cf. Figure 5.4, only in D-LIST, D-IF, D-LET, and D-APP) and so situations where semantically equivalent functions are treated as non-equal are less common than one might imagine.

Function normalization The ↓-operator tries to reduce functions to a common syntactical form. As shown in Figure 5.5, the ↓-operator's implementation starts by replacing all functions and function compositions with the concrete lambda terms defined in Figure 4.3 (via `desugar`). It then iteratively applies `applyFunGens` and `eval`, until `eval` does not modify f_2 and so $f_2 = f_3$. `applyFunGens` f_1 applies any function generators (cf. Section 4.3.3) in f_1, replacing them with concrete lambda-abstractions, and `eval` f_2 applies the language's evaluation rules (cf. Figure 3.2) to all terms and sub-terms in f_2, until no more can be applied. This normalization function works in many situations, including the examples in this chapter (cf. Section 4.5). However, Chapter 6 presents a more nuanced approach that can find solutions to equations that this algorithm cannot. We now prove that ↓ is sound (i.e., that ↓ f is semantically equivalent to f), and that it terminates.

Lemma 7 (Soundness of ↓)**.** $\downarrow f \equiv f$.

Proof. $f_0 \equiv f$ by definition of `desugar` (since it replaces library function variables with their semantically equivalent definitions). $f_2 \equiv f_1$ by soundness of `applyFunGens`. $f_3 \equiv f_2$ by definition of `eval`, since it applies the reduction rules defined in the operational semantics shown in Figure 3.2. Finally, line 6 is sound trivially, and the recursive call `normalize` f_3 is sound inductively (i.e., by soundness of ↓). □

Lemma 8 (Termination of ↓)**.** For any input ↓ will always terminate.

Proof. `desugar` is only applied once and always terminates. Now consider the pair (m, n) where m is the number of inapplicable function generators in f, and n is the number of

reducible expressions in f. `eval` always terminates since primitive recursion is not currently possible in Flocc (cf. Section 3.3), and `eval` does not apply built-in combinators or library functions. `eval` therefore makes $n = 0$ and reduces m, since it reduces all reducible expressions, including those that are making function generators (cf. Section 4.3.3) inapplicable. `applyFunGens` may increase n, but cannot increase m since applying function generators never creates more. Therefore, every application of `normalize` decreases (m, n) with respect to the lexicographical order, such that eventually $m = 0$ and $n = 0$, at which point $f_3 = f_2$ on line 6, and $\downarrow$ terminates. □

Projection functions are those expressions that can be built from the non-terminal f in Figure 5.6. That is, expressions where all functions are (desugared to be) lambda abstractions, and expressions can only be applications of such functions, tuple expressions, variables, and scalar literals. Fully simplified projection functions are those that can be built from f' in Figure 5.6, i.e. projection functions which contain no function applications. We now show that every projection function can be reduced to a fully simplified projection function, and that such a simplified projection function is a unique canonical form of all the projection functions that can be reduced to it.

$$f ::= \backslash x \texttt{ -> } e$$

$$e ::= \text{Id} \mid f\ e \mid (e_1, \ldots, e_k) \qquad f' ::= \backslash x \texttt{ -> } e'$$

$$x ::= \text{Id} \mid \_ \mid (x_1, \ldots, x_k) \qquad e' ::= \text{Id} \mid (e'_1, \ldots, e'_k)$$

FIGURE 5.6: Syntax of projection functions

Lemma 9 (Simplification of projection functions)**.** Every projection function f can be converted into a fully simplified projection function f', and $\downarrow f$ performs this simplification.

Proof. We proceed by induction over expressions built from f in Figure 5.6.
Base case: the most deeply nested expressions in f must be either variables, or tuples of variables, and are thus already in fully simplified form.
Inductive case: every expression e_k in f must be either a variable, tuples of expressions, or a function application. If e_k is a variable or tuple of variables, then it is already in fully simplified form. If e_k is a function application $f_k\ e_1$, then e_1 can always be reduced to a variable or tuple of variables e'_1, by the inductive hypothesis and E-App2. If f_k is of the form `\(`$x_0, \ldots, x_n$`) -> (`$e_0, \ldots, e_n$`)` then E-AppTup can be applied to decompose it into successive applications of the form $\backslash x$ `->` e. Once f_k is of the form $\backslash x$ `->` e beta-reduction (i.e. E-App) can be applied to e_k to yield an expression e'_k which is a variable, or tuple of variables. Therefore, all function applications can be eliminated from projection functions, by recursively applying E-AppTup, E-App, and E-App2, which converts them into fully simplified projection functions. `eval` in $\downarrow$ applies these

rules until no more apply, and so $\downarrow$ converts projection functions into fully simplified ones. □

Now we show that every pair of projection functions are semantically equal, exactly when their fully simplified forms are syntactically equal, up to variable renaming. That is, fully simplified forms of projection functions are canonical.

Lemma 10 (Completeness of $\downarrow$ for projection functions)**.** $\forall f, g : (f \equiv g) \Leftrightarrow ((\downarrow f) = (\downarrow g))$, where f and g are projection functions.

Proof. We show by contradiction that full simplification of projection functions is canonical. Assume there exist f and g s.t. $f \equiv g$, but $(\downarrow f) \neq (\downarrow g)$. Since $(\downarrow f) \neq (\downarrow g)$, $\downarrow f$ and $\downarrow g$ must differ syntactically in some way. They must both be of the form $\backslash x \texttt{ -> } e'$, so either $x_f \neq x_g$ or $e'_f \neq e'_g$, up to variable renaming. x_f and x_g can either be variables, or (possibly nested) tuples of variables. If $x_f \neq x_g$ up to variable renaming, e.g., $x_f = (a, b)$ and $x_g = c$, then f and g's input types would differ, which would contradict $f \equiv g$, and so $x_f = x_g$. If e'_f was a variable and e'_g a tuple, or visa versa, then their types would differ since the most-general type for a variable is a single type variable α, and the most general-type for a tuple is a tuple of type variables $(\alpha_0, \ldots, \alpha_n)$, and so either they are both variables, or both tuples. If they are both variables then they must be equal up to variable renaming, because if they were not they would return different arguments and contradict $f \equiv g$ (since $f \equiv g \implies \forall x\ (f\ x) = (g\ x)$). If they are tuples $(e_{f_1}, \ldots, e_{f_m})$ and $(e_{g_1}, \ldots, e_{g_n})$, then m must equal n or else their types would differ. If they are tuples of the same arity, then $\forall k \in \{1, \ldots, m\}\ e_{f_k}$ must equal e_{g_k} by induction. Therefore, $\forall f, g : f \equiv g \implies (\downarrow f) = (\downarrow g)$, i.e., $\forall f : \downarrow f$ is canonical. Furthermore, $\forall f, g : (\downarrow f) = (\downarrow g) \implies f \equiv g$, since $(\downarrow f) = f$ and $(\downarrow g) = g$ by soundness of $\downarrow$, and $\forall f, g : f = g \implies f \equiv g$, because all syntactically equal functions are also semantically equal. Therefore, $\forall f, g : (f \equiv g) \Leftrightarrow ((\downarrow f) = (\downarrow g))$. □

5.1.3 Unifying functions

The other problem is that type inference algorithms need to *unify* equations between type terms. Since function equality is undecidable in general, it follows that unification of functions, known as *higher-order unification*, is also undecidable in general [94, 111, 136]. We must therefore adopt a sound but incomplete solution here as well.

Just as syntactic equality is an under-approximation of semantic equality, so syntactic unification is of semantic unification. For example, the equation $f \cdot \texttt{id} \doteq \texttt{fst} \cdot g$ unifies syntactically with substitutions $[f \mapsto \texttt{fst}, g \mapsto \texttt{id}]$. However, syntactic unification will not solve $f \cdot \texttt{id} \doteq \texttt{fst}$ to yield $[f \mapsto \texttt{fst}]$ since the sides are not the same shape, and it cannot make them the same shape, because it does not "know" that $f \cdot \texttt{id} \equiv f$.

$$\frac{}{G \cup \{t \doteq t\} \rightsquigarrow G}\ \textsc{DeleteSyn}$$

$$\frac{t_1 \in T_f \qquad t_2 \in T_f \qquad (\downarrow t_1) = (\downarrow t_2)}{G \cup \{t_1 \doteq t_2\} \rightsquigarrow G}\ \textsc{DeleteFun}$$

$$\frac{t_1 \in T_d \qquad t_2 \in T_d \qquad vars(t_1) = vars(t_2)}{G \cup \{t_1 \doteq t_2\} \rightsquigarrow G}\ \textsc{DeleteDimSet}$$

$$\frac{f \neq g \qquad f \notin T_f \qquad f \notin T_d}{G \cup \{f(s_0, \ldots, s_k) \doteq g(t_0, \ldots, t_m)\} \rightsquigarrow \bot}\ \textsc{Conflict}$$

$$\frac{}{G \cup \{f(s_0, \ldots, s_k) \doteq f(t_0, \ldots, t_k)\} \rightsquigarrow G \cup \{s_0 \doteq t_0, \ldots, s_k \doteq t_k\}}\ \textsc{Decompose}$$

$$\frac{}{G \cup \{f(\ldots) \doteq x\} \rightsquigarrow G \cup \{x \doteq f(\ldots)\}}\ \textsc{Swap}$$

$$\frac{x \notin T_f \qquad x \in vars(t_0, \ldots, t_k)}{G \cup \{x \doteq f(t_0, \ldots, t_k)\} \rightsquigarrow \bot}\ \textsc{Check}$$

$$\frac{x \notin vars(t) \qquad x \in vars(G)}{G \cup \{x \doteq t\} \rightsquigarrow [x \mapsto t]G \cup \{x \doteq t\}}\ \textsc{Eliminate}$$

FIGURE 5.7: DDL type unification algorithm U

Any attempt to solve such equations means exploring a search space of different possible substitutions, and will therefore require backtracking since some choices may lead to dead ends. This will involve some search over the possible values for the abstract functions/function variables. Multiple possible values for type variables also implies that most general unifiers (MGUs) will no longer exist, although the meaning/relevance of MGUs for distributed data layouts is unclear anyway.

For this chapter we adopt an approximation called U (cf. Figure 5.7), that works for many example programs, including those in Section 4.5. U extends the non-deterministic unification algorithm of Martelli and Montanari [141], which transforms a set of equations G by applying the rules defined in Figure 5.7 until no more rules apply or it returns $\bot$ (i.e., fail). Our implementation of this algorithm iterates over the equations in G. For each equation, U transforms it using the first rule from Figure 5.7 that applies, or if none apply, leaves it unchanged. Once all equations have been visited, it starts again, iterating over all the equations in the transformed set G'. U stops when the last pass over G, did not transform any equations (i.e., no rules apply to any of the equations). If the remaining set of equations $G = \{t_1 \doteq t'_1, \ \ldots, \ t_n \doteq t'_n\}$ contains any equations that are not of the form $x \doteq f(\ldots)$ (i.e., not substitutions) then unification fails. Otherwise, if the remaining equations in G are all of the form $x \doteq f(\ldots)$ where every variable x only appears on the left-hand side of one equation, we say that G is in (fully) *solved*

form and unification succeeds with substitutions $\sigma = \{(t_1, t'_1),\ \ldots,\ (t_n, t'_n)\}$.

In addition to the normal constraint deletion rule DeleteSyn, we have DeleteFun which discharges a constraint if two normalized functions (members of T_f, the syntactic domain of embedded functions), are syntactically equal. DeleteDimSet also deletes a constraint between dimension identifier sets (e.g., sets of topology dimensions) if the sets of type variables they contain are equal. Here, most general unifiers still exist, and no-backtracking is required, since the extra rules only delete constraints, i.e., do not generate any substitutions.

The DDL types for the combinator implementations in this chapter typically either use constant functions (e.g., `union` uses `fst`), Π-bound functions (e.g., `eqJoin1`, `groupReduce2`), and/or a single type variable on one side (e.g., the input) and compositions involving that variable and constant/Π-bound functions on the other (e.g., the output). This means that if a valid set of combinator implementations are used, non-trivial function expressions will be inferred from more basic ones, and can then just be compared where they meet, and deleted if semantically equivalent by DeleteFun. For example, `mapArrInv2` has the DDL type:

```
mapArrInv2 :: Π(_,_,f_inv,_) : (i->j, (i,v)->w, j->i,
  DArr i v g d m) -> DArr j w (g · f_inv) d m
```

Here the input g will unify with any term, so at some point g will be replaced with some term t_1 by Eliminate. Then when $g \cdot f_{inv}$ is unified with the input type of its consuming function application t_2, it will be $t \cdot f_{inv}$ and can be compared and deleted if equal to t_2 by DeleteFun. So whenever the consumer's input type t_2 is a simple variable x or an expression inferred from the other direction, our equality check approximation DeleteFun will suffice.

This solution works whenever function compositions unify syntactically, or are equal by the approximation above. This cannot solve equations like $f \cdot \texttt{id} = \texttt{fst}$ unless f becomes `fst` by some other constraint/substitution. That is, we do not search the space of possible values for function variables, but rely on them becoming matching terms via constraints created for other parts of the program. For example, iteration might require a state transformer function to have type $s \to s$. That is, the input data layout has to be the same as the output. In this case the input may have partition function f and the output some composition of constant functions, Π-bound functions, function generators, and f (e.g., $f \doteq (\texttt{id} \otimes \texttt{id}) \cdot f \cdot \texttt{id}$), that when normalized becomes $f \doteq f$. This will unify correctly. The `floyd` all-pairs shortest path algorithm example in Section 4.5 does this. We introduce more complex DDL types in Chapter 6 where inference cannot simply build terms from inputs to outputs, or visa versa, and therefore also present an extended unification algorithm in Section 6.4, that can deal with such types.

We now prove that U terminates for all inputs, whatever non-deterministic choices are made.

Theorem 11 (Termination of U)**.** *U terminates for any input.*

Proof. Consider the triple (l, m, n) where l is the number of variables that occur more than once in G, m is the number of non-variables on the left hand side of constraints, and n is the cardinality of G. Applying the delete rules (DeleteSyn, DeleteFun and DeleteDimSet) reduce n and cannot increase m or l. Applying Decompose, Conflict, and Swap decrease m, and cannot increase l, and Eliminate and Check decrease l. Thus every rule decreases (l, m, n) according to a lexicographical ordering, which can only occur finitely many times. Therefore, U always terminates. □

We now prove that U is sound. That is, if U succeeds, i.e., returns a set of constraints that are all substitutions, then those substitutions make all constraints equivalent (i.e, semantically equal for embedded functions, and dimension sets, and syntactically equal for all other terms).

Theorem 12 (Soundness of U)**.** *If G has no unifier, then U terminates with $\perp$ or a set of equations that are not in solved form. Otherwise, if U terminates with success, then G has been transformed into an equivalent set G', which is in solved form.*

Proof. In syntactic unification there are two kinds of equations that have no unifiers. First, an equation of the form $f'(t'_1, \ldots, t'_n) \doteq f''(t''_1, \ldots, t''_m)$ has no unifier if $f' \neq f''$ (Thm 2.1 in [141]) or if $n \neq m$. Second, an equation of the form $x \doteq t$ where x occurs in t and $x \neq t$ (Thm 2.2 in [141]). Decompose excludes the former and Eliminate the latter, whilst Conflict and Check return $\perp$ for the former and latter respectively. Then for functions $f, g \in T_f$, $(f = g) \implies (f \equiv g)$ means that $(f \not\equiv g) \implies (f \neq g)$, (and similarly for dimension sets), and so equations with no semantic unifiers, also have no syntactic unifiers, and therefore terminate with $(f \doteq g) \in G'$, which is not in solved form. So if G has no syntactic unifiers, U returns $\perp$. Otherwise, if it has no semantic unifiers then G' is not in solved form.

Two sets of equations G and G' are *equivalent* if they have the same sets of unifiers. They are in *solved form* if they are $x_j \doteq t_j, j = 1, \ldots, k$ and for every variable x_j and term t_j with $x_j \notin vars(t_j)$, x_j occurs only there (pp. 260-1 of [141]). All the rules apart from DeleteFun and DeleteDimSet in Figure 5.7 are shown to preserve the sets of all unifiers in [141]. This is obvious for Swap and DeleteSyn. For Decompose any substituion that satisfies $s_0 \doteq t_0, \ \ldots, \ s_k \doteq t_k$, also satisifies $f(s_0, \ \ldots, \ s_k) \doteq f(t_0, \ \ldots, \ t_k)$, and conversely for the recursive definition of a term. For Eliminate any equation $x = t$ belongs to both G and G' and thus any unifier σ (if it exists) of G or G' must unify x and t; that is $\sigma x = \sigma t$. Now let $t_1 \doteq t_2$ be any other equation of G, and let $t'_1 \doteq t'_2$ be the transformed form of $t_1 \doteq t_2$ in G'. Since t'_1 and t'_2 have been obtained

by substituting every occurrence of x in t_1 and t_2 respectively we have $\sigma t_1 = \sigma t_1'$ and $\sigma t_2 = \sigma t_2'$. Thus, any unifier of G is also a unifier of G', and vice versa. CONFLICT and CHECK only apply if G has no unifiers. The additional rules DELETESYN and DELETEDIMSET also preserve the sets of unifiers. For any $t_1, t_2 \in T_f$, if $(t_1 \doteq t_2) \in G$ has syntactic unifiers, then these are preserved by all the rules, and DELETEFUN acts like DELETESYN. If $(t_1 \doteq t_2) \in G$ has no syntactic unifiers, then removing $(t_1 \doteq t_2)$ will not introduce any, and so G' will have the same unifiers as G. A similar argument follows for DELETEDIMSET. Finally, if SWAP, DELETE, and DECOMPOSE cannot be applied, then G' is in solved form. □

U is not complete. This is because unification can fail for constraints that are semantically equivalent, but not equal by $\downarrow$, or where substitutions exist that make constraints equal by $\downarrow$, but where these substitutions can not be inferred syntactically. However, U is complete with respect to syntactic equality, such that if a sequence of rule applications exist that convert the equations into solved form, then this sequence will be found. Here, a sequence of rule applications $\overset{r_i,e_j}{\rightsquigarrow}$, $\overset{r_{i+1},e_{j+1}}{\rightsquigarrow}$, ... applies rule i to equation j, and then rule $i+1$ to equation $j+1$ etc.

Theorem 13 (Completeness of U w.r.t sequence of rule applications)**.** If a sequence of rule applications $\overset{r_i,e_j}{\rightsquigarrow}$, $\overset{r_{i+1},e_{j+1}}{\rightsquigarrow}$, ... exists transforming G into solved form G', then U will transform G into this form.

Proof. U consists of an inner loop that iterates over all the equations in G, applying the first rule in Figure 5.7 that applies to each, and an outer loop which keeps iterating over G, until no more rules apply to any of the equations. If a single sequence of rule applications exists that transforms G into solved form, then U will clearly find this, by performing the only rule application $\overset{r_i,e_j}{\rightsquigarrow}$ possible for every iteration of the outer loop. To show that this holds when multiple rule applications can be performed, we show that deferring a valid rule application $\overset{r_i,e_j}{\rightsquigarrow}$ after some intermediate sequence of applications $\overset{r_k,e_l}{\rightsquigarrow}, \ldots, \overset{r_{k+n},e_{l+m}}{\rightsquigarrow}$, leads to the same final solution as applying $\overset{r_i,e_j}{\rightsquigarrow}$ and then $\overset{r_k,e_l}{\rightsquigarrow}, \ldots, \overset{r_{k+n},e_{l+m}}{\rightsquigarrow}$. This means that where multiple rule applications are possible, any choice will lead to the same eventual solution set.

The only rule that can alter other equations in the set is ELIMINATE. This rule applies a substitution $[x \mapsto f(\ldots)]$ (which is a unifier of G) to all the other equations in G. Therefore, any intermediate sequence of applications $\overset{r_k,e_l}{\rightsquigarrow}, \ldots, \overset{r_{k+n},e_{l+m}}{\rightsquigarrow}$, can be treated as a set of substitutions σ, that contains the substitutions caused by the ELIMINATE rule applications. It therefore suffices to show, that deferring any rule application after some intermediate substitutions σ, leads to the same result as applying it immediately and then applying σ. We now show that this holds for every rule in Figure 5.7:

Case DELETE-SYN:: This rule removes $\{t \doteq t\}$ from G. Deferring this transformation yields $\{\sigma t \doteq \sigma t\}$, which will still be removed.

Case DELETE-FUN:: This rule removes $\{t_1 \doteq t_2\}$ from G where $(\downarrow t_1) = (\downarrow t_2)$. Any type variables in t_1 and t_2 stand for abstract functions. Thus if $(\downarrow t_1) = (\downarrow t_2)$ then $(\downarrow \sigma t_1) = (\downarrow \sigma t_2)$ since any substitutions in t_1 and t_2 will be simplified in the same way by $\downarrow$. Therefore, $\{\sigma t \doteq \sigma t\}$ will still be removed.

Case DELETEDIMSET:: Similar to DELETEFUN.

Case CONFLICT:: Applying σ will not change the functions f and g, their arities k and m, or their syntactic domains, and so if this rule applies to G and returns $\bot$, it will also apply to σG and return $\bot$.

Case DECOMPOSE:: Applying σ will not change the function f or its arity k, so if this rule applies to G it will also apply to σG. Furthermore, applying σ before the rule application yields $\sigma\{f(s_0, \ldots, s_k) \doteq f(t_0, \ldots, t_k)\} \rightsquigarrow \{\sigma s_0 \doteq \sigma t_0, \ldots, \sigma s_k \doteq \sigma t_k\}$ and applying it after the application yields the same $\sigma\{s_0 \doteq t_0, \ldots, s_k \doteq t_k\} = \{\sigma s_0 \doteq \sigma t_0, \ldots, \sigma s_k \doteq \sigma t_k\}$.

Case SWAP:: If σ does not contain x, or maps x to another variable y then applying σ before the rule yields $\sigma\{f(\ldots) \doteq x\} = \{\sigma f(\ldots) \doteq y\}$ and applying it after yields the same $\{\sigma f(\ldots) \doteq y\} \rightsquigarrow \{y \doteq \sigma f(\ldots)\}$. If σ contains $x \mapsto g(\ldots)$ then $\sigma\{f(\ldots) \doteq x\} = \{\sigma f(\ldots) \doteq g(\ldots)\}$, in which case SWAP no longer applies. However, in that case $\sigma\{x \doteq f(\ldots)\} = \{g(\ldots) \doteq \sigma f(\ldots)\}$ which is equivalent to $\{\sigma f(\ldots) \doteq g(\ldots)\}$, since DECOMPOSE and CONFLICT will apply in the same way, and the ordering of the final equations is determined by latter applications of SWAP.

Case CHECK:: If σ does not contain x or maps x to another variable y, then if $x \in vars(t_0, \ldots, t_k)$ then $y \in vars(\sigma t_0, \ldots, \sigma t_k)$ and CHECK will still return $\bot$. Otherwise if σ contains $x \mapsto g(\ldots)$, then CHECK will no longer apply, but if $x \in vars(t_0, \ldots, t_k)$ then $\{g(\ldots) \doteq \sigma f(t_0, \ldots, t_k)\}$ will be an unsolvable constraint (i.e., cannot be transformed to solved form) and so unification will still fail.

Case ELIMINATE:: If σ does not contain x, or maps x to another variable y, then ELIMINATE will still apply after applying σ, and will yield $\sigma G \cup \{y \doteq \sigma t\}$. If σ contains $x \mapsto g(\ldots)$ then ELIMINATE will no longer apply, however in this case applying σ after the ELIMINATE application will lead to the same equation $\{g(\ldots) \doteq \sigma t\}$, and will transform all the other equations in G in the same way as applying σ before.Thus U is complete w.r.t. sequences of rule applications.

□

5.1.4 Variables bound in outer scopes

A difficulty caused by dependent type schemes is that lambda abstractions bound to type variables can reference variables bound outside them. If such a lambda term is

lifted into a type, and then by substitution ends up in a DDL type for an expression which is above the variable binding, the function cannot be used since it cannot access the variable it references. For example, Figure 5.8 shows a program that could cause such an error, and Figure 5.9 a possible, and erroneous, typing for it.

```
1 let x = readMap(...) in
2 let y = reduce (snd, addi, x) in
3 let z = groupReduce (\(k,v)->muli(k,y), snd, addi, x) in z
```

FIGURE 5.8: Example program where a variable escapes during type inference

x	:	`DMap Int Int q (\(k, v) -> muli (k, y)) d m`
reduce	:	$((\texttt{Int}, \texttt{Int}) \rightarrow \texttt{Int}, (\texttt{Int}, \texttt{Int}) \rightarrow \texttt{Int}, \texttt{DMap Int Int q } \alpha \texttt{ d m})) \rightarrow \texttt{Int}$
groupReduce	:	$(\ldots, \texttt{DMap Int Int q (\(k, v) -> muli (k, y)) d m}) \rightarrow \texttt{DMap Int Int q fst d m}$

FIGURE 5.9: Example DDL types where a variable escapes during type inference

Here, the inference process propagates the function `\(k,v) -> muli (k,y)` back up to `x`, but the function references `y`, which is defined after `x`. So the compiler would be unable to generate code to distribute `x` since its distribution is defined in terms of a value that is computed after it. To prevent this, the compiler checks that partition functions do not reference variables defined after the expressions the pertain to.

5.1.5 Default parameter values

After unification has taken place, some type variables may still remain abstract, where the data layout would be valid for any value of them. Before code generation, we replace these variables with concrete values. Different values can be used to explore different concrete DDLs, but we define default values in the table below:

Parameter type	Default value
Partition function	`id`
`DMap` distribution mode	`Hash`
`DList` distribution mode	`Blk`
`DArr` local layout function	`id`
`DMap` local layout function	`nullF`
Dimension ID that is only mirrored	`()`

5.2 Redistribution insertion

We can now enumerate different choices of combinator implementations and infer DDL types for them, if valid types exist. However, the best/fastest distributed-memory programs often involve redistributing some of their collections (i.e., broadcasting an input matrix, to mirror it on all nodes) and changing their local layouts (cf. Section 6.1). We therefore need to consider solutions that redistribute their collections using redistribution/re-layout functions (cf. Figure 6.2, Figure 6.3, Figure 6.5). These can be viewed as implementations of the high-level identity function, or type coercions/casts in our DDL type system. They have different DDL types, converting one data layout to another, by mapping DDL type parameters from one value to another value, while leaving others constant. For example, `repartMap : DMap k v q1 f1 d1 m -> DMap k v q2 f2 d2 m` can change a map partitioned by any function `f1` and mode `q1` along any dimension `d1`, to be partitioned by any other function `f2` and mode `q2` along any other dimension `d2`, whilst it remains mirrored along `m`. Type inference infers values for these parameters, that are then fed to the code generator, which uses them to generate code specialized for the specific input and output data layouts.

The search space of combinator implementation choices for a program is already exponential in the number of combinator applications $O(N^m)$[1], and considering possible redistributions makes it even bigger $O(N^m m P^{P-1})$. However, redistribution functions are only useful when they allow more efficient combinator implementations to be used in a program (e.g., using `eqJoin1` rather than `eqJoin2`). This means we can restrict our attention to uses of redistribution functions that make valid, otherwise invalid choices of combinator implementations. To do this we need an algorithm that takes a program with DDL types that do not type-check, and inserts redistribution function applications at suitable locations to make it type-check.

Redistribution functions Although our type inference algorithm is not complete we can provide a set of redistribution functions, that can be inserted to make any combination of combinator implementations type-check. To do this we need to provide a set of redistribution functions for each distributed collection type, that can be combined to map any of the DDL type parameters from one value to another. This can either be by enumerating concrete values, like `Stm` to `Tree` and `Tree` to `Iter`, or by using type variables like `f1` to `f2`. Then we can search for chains of these functions, to map one DDL type to another. We provide such complete sets of redistribution functions for `DMap`s, `DArr`s and `DList`s in our compiler.

[1] Here N is the number of combinator implementations per combinator, m is the number of combinator applications, and P is the number of redistribution functions.

Identifying locations/expression to redistribute To automatically insert redistributions to make programs type-check, we need to be able to identify where the DDL types in a program are broken. We automatically identify these locations by extending our constraint-based type inference algorithm $\mathrm{T}_{\text{infer}}$ to carry provenance about which program expressions generated which types, so that when a constraint fails to unify, we can identify which expressions caused the problem.

We maintain this information by extending our types, which are implemented as expressions of the form $\texttt{T}(t_1, \ldots, t_n)$, to carry labels to record what expression IDs the type came from. We denote this $\texttt{T}_l$, where `T` is the type, and l is a set of expression IDs. We then modify the type rules to initialize types with the expression IDs that they came from. Here we use the function L defined in Figure 5.10 to add a set of expression IDs to a type's terms and subterms. Then we redefine substitution application to combine the expression IDs from the variable being replaced, and its replacement (cf. Figure 5.10). This means that during unification, whenever we eliminate a type variable X_{l_1} by substitution, all of its labels l_1 (the expression IDs of the expressions that contributed to it) are added to the type t_{l_2} replacing it, so that for a given type term or subterm (e.g., a single type parameter like a partition function) we know which expressions created it. This approach finds at least two broken expressions for each broken constraint— one for the LHS, and one for the RHS of the constraint. These may relate to the expression that produced the value, and one of the expressions that is trying to consume it.

This approach does not record all the expressions that share a given type, but just some of the expressions that created it (e.g., including function applications that produce values of a given type, and other applications that consume these values). For example, D-If constrains e_2's type T_2 to equal e_3's type T_3, but only returns T_2, and so only e_2, and e_2's creator(s) will appear in the expression ID list of any broken constraint. This is not a big problem for two reasons. Firstly, in a situation like the one above, where both branches of the `if` disagree with some consumer of the `if`, it would be better to redistribute the whole `if`, rather than both the `then` and `else` branches. Furthermore, our redistribution insertion algorithms are iterative, so it can insert a redistribution for e_2 in one iteration, and then insert another to make e_3 match e_2's DDL during the next.

Function choice heuristic There may be multiple redistribution functions, or chains of redistribution functions, that convert one DDL type to another. In this case we want some kind of metric to let us choose the *best* (fastest) ones to use. A simple metric that counts the number of redistribution functions is helpful, so that for example `repartMap` is preferred over `repartMap`·`repartMap`. However, some individual redistribution functions may be more expensive than others, and so some chains of functions may be faster than individual functions. To address this we adopt a very simple integer constant cost for each redistribution/re-layout function, to encode the fact that, for example, performing a local re-layout like changing a matrix from column-major to row-major

$$
\begin{aligned}
\mathrm{L}_{\mathrm{get}} &: \mathsf{Term} \to \mathcal{P}\,(\mathsf{Label}) \\
\mathrm{L}_{\mathrm{get}}\ (\mathtt{T}_{(l_1)}(t_1\ \ldots\ t_n)) &= l_1 \cup (\mathrm{L}_{\mathrm{get}}\ t_1)\ \cup\ \ldots\ \cup\ (\mathrm{L}_{\mathrm{get}}\ t_n) \\
\mathrm{L}_{\mathrm{get}}\ x_{(l_1)} &= l_1 \\
\mathrm{L}_{\mathrm{get}}\ f_{(l_1)} &= l_1
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{L} &: \mathcal{P}\,(\mathsf{Label}) \times \mathsf{Term} \to \mathsf{Term} \\
\mathrm{L}\ l_1\ (\mathtt{T}_{(l_2)}(t_1\ \ldots\ t_n)) &= (\mathtt{T}_{(l_1\ \cup\ l_2)}((\mathrm{L}\ l_1\ t_1)\ \ldots\ (\mathrm{L}\ l_1\ t_n))) \\
\mathrm{L}\ l_1\ x_{(l_2)} &= x_{(l_1\ \cup\ l_2)} \\
\mathrm{L}\ l_1\ f_{(l_2)} &= f_{(l_1\ \cup\ l_2)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Subst} \times \mathsf{Term} \to \mathsf{Term} \\
[X_{(l_1)} \mapsto t_{(l_2)}]\ (\mathtt{T}_{(l_3)}(t_1\ \ldots\ t_n)) &= (\mathtt{T}_{(l_3)}(([X_{(l_1)} \mapsto t_{(l_2)}]\ t_1)\ \ldots\ ([X_{(l_1)} \mapsto t_{(l_2)}]\ t_n))) \\
[X_{(l_1)} \mapsto t_{(l_2)}]\ X_{(l_3)} &= \mathrm{L}\ (l_1\ \cup\ l_3)\ t_{(l_2)} \\
[X_{(l_1)} \mapsto t_{(l_2)}]\ Y_{(l_3)} &= Y_{(l_3)} \\
[X_{(l_1)} \mapsto t_{(l_2)}]\ f_{(l_3)} &= f_{(l_3)}
\end{aligned}
$$

FIGURE 5.10: $\mathrm{L}_{\mathrm{get}}$ returns all labels attached to a type and its sub-terms; L adds labels to a type and its sub-terms; and applying substitutions combines labels from the variables, and target terms.

is faster than re-distributing the matrix across the network to achieve the same effect. We therefore give redistribution functions that communicate over the network a cost at least an order of magnitude greater than local re-layout functions (cf. Section 6.1). More complex cost heuristics are possible, but these are sufficient to choose between different redistribution/re-layout functions that achieve the same effect. Table Figure 5.11 gives costs for some of the `DMap` redistribution functions in our implementation.

Function	Cost	Behavior
`readVMap`	1	Iterates over sorted vector.
`readHMap`	1	Iterates over hashmap.
`saveVMap`	5	Saves a stream of values in a sorted vector.
`sortVMap`	100	Re-sorts a sorted vector using a new sort key.
`sieveSMap`	10	Filters values that do not belong on this node from a stream.
`mirrVMap`	200	Mirrors vector partitions over a new topology dimension.
`repartVMap`	500	Re-partitions sorted vector using a new partition function.

FIGURE 5.11: Example cost values for redistribution and local re-layout functions.

```
1  R1 : P (Fun) × Program → P (Program × TypeEnv)
2  R1 funs P =
3    (Γ, C) = Tinfer P
4    return (R1′ funs P C)
5
6  R1′ : P (Fun) × Program × Constraint → P (Program × TypeEnv)
7  R1′ funs P (a ≐ b) =
8    locations = (Lget a) ∪ (Lget b)
9    redistFuns = (Rfuns funs ∀c . a → c) ∪ (Rfuns funs ∀c . b → c)
10              ∪ (Rfuns funs ∀c . c → a) ∪ (Rfuns funs ∀c . c → b)
11   options = ∅
12   for each l ∈ locations
13     for each f ∈ redistFuns
14       P' = insertFunAt P f l
15       (Γ′, C') = Tinfer P′
16       if C' = () then solutions = solutions ∪ {(Γ′, C′)}
17       else if C' ≠ C then options = options ∪ {(P′, C′)}
18   for each (P',C') ∈ options
19     solutions = solutions ∪ (R1′ funs P′ C′)
20   return solutions
```

FIGURE 5.12: Automatic redistribution function insertion algorithm 1.

Algorithm 1 Our first algorithm for automatic redistribution insertion is function R_1 in Figure 5.12. Here we fix one broken constraint `C` at a time, trying progressively longer chains of redistribution functions (`redistFuns`), inserting them at all the possible locations (`locations`) returned by the labels of the types in the broken constraint. For each possible `P'`, we infer the types, and if inference succeeds we add it to `solutions`, otherwise if inference fails on a new constraint we add it to `options`. We then call R_1' recursively for each option, adding all solutions to the `solutions` set. Here $R_{\texttt{funs}}$ returns chains of redistribution/re-layout functions from `funs` that unify with the type scheme given, and `insertFunAt` returns a new program where the redistribution/re-layout functions `f` are applied to the expression with ID `l`.

This approach is sound, since it only returns solutions that type-check, and is complete, as it tries all valid combinations of redistribution functions inserted at all locations that could have caused the broken constraint. However, it is not efficient. One reason for this is that we cannot tell which of the types in the broken constraint should be the source and target DDL types for the redistribution, and so we are forced to try all chains of redistribution functions whose source or target type unify with one side of the constraint (see lines 9 and 10 in Figure 5.12). This means `redistFuns` is often very large. Another reason is that R_1' is called recursively for each combination of location and redistribution function that fixes the current constraints. This means that the algorithm has very poor computational complexity, in the number of times it calls T_{infer} to re-infer the types . Say we have m broken constraints, which could each be fixed by inserting a redistribution in one of L possible locations. Then for each location there are a maximum R redistribution functions (or chains thereof) to try. Of those that

fix the constraint but aren't yet solutions (i.e., where there is a further constraint `C'` to fix) we recursively call R_1' for the all of them. This gives a computational complexity of $O(LR^m)$, i.e., exponential in the number of broken constraints where R is typically very large. We can improve this by changing line 18, to only iterate over the best N `options` found by computing the redistribution cost metric for each. This makes the complexity $O(LRN^{m-1})$, and therefore faster when $N < LR$.

Algorithm 2 Although R_1 is sound, it is slow, especially for programs with several broken constraints, and when calling it repeatedly for each possible combination of combinator implementations in the search space. Our second approach R_2 shown in Figure 5.13, is much faster than R_1 because it fixes multiple constraints at a time, avoids repeatedly re-inferring all the types for the program, and prunes the solutions early using the redistribution cost heuristic.

First on line 3, R_2 uses R_1 to fix all the broken constraints in a program by using the function `cast :: a -> b` (i.e., a dummy cast function) to fix the constraints. This is fast since, there is only one redistribution function to choose for each break (so in R_1 we have $R = 1$). This may return multiple programs as solutions in `Ps`, formed by inserting `cast` at different locations to fix the constraints. If a single program which is the same as the original is returned (i.e., with no `cast`s inserted), then R_2 terminates, and this program is returned on line 4. Otherwise, for each of these solutions, we use the type environment inferred Γ to find the type $a \to b$ for each `cast` function application (cf. Line 9). Then for each of these, we find chains of redistribution functions that unify with $a \to b$ (line 10), choose the best (line 11), and replace the `cast` with the redistribution function application chosen (line 12). This fixes multiple constraints at a time, and knows the exact type $a \to b$ to look for when considering possible redistribution functions. Here `findApps` returns the set of expression IDs in the program that apply the function given, `getExpType` returns the type in the typing environment for the expression provided, and `replaceExp` returns a new program where one expression has been replaced with another.

However, `cast`'s type $\forall a, b : a \to b$ is more general than specific redistribution function chains (i.e., it maps any DDL to any other DDL). This means that inserting a `cast` application may stop some DDL type information propagating during type inference, such that other broken constraints might not manifest, and the types inferred for some `cast` applications may be too general, so that further redistributions may be required to fully fix the constraint(s). For example, in a program with two broken constraints, the second one's $a \to b$ type may have one conflicting type parameter, when there are actually two. To fix this problem, we iterate (line 15), by calling R_2 until all constraints are fully fixed. This works because most redistribution functions change one DDL type parameter whilst leaving the others unchanged, so additional conflicting type parameters can be fixed by inserting additional redistributions. This solution should be much

```
1  R2 : P(Fun) × Program → Program × TypeEnv
2  R2 funs P =
3    Ps = R1 {cast : ∀a, b . a → b} P
4    if Ps = {P} then return (P′, Γ)
5    Ps′ = {}
6    for each (P′, Γ) ∈ Ps
7        castApps = findApps P' cast
8        for each (f e) ∈ castApps
9          (a → b) = getExpType Γ f
10         redistFuns = Rfuns funs (a → b)
11         f' = arg min_{x∈redistFuns} cost(x)
12         P' = replaceExp P' f f'
13       Ps′ = Ps′ ∪ P'
14   P' = arg min_{x∈Ps′} cost(x)
15   return (R2 funs P')
```

FIGURE 5.13: Automatic redistribution insertion algorithm 2.

faster than R_1 because it calls T_{infer} much less. This is because it fixes several broken constraints in one iteration, and because it knows the exact type required for the redistribution's domain and co-domain, by looking at the type inferred for the cast function instance, so that `redistFuns` is much smaller.

If we have a redistribution function that transforms each type parameter, and there are a maximum n parameters then we will need a maximum n iterations to fix all the constraints (if we ignore the fact that sometimes broken constraints may not be revealed until others are fixed). n is always a small constant (i.e., 4 for `DArr`, `DMap`, and `DList`), and so can be ignored when considering complexity. Then, if there are m broken constraints, R_1 infers the types LN^{m-1} times (since when using `cast` $R = 1$). So the the total number of calls to T_{infer} required is $O(LN^{m-1})$. This is faster than R_1 because L is typically small (e.g., 2), whereas R is typically very large (e.g., 100). These small constants make R_2 faster than R_1, but it will still be slow for larger programs, since it is still exponential in the number of broken constraints. The only way to avoid this is to make $N = 1$, or to modify R_1' in some other way to regularly pick the best few partial solutions, and continue to grow those towards full solutions. This means that not all possible locations will be considered, but in practice whether a redistribution is inserted just after a producer or just before a consumer, will often have no effect on the performance (i.e., if there is only one consumer).

5.3 Concluding remarks

In this chapter we have presented an automatic type inference algorithm for our DDL type system. This includes a system of typing rules that allow types and typing constraints to be derived from a program, a way of normalizing functions so the they can be compared, and a unification algorithm for solving the typing constraints. We have

proved that these algorithms terminate for any inputs, and that they are sound. We have also proved that our normalization operation is complete for projection functions, and that our unification algorithm is complete with respect to possible sequences of rule applications. We have also introduced the concept of automatic redistribution insertion, that is, automatically inserting DDL type-casts into Flocc plans to make them type-check, and presented two algorithms that implement it. We have implemented the type inference algorithm, and redistribution Algorithm 1 in our prototype compiler (cf. Chapter 7), and used them to automatically derive DDL types for some example programs in Chapter 8. One limitation of our current implementation is the time complexity of this redistribution insertion Algorithm 1. We have therefore presented an improved redistribution insertion algorithm, Algorithm 2 in Figure 5.13, with better time complexity, and future work would implement this algorithm in our compiler.

Chapter 6

Extended Distributed Data Layout Types

So far we have seen how to use types to characterize the distributed data layouts (DDLs) of distributed maps, arrays, and lists, and different distributed memory implementations of high-level combinators involving these (cf. Chapter 4). We have also seen how data distribution information can be statically derived for different choices of combinator implementations in programs, via type inference and automatic redistribution insertion (cf. Chapter 5).

A major strength of our approach is its extensibility. New combinators can be added simply by declaring their functional types, and the DDL types and back-end templates of their implementations. Furthermore, the system can be extended with new types without altering the underlying framework. For example, collections like spatially indexed maps (`Spatial`), or trees (`Tree`), and their distributed equivalents (`DSpatial` and `DTree`), could be added by simply adding them to a configuration file (since all types are implemented as s-expressions). This extensibility is a clear benefit of this approach over collection-specific techniques.

This chapter demonstrates the extensibility of our approach even further. In Section 6.1 we show how to extend our DDL types to encode local layout information about collections. In Section 6.2 we show how we can add more flexibility to our partition and layout functions, and in Section 6.3 we show how to extend our distributed array DDL types to support more complicated DDLs (i.e., involving different block sizes, ghosted regions, virtual offsets, and axis-reflection). Finally, in Section 6.4 we give another approximation of higher-order function unification that can be used to find solutions (i.e., valid typings) for Flocc plans involving these extended types. This is not a complete list of possible extensions, but rather a collection of useful example extensions to demonstrate the extensibility of our approach.

dt ::= `DArr` ... m_{arr} f | `DMap` ... m_{map} f | `DList` ... m_{list}

m_{arr} ::= `Mem` | `Iter` | `Stm`

m_{map} ::= `Hash` | `Tree` | `Vec` | `Iter` | `Stm`

m_{list} ::= `Link` | `Vec` | `Iter` | `Stm`

g ::= ... | `rem` f

FIGURE 6.1: Local data layout type parameters

6.1 Local data layouts

In addition to distribution information, we also use DDL types to specify how to store collections locally in memory. For example, multidimensional arrays can be stored in different ways in memory, e.g., in row-major or in column-major order. We specify the layout of an n-dimensional `DArr` by adding a *layout function* to the type. This is a permutation function which maps the array's indices to an n-tuple, whose order dictates how to order the indices in memory. Hence, `\(x,y) -> (x,y)` means row-major order, and `\(x,y) -> (y,x)` means column-major. This can express very similar constraints to partition functions. For example, `groupReduceArr2` has the full type[1]

```
Π(f,_,_,_) :
  (i1->i2, (i1,v1)->v2, (v2,v2)->v2, DArr i1 v1 pf d m Stm ((f ⊗ (rem f)) · Δ))
    -> DArr i2 v2 id d m Stm id
```

where `rem f` projects all the parts of the input tuple that `f` does not already project, such that `(f⊗(rem f))` defines a complete permutation of all the array's indices. Here, we force the first indices to be the group's key indices (projected by `f`), followed by the rest `rem(f)`. This improves cache-line usage by ensuring that elements in the same group are adjacent in memory. We use the same technique to specify the indexing schemas of `DMap`s. We also use flags in the types to specify the local storage modes (e.g., hash table/binary tree/sorted vector/iterable collection/stream of values) for `DMap` and `DList`.

The following subsections briefly explain the local layouts for some of the combinator implementations. The syntax for the additional local layout DDL type parameters is shown in Figure 6.1.

`rem` function generator `rem` is an additional function generator that takes a function f and returns another function that is the complement of f, i.e., projects all the parts

[1] Note that although these types are complex, the end-user does not need to see, write, or understand them. They are internal to the compiler, and allow the user to ignore all the complex layout information that they contain.

of the input tuple that f does not already project. We use it in `DArr` local layout functions to ensure that they project all of their argument values, and therefore define a permutation of all the array's dimensions.

6.1.1 Local array layouts

The two additional `DArr` type parameters are a local storage mode m_{arr} and a local layout function which defines which order the array's indices should be stored in (i.e., `id` would be row-major, and `swap` column major). The modes are `Mem` which stores the whole array contiguously in memory, `Iter` which provides an iterable interface to the array, and `Stm` which represents a stream of elements in some order (defined by f), which can be consumed in this sequence. Figure 6.2 shows the extra type parameters for some of the implementations of the combinators in Figure 3.6. Here we use numerical suffixes to denote different distributed algorithms, upper case letters to denote different types for the same distributed algorithm, and roman numerals to denote different local layout types for the same distributed algorithm.

The `mapArrInv` combinators work on value streams in any order. For `eqJoinArr1A` and `eqJoinArr2i` both input arrays are ordered using their respective key emitter functions f and g, pairwise composed with `rem` f and `rem` g so that they return a permutation of *all* the array's indices. This allows the implementation to perform a sort of merge-join which consumes the elements in key order, and therefore avoids having to repeatedly rescan the whole right-hand array. Consuming elements in this order means that the result elements are ordered primarily by the left-hand element indices, and then secondarily by the right-hand, which is reflected in the result type by pairwise composing the left-hand and right-hand layout functions. `eqJoinArr2ii` and `eqJoinArr2iii` accept element streams in any order, and so are forced to use a nested-loop $O(n^2)$ join, which rescans the whole right-hand array for every element consumed from the left-hand. This is same behavior is reflected in `eqJoinArr2ii` and `eqJoinArr2iii` by pairwise composition and function generators respectively. The mirror images of all these join types are available via *replacement rules* in Chapter 7, which define which high-level combinators can be replaced by what expressions involving combinator implementations.

`groupReduceArr1i` and `groupReduceArr1ii` accept element streams in any order, and therefore must store there results contiguously via `Mem`, so that multiple values for the same result index can be aggregated correctly. `groupReduceArr2` requires its input elements to be ordered using the index projection function f, which means that multiple values for the same index are consumed in contiguous blocks, and so the result is an element stream `Stm` ordered by the new indices (via `id`). The various *re-layout* functions `readArr` to `saveArr` act like type-casts to convert between storage modes.

```
mapArrInvA :: Π(f,_,_,_) : (i->j, j->i, (i,v)->w,
  DArr ⟨...⟩ Stm (g · f) -> DArr ⟨...⟩ Stm g
mapArrInvB :: Π(_,f⁻¹,_,_) : (i->j, j->i, (i,v)->w,
  DArr ⟨...⟩ Stm g) -> DArr ⟨...⟩ Stm (g · f⁻¹)

eqJoinArr1A   :: Π(f,g,_,_) : (i->k, j->k, DArr ⟨...⟩ Stm (f⊗(rem f))·Δ,
  DArr ⟨...⟩ Iter (g⊗(rem g))·Δ)
  -> DArr ⟨...⟩ Stm ((f⊗(rem f))·Δ ⊗ (g⊗(rem g))·Δ)
eqJoinArr2i   :: Π(f,g,_,_) : (i->k, j->k, DArr ⟨...⟩ Stm (f⊗(rem f))·Δ,
  DArr ⟨...⟩ Iter (g⊗(rem g))·Δ)
  -> DArr ⟨...⟩ Stm ((f⊗(rem f))·Δ ⊗ (g⊗(rem g))·Δ)
eqJoinArr2ii  :: (i->k, j->k, DArr ⟨...⟩ Stm f,
  DArr ⟨...⟩ Iter g) -> DArr ⟨...⟩ Stm (f ⊗ g)
eqJoinArr2iii :: (i->k, j->k, DArr ⟨...⟩ Stm (fstFun f),
  DArr ⟨...⟩ Iter (sndFun f)) -> DArr ⟨...⟩ Stm f

groupReduceArr1i  :: (i->j, (i,v)->w, (w,w)->w, w,
  DArr ⟨...⟩ Stm f) -> DArr ⟨...⟩ Mem id
groupReduceArr1ii :: (i->j, (i,v)->w, (w,w)->w, w,
  DArr ⟨...⟩ Stm f) -> DArr ⟨...⟩ Mem g
groupReduceArr2   :: Π(f,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr ⟨...⟩ Stm ((f⊗(rem f))·Δ) ) -> DArr ⟨...⟩ Stm id
subArr :: (i, i,  DArr ⟨...⟩ Mem f) -> DArr ⟨...⟩ Mem f

readArri  :: DArr ⟨...⟩ Mem f -> DArr ⟨...⟩ Iter f
readArrii :: DArr ⟨...⟩ Mem f -> DArr ⟨...⟩ Iter g
iterArr   :: DArr ⟨...⟩ Iter f -> DArr ⟨...⟩ Stm f
saveArri  :: DArr ⟨...⟩ Stm f -> DArr ⟨...⟩ Mem f
saveArrii :: DArr ⟨...⟩ Stm f -> DArr ⟨...⟩ Mem g
```

FIGURE 6.2: Local layout type parameters for array combinator implementations.

6.1.2 Local map layouts

`DMap`'s additional type parameters are a local storage mode m_{map} and a layout function f. The storage mode can be a hashmap `Hash`, a binary tree `Tree`, a sorted vector `Vec`, an iterable collection `Iter`, or a key-value stream `Stm`. The layout function defines the key that hashmaps and treemaps are indexed by, and the order that sorted vectors, iterable interfaces, and key-value streams are sorted by. (These could be supplemented with disk-backed storage modes, and implemented via STLXXL [67]). Figure 6.3 shows the extra DDL type parameters for some of the combinator implementations.

The `map` and `reduce` combinators (cf. Figure 6.3) work on streams in any order. `eqJoin1A` and `2i` perform merge joins similar to `eqJoinArr1A` and `eqJoinArr2i`. `eqJoin2ii` and `2iii` perform (less efficient) nested loop joins like `eqJoinArr2ii` and `eqJoinArr2iii`, and work on any input layouts f and g, and any output layout f respectively. `allPairsAii` performs a merge self-join, and `allPairsAii` performs a nested-loops self-join. `groupReduce1i` and `1ii` accept input streams in any order, storing the result map in a hashmap or binary tree respectively. `groupReduce2` works like `groupReduceArr2` requiring its input values to be ordered using the key-emitter function, and can therefore return a stream of values in the same order.

`union` requires its inputs to both be ordered by key (via `fst`) so that it can perform a sorted merge, where the inputs are visited in step returning either the value from the left-hand input, if its key is lexicographically before or equal to the right-hand, or otherwise the value from the right-hand input. `intersect1i` performs a similar kind of merge, but `intersect1ii` to `intersect1iv` allow the left-hand input to be sorted in any way, as long as the right-hand is stored as a hashmap, binary tree, or sorted vector, which is indexed by key (hence the `fst`), so that it can be probed to check if it contains a key that corresponds with the key-value from the left-hand input. `intersect1v` performs the less efficient nested-loops algorithm. `diff1i` performs a merge, `diff1ii` to `diff1iv` probe hashmaps, binary trees, and sorted vectors, and `diff1v` uses nested-loops. The re-layout functions in Figure 6.4 convert between the different storage modes and orderings.

6.1.3 Local list layouts

The only additional parameter for `DLists` is m_{list} which can be a linked list `Link`, array-list `Vec`, iterable collection `Iter` or stream `Stm`. Figure 6.5 shows the extra DDL type parameters for some of the combinator implementations. Most of these combinators work on element streams, but `appendList1i` takes advantage of lists stored as linked lists, by appending them using pointers, rather than having to duplicate them.

```
map :: Π(f,_) : ((i,v)->(j,w),
  DMap Stm ⟨...⟩ (g · f)) -> DMap ⟨...⟩ Stm g
eqJoin1A :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap ⟨...⟩ Stm f,
  DMap ⟨...⟩ Iter g) -> DMap ⟨...⟩ Stm (((f · lft) ⊗ (g · rht))·Δ)
eqJoin2i :: Π(f,g,_,_) :((i,v)->k, (j,w)->k, DMap ⟨...⟩ Stm f,
  DMap ⟨...⟩ Iter g) -> DMap ⟨...⟩ Stm (((f · lft) ⊗ (g · rht))·Δ)
eqJoin2ii :: ((i,v)->k, (j,w)->k, DMap ⟨...⟩ Stm f,
  DMap ⟨...⟩ Iter g) -> DMap ⟨...⟩ Stm (((f · lft) ⊗ (g · rht))·Δ)
eqJoin2iii :: ((i,v)->k, (j,w)->k, DMap ⟨...⟩ Stm (lftFun f),
  DMap ⟨...⟩ Iter (rhtFun f)) -> DMap ⟨...⟩ Stm f
allPairsAi :: Π(f,_) : ((i,v)->k, DMap ⟨...⟩ Iter f) ->
  DMap ⟨...⟩ Stm (((f · lft) ⊗ (f · rht))·Δ)
allPairsAii :: ((i,v)->k, DMap ⟨...⟩ Iter f) ->
  DMap ⟨...⟩ Stm (((f · lft) ⊗ (f · rht))·Δ)
groupReduce1i :: ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap ⟨...⟩ Stm f) -> DMap ⟨...⟩ Hash fst
groupReduce1ii :: ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap ⟨...⟩ Stm f) -> DMap ⟨...⟩ Tree fst
groupReduce2 :: Π(f,_,_,_) : ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap ⟨...⟩ Stm f) -> DMap ⟨...⟩ Stm fst
reduce :: ((k,v)->s, (s,s)->s, s, DMap ⟨...⟩ Stm f) -> s
union :: (DMap ⟨...⟩ Stm fst, DMap ⟨...⟩ Iter fst) -> DMap ⟨...⟩ Stm fst
intersect1i :: (DMap ⟨...⟩ Stm fst, DMap ⟨...⟩ Iter fst) -> DMap ⟨...⟩ Stm fst
intersect1ii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Hash fst) -> DMap ⟨...⟩ Stm f
intersect1iii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Tree fst) -> DMap ⟨...⟩ Stm f
intersect1iv :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Vec fst) -> DMap ⟨...⟩ Stm f
intersect1v :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Iter g) -> DMap ⟨...⟩ Stm f
diff1i :: (DMap ⟨...⟩ Stm fst, DMap ⟨...⟩ Iter fst) -> DMap ⟨...⟩ Stm fst
diff1ii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Hash fst) -> DMap ⟨...⟩ Stm f
diff1iii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Tree fst) -> DMap ⟨...⟩ Stm f
diff1iv :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Vec fst) -> DMap ⟨...⟩ Stm f
diff1v :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Iter g) -> DMap ⟨...⟩ Stm f
```

FIGURE 6.3: Local layout type parameters for map combinator implementations.

```
readTreeMap   :: DMap ⟨...⟩ Tree f -> DMap ⟨...⟩ Iter f
readHashMap   :: DMap ⟨...⟩ Hash f -> DMap ⟨...⟩ Iter nullF
readVecMap    :: DMap ⟨...⟩ Vec f -> DMap ⟨...⟩ Iter f
readIterMap   :: DMap ⟨...⟩ Iter f -> DMap ⟨...⟩ Stm f
saveTreeMap   :: DMap ⟨...⟩ Stm f -> DMap ⟨...⟩ Tree g
saveHashMap   :: DMap ⟨...⟩ Stm f -> DMap ⟨...⟩ Hash g
saveVecMap    :: DMap ⟨...⟩ Stm f -> DMap ⟨...⟩ Vec f
sortVecMap    :: DMap ⟨...⟩ Vec f -> DMap ⟨...⟩ Vec g
```

FIGURE 6.4: Local layout type parameters for map re-layout functions.

```
zip            :: (DList ⟨...⟩ Stm, DList ⟨...⟩ Iter) -> DList ⟨...⟩ Stm
mapList        :: (v->w, DList ⟨...⟩ Stm) -> DList ⟨...⟩ Stm
reduceList     :: ((v,v)->v, v, DList ⟨...⟩ m) -> v
filterList     :: (v->Bool, DList ⟨...⟩ Stm) -> DList ⟨...⟩ Stm
crossList1     :: (DList ⟨...⟩ Stm, DList ⟨...⟩ Iter) -> DList ⟨...⟩ Stm
concatList1i   :: (DList ⟨...⟩ Link, DList ⟨...⟩ Link) -> DList ⟨...⟩ Link
concatList1ii  :: (DList ⟨...⟩ Stm, DList ⟨...⟩ Stm) -> DList ⟨...⟩ Stm
findInList     :: (v -> w, w, (v,w), DList ⟨...⟩ Stm) -> (Int,(v,w))

readVecList   :: DList ⟨...⟩ Vec -> DList ⟨...⟩ Iter
readLinkList  :: DList ⟨...⟩ Link -> DList ⟨...⟩ Iter
iterList      :: DList ⟨...⟩ Iter -> DList ⟨...⟩ Stm
saveVecList   :: DList ⟨...⟩ Stm -> DList ⟨...⟩ Vec
saveLinkList  :: DList ⟨...⟩ Stm -> DList ⟨...⟩ Link
```

FIGURE 6.5: Local layout type parameters for list combinator implementations.

6.2 More flexible functions

Some of the combinator implementation DDL types in Chapter 4 (cf. Section 4.3.1) are more specific than they have to be. For example, for

```
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (f·fst) d m
```

with f = `\((`x`,` y`),` z`) -> (`x`,` y`)`, the left-hand map could actually be partitioned by f_1 = `\((`x`,` y`),` z`) ->` x or f_2 = `\((`x`,` y`),` z`) ->` y instead of f. Either of these partition

```
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k,
     DArr i v (α·f) d m, DArr j w (α·g) d m)
  -> DArr (i,j) (v,w) (α·f·fst) d m
groupReduceArr2 :: Π(f,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
     DArr i v (α·f) d m)
  -> DArr j w α d m

eqJoin1A :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k,
     DMap i v p (α·f) d m, DMap j w p (α·g) d m)
  -> DMap (i,j) (v,w) p (α·f·lft) d m
allPairsA :: Π(f,_) : ((i,v)->k,
     DMap i v p (α·f) d m)
  -> DMap (i,i) (v,v) p (α·f·lft) d m
union :: (DMap k v p (α·fst) d m, DMap k v p (α·fst) d m)
  -> DMap k v p (α·fst) d m
intersect1 :: (DMap k v p (α·fst) d m, DMap k w p (α·fst) d m)
  -> DMap k v p (α·fst) d m
diff1 :: (DMap k v p (α·fst) d m, DMap k w p (α·fst) d m)
  -> DMap k v p (α·fst) d m
```

FIGURE 6.6: Extended DDL types for combinator implementations.

functions would ensure that all elements for a given value of (x, y) would be co-located on the same node. This intuitively seems like a kind of sub-typing, where f_1 and f_2 are subtypes of (and are therefore subsumed by) f. However, although classical sub-typing might work for arrays, it would not for maps, because applying a `Hash` function to f_1's output x may lead to a different distribution to applying one to f's output (x, y), i.e., hash functions may not respect a lexicographical ordering of input tuples. We therefore show how we can encode this flexibility into the DDL types without sub-typing, as follows.

6.2.1 Extended partition functions

For both map and array partition functions, functions that return fewer arguments than strictly required can be used, as long as they are used consistently. For example, with `eqJoinArr1A`, f_1 can be used instead of f, as long as it is used for the left-hand argument map, the output map, and a consistent version of g is used for the right-

```
let triEnum = (\E :: Map (Int,Int) () ->
  -- find degree of all vertices
  let D1 = groupReduce (fst.fst, \_->1, addi, E) in
  let D2 = groupReduce (snd.fst, \_->1, addi, E) in
  let D = map (\(k,v)->(k,addi v), eqJoin (fst,fst,D1,D2)) in

  -- identify edges by vertex with lower degree
  let E1 = eqJoin (fst, fst, E, D) in
  let E2 = eqJoin (snd.fst, fst, E1, D) in
  let E3 = map (\(((_,v1),v2),((_,d1),d2)) ->
    (if lti (d1,d2) then (v1,v2) else (v2,v1), ()), E2) in

  -- for each edge, find all angles
  let A = allPairs (fst, E3) in

  -- for each angle, see if it is closed
  let T1 = eqJoin (\(((a,_),(_,b)),_)->(a,b), fst, A, E3) in
  let T2 = eqJoin (\(((a,_),(_,b)),_)->(b,a), fst, A, E3) in
  map (\((((v1,_),_),(v2,v3)),_)->((v1,v2,v3),()),union(T1,T2)))
```

FIGURE 6.7: Triangle enumeration (MinBucket algorithm)

hand argument map. We can encode this in the DDL types by sequentially composing an unknown/abstract function variable α with the original partition function. This abstract function α can be instantiated with any projection function, and thus return a subset of f's original result. For example

```
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k, DArr i v (α·f) d m,
  DArr j w (α·g) d m) -> DArr (i,j) (v,w) (α·f·fst) d m
```

would also be a valid DDL type for `eqJoinArr1A`. Here, $\alpha =$ `fst` yields $f_1 = \backslash((\mathtt{x},\mathtt{y}),\mathtt{z}) \to \mathtt{x}$ and $\alpha =$ `snd` yields $f_2 = \backslash((\mathtt{x},\mathtt{y}),\mathtt{z}) \to \mathtt{y}$. This technique can be used for a number of the map and array DDL types, some of which are shown in Figure 6.6. For example, the extended `eqJoin1A` type allows an implementation of the MINBUCKET triangle enumeration algorithm (cf. Figure 6.7) to be derived, that cannot be with the original DDL types. The constraints caused by the `eqJoin`s and `union` at the end of this example preclude the use of local `eqJoin1`s in the original DDL types, but the flexibility added by the variable α in the extended types permits this. Here, the parametric polymorphism in our DDL types allows us to encode even more precise constraints between input and output data layouts, than those in Chapter 4. The unification algorithm given in Chapter 5 is unlikely to find a suitable value for α in most cases, since it is no longer a simple case of binding a function to a type variable on one side, and composing it with some other variable on the other. However, we present an extended unification technique based on an equational theory of projection functions in Section 6.4 that can find suitable values for these abstract function variables.

```
eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k,
     DArr ⟨...⟩ Stm  f⊙α, DArr ⟨...⟩ Iter g⊙β)
  -> DArr ⟨...⟩ Stm ((f⊙α)·fst)⊙((g⊙β)·snd))
eqJoinArr2ii :: Π(f,g,_,_) : (i->k, j->k,
     DArr ⟨...⟩ Stm h, DArr ⟨...⟩ Mem g⊙α)
  -> DArr ⟨...⟩ Stm (h·fst)⊙((g⊙α)·snd)
eqJoinArr2iii :: (i->k, j->k,
     DArr ⟨...⟩ Stm  f, DArr ⟨...⟩ Iter g)
  -> DArr ⟨...⟩ Stm (f·fst)⊙(g·snd)
groupReduceArr2 :: Π(f,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
     DArr ⟨...⟩ Stm f⊙α)
  -> DArr ⟨...⟩ Stm id

eqJoin1A :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k,
     DMap ⟨...⟩ Stm  f⊙α, DMap ⟨...⟩ Iter g⊙β)
  -> DMap ⟨...⟩ Stm ((f⊙α)·lft)⊙((g⊙β)·rht)
eqJoin2ii :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k,
     DMap ⟨...⟩ Stm h, DMap ⟨...⟩ Vec g⊙α)
  -> DMap ⟨...⟩ Stm (h·lft)⊙((g⊙α)·rht)
eqJoin2iii :: ((i,v)->k, (j,w)->k,
     DMap ⟨...⟩ Stm  f, DMap ⟨...⟩ Iter g)
  -> DMap ⟨...⟩ Stm (f·lft)⊙(g·rht)
allPairsAi :: Π(f,_) : ((i,v)->k,
     DMap ⟨...⟩ Iter f⊙α)
  -> DMap ⟨...⟩ Stm ((f⊙α)·lft)⊙((f⊙α)·rht)
intersect1i :: (
     DMap ⟨...⟩ Stm  fst⊙α, DMap ⟨...⟩ Iter fst⊙α)
  -> DMap ⟨...⟩ Stm  fst⊙α
intersect1iv :: (
     DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Vec fst⊙α)
  -> DMap ⟨...⟩ Stm f
```

FIGURE 6.8: Extended local layout type parameters for combinator implementations.

6.2.2 Extended local layout functions

Local data layout functions can also be made more flexible in a similar fashion. The sequence of arguments returned by the local layout function gives the value that the collection is ordered or indexed by. So for $f = \backslash((\mathtt{x},\mathtt{y}),\mathtt{z}) \rightarrow (\mathtt{x},\mathtt{y})$ the collection would be sorted first by x and then by y using a lexicographical ordering. So if a collection needs to be ordered by $f_1 = \backslash((\mathtt{x},\mathtt{y}),\mathtt{z}) \rightarrow \mathtt{x}$, it would be valid to sort it using f, since a collection sorted by (x, y) values is also sorted by x. We can therefore extend local layout functions to return additional arguments on the right-hand side of the result tuple, without breaking the ordering. Note that we flatten result tuples such that ordering by $((x, y), z)$ is equivalent to ordering by $(x, (y, z))$. We use a new *associative* pairwise composition operator $\odot$ to capture this flattening, where $\alpha \odot \beta$ is a syntactic sugar for `\`x `->` `(`$\alpha\ x,\ \beta\ x$`)`.

To encode this in the DDL types we use the $\odot$-operator to pairwise compose an abstract function variable on the right-hand side of our original local layout function. For example,

```
groupReduce2 :: Π(f,_,_,_) : (i->j, (i,v)->w, (w,w)->w,
     DMap ... Stm (f⊙α) )
  -> DMap ... Stm (fst⊙α)
```

gives a valid local layout for `groupReduceArr2`. Similar types for the other combinators is shown are Figure 6.8. These more flexible types are also unlikely to unify using the algorithm given in Chapter 4. However, unification using an equational theory of *permutation functions* shown in Section 6.4.2, does unify them.

6.3 Extended distributed array types

The `DArr` type given in Chapter 4 can be extended to allow more efficient implementations of some of the array combinators. The syntax of these extended types is shown below in Figure 6.9. This type extends `DArr` with 5 extra function parameters, all of

$$dt ::= \texttt{DArr}\ \ldots\ f_{bs}\ f_{dir}\ f_{off}\ f_{ghl}\ f_{ghr}$$

FIGURE 6.9: `DArr` extended syntax

which take array indices and increment, decrement, or scale them by some amount (i.e., have type `i -> i`, where `i` is a tuple of integers). Here $+(i)$ and $\times(i)$ are sugars for $\backslash\mathtt{x} \rightarrow \mathtt{addiv}\ (\mathtt{x}, i)$ and $\backslash\mathtt{x} \rightarrow \mathtt{muliv}\ (\mathtt{x}, i)$ respectively, as shown in Figure 6.14, where `addiv` and `muliv` lift integer addition and multiplication to index vectors (i.e., tuples of integers).

The original `DArr` type in Chapter 4 uses a block-cyclic distribution with a fixed block size. Here we use an additional parameter f_{bs}, which returns an integer value for each array index, to make these block sizes configurable for each dimension of the array.

In order to reverse the direction of array indices in Chapter 4, `reflectArr` has to swap array partitions over the network, and reverse the order of each one locally. Here, we introduce a new function parameter f_{dir} which returns a positive value if the direction of a given array dimension is as stored, and a negative value if it should be treated as reversed. This allows us to implement `reflectArr` in-place without actually having to swap or reverse any of the array partitions.

Finally, the `DArr` type in Chapter 4 does not support ghosting, i.e. replicating fringes of elements from adjacent partitions. This means that the `shiftArrR` and `shiftArrL` combinators that increment and decrement the indices of arrays always require communication between neighboring nodes. Successive applications also require successive communications. We avoid this here by extending `DArr` with the f_{off} function to specify the size of any offset/displacement of the array indices, and f_{ghl} and f_{ghr} to specify the sizes of any ghost fringes for all the dimensions of the array. Here, f_{ghl} specifies fringes to left (i.e., overlap with a previous partition, in the negative direction), and f_{ghr} specifies fringes to the right (i.e., positive direction).

Figure 6.10 and Figure 6.11 show extended DDL types for the key `DArr` combinators. The first four `intRangeArr` implementations generate arrays with block-cyclic distributions and block sizes 1 (i.e., cyclic), 10, 100, and 1000 respectively. `intRangeArr5` has block size $y - x$ (i.e., the size of the array) and therefore stores all elements on node 0. `intRangeArr6` uses block size $(y - x)/dimSizes(\texttt{d1})$ which corresponds to a blocked distribution, with any remaining elements on node 0.

The `shiftArrL1` and `shiftArrR1` types translate an array's indices in the negative/-positive direction by performing communication and so do not affect the virtual offset or fringes. `shiftArrL2` and `shiftArrR2` however, consume part of their right-hand and left-hand fringes respectively, and so do not perform any communication. Their types therefore force their input arrays to have sufficient ghost fringes available by composing $+(\texttt{o})$ with the fringes for the output. Then `shiftArrL3` and `shiftArrR3` also act without communication by decreasing or increasing the virtual offset so that the array's indices are displaced by different amounts. These virtual offsets can then be made concrete by applying the `shRedistArr` redistribution function to actually shift the array by the amount specified. Similarly, `ghRedistArr` changes an arrays fringes, adding or removing them.

The `subArr` type does not affect the parameters, as it just discards any parts of the array partitions that are not in the sub-array. `scaleArr` scales the array indices by the factors in the first argument, and therefore just scales the block size, virtual offset, and ghost sizes by this amount, and performs no communication. `reflectArr` changes the

```
intRangeArr1 :: (Int,Int,Int)->DArr Int () ⟨...⟩ +(1) dir of ghl ghr
intRangeArr2 :: (Int,Int,Int)->DArr Int () ⟨...⟩ +(10) dir of ghl ghr
intRangeArr3 :: (Int,Int,Int)->DArr Int () ⟨...⟩ +(100) dir of ghl ghr
intRangeArr4 :: (Int,Int,Int)->DArr Int () ⟨...⟩ +(1000) dir of ghl ghr
intRangeArr5 :: Π(x,y,_) : (Int,Int,Int)
  -> DArr Int () ⟨...⟩ +(y − x) dir of ghl ghr
intRangeArr6 :: Π(x,y,_) : (Int,Int,Int)
  -> DArr Int () f d1 d2 +((y − x)/dimSizes(d1)) dir of ghl ghr

shiftArrL1 :: (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs dir of ghl ghr
shiftArrR1 :: (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs dir of ghl ghr
shiftArrL2 :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of ghl (ghr·+(o)))
  -> DArr ⟨...⟩ bs dir of ghl ghr
shiftArrR2 :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of (ghl·+(o)) ghr)
  -> DArr ⟨...⟩ bs dir of ghl ghr
shiftArrL3 :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs dir (of·−(o)) ghl ghr
shiftArrR3 :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs dir (of·+(o)) ghl ghr

subArr :: (i, i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs dir of ghl ghr
scaleArr :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ (bs·×(o)) dir (of·×(o)) (ghl·×(o)) (ghr·×(o))
reflectArr :: Π(o,_) : (i, DArr ⟨...⟩ bs dir of ghl ghr)
  -> DArr ⟨...⟩ bs (dir·×(o)) dir of ghl ghr

mapArrInv1 :: Π(f,_,_,_) : (i->j, j->i, (i,v)->w,
  DArr ⟨...⟩ (bs·f) (dir·f) (of·f) (ghl·f)  (ghr·f) ->
  DArr ⟨...⟩ bs dir of ghl ghr
mapArrInv2 :: Π(_,f⁻¹,_,_) : (i->j, j->i, (i,v)->w,
  DArr ⟨...⟩ bs dir of ghl ghr) ->
  DArr ⟨...⟩ (bs·f⁻¹) (dir·f⁻¹) (of·f⁻¹) (ghl·f⁻¹) (ghr·f⁻¹)
```

FIGURE 6.10: Extended DDL type parameters for `DArr` combinator implementations.

direction of any array indices for which the corresponding part of the parameter tuple is negative, and therefore just multiplies the array direction by this tuple, to change the directions of the corresponding array indices.

The `mapArrInv` types apply their index transformer (i.e., projection/permutation) functions to the results of the `DArr` parameters, and therefore return the parts of these parameters that correspond to the indices in the new array.

For `eqJoin1A` we modify the basic type, to take four parameter functions: a function that returns a pair with key indices on the left and any other indices on the right, and the inverse of this function, for both array arguments. This could be avoided using the `rem` function generator, and by introducing an `inv` function generator that returns the inverse of a permutation function, but we make the parameters explicit here for clarity. We then use these functions in the types to separate the key indices from the others, so that we can use `bsK` and `dirK`, etc., for the key indices from both array arguments, and `bs1`, `bs2`, `dir1 dir2`, etc., for the others. This forces the block sizes, array directions, and array offsets to be the same for the key indices from both argument arrays, but permits any values for the other indices. We use the same technique for the fringe sizes `ghl` and `ghr`, apart from the fact that we permit the fringes of the argument arrays to be bigger than needed for the output by using $+(l1)$, $+(r1)$, $+(l2)$, $+(r2)$ for the left and right fringes of the first and second argument arrays respectively.

`groupReduceArr2a` accepts any offset and fringes, keeping the offsets and throwing any fringes away, where as `groupReduce2b` keep both offsets and any fringes for the group indices.

Finally, the `dirRedistArr`, `shRedistArr`, `bsRedistArr`, and `ghRedistArr` redistribution functions change the array's direction, offset, block size, and ghost fringes respectively, and all require communication. `ghRedistArr2` on the other hand, reduces the sizes of an array's ghost regions inplace by discarding the parts that are no longer needed.

Only fairly simple uses of these types will unify using the algorithm in Chapter 5, since the algorithm does not know about the associativity and commutativity of integer addition, and about how addition and multiplication can be inverted. However, the equational theory in Section 6.4.3 can be used to unify such functions. The Jacobi 2D example program (cf. Figure 6.12) is a good motivation for these extended `DArr` types. Using the `shiftArrL2/R2` variants (and the equational theory) we get a DDL type for `next` of

```
DArr (Int,Int) Float id d1 d2 bs dir id id (+((0,1)) · +((1,0))) (+((1,0)) · +((0,1)))
  -> DArr (Int,Int) Float id d1 d2 bs dir id id id
```

which is the same as having a 1-element fringe on both sides of both array dimensions. Then inserting a single `ghRedistArr` can exchange all these fringes at once, making

```
eqJoinArr1A :: Π(f,f_inv,g,g_inv,_,_) :
  (i->(k,i')), (k,i')->i, j->k, (k,j')->j,
  DArr i ⟨...⟩ (f_inv·(bsK⊗bs1)·f) (f_inv·(dirK⊗dir1)·f) (f_inv·(ofK⊗of1)·f)
               (f_inv·((ghlK·+(l1))⊗ghl1)·f) (f_inv·((ghrK·+(r1))⊗ghr1)·f),
  DArr j ⟨...⟩ (g_inv·(bsK⊗bs2)·g) (g_inv·(dirK⊗dir2)·g) (g_inv·(ofK⊗of2)·g)
               (g_inv·((ghlK·+(l2))⊗ghl2)·g) (g_inv·((ghrK·+(r2))⊗ghr2)·g))->
  DArr (k,(i,j)) ⟨...⟩ (bsK⊗(bs1⊗bs2)) (dirK⊗(dir1⊗dir2)) (ofK⊗(of1⊗of2))
                       (ghlK⊗(ghl1⊗ghl2)) (ghrK⊗(ghr1⊗ghr2))

groupReduceArr2a :: Π(f,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr ⟨...⟩ bs dir of ghl ghr) ->
  DArr ⟨...⟩ (bs·of) (dir·of) (f·of) id id
groupReduceArr2b :: Π(f,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr ⟨...⟩ bs dir of ghl ghr) ->
  DArr ⟨...⟩ (bs·of) (dir·of) (f·of) (f·ghl) (f·ghr)

dirRedist   :: DArr ⟨...⟩ bs dir of ghl ghr ->
               DArr ⟨...⟩ bs id of ghl ghr
shRedistArr :: DArr ⟨...⟩ bs dir of1 ghl ghr ->
               DArr ⟨...⟩ bs dir id ghl ghr
bsRedistArr :: DArr ⟨...⟩ bs1 dir of ghl ghr ->
               DArr ⟨...⟩ bs2 dir of ghl ghr
ghRedistArr :: DArr ⟨...⟩ bs dir of ghl1 ghr1 ->
               DArr ⟨...⟩ bs dir of ghl2 ghr2
ghRedistArr2 :: DArr ⟨...⟩ bs dir of (ghl·+(l)) (ghr·+(r)) ->
                DArr ⟨...⟩ bs dir of ghl ghr
```

FIGURE 6.11: Extended DDL type parameters for `DArr` combinator implementations.

```
let N = 100 :: Int in
let jac = (\X :: DArr (Int,Int) Float ->
  let shj = \(f,d,A) -> mapArrInv (fst, dup, addf.snd,
    eqJoinArr (id, id, A, f (A, d))) in
  let next = \A ->
    let A' = shj (shiftArrR, (0,1), shj (shiftArrR, (1,0),
               shj (shiftArrL, (1,0), shj (shiftArrL, (0,1), A)))) in
    mapArrInv (id, id, \(_,x) -> divf (x, 4), A') in
  while (\(V,k) -> ((next V, addi (k,1)), lti (k,N)), (X, 0))) in ...
```

FIGURE 6.12: Jacobi 2D stencil

$$f ::= f_1 \cdot f_2 \mid f_2 \otimes f_2 \mid \mathrm{id} \mid \Delta \mid \Pi_1 \mid \Pi_2 \mid (f) \mid x$$

$$\begin{aligned} g ::= {} & g_1 \cdot g_2 \mid g_1 \otimes g_2 \mid \mathrm{id} \mid (g) \mid x \\ \mid {} & +(i) \textit{ (index vector addition)} \\ \mid {} & -(i) \textit{ (index vector subtraction)} \\ \mid {} & \times(i) \textit{ (index vector multiplication)} \end{aligned}$$

$$i ::= \mathbf{0} \mid \mathbf{1} \mid i^{-1} \mid x$$

$$h ::= h_1 \cdot h_2 \mid h_1 \odot h_2 \mid \mathrm{null} \mid f$$

FIGURE 6.13: Projection (f), indexing (g), and permutation (h) function syntax.

`next` input's fringe parameters `id` (i.e., no fringe), such that it unifies with `s -> s` and can be used as the `while` loop's parameter function.

6.4 Extended unification of functions

As already mentioned, the approximation of function unification that is used in the unification algorithm in Chapter 5, is not sufficient for the extended types discussed in this chapter. For these types we need a better approximation to full higher-order unification. To do this we encode the subset of functions we are interested in in first-order logic. We then use E-prover [183], which is a fully automatic equational theorem prover for first-order logic, which also returns values for any existentially quantified variables in conjectures, thus providing the substitutions for function variables that we need. This approach is automatic, solves equations between embedded functions quickly, and allows us to extend our unification algorithm to include other equational theories as needed.

In this section we give an equational theory for projection functions, and additional axioms for permutation functions and indexing functions. These can be used for partition functions, local layout functions, and the additional `DArr` type parameters, respectively. These could also be used to support further similar extensions, and more equational

$$
\begin{array}{lcl}
\beta \cdot \alpha & \stackrel{def}{=} & \backslash \mathtt{x} \to (\beta(\alpha\ \mathtt{x})) \\
\alpha \otimes \beta & \stackrel{def}{=} & \backslash(\mathtt{x}, \mathtt{y}) \to (\alpha\ \mathtt{x},\ \beta\ \mathtt{y}) \\
\mathrm{id} & \stackrel{def}{=} & \backslash \mathtt{x} \to \mathtt{x} \\
\Delta & \stackrel{def}{=} & \backslash \mathtt{x} \to (\mathtt{x}, \mathtt{x}) \\
\Pi_1 & \stackrel{def}{=} & \backslash(\mathtt{x}, \mathtt{y}) \to \mathtt{x} \\
\Pi_2 & \stackrel{def}{=} & \backslash(\mathtt{x}, \mathtt{y}) \to \mathtt{y} \\
 & & \\
\mathbf{0} & \stackrel{def}{=} & (0,\ \ldots,\ 0) \\
\mathbf{1} & \stackrel{def}{=} & (1,\ \ldots,\ 1) \\
i^{-1} & \stackrel{def}{=} & \mathtt{diviv}\ (\mathbf{1}, i) \\
+(i) & \stackrel{def}{=} & \backslash \mathtt{x} \to \mathtt{addiv}\ (\mathtt{x}, i) \\
-(i) & \stackrel{def}{=} & \backslash \mathtt{x} \to \mathtt{subiv}\ (\mathtt{x}, i) \\
\times(i) & \stackrel{def}{=} & \backslash \mathtt{x} \to \mathtt{muliv}\ (\mathtt{x}, i) \\
 & & \\
\alpha \odot \beta & \stackrel{def}{=} & \backslash \mathtt{x} \to (\alpha\ \mathtt{x},\ \beta\ \mathtt{x}) \\
\mathrm{null} & \stackrel{def}{=} & \backslash \_ \to ()
\end{array}
$$

FIGURE 6.14: Pointed definitions of point free functions.

theories could be added to deal with other possible extensions, as required. These theories all use a point-free formulation of functions and function comprehension that we prove sound by translating into pointed form (cf. Appendix D). The syntactic domains for these functions are f, h, and g respectively in Figure 6.13. Here, x is an identifier or expression in the base language, which is considered to be a constant in the theories, i.e., some function we do not know about. The semantics of the constant functions and composition operators are given in Figure 6.14 by translating them into pointed form. The rest of this section explains these theories and gives their axiomatizations.

6.4.1 Projection function theory

The equational theory of projection functions given in Figure 6.15 involves pairs and function composition. Supporting pairs, rather than n-ary tuples is not a limitation in this context, since any Flocc program can be converted into an equivalent program with only pairs, by converting tuples to nested pairs. Here $\cdot$ is sequential composition, $\otimes$ is pairwise composition, Δ is pairwise duplication, id is the identity function, and Π_1 projects out the left part of a pair, and Π_2 the right. $\cdot$ is associative.

$$f \cdot \mathrm{id} = f \tag{6.1}$$
$$\mathrm{id} \cdot f = f \tag{6.2}$$
$$\Pi_1 \cdot \Delta = \mathrm{id} \tag{6.3}$$
$$\Pi_2 \cdot \Delta = \mathrm{id} \tag{6.4}$$
$$(\Pi_1 \otimes \Pi_2) \cdot \Delta = \mathrm{id} \otimes \mathrm{id} \tag{6.5}$$
$$(\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta = \mathrm{id} \otimes \mathrm{id} \tag{6.6}$$
$$(f_1 \otimes f_2) \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta = (\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (f_2 \otimes f_1) \tag{6.7}$$
$$\Pi_1 \cdot (f_1 \otimes f_2) = f_1 \cdot \Pi_1 \tag{6.8}$$
$$\Pi_2 \cdot (f_1 \otimes f_2) = f_2 \cdot \Pi_2 \tag{6.9}$$
$$\Pi_1 \cdot (f_1 \otimes f_2) \cdot \Delta = f_1 \tag{6.10}$$
$$\Pi_2 \cdot (f_1 \otimes f_2) \cdot \Delta = f_2 \tag{6.11}$$
$$\Delta \cdot f = (f \otimes f) \cdot \Delta \tag{6.12}$$
$$(f_1 \cdot f_2) \cdot f_3 = f_1 \cdot (f_2 \cdot f_3) \tag{6.13}$$
$$(f_1 \otimes f_2) \cdot (f_3 \otimes f_4) = (f_1 \cdot f_3) \otimes (f_2 \cdot f_4) \tag{6.14}$$

FIGURE 6.15: Equational theory of projection functions.

$$h \cdot \mathrm{id} = h \tag{6.15}$$
$$\mathrm{id} \cdot h = h \tag{6.16}$$
$$h \odot \mathrm{null} = h \tag{6.17}$$
$$\mathrm{null} \odot h = h \tag{6.18}$$
$$(h_1 \cdot h_2) \cdot h_3 = h_1 \cdot (h_2 \cdot h_3) \tag{6.19}$$
$$(h_1 \odot h_2) \odot h_3 = h_1 \odot (h_2 \odot h_3) \tag{6.20}$$
$$(h_1 \odot h_2) \cdot h_3 = ((h_1 \cdot h_3) \odot (h_2 \cdot h_3)) \tag{6.21}$$

FIGURE 6.16: Additional axioms for equational theory of permutation functions.

6.4.2 Permutation function theory

Figure 6.16 contains additional axioms to supplement those in Figure 6.15, yielding an equational theory of permutation functions. Here all that matters is the flattened order of arguments returned, not the specific bracketing of pairs. For this reason, we introduce a new pairwise function composition $\odot$ which is associative. Here, $\alpha \odot \beta$ duplicates its argument and then returns α applied to it on the left, and β applied to it on the right. Again, since we are only interested in the flattened order of arguments returned (as this defines the lexicographical order of the collection) the null function that returns the unit value can always be ignored.

$$i^{-1^{-1}} = i \tag{6.22}$$
$$\mathbf{1}^{-1} = \mathbf{1} \tag{6.23}$$
$$+(i) \cdot -(i) = \mathrm{id} \tag{6.24}$$
$$-(i) \cdot +(i) = \mathrm{id} \tag{6.25}$$
$$\times(i) \cdot \times(i^{-1}) = \mathrm{id} \tag{6.26}$$
$$\times(i^{-1}) \cdot \times(i) = \mathrm{id} \tag{6.27}$$
$$g \cdot \mathrm{id} = g \tag{6.28}$$
$$\mathrm{id} \cdot g = g \tag{6.29}$$
$$(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3) \tag{6.30}$$
$$(g_1 \otimes g_2) \cdot (g_3 \otimes g_4) = (g_1 \cdot g_3) \otimes (g_2 \cdot g_4) \tag{6.31}$$

FIGURE 6.17: Additional axioms for equational theory of indexing functions.

6.4.3 Indexing function theory

Figure 6.17 gives the additional axioms for our equational theory of projection and indexing functions. These functions add, subtract, and multiply array indices by constants, runtime values, and the reciprocals of these. It is therefore able to invert any sequential composition of such functions, and solve any equation involving them. We encode both permutations of these inversion axioms (Equations (6.24), (6.25), (6.26), and (6.27)) since this makes the proof search quicker than using explicit commutativity axioms. The only condition that must be checked, is that indices are never multiplied or divided by zero.

6.4.4 Implementing unification with equational theories

To use these theories in our unification algorithm we first replace the DELETEFUN, CONFLICT, ELIMINATE rules in Figure 5.7 with those in Figure 6.18 (and add H, I, and J to the others). Note the syntactic domains T_f, T_g, and T_h refer here to the nonterminals in Figure 6.13, rather than those in earlier chapters. Here G are the original constraints, H are constraints between projection functions, I between permutation functions, and J between index functions. These rules "siphon off" these three kinds of constraints between embedded functions into separate sets while unifying the other constraints. If unification succeeds (i.e., G only contains substitutions, and does not infer $\bot$), then we can try to solve the constraints in H, I, and J using the E-prover and our equational theories of projection functions, permutation functions, and index functions respectively. We encode each of these three sets of constraints as a conjecture, where each conjecture is a conjunct between all the equations in the relevant set, which existentially quantifies over all free variables in the equations. We then run the theorem prover three times,

$$\frac{t_1 \in T_f \qquad t_2 \in T_f}{\langle G \cup \{t_1 \doteq t_2\},\ H,\ I,\ J\rangle \ \rightsquigarrow\ \langle G,\ H \cup \{t_1 \doteq t_2\},\ I,\ J\rangle}\ \textsc{DeleteProjFun}$$

$$\frac{t_1 \in T_g \qquad t_2 \in T_g}{\langle G \cup \{t_1 \doteq t_2\},\ H,\ I,\ J\rangle \ \rightsquigarrow\ \langle G,\ H,\ I \cup \{t_1 \doteq t_2\},\ J\rangle}\ \textsc{DeletePermFun}$$

$$\frac{t_1 \in T_h \qquad t_2 \in T_h}{\langle G \cup \{t_1 \doteq t_2\},\ H,\ I,\ J\rangle \ \rightsquigarrow\ \langle G,\ H,\ I,\ J \cup \{t_1 \doteq t_2\}\rangle}\ \textsc{DeleteIndexFun}$$

$$\frac{f \neq g \qquad f \notin T_f \qquad f \notin T_g \qquad f \notin T_h \qquad f \notin T_d}{\langle G \cup \{f(s_0,\ldots,s_k) \doteq g(t_0,\ldots,t_m)\}, H,\ I,\ J\rangle \ \rightsquigarrow\ \bot}\ \textsc{Conflict}$$

$$\frac{\begin{array}{c} x \notin vars(t) \\ x \in vars(G) \vee x \in vars(H) \vee x \in vars(I) \vee x \in vars(J) \end{array}}{\begin{array}{c} \langle G \cup \{x \doteq t\},\ H,\ I,\ J\rangle \ \rightsquigarrow \\ \langle [x \mapsto t]G \cup \{x \doteq t\},\ [x \mapsto t]H,\ I\{x \mapsto t\},\ [x \mapsto t]J\rangle \end{array}}\ \textsc{Eliminate}$$

FIGURE 6.18: Modified DDL type unification algorithm

$$\begin{array}{lcl} bargs(\mathtt{x}, f) & = & \{\mathtt{x} : f\} \\ bargs((x_1, x_2), f) & = & bargs(x_1, \Pi_1 \cdot f) \cup bargs(x_2, \Pi_2 \cdot f) \\ bargs(\_, f) & = & \{\} \end{array}$$

FIGURE 6.19: Definition of *bargs* (Bind projection functions to argument variables.)

once for each theory, with the constraints that pertain to them.[2]

Finally, to convert our constraints into equations for our theories, we have to convert all lambda-abstractions into point-free form. This is quite straightforward. The only place that pointed functions can enter our types is at dependent type schemes. All other functions are already point-free. To convert these abstractions into point free form we first remove all let-expressions by replacing any let-bound variables by their bound expressions, and converting all if-expressions to `ifF` combinator function applications. All tuples must also be converted to nested pairs. Note that functions containing list literals cannot be converted to point-free form unless the literal is first converted into `cons` function applications. Then we apply the function *pf* as shown in Figure 6.20 and Figure 6.19. For example, to convert `\(`x`, (`y`, `z`)) -> (`f y`, `g `(`z`, `x`))` into point-free form we get $\Gamma(x) = \Pi_1 \cdot \mathtt{id}$, $\Gamma(y) = \Pi_1 \cdot \Pi_2 \cdot \mathtt{id}$, and $\Gamma(z) = \Pi_2 \cdot \Pi_2 \cdot \mathtt{id}$. Then, *pf* returns $((f \cdot \Pi_1 \cdot \Pi_2 \cdot \mathtt{id}) \otimes (g \cdot ((\Pi_2 \cdot \Pi_2 \cdot \mathtt{id}) \otimes (\Pi_1 \cdot \mathtt{id})) \cdot \Delta)) \cdot \Delta$.

[2] Note that in E-prover we actually use the keyword `question` rather than `conjecture` since this gives us values for all existentially quantified variables, if the conjecture is proved true.

$$
\begin{array}{lcll}
pf(\Gamma, \mathtt{x}) & = & \Gamma(\mathtt{x}) & \text{if } \mathtt{x} \in \Gamma \\
 & = & \mathtt{x} & \text{if } \mathtt{x} \text{ is a function} \\
 & = & \bot & \text{otherwise} \\
pf(\Gamma, l) & = & \mathtt{constfun}(l) & \\
pf(\Gamma, (e_1, e_2)) & = & (pf(\Gamma, e_1) \otimes pf(\Gamma, e_2)) \cdot \Delta & \\
pf(\Gamma, e_1\ e_2) & = & pf(\Gamma, e_1) \cdot pf(\Gamma, e_2) & \\
pf(\Gamma, \backslash x \to e) & = & pf(\Gamma \cup \mathit{bargs}(x, \mathtt{id}), e) &
\end{array}
$$

FIGURE 6.20: Definition of pf (Convert to point free form.)

6.5 Concluding remarks

In this chapter we have shown how our DDL types can be extended to encode local data layouts, more flexible functions, and more expressive array distributions. These demonstrate the extensibility of our approach, and how expressive dependent DDL types can be. Of these extensions, our implementation (described in the next chapter) only currently implements local data layouts. However, we have also shown how we can extend our type inference (unification) algorithm to better approximate higher-order unification via equational theories. These theories are proved sound in Appendix D.

Chapter 7

Implementation

We have implemented a prototype code generator for Flocc that produces MPI implementations in C++, in about 25,000 lines of Haskell code. This supports, at the high-level, three different distributed data types (i.e., `DList`, `DArr`, and `DMap`) with in total 40 combinators, which are implemented in 58 low-level (i.e., C++) variants. Both the language and the generator can easily be extended by defining new types, combinators (cf. Section 7.3.1), and replacement rules (cf. Section 7.3.2), and giving the corresponding combinator implementations in the form of templates (cf. Section 7.4.6).

The architecture of our prototype code generator is shown in Figure 7.1. Flocc programs are first parsed and type checked as per the grammar and type system described in Chapter 3. Chapters 4, 5, and 6 describe how we synthesize concrete plans by inferring DDL types and automatically insert redistribution type casts, for different sets of specific combinator implementations, to use for these programs. In this chapter we explain how the front-end, plan synthesis, and back-end of our code generator are practically implemented, and how we use performance-feedback to search for *good* distributed-memory implementations of high-level Flocc programs.

7.1 Overview

Our code generator's high-level process is illustrated in Figure 7.1. After 1) AST parsing, 2) type-checking, and pre-processing, the generator loads 11) the combinator implementation rules and 10) their DDL types, and then 3) uses the type inference described in Chapter 5 to find possible distribution plans, together with the corresponding DDL types. For each plan, it 4) converts the AST into a data flow graph (DFG), replacing the high-level combinators with the implementations from the plan. The generator then 5) traverses this DFG, applying expression templates for each combinator. The templates generate blocks of C++ code to perform the corresponding operation, specialized to

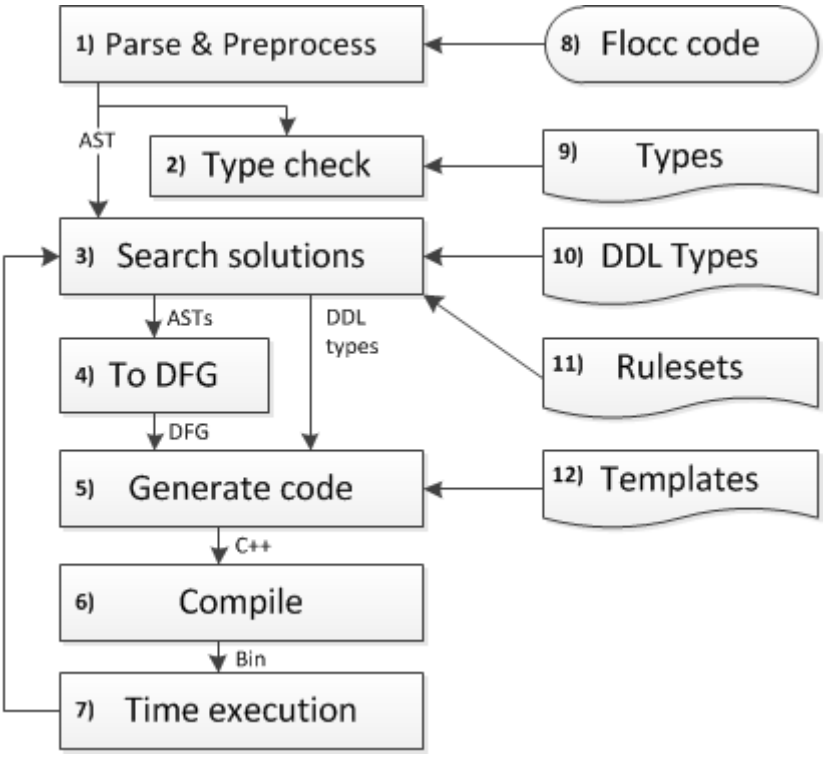


FIGURE 7.1: Feedback-directed code generation for Flocc

the DDL type from the plan. These plan synthesis and code generation phases form the inner loop of 3) the solution search. The search applies these phases to enumerate different C++ implementations from different choices of combinator implementations for function applications in the input program. Each generated program is then 6) complied, 7) executed, and performance statistics (i.e., execution time) are fed back into the search. Thus, the code generator searches for the fastest concrete distributed memory implementation of a given input program, according to some performance metric; in our case the overall execution time.

7.2 Front end

We have used Alex [70] and Happy [140] as lexer, and parser generators for our input language and the different definition languages (i.e., functional and DDL types and replacement rules). After parsing a Flocc program, a straightforward implementation of Algorithm W [62] infers the functional types for all the expressions in the program's AST.

After parsing and inference of the functional data types, the preprocessor expands all tuple-typed variables to tuples of variables, and replaces all function-typed variables with the lambda-abstractions they are bound to. This ensures that all Π-bound lambdas are directly available at function application expressions. It also expands lambda-

abstractions applications, so that different applications can have different DDLs.

7.3 Plan synthesis

The plan synthesis phase implements the DDL type inference algorithm, and redistribution insertion Algorithm 1 presented in Chapters 4 and 5. The current prototype does not use the extended inference algorithm presented in Chapter 6.

7.3.1 Type declarations

Chapter 4 does not specify how to marshal DDL type declarations. Our implementation uses a common file type for all functional and DDL type definitions, and common code to parse and lex them. It uses a single extendable data type to represent all types as S-expressions with variables, i.e., $(f\ t_1\ \ldots\ t_n)$, where f is a label, and t_1 to t_n are nested expressions, or variables. Arrows for function abstractions and tuples are also included to improve readability. For example, Figure 7.1 and Figure 7.2 show the marshalled functional and DDL types, for `eqJoin1` respectively.

```
1 eqJoin :: forall k1,k2,v1,v2,x =>
2   ((k1,v1) -> x, (k2,v2) -> x,
3    (Map k1 v1), (Map k2 v2)) -> (Map (k1,k2) (v1,v2))
```

Listing 7.1: Data type definition

```
1 eqJoin1A (f1,f2,_,_) ::
2   ((k1,v1) -> k, (k2,v2) -> k,
3     DMap Stm k1 v1 sf1 f1 dim0 mdim,
4     DMap Vec k2 v2 f2 f2 dim0 mdim2) ->
5   DMap Stm (k1,k2) (v1,v2) (FAssocPair (FSeq sf1 FLft) (FSeq f2 FRht))
6                            (FSeq f1 FLft) dim0 ()
```

Listing 7.2: DDL type definititon

Internally these type terms are extended with labels which carry the expression IDs of the expressions that generated them as shown in Figure 5.10, to support the redistribution insertion technique described in Section 5.2.

7.3.2 Combinator implementation rules

Each of Flocc's high-level combinators can be implemented in multiple versions. Our automatic distributed-memory implementation search algorithm hinges on being able to enumerate different choices of these combinator implementations for the function

applications in a Flocc program. We therefore need to declare which combinator implementations relate to which high-level combinators.

For our code generator, the sets of possible implementations are explicitly defined in easily extendable *rule sets*. Each rule has on the left-hand side a combinator's name and formal argument(s). The right-hand side then lists one or more low-level expressions that implement the high-level combinator. For example, the rule

```
countMap mp =>
  countVMap mp
| countVMapMirr mp
| reduceSMap(\_ -> 1, +, 0, mp);
```

specifies three implementation alternatives for the `countMap` combinator. The first two work on vector-based map implementations, where `countVMapMirr` is only applicable if the map is replicated, as its type fixes the partition function to be `nullF`. The third one actually iterates over the stream representation of the map, counting and summing the counts using `MPI::AllReduce`.

This approach also minimizes the number of combinator implementations required, since rules can encode left and right variants using the same function with its inputs and outputs swapped around. For example, Figure 7.3 shows some of the replacement rule for `eqJoin`, where `eqJoin1A` and `eqJoin1B` both perform the join in-place to maps aligned on their join key, and `eqJoin2` partitions the left-hand map and mirrors the right-hand map. The last alternative switches the two input maps by applying `eqJoin2` to `(x',x)` instead of `(x,x')`, and then uses `mapStm` to switch the output pairs back, such that the left-hand map is mirrored and the right-hand map partitioned.

```
1 eqJoin (f1,f2,x,x') =>
2     eqJoin1A (f1, f2, x, x')
3   | eqJoin1B (f1, f2, x, x')
4   | eqJoin2 (f1, f2, x, x')
5   | mapStm (\((k,k'),(v,v')) -> ((k',k),(v',v)),
6             eqJoin2 (f2,f1,x',x,))
7   ...
```

Listing 7.3: Replacement rule file snippet

7.3.3 Representing solutions

The solution search described in Section 7.5 needs some means of representing possible solutions that supports different search algorithms, including genetic searches. We use lists of integers for this purpose, with an element for each combinator function application in the input program, in the order they appear in the input program. Each element

records which rule alternative should be used for each high-level combinator. Then we include a final element to specify which of the possibly multiple ways to insert redistributions found by the insertion algorithm, should be chosen. This representation allows us to search depth first by incrementing the elements in lexicographical order, and to apply genetic searches by treating these list elements as individual genes.

To translate these lists into concrete plans, first each combinator application in the source program is replaced with the rule alternative specified (with all variables replaced with fresh names). We then perform redistribution insertion to infer the DDL types and insert any necessary redistributions. This may return multiple solutions which use different redistribution functions inserted at different points in the program. If this happens we sort the solutions by their redistribution costs, and select the i-th alternative, where i is the value of the last integer (i.e., gene) in the list. We then generate C++ code from these plans as described in the section that follows (cf. Section 7.4).

7.4 Back End

Making Flocc a functional programming language has an number of benefits including supporting type inference, polymorphic functions, passing functions as arguments, and the data distribution planning technique in Chapters 4 to 6. However, generating efficient imperative code from a functional programming language involves a number of challenges. In this section we explain how we have addressed these to implement our current code generator prototype.

7.4.1 Expression evaluation

The first challenge in generating C++ code from Flocc programs is how to convert an applicative language into an imperative one; i.e., a language with expressions that return values into operations that effect variables. We convert the AST into a data flow graph (DFG), replacing all literals, tuple expressions, function applications etc. with nodes. At let-expressions, we generate nodes for the bound expressions, and then add them to a context that binds variable names to nodes; when a variable is visited, an edge is created from the bound expression's node to the consuming expression's node.

Each DFG node is an object that can contain literals, types, C++ variable names, statements, and expressions; in particular, each node stores the C++ variable name(s) of its output value(s). Before the code is generated, the DFG is visited to create fresh C++ variable names for all intermediate values, and perform any other node initialization. Library function and combinator application nodes are initialized using templates which define how to generate code for that operation. The template instances are initialized with the node identifiers of their inputs, and their concrete DDL types, which may

include (nested) DFGs for parameter and partition functions. The code generator traverses the DFG depth first, adding the C++ declarations for intermediate and output variables, and statements to compute the correct values. This produces a C++ program with statements which compute the value(s) for the original functional Flocc program by assigning the expressions to intermediate variables.

In order to generate an efficient program, we must choose an efficient evaluation order, which in our case means choosing a good DFG traversal order. We could search for an optimal traversal order similar to the way we search for optimal distribution plans, but this would substantially increase the search space. We instead use a *deepest depth first* traversal, i.e., we evaluate the deepest subtrees first. This heuristic produces good results, as values are computed just before they are used, and can be freed again quickly.

7.4.2 Function handling

In fully functional languages (i.e., that treat functions as first-class objects which can be created and composed at runtime), there is no general way to determine statically what function a function application will actually apply. Functions are therefore implemented as thunks, i.e., pointers to a code block with an environment to store any captured variables. This has a considerable negative impact on performance. We have instead made Flocc a "semi-functional" language. That is, we allow polymorphic and higher-order functions, as long as the code generator can statically determine which function abstraction (i.e., lambda-term) is being used at applications. It therefore performs a semantic check after parsing, to make sure that this is the case. This allows us to inline functions, and, in particular, to lift them into the type language during DDL type inference.

When an application node is visited during DFG traversal, the code for any lambda-abstractions stored as nested DFGs is generated by setting the variable names for the DFGs input and output variable(s), and then traversing it in the same way as the top-level DFG.

7.4.3 Copy avoidance

The expression evaluation described in Section 7.4.1 in effect produces code in static single assignment form. For scalar variables, the C++ compiler will eliminate these repeated copies and replace them by in-place updates, but for the collection types the code generator has to produce code that avoids unnecessary copying.

We therefore introduce local *stream* storage types for all high-level collections (cf. Chapter 6), and use re-distribution functions to type-cast between memory-resident collections and local streams. The combinator implementations then consume and produce

such streams instead of explicit collections. The templates for these combinator implementations declare a public *stream variable*, which always stores the current collection element like an iterator, and is accessible to all consumer nodes. The consumer nodes use this variable to generate a *consumer* code block, which is then passed back to the stream producer, and spliced into the producer's loop nest. So, although collections are not transformed in place, new collections are only created when needed. This system also performs a kind of loop fusion, since multiple stream consumer nodes are spliced into the same loop nest. Note that stream consumers can also generate initialization and finalization statements which are spliced in before and after producer loops, respectively.

7.4.4 Collection storage

The local storage of collections can have a large impact on the performance of the generated MPI implementations. For example, maps can be stored as streams, hash tables, binary trees, sorted vectors, or unsorted vectors. The DDL types therefore also include information on how to store the collections in local memory (cf. Section 6.1). They can be stored with different key indices using DDL functions like the partition function. Local streams (cf. Section 7.4.3) also include a DDL function to define in which order the values are produced.

This local layout aspect ensures that efficient data structures are used for storing collections for different operations, and that (unless re-distribution functions are inserted that act as explicit conversions) only combinator implementations with DDL types with matching local layout information are used together. Therefore, we do not have to address local storage for collections during code generation. In fact, we can even frequently avoid using resizable containers by preallocating collections to be the right size. This is achieved by adding expressions to compute the size of a collection, to a producer node's public environment so that it is available to any consumer nodes.

Another issue that must be addressed, however, is how and when to free collections. We currently use Boost smart pointers [88] to either deallocate when a collection goes out of scope (using `scoped_ptr`), or when all references to it have gone out of scope (using `shared_ptr`). Analyses do exist [82] to eliminate some instances of reference counting for DFGs, but we do not implement these in our current prototype.

7.4.5 Tuple storage

We store tuples in two ways. Whenever possible we store them using a separate variable for each element of the tuple, by passing around trees of C++ variable names, rather than simple strings. However, when tuples need to be stored in a collection, we pack the values into a struct. We declare a struct with comparison functions and hash code generation for every tuple type used in a program.

```
1  -- /mapList template
2  t23 :: Monad m => Template m
3  t23 (Tup [(vt1 :-> wt1),
4            (Lf (LfTy "DList" [vt2, lm1, pm1, pd1, md1]))] :->
5            (Lf (LfTy "DList" [wt2, lm2, pm2, pd2, md2])))
6     (LFun _ (LTup _ [funN, inN]) outN)
7     | vt1 == vt2 && wt1 == wt2 &&
8       lm1 == nameTy "Stm" && lm2 == nameTy "Stm" = do
9       -- check in and out dims are the same
10      assertM (return $ pd1 == pd2) $ "par dims don't match"
11      assertM (return $ md1 == md2) $ "mirror dims don't match"
12      -- get input stream vars
13      getVar (Lf "v1") inN "streamVar"
14      -- declare and set output stream vars
15      newVar (Lf "v2") wt1
16      runGenV "declareVar" "decStmVar" [Lf "v2"]
17      setVar outN "streamVar" (Lf "v2")
18      -- if we know the list's length, pass it on
19      ifVarExists "listCount" inN "listCount"
20        (setVar outN "listCount" $ Lf "listCount") -- then
21        (return ()) -- else
22      -- when gen is called, generate loop
23      setFun outN "genConsumers" nt (\_ -> do
24        -- generate map fun implementation
25        genFunV "funCode" funN (Lf "v1") (Lf "v2")
26        -- gen consumers
27        callAll outN "genConsumers" nt
28        getCode "init" outN "initStream"
29        getCode "fin" outN "finStream"
30        getCode "consume" outN "consumers"
31        -- add these consumers to producer
32        addCode inN "consumers" $ "<decStmVar><funCode>\n<consume>"
33        addCode inN "initStream" "<init>"
34        addCode inN "finStream" "<fin>"
35        return ())
36 t23 t n = terr' t n
```

FIGURE 7.2: mapList template

7.4.6 Code templates

We implemented our code generation templates as functions which create and initialize objects using monadic side-effects, similar to the way object-orientation is simulated in JavaScript. The "objects" include private and public variables, which may store types, functions, C++ variables, expressions, and statements. Some of the key monadic actions are `newVar`, `runGenV`, `setVar`, `getCode` and `addCode`. `newVar` *varName varType* creates a new local C++ variable called *varName* of type *varType*,
`runGenV` "`declareVar`" *varName argVars* binds the result of running the `declareVar` expression generator with arguments *argVars* to the local variable *varName*, and
`setVar` *node destVarName srcVarName* binds the value of *srcVarName* to the public member of *node* called *destVarName*. `getCode` *destVarName node srcVarName* assigns

any code bound to the public member of *node* called *srcVarName* to the local variable *destVarName*, and `addCode` *node destVar template* replaces all the meta-variables between angle brackets in *template* with the values of any local variables of the same name, and then appends the resulting code to the public variable *destVar* of *node*.

Figure 7.2 shows as an example the `mapList` template for distributed lists. It generates code to apply a function of type `vt1 -> wt1` to a stream of `vt1` values, producing a new stream of `wt1` values. Lines 3 to 8 pattern match on the template instance's concrete DDL type, and neighboring nodes, and line 10 and 11 check that the type parameters are as expected. Line 13 calls `getVar` which gets the C++ stream variable(s) from the `inN` node, and binds it to `v1` in the local environment. Lines 15 to 17 then create a new C++ stream variable `v2` of type `wt1`, creates a statement to declare it (bound to `decStmVar`), and assigns it to the public variable `streamVar` in node `outN` so that all consumer nodes can access it. Lines 19 to 21 try to get a C++ expression called `listCount`, which holds the global size of the distributed list, from `inN`, and assigns it to `outN` if it exists. Then lines 23 to 35 create a Haskell function `genConsumers` which the input node `inN` calls to generate the `init`, `fin`, and `consume` code blocks to splice before, after, and into its producer's loop. Line 25 uses `getFunV` to generate the code for the map function stored at the `funN` node, applied to `v1` and storing its result in `v2`. The `genConsumers` function then generates these three blocks of C++ code (i.e., `init`, `fin`, and `consume`), by calling `genConsumers` on all its output nodes using `callAll` (lines 27 to 30), and then prepending the `decStmVar` and map function's implementation `funCode` before its consumers `consume` blocks. Finally, line 34 reports an error if the concrete type does not match the pattern expected.

7.4.7 Example output

Figure 7.3 shows the main code for a distributed implementation of dot product (cf. Figure 3.10), generated by our implementation. The whole snippet is the implementation of a `readList` stream-producing combinator implementation. Lines 4 to 7 initialize the accumulator (i.e., `v17`) for locally sum reducing the product pairs, and lines 9 to 12 prepare to read the local partition of the two lists. Lines 14 to 31 read the first list (i.e. `v11`) and lines 15 to 30 implement the `zip`'s consume block which reads an element of the second list (i.e. `v6`) for every element of the first. Lines 17 to 26 show the `mapList` combinator template's `consume` block, which here computes the product of the current pair of elements (line 19), and adds them to the current accumulator (lines 21 to 24). The `reduceList` template's `fin` block then aggregates all the local accumulates using the `Allreduce` MPI function, returning the resultant dot product in `v17`.

```
1  // BEGIN readList
2  int v14 = 0;
3
4  // BEGIN reduceList init
5  double v17(v1);
6  double v18;
7  // END reduceList init
8
9  std::vector<double >::iterator v15(v6->begin());
10 v14++;
11 std::vector<double >::iterator v12(v11->begin());
12 std::vector<double >::iterator v13(v11->end());
13
14 for (; v12 != v13 && v14 > 0; v12++) {
15   // BEGIN zip consume:
16   if (v15 < v6->end()) {
17   // BEGIN mapList consume:
18       double v16;
19       v16 = (*v12) * (*v15);
20
21       // BEGIN reduceList consume
22       v18 = v17 + v16;
23       v17 = v18;
24       // END reduceList consume
25
26   // END mapList consume
27   }
28   else if (v15 == v6->end()) v14--;
29   v15++;
30   // END zip consume
31 }
32
33 // BEGIN reduceList fin
34 v18 = v17;
35 cartComm.Allreduce(&v18, &v17, sizeof(double), MPI_PACKED, v103);
36 if (cartComm != cartComm) {
37   cartComm.Bcast(&v17, sizeof(double), MPI_PACKED, rootRank);
38 }
39 // END reduceList fin
40 // END readList
```

FIGURE 7.3: Dot product generated C++ snippet (tidied)

7.5 Feedback-Based Implementation Search

Sections 7.2 to 7.4 describe how to generate *one* solution for a Flocc program. However, since each combinator can have different implementations, with different communication patterns and performance characteristics, we need to explore different combinations to find good overall solutions.

Typically, each combinator has one or two efficient implementations that use some preferred data distributions for their arguments (e.g., `groupReduce2` and `eqJoin1A`), and several others that are less efficient, but that have less stringent constraints on the input and output distributions (e.g., `groupReduce1` and `eqJoin2`). Re-distribution functions can be used to resolve incompatible constraints, and allow further implementations, but with a performance penalty (e.g., `groupReduceSMap·readVMap·sortVMap`). Good distributed implementations are therefore often a trade-off between giving some combinator applications their best implementations (and thus their preferred data distributions) and others worse ones, such that the type constraints are satisfied.

Initially, we ordered the combinator implementations in our replacement rules from fastest to slowest, and then to sought good DDL plans by starting with the fastest implementation for each combinator application, and then using progressively slower implementations until the plan type-checked. The problem with this approach was that it did not consider plans that involved data redistributions, and scaled exponentially with the number combinator applications. We therefore developed our automatic redistribution insertion technique (cf. Section 5.2) which allows us to make any DDL plan type-check. This involves using a simple cost metric to estimate the comparative performance costs of different redistributions. We have considered using this metric to compare whole plans, but experiments showed (cf. Section 8.3) that although this metric is sufficient for comparing different chains of redistributions with equivalent types, it is not a good predictor of an overall plan's performance. More sophisticated performance cost estimates would require estimates of the sizes and characteristics of runtime data, which could be very difficult to infer statically. Therefore, rather than try to develop more sophisticated cost estimates, we have used performance-feedback-based searches to find good overall solutions. This approach has the advantage of being very accurate and portable, since it uses real empirical performance data rather than relying on some model of a system's performance characteristics. Furthermore, such searches have been shown to work very well for other code generation problems [5, 87, 197].

This section describes different performance-feedback-based search algorithms that we have implemented for this purpose. Our experiments in Section 8.3 investigate the performance of these different search algorithms, by comparing the performances of the solutions they return, and how fast they converge to find these solutions.

7.5.1 Dimensions of the search heuristics

We have implemented different performance-feedback-based search heuristics, which can be distinguished along four dimensions. These dimensions are as follows.

Search algorithm The search algorithm controls, amongst other things, the order in which candidate solutions are visited. We can use simple depth first searches, where the combinator applications can be considered either from input to output, or vice versa. We could also use a greedy strategy where combinators like `groupReduceMap` that are often computationally intensive are explored first, and "cheaper" combinators like `mapMap` later. In addition, we can use random or genetic searches.

Termination The termination condition controls when to stop the search, ideally with an optimal or a near-optimal solution. The simplest conditions consider all candidates (i.e., *exhaustive search*) or terminate when some pre-determined budget has been exhausted, returning the best implementation found so far. We can also terminate when the current candidate is "substantially better" than the previous solutions (e.g., when it is twice as fast as the current mean, or faster than the current mean minus twice the standard deviation), or when the gradient of the search levels off.

Pruning Pruning tries to determine statically whether candidates can be ignored completely, without executing them. This can be seen as an approximation of static cost estimates. Our current pruning techniques are based on the data re-distributions that a solution performs, measured either by simply counting the number of its re-distribution operations, or, more effectively, by weighting the different operations differently. We maintain some threshold score from the solutions explored so far (e.g., average) and prune any solutions whose score exceeds this threshold.

Runtime pruning We can also set a runtime bound that decides how long a candidate solution is executed before terminating it. This can be seen as a dynamic version of pruning, and similar approaches as described above can be used.

7.5.2 Implemented search heuristics

For each dimension we have implemented several alternatives. In Chapter 8 we evaluate different combinations of these alternatives to find which combination finds the best overall plans, in the best overall search time. The main differentiating factor is the overall search algorithm, which we describe in detail here. All search algorithms are combined with different termination conditions, pruning, and runtime pruning techniques as outlined in Section 7.5.1.

Depth first The depth first heuristics exhaustively search all solutions by enumerating all combinations of combinator implementations for each combinator in the Flocc program, in the order in which the implementations are given in the rule sets. We use two variants that explore the combinators in data flow order from input to output, and vice versa.

Random The random search heuristics maintains a fixed-size population of candidate solutions, and iteratively chooses the n best to produce a new generation. It creates new children by applying up to two random mutations to each of the chosen solutions.

Genetic Genetic searches also iteratively improve a fixed-size population of candidate solutions. They start with some random candidates, evaluate their runtimes, and use the fastest $n\%$ of the population (but at least 2) as parents. The next generation is then created by nondeterministically choosing two parents, and combining them using a crossover operation. In the crossover, each combinator's implementation is randomly chosen from one of the parents. Finally, each child's combinator implementations may be randomly mutated with a given probability. The search algorithm thus has three parameters, the size of the population, the fraction of candidates used as parents, and the probability of mutating a given gene.

Genetic (without crossover) A second version of the genetic search only chooses a single parent to create each child and only performs mutations but no crossovers. Furthermore, instead of mutating a combinator application by choosing a completely random implementation, this works by randomly choosing either the previous or the next implementation. This exploits the fact that the combinator replacements are ordered from best to worst, and so neighboring combinator implementations are likely to be better than completely random ones.

7.5.3 Performing the search

During a search, the current search state is regularly serialized and saved to disk so that searches can be resumed after interruptions or machine failures. This includes the current search population, and a cache of the runtimes of all currently visited solutions. This cache speeds up the search by allowing us to avoid repeating performance measurements of already visited solutions.

As explained in Section 7.3.3 solutions in the search space are represented as lists of integers. The chosen search algorithm determines the order in which solutions in the search are visited, however for each visit a number of steps occur. First, if the solution does not appear in the cache, the plan is obtained by running the redistribution insertion

and DDL type inference algorithms. Then C++ code is generated from this plan, and this code is compiled using the chosen compiler (which is a configurable parameter) and with the highest optimization level available (i.e., `-O3`). Our experiments in Chapter 8 use Intel's `icc` and GNU's `gcc`. At this point the compiled executable is run, and performance measurements (i.e., total runtime) made. For the experiments in Section 8.3 we use random input data generated using a uniform distribution, although in practice more realistic test data could be supplied by the programmer. If execution exceeds some fixed budget (i.e., maximum execution time) the process is terminated. The current implementation runs executables locally, using as many cores as are available. However, any production implementation would need to support submitting jobs through SSH and PBS (or similar), to evaluate performance on a real cluster.

We have used this implementation to generate distributed-memory implementations for several example programs (cf. Chapter 8). However, finding these example programs revealed a limitation with the current implementation. That is, that especially when searching both the space of distributed and *local* data layouts, different implementations can have performances that differ by 2 orders of magnitude or more. This means that either the search becomes very slow, waiting for some very poor implementations to finish executing, or it must terminate them early, and lose important performance information. A piece of future work would therefore be to change the implementation to allow implementations to be generated that run on different sizes (or *complexities*) of test data. Searches could then start by evaluating implementations on small (or less complex) data sets, and then increase the size (or complexity) once very poor implementations have been discarded. This approach could also be used, to start evaluating implementations on local cores, and then once the whole population runs well locally, could be evaluated on a real cluster.

7.6 Concluding remarks

In this chapter we have given an overview of how our prototype Flocc compiler works, and described some of the practicalities involved in implementing the DDL plan synthesis technique described in Chapters 4 to 6. We have particularly focused on how our implementation generates C++ code from Flocc plans, and how our performance-feedback-based plan search is implemented. The following chapter (cf. Chapter 8) presents some experimental and conceptual evaluation of our approach that uses our proof of concept compiler to synthesize distributed-memory implementations for several example programs.

Chapter 8

Evaluation

In Chapters 3 to 7 of this thesis, we have presented a new technique for automatically synthesizing implementations of data-parallel programs that targets clusters, and explained its implementation. This fulfills our research objectives (i.e., 1 to 6). In addition to this we have performed some experimental and conceptual evaluation, that we present in this chapter, which demonstrates that we have fulfilled research objective 6 (i.e., that our implementation works in practice). A complete and comprehensive evaluation of our approach is outside the scope of this PhD thesis, but the content of this chapter demonstrates that our approach works in practice.

Our initial experiments focused on implementing our DDL type inference algorithm, and inferring DDL types for candidate programs. We then started to implement the back-end of our compiler, to investigate whether we could generate code from these programs that would give performance close to classic textbook MPI implementations. This involved extending our implementation's DDL types to include local data layout information and storage mode information. We then used this back-end to generate implementations from the DDL (and local layout) plans, and compared their performance to textbook versions, by running them on a Linux cluster. The results of these experiments are presented in Section 8.2.

After this we continued to investigate how we could find the best DDL plans for input programs, and generate code from them. This included developing our automatic redistribution insertion technique. After implementing redistribution insertion Algorithm 1 (cf. Figure 5.12) we implemented a performance feedback-based search, and investigated how long different search heuristics took to find the optimal solutions of four map-based example programs. The results of these experiments are presented in Section 8.3.

Finally, we have evaluated the conceptual merits of our approach, which we present in Section 8.4. This includes comparing the expressiveness our input language and programming model to other data-parallel programming languages, and how the expressiveness of our DDL types compare to the data distributions possible in other languages.

Problem	Flocc	Comparison		Types
Matrix multiply (cf. Fig 3.7)	5	C/MPI	89	Arr
Floyd's all pairs shortest path	15	C/MPI	88	Arr
Jacobi 2D (cf. Figure 6.12)	8	C++/MPI	120	Arr
Successive over-relaxation (red/black)	18	C/MPI	289	Arr
N-body (gravitational)	38	C/MPI	153	Arr
K-means clustering (cf. Figure 8.9)	36	C/MPI	114	Map
Triangle enum (cf. Fig 6.7)	12	C++/MR-MPI	263	Map
R-MAT graph generation (cf. Figure 8.10)	35	C++/MR-MPI	148	Map
PageRank	11	Java/Hadoop	157	Map
Histogram (cf. Fig 3.9)	6	C++/MPI	204	Map
Apriori association mining	14	Java/Hadoop	371	Map
Dot product (cf. Fig 3.10)	3	C++/MPI	35	List
Standard deviation	6	C/MPI	38	List
Simple linear regression	10	C++/MPI	47	List
Word count	3	Java/Hadoop	48	List & Map
Grep	2	Java/Hadoop	59	List

FIGURE 8.1: Comparative code sizes (code lines without comments and IO code)

8.1 Flocc Programs

Figure 8.1 lists some example problems that we have programmed in Flocc, and compares the code sizes of these programs with programs written in other languages.[1] These include numerical/array-based problems usually written in HPF, map-based problems usually written using MapReduce, and list-based ones usually written in MPI/MapReduce. All these problems were written in Flocc, demonstrating the versatility this high-level language approach. Furthermore, the Flocc implementations are between 3% (Histogram) and 32% (K-means) of the size of the comparisons (12% on average). This illustrates the potential productivity gains of this high-level language approach.

8.2 Automatic code generation

In this section we investigate whether generating cluster implementations from Flocc programs is viable, and in particular whether the performance of these implementations is reasonable (i.e., close to hand-coded equivalents). To do this we have used the back-end of our implementation (cf. Chapter 7) to generate MPI implementations in C++, for five Flocc programs, and then compared their performances to independently hand-

[1] See http://www.flocc.net/hlpp14/codesizes.html for details.

Program	Speedup	Compiler	Data
Dot product	4.96×	gcc -Ofast	2.2GB
Simple linear regression	137×	gcc -Ofast	3GB
Standard deviation	98.6×	gcc -Ofast	3GB
Histogram	31.5×	gcc -Ofast	32MB
Matrix multiply	342×	gcc -Ofast	1.4MB

FIGURE 8.2: Performance comparison of Flocc generated code vs. PLINQ implementations using 4-cores

coded versions of the same programs. All of the generated implementations compiled and executed successfully, and returned the correct answers, showing that we can generate working cluster implementations from Flocc plans/programs. In addition, for Section 8.3 we have used our Flocc compiler to automatically generate 170 different working MPI implementations from our histogram example program and three other Flocc programs. So we can clearly generate distributed-memory implementations from Flocc programs.

To evaluate whether generated implementations can give acceptable performance, we have compared the runtimes of these five programs to that of hand-coded versions. In Section 8.2.1 we compare the implementations to PLINQ versions, and in Section 8.2.2 we compare them to MPI versions written in C++.

8.2.1 Comparison with PLINQ

We first compare with PLINQ, because it also auto-parallelizes programs written in a functional language, and so is the most closely related approach. It is also one of the most modern and accessible of the high-level auto-parallelizing approaches.

We have generated implementations of five example Flocc programs using our prototype tool and compared them to PLINQ [72] using all cores on a 64-bit quad-core 2.67GHz workstation with 12GB memory. Figure 8.2 shows the average speedups of the Flocc versions compared to the PLINQ implementations. All Flocc programs drastically outperformed the PLINQ implementations. This is most likely because PLINQ is choosing poor job partitionings. However, it is also limited by the fact that unlike Flocc, it cannot inline lambda-abstractions, and does not distinguish between, and so cannot optimize for, arrays, lists, and maps. The Flocc implementation of matrix multiplication outperformed PLINQ by over 340×. This is most probably because PLINQ cannot directly query multidimensional arrays.

8.2.2 Comparison with MPI

Although PLINQ is the most closely related approach in some ways, most HPC applications are written using MPI, and so it is important that our approach yields similar performance to MPI implementations. Note that we were unable to compare our approach against Hadoop or DryadLINQ, since we do not have access to clusters that support these technologies.

We compare the performance of our five Flocc programs with that of näive hand-coded MPI implementations solving the same tasks (cf. Figure 8.8). These hand-coded versions are straightforward textbook implementations of the same tasks, rather than highly tuned BLAS implementations, since we are not aiming to compete with such implementations. This is for a number of reasons.

Firstly, we are currently only trying to show the overall *viability* of our approach. Our main aim is to develop a fully automated distributed-memory synthesis technique that works for multiple collection types, i.e., to do *automatically* what a *non-expert user* can do manually. Our major research effort has therefore been focused on automatically synthesizing suitable data distributions in a way that works for multiple collection types. Since this type-based DDL synthesis technique is our main novelty, we chose to prioritize developing an end-to-end prototype that shows the efficacy of the whole approach, and demonstrates all aspects (and especially the most novel ones, i.e., DDL type inference, redistribution insertion, and plan search), rather than fine-tuning back-end template implementations. The problem of generating high-performance code from a functional language (and particularly array-based programs) has already been addressed in languages like SAC [182] and SISAL [82]. Furthermore, auto-tuners like PHiPAC [20] and Spiral [166] are already able to generate high-performance code that can rival hand optimized BLAS implementations. For these reasons, and because writing good back end templates for array combinators is harder than maps and lists, we have focused on implementing templates for maps and lists, and mainly restrict our evaluation to map and list-based programs.

Experimental Set-Up

For each of the Flocc-examples we generated the implementation of the most efficient DDL plan that the type inference produces (i.e., the first of each of the solutions discussed in Section 4.5). We then either found näive textbook implementations of the same problems on the web (the kind a non-expert user could write), and modified them slightly so that they are comparable with the generated versions, or hand-coded implementations ourselves; see below for details.

For the evaluation we used a large third-generation cluster[2] with approximately 1000 Westmere compute nodes, each with two 6-core 2.67Ghz processors, 4GB of RAM per node, and an InfiniBand network for interprocess communication. The generated implementations of the matrix multiplication generate random input data on the root node and then distribute this using the derived strategy. For the histogram implementations, the data is generated locally at each node because the runtimes would otherwise be dominated by the initial data distribution. We use double-precision floating point numbers as payload data in all cases, and compile with `icc` using the `-O3` optimization flag. Note that we also compile using `gcc` in cases where this gives a performance that is notably different to `icc` (e.g., for the matrix multiply). We compare their performances for varying numbers of nodes; the reported runtimes are the average of at least three runs.

Matrix multiplication For the comparison, we used three different näive matrix multiplication implementations.[3] [4] [5] We compiled all four programs with `gcc` (v4.1.2) and Intel's `icc` (v11.1) to see the effects of different compilers and evaluated them over the multiplication of two 3000x3000 dense matrices.

Histogram Here, we modified a version[6] for comparison, to use `doubles` rather than `ints`, and to generate partitions locally, rather than scattering all the data from the root node. Both programs were compiled with `icc` (v11.1) and used to compute 100-bucket histograms of in total 1.07 billion values (i.e., 8GB of data).

Standard deviation For comparison we wrote a C++ MPI implementation that computes the standard deviation of an array of arbitrary `doubles`. Both programs were compiled with `icc` (v13.1.2), and the manual version was also compiled with `gcc` (v4.1.2), and applied to 400 million (2.98GB of) values.

Simple linear regression We wrote a C++ MPI implementation to perform simple linear regression on two arrays of `doubles`. Both programs were compiled with `icc` (v13.1.2) and applied to 200 million (1.49GB of) values.

Dot product We implemented a C++ MPI comparison to compute the dot product of two randomly generated aligned arrays of `doubles`, such that they did not have to be repartitioned before the dot product could be computed. The generated version also

[2] http://cmg.soton.ac.uk/iridis
[3] www.cs.hofstra.edu/~cscccl/csc145/imul.c
[4] www.eecg.toronto.edu/~amza/ece1747h/homeworks/examples/ MPI/other-examples/mmult.c
[5] www.cs.arizona.edu/classes/cs522/fall12/examples/mpi-mm.c
[6] http://penguin.ewu.edu/~trolfe/CCSC2002/MPI/MPI_Hist.C

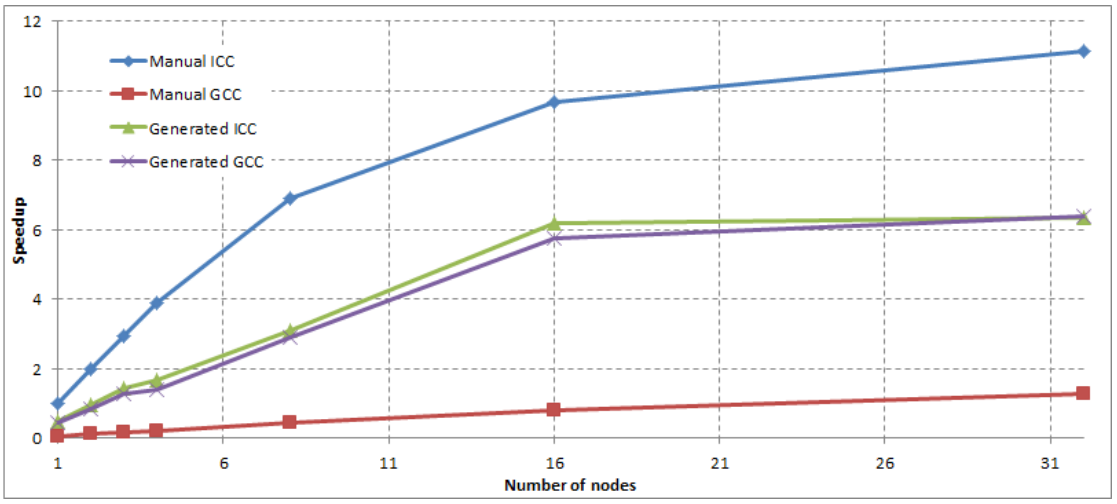


FIGURE 8.3: Performance of matrix multiplications

used an aligned distribution, where the two input arrays are distributed using a cyclic distribution on the same cluster dimension. The programs were both compiled with `icc` (v13.1.2) and applied to a pair of arrays with 200 million (1.49GB of) and 400 million (2.98GB of) values respectively.

Results

The results of the performance comparisons follow. The graphs here show the relative speedups of the manual MPI and generated Flocc versions by calculating all the speedups relative to the fastest single node execution time. In all the examples below this is the execution time of the manual MPI implementation compiled with `icc`.

Matrix multiplication For the matrix multiplication the results vary greatly with the applied compiler. For `gcc`, the generated code outperforms the manual implementation by a factor of about six, independent of the number of nodes (see Figure 8.3). For `icc`, the situation is reversed, and the manual implementation outperforms the generated code, albeit only by a factor of about two. The reason the generated version works so well with `gcc` is that it uses the local memory layout optimization described in Section 6.1 and so stores the second matrix in transposed form, for which `gcc` generates optimized code. The reason the generated `icc` version performs worse than the manual version, is that the manual version loops over a global static array, and `icc` seems to optimize for this special case.

Histogram Figure 8.4 shows that the generated version is approximately 30-40% slower than the hand-coded version. This is because our implementation of `groupReduce1` uses a hash map, so it can work with any key type, whereas the hand-coded version uses

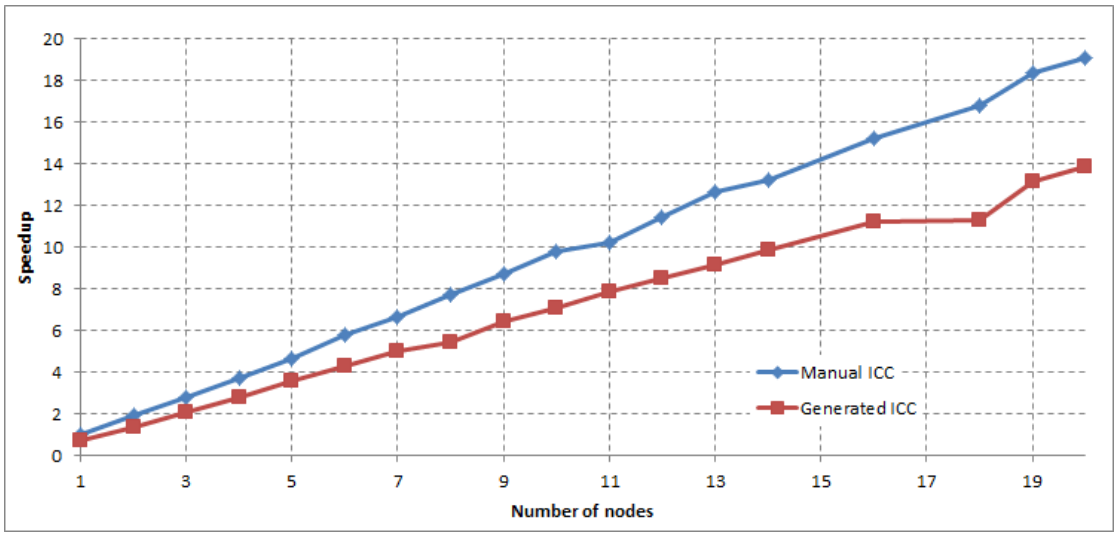


FIGURE 8.4: Performance of histogram implementations

an array. This could be addressed by providing a special version of `groupReduce` that outputs arrays, by implementing an *array* local layout mode for maps that can be stored as arrays, or by using the integer domain analysis mentioned in Section 9.2 to detect when we can store maps as arrays.

Standard deviation Figure 8.5 shows that the speedup of the Flocc version is a maximum of 32% less (on 8 nodes) than the manual version's, and 15% less on average. This is a minor difference, and is likely because the generated version uses `MPI::Allreduce`, which mirrors its result on all nodes, rather than `MPI::Reduce`, which stores its result only on the root node, and as the speedups tail off this extra overhead begins to dominate. This effect only occurs as the speedups tail off, and so in practice would only be experienced when using too many nodes for the size of dataset.

Simple linear regression Figure 8.6 shows that the generated version's speedup is very close to (within 17% of) the manual implementation's on 1 to 16 nodes, exceeding the `icc` manual version by 14% on 9 nodes. As the number of nodes increases, the performance of the generated version drops off to 61% slower on 32 nodes. This may partly be because the manual version iterates over a struct of arrays, rather than an array of structs, and also because the generated version uses `MPI::Allreduce`, rather than `MPI::Reduce`.

Dot product Figure 8.7 shows that the performance of the manual and generated implementations of dot product are identical for all but the last data point (on 32 nodes).

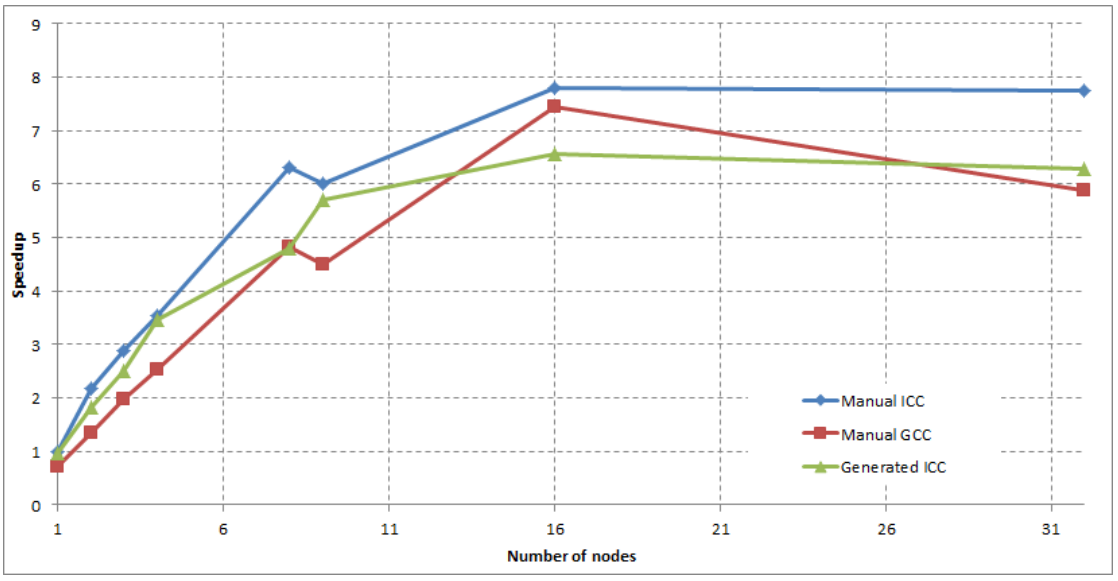


FIGURE 8.5: Performance of standard deviation implementations

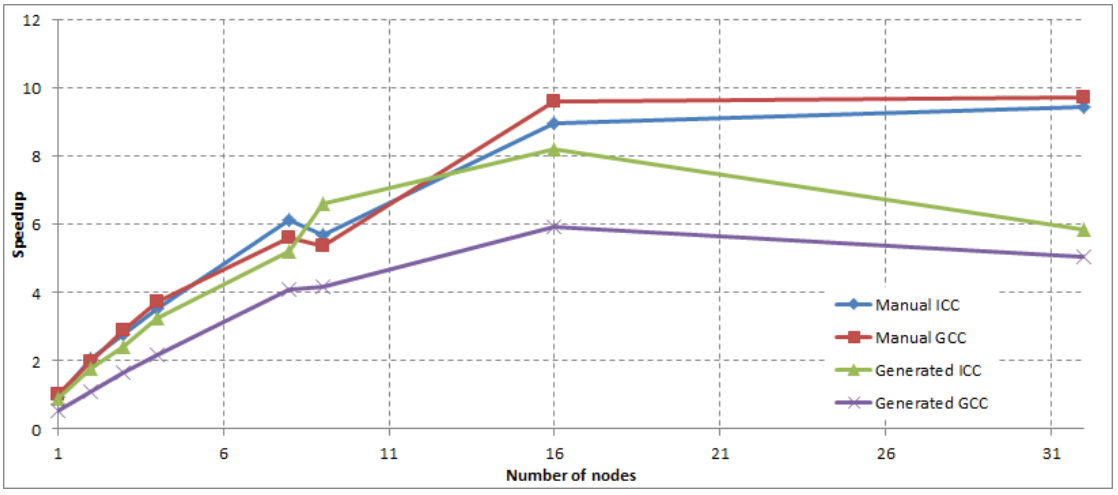


FIGURE 8.6: Performance of simple linear regression implementations

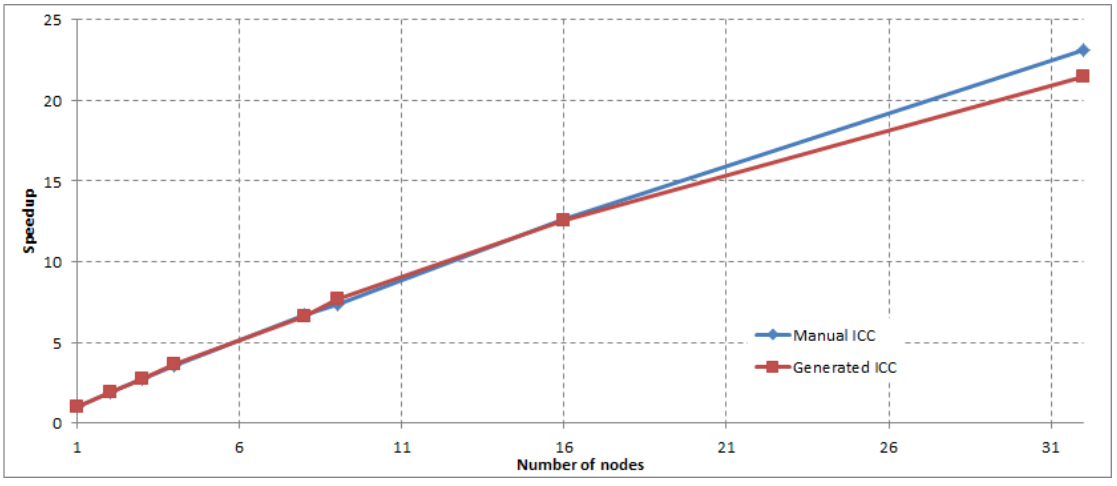


FIGURE 8.7: Performance of dot product implementations

Program	Average speedup	Compiler	Data
Dot product	0.99×	icc -O3	4.5GB
Simple linear regression	0.89×	icc -O3	3GB
	0.61×	gcc -O3	3GB
Standard deviation	0.88×	icc -O3	3GB
	1.00×	gcc -O3	3GB
Histogram	0.73×	icc -O3	8GB
Matrix multiply	6.14×	gcc -O3	140MB
	0.49×	icc -O3	140MB

FIGURE 8.8: Average performance of Flocc generated code compared with manual MPI implementations on 1 to 32 nodes

Discussion

The generated versions compete well with (i.e., came within 51% of the speed of) hand-coded MPI versions (cf. Figure 8.8), demonstrating the viability of our approach for these sort of data-parallel programs. The dot product and simple linear regression compiled with ICC, and the standard deviation, were nearly identical to the hand-coded versions. The linear regression when compiled with ICC was 39% slower because it used an array of structs, rather than a struct of arrays. The histogram was 27% slower, because it used a hash table, and the comparison used an array. The matrix multiply was 6× faster than the hand written code when compiled with GCC, since our tool optimized the layout of `B` to be column-major, but was 51% slower when compiled with ICC. This is because the manual version iterates over global arrays, and ICC seems to optimize for this case. An additional `Global` array storage mode (see Section 6.1) and corresponding templates, would cater for this situation. Similarly, it may be worthwhile adding DDL types for scalars that should be stored on a single node, so implementations can use `MPI::Reduce` instead of `MPI::Allreduce`.

These results could be improved further by optimizing the existing, and adding additional, back-end templates, however, they are sufficient to indicate that the approach is viable in practice.

8.3 Automatic plan generation

In this section we use our compiler implementation's performance-feedback-based plan generation search to find MPI implementations of four example programs, and to compare the average search times of different search heuristics. To do this we have chosen four map-based Flocc programs, and used our system to search for good distributed-memory implementations for them. These examples are larger than those used in the

previous section, and exercise all phases of our implementation, including parsing, type inference, DDL type inference, redistribution insertion, plan search, and code generation. Our implementation synthesizes 170 viable implementations in total— between 30 and 48 per example.

8.3.1 Experimental setup

We took four Flocc programs (see below) that represent map-based versions of different algorithms. Two of these programs (i.e., `rmat` and `mandel`) take a problem-size as input, and for the other programs (i.e., `kmkernel` and `hist`) we randomly generated a uniformly distributed appropriate input data set, see Table 8.1 for details. Our current evaluation only uses a single input data set for each program. Future work, could investigate to what extent this effects the choice of best solution. We used our code generator to exhaustively generate all possible variants. This included trying two different ways of inserting redistribution functions per solution. We then compiled each generated MPI/C++ implementation using the `GCC` compiler (v4.6.3) with the `-O3` optimization flag. We finally ran the compiled binaries on the corresponding input data, and measured and cached their run times, so that we could use them to compare the efficiency of different search heuristics.

Test programs For the experiments we used the histogram computation shown in Chapter 3 (cf. Figure 3.9), a k-means clustering kernel (cf. Figure 8.9, `kmkernel`), a random matrix generation (cf. Figure 8.10, `rmat`), and a Mandelbrot set computation (cf. Figure 8.11, `mandel`); note that the listings omit all scalar functions for brevity. Most of these programs are well-known. `rmat` generates a sparse random adjacency matrix representing a graph; the choice of its parameters controls the degree distribution of the resulting graph [38].

These Flocc programs use between four and eight high-level combinators; in total, they use 12 of the 15 `Map` combinators. The programs yield between 30 and 96 different MPI/C++ implementations, with sizes ranging from 700 to 3500 lines of C++ code. Table 8.1 contains more details; here, $|C|$ and $|S|$ denote the numbers of combinators in the program and the number of generated solutions. Table 8.1 also shows, for each Flocc program, the ratio of the best runtime to the worst runtime, as well as the average runtimes and standard deviation of the runtimes, presented as fractions of the best runtime for each program.

Experimental environment We have run the generated programs on a standard quad-core (Intel Xeon W3520/2.67GHz) x64 desktop with 12GB of memory, running

```
let kmkernel = (\(points, clusters) ->
  -- distances between points and clusters &
  -- new points with their closest clusters
  let points' = groupReduceMap (
    \((pid,_),_) -> pid,
    \((pid,cid),((ncid,ocid,ppos,d),cpos)) ->
      (cid, ocid, ppos, distPoints (cpos, ppos)),
    \((nc1,oc1,p1,d1),(nc2,oc2,p2,d2)) ->
      (if (d1 < d2)
       then (nc1,oc1,p1,d1)
       else (nc2,oc2,p2,d2)),
    (-1,-1,(-1.0,-1.0),9000000.0),
    crossMaps (points, clusters)) in
  -- new cluster centres (avg member pos)
  let clusters' = groupReduceMap (
    \(pid,(ncid,ocid,ppos,d)) -> ncid,
    \(pid,(ncid,ocid,ppos,d)) -> (ppos, 1),
    \((sum1,tot1),(sum2,tot2)) ->
      (addPoints(sum1,sum2), (tot1 + tot2)),
    ((0.0,0.0),0), points') in
  let clusters'' = mapMap (
    \(cid,(psum, ptot)) ->
      (cid, divPoint (psum, toFloat ptot)),
       clusters') in
  -- count how many memberships changed
  let totalChanged = reduceMap (
    \(pid,(ncid,ocid,ppos,d)) ->
      (if (ncid == ocid) then 0 else 1),
      +, 0, points') in
  (points', clusters', totalChanged)) in ...
```

FIGURE 8.9: K-means kernel

```
let rmat = (\N ->
 loop (\E-> (
  -- calc more egdes
  let size = countMap E in
  let ints = intRangeMap (1,N-size,1) in
  let E' = mapMap (\k ->
    (k, genE (a0,b0,c0,d0,delta0)), ints) in
  -- remove duplicates
  let E'' = groupReduceMap
    (snd, \_->(), fst, (), E') in
  -- combine with existing
  union (E'', E)),
  (\E -> (countMap E)<N), emptyMap())) in ...
```

FIGURE 8.10: Random adjacency matrix generation

Ubuntu 12.04 LTS as operating system. We used the OpenMPI 1.6.5 implementation.

[1]The number of plans was actually higher than this (i.e., 48 for `hist` and 102 for `mandel`), but some of their generated implementations could not be used because interactions between certain combinations of templates led to errors.

```
let mandel = (\(xs, ys) ->
 let axes = crossMaps ( -- make axes
   intRangeMap (-2.5*xs,1*xs,1),
   intRangeMap (-1*ys,1*ys,1)) in
 -- evaluate at each point
 mapMap (\((x,y),_) -> ((x,y), escape_time (
   (toFloat x) / (toFloat xs),
   (toFloat y) / (toFloat ys))), axes)) in ...
```

FIGURE 8.11: Mandelbrot set

Program	$\|C\|$	$\|S\|$	∅ size	training data	$t_{best} : t_{worst}$	mean	variance
`hist`	4	44[1]	886	8,000,000 entries	1 : 38.5	19.6	11.6
`kmkernel`	5	30	1235	100 points, 5 clusters	1 : 11.0	2.79	2.05
`mandel`	4	48[1]	752	140x20 pixel	1 : 2.56	1.67	0.548
`rmat`	8	96	3482	1000 edges	1 : 57.3	4.90	11.62

TABLE 8.1: Characteristics of test programs

8.3.2 Results

Solutions The graphs in figures 8.12, 8.13, 8.14 and 8.15 show in blue the speedups (actually slowdowns), of the different solutions, relative to the fastest solution for each example program. They also show in red redistribution cost estimates, which were discussed on page 93. The individual solutions are shown from left to right in order of a depth first traversal. Here the amount of global computation remains the same; the difference in performance comes from different data re-distribution times and local access

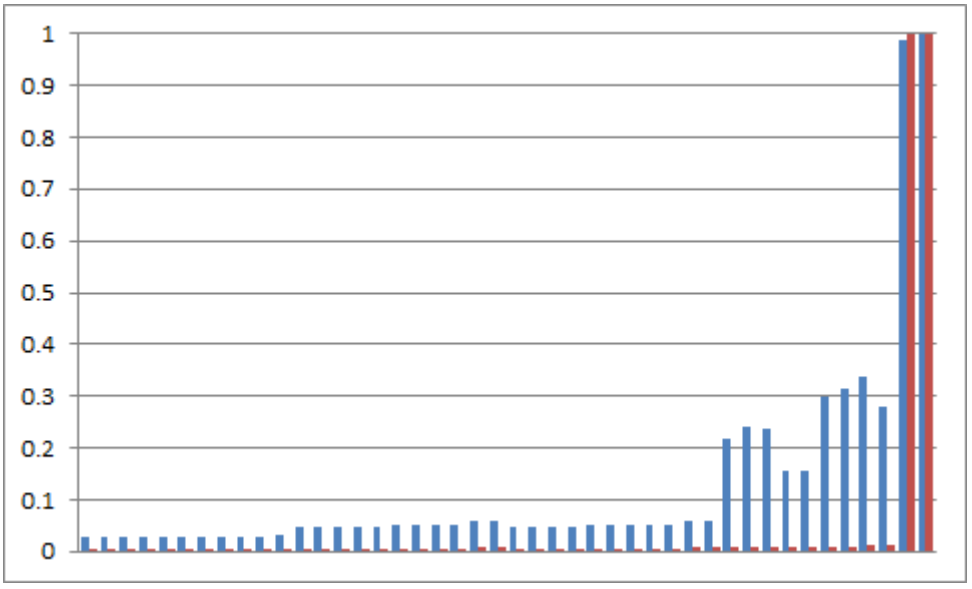


FIGURE 8.12: Relative performance of 44 Histogram implementations

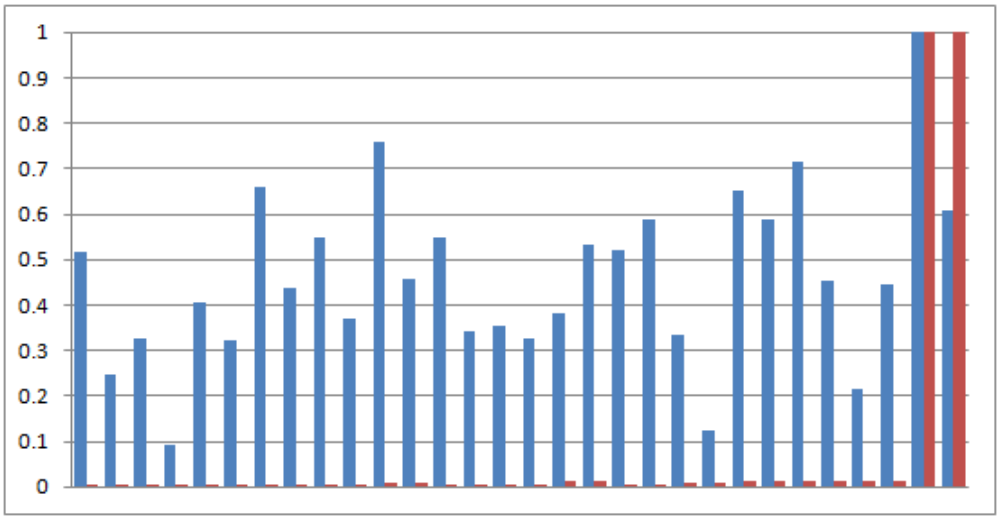


FIGURE 8.13: Relative performance of 30 Kmeans implementations

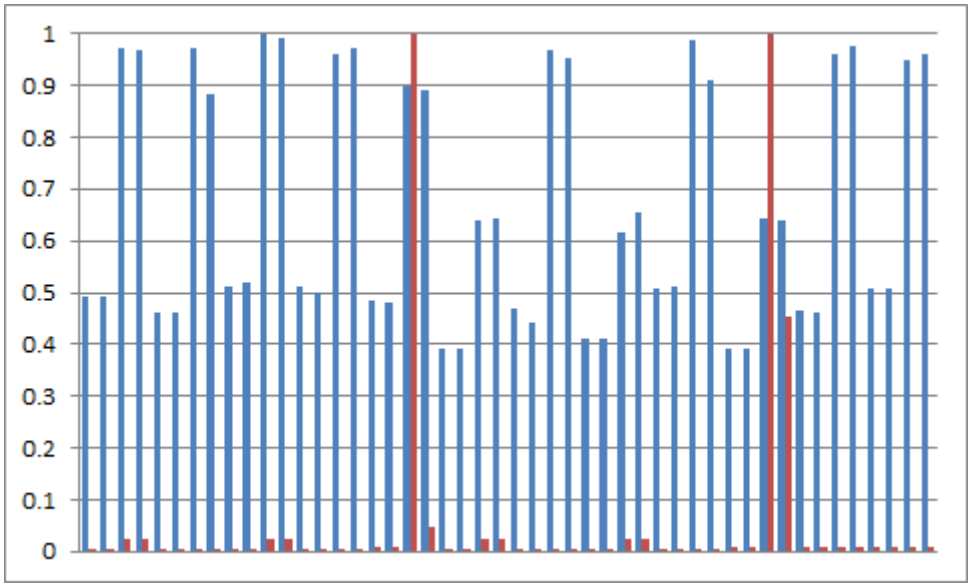


FIGURE 8.14: Relative performance of 48 Mandelbrot set implementations

overheads. The difference between the runtimes of the best and worst solutions is more than an order of magnitude (11×) for `kmkernel`, and almost two orders of magnitude for `hist` (39×) and `rmat` (57×), which is due to the fact that radically different data distributions and layouts are being explored. What is surprising is that this difference is still large (2.6×) for `mandel`, which only varies the way it distributes the (x, y) plane, showing that this kind of search can improve solutions even for seemingly straightforward examples.

For `kmkernel` and `hist`, the best solution is closely aligned with the minimum redistribution cost. Here using sub-optimal combinator implementations which have more general DDL types gives better performance than inserting redistributions to allow more efficient combinator implementations to be used. For `hist`, the best solution does an exaustive

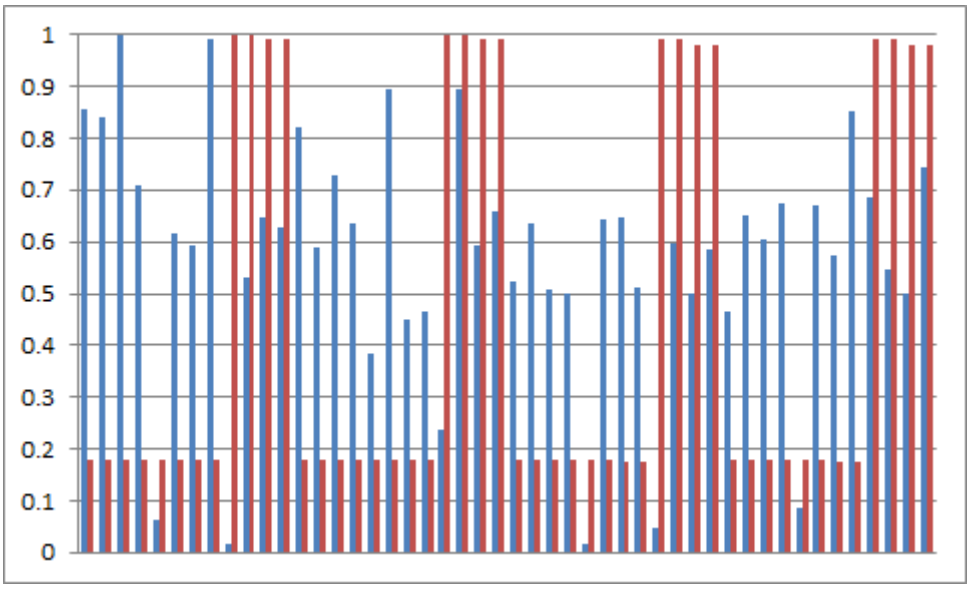


FIGURE 8.15: Relative performance of 48 R-mat implementations

search (less efficient) to find `minV` and `maxV`, and the worst, re-partitions and sorts `D` by value to make `minV` and `maxV` the first and last elements. The best also group-reduces `D'` using a hash-map and inter-node communication, whilst the worst re-partitions it by value and then group-reduces in-place. Similarly, the best `kmkernel` solution partitions `points`, mirrors `clusters`, and then performs the group-reduces using hash-maps and inter-node communication, rather than re-distributing so that this communication can be avoided. This shows us that the fastest solutions can be those that use less efficient DDLs and distributed algorithms to avoid expensive redistributions. Therefore any search heuristic that gives unequal preference to the fastest and most specialized combinator implementations, may miss optimal solutions, or take a long time to find them.

For `mandel` the best solutions (having speedups between 0.9 and 1), all distribute the x-axis of `axes` using hash partitioning, rather than range partitioning. Here, the `escape_time` algorithm takes much longer for points inside the set than those outside, and partitioning into contiguous blocks concentrates these more computationally intensive points in certain partitions. Hash partitioning on the other hand balances the load better. This is an effect that could not easily be predicted by a static cost-estimate, making this a good example for the need for using performance-feedback in the search. `rmat` also shows the opposite effect to `hist` and `kmkernel`. Here, the best solution repartitions and sorts `E'` by its edge value, allowing the group-reduce to proceed in-place reducing one group at a time, and the union to proceed in-place as well. For both `mandel` and `rmat` the minimum redistribution cost does not correlate with the best performance, and the maximum cost does not correlate with the worst performance. In fact, the minimum redistribution cost (tallest red bar) for `rmat` corresponds to one of the two worst solutions in practice. This lack of correlation between redistribution cost

Program	Average speedups			% of solutions that are faster	
	1^{st}	2^{nd}	Increase for 1^{st}	1^{st}	2^{nd}
`hist`	0.1338	0.1245	6.93%	50%	50%
`kmkernel`	0.52906	0.3954	25.3%	83.3%	16.6%
`mandel`	0.6741	0.6673	1.01%	58.8%	41.6 %
`rmat`	0.4809	0.6689	-28.1%	33.3%	66.6%
Overall	-	-	1.29%	56.3%	43.7%

TABLE 8.2: Comparison between different redisribution insertion solutions

and real performance, shows that in some cases the fastest solutions redistribute their data, and that different distributions that both use redistribution functions can have drastically different performance characteristics.

Redistribution variants For the four example programs, we used the redistribution insertion technique described in Section 5.2 to synthesize two plans for every combination of combinator implementations. For each combination, these two plans were chosen by prioritizing plans with lower redistribution cost estimates, such that the first plan's cost is less than or equal to the second's. To discover whether or not only using the plan with the lowest redistribution cost is sufficent, or conversely whether more variants should be considered, we have calculated the average speedups for the first and second variants, and how often the first variant's performance exceeded the second's. These values are shown in Table 8.2.

For `kmkernel` the first variants are considerabley faster than the second, but this is not true in general. The first variants for `hist` and `mandel` only have slightly better speedups on average, and are better only slightly more often than they are worse. Furthermore, for `rmat` the first variants are 28% slower on average, and are slower 66% of the time. Overall, second variants give better performance nearly half (43.7%) of the time. This means that we certainly cannot only consider the variant with the lowest redistribution cost estimate, and suggests that it may be worthwhile considering more than two variants per combination in future implementations.

8.3.3 Search algorithms

In this section we use the cached runtimes for the implementations of our four example programs, to compare the efficacy of different search algorithms. We ran the search loop for different search heuristics, trying different combinations and values of the search parameters discussed in Section 7.5, giving us 946 different heuristics. We ran the

TABLE 8.3: Fastest search heuristics to find a good solution, sorted by percentage of total runtime elapsed before finding the ultimate solution.

Search algorithm	To solution	
	% solutions explored	% runtime
randomChanges2(1par,4chi,1mut)	26.2%	17.8%
geneticVisitor1(5)	24.6%	18.5%
randomChanges4(1par,3chi,1mut)	27.8%	19.0%
...	...	...
randomChanges2(1par,4chi,1mut)	26.2%	17.8%
geneticVisitor1(5)	24.6%	18.5%
geneticVisitor1(4)	24.6%	18.7%
randomChanges4(1par,3chi,1mut)	27.8%	19.0%
geneticVisitor1(6)	25.2%	19.0%

non-deterministic heuristics 500 times each to produce the results and compared the effectiveness (e.g., convergence speed) of the different heuristics on these examples.

Convergence speed The first measure we use to compare the different search heuristics is how long it takes for them to find the best solution. We measured this by calculating the percentage of the total runtime (i.e., the sum of the runtimes for all the candidate implementations), which we know from the exhaustive search, elapsed before the *best* solution was found, and taking the average over all runs of each search heuristic for all four example programs. The percentages of the total runtimes are listed in the third column (% runtime) of Table 8.3. The second column (% solutions explored) lists the percentage of the overall search space explored before finding the search's final solution. Table 8.3 lists the fastest algorithms to converge to a solution, i.e., the algorithms with the lowest percentage runtime elapsed before finding a solution, on average. The table is sorted by percentage of total runtime expended before finding the best solution, where the first three lines (before the ...) show algorithms which found the global optimum at least 90% of the time, and lines 4–8 show algorithms that found a solution whose performance was within 10% of the fastest runtime on average. Here, randomChanges(x par, y chi, z mut) is an genetic search without crossover that generates y offspring from the x best parents, by applying z random mutations to each. geneticVisitor(k) is a genetic search with crossover, that generates a population of k children from randomly chosen pairs of parents drawn from the best third of the current population, and mutates a given gene (i.e., combinator choice) with probability of 0.2.

The best algorithms here were genetic searches, with a constant runtime pruning timeout. The genetic search without crossover with a single parent generating 4 children, and a single mutation applied to each, was fastest to converge to a good solution, finding

TABLE 8.4: Fastest search heuristics to terminate after finding a good solution.

Search algorithm	Termination condition	To termination	
		% sols	% runtime
geneticVisitor1(5)	searchTilHalfTried	51.5%	36.7%
randomChanges2(1par,4chi,1mut)	tilNoneNew	56.9%	38.6%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=4,th=0.2)	56.6%	38.9%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=3,th=0.1)	57.4%	39.0%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=2,th=0.2)	57.5%	39.1%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=4,th=0.1)	57.3%	39.1%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=2,th=0.1)	57.4%	39.1%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=3,th=0.2)	57.4%	39.3%
geneticVisitor1(7)	searchTilSameCosts(n=4,th=0.2)	67.2%	53.0%
...	...	...	...
geneticVisitor1(4)	searchTilHalfTried	51.4%	36.7%
geneticVisitor1(5)	searchTilHalfTried	51.5%	36.7%
geneticVisitor1(6)	searchTilHalfTried	51.5%	37.0%
geneticVisitor1(8)	searchTilHalfTried	51.5%	37.7%
geneticVisitor1(7)	searchTilHalfTried	51.5%	37.8%
geneticVisitor1(9)	searchTilHalfTried	51.5%	37.9%
randomChanges2(1par,4chi,1mut)	searchTilSameCosts(n=2,th=0.2)	57.5%	39.1%
reverseDepthFirstBwd	searchTilHalfTried	51.5%	48.6%
randomChanges6(1par,3chi,2mut)	searchTilSameCosts(n=4,th=0.1)	49.9%	49.2%

its best solution after 17.8% of the total runtime. The second fastest to converge was the genetic search with crossover and population sizes of 4, 5 or 6, finding their solution after 18.5% to 19.0% of the total runtime. These differences in convergence speed are too small to be conclusive, but certainly indicate the genetic searches are good candidates for our performance-feedback-based search. Reverse depth first searches also performed well in our experiments, but this is an anomaly caused by the fact that `hist` and `kmkernel`'s best solutions happened to be within the last two points in the search space. Such searches are not expected to perform well in general. Note that greedy searches almost never found the optimum, returning solutions which were on average three times the cost of the optimum.

Termination Table 8.4 shows the search heuristics that were the fastest to terminate after finding their best solution. As in Table 8.3, the lines 1–9 (before the ...) list heuristics that found the global optimum at least 90% of the time, and lines 10–18 list heuristics that found solutions within 10% of the optimal performance on aver-

age. searchTilHalfTried searches half the search space, tilNoneNew searches until the population stops changing (the same solutions keep being visited, i.e., the number of visits exceeds 5× the search space size), and searchTilSameCosts(n=n, th=t) terminates when the last n solutions visited are within $\pm t$ of the last solution (i.e., when the search gradient levels off).

The most successful termination conditions for the good genetic searches either waited for the search gradient to reduce until the k most recent ($k = 2, 3, 4$) candidates had runtimes within 20% of each other (for searches without crossover), or searched half the state space (for searches with crossover). These successfully detected that they had found the global optimum over 90% of the time (resp. found solutions within 10% of the optimum performance) after 36.7-39.3% of the total runtime. This is 2.5× faster than an exhaustive search, but is still twice the time required for the search algorithm to actually find the best solution. This illustrates the difficulty in accurately recognizing when a search has found the global optimum, and suggests that terminating the search after a fixed budget (e.g., elapsed time) has been expended, may be the best approach.

Pruning Search heuristics that used the re-distribution cost estimates to prune candidate solutions performed poorly and failed to find good solutions in general. This is not surprising as Figure 8.14 and Figure 8.15 show that for `mandel` and `rmat` these costs are poor predictors of true performance, since for some of the implementations of these programs data redistribution time is amortized by later data access. Furthermore, the fact that a given set of weights works precisely for some examples (i.e., `hist` and `kmkernel`), and is actually counterproductive for another, means we are unlikely to find a good general setting.

Runtime pruning Runtime pruning is where we force an executing candidate implementation to terminate, after some maximum duration, before it has finished running. We tried different ways of choosing this maximum duration, and varying it during the search. These were timing out runs after the current best runtime, twice the best current runtime, or after some fixed maximum runtime.

In general the searches that just used a constant cutoff timeout worked best. The best of these found the global optimum almost always. Timing out runs after the current, or twice the current, best runtime, threw away too much information to find good solutions, finding the global optimum only very rarely.

Summary In this section we have demonstrated that our compiler implementation's performance-feedback-based searches can automatically generate DDL plans and corresponding implementations for Flocc input programs. We have also seen that such

searches should consider plan variants that use redistributions in different ways, and that the best search heuristics are genetic searches, with a fixed termination budget, and constant runtime pruning.

8.4 Capabilities

In this section we evaluate some of the conceptual merits of our approach. In Section 8.4.1 we evaluate the conceptual flexibility of our framework on the language level, by showing how the data-parallel features of programming languages from three programming models, can all be mapped onto Flocc combinators. In Section 8.4.2 we present a similar comparison, that shows the flexibility of our framework with regard to the data distribution it supports. Finally, in Section 8.4.3 we discuss some of the conceptual benefits and limitations of our approach.

8.4.1 Language comparison

In this section we compare Flocc's data-parallel features to those of other programming languages. Details of this comparison are listed in Appendix E, which shows for each comparison language, how its data-parallel features can be mapped onto Flocc combinators (which are listed in Appendix B) in a straightforward way.

The first languages we compare with are Fortran 90 [3] and HPF [135]. Fortran 90 and HPF both support array-based data-parallelism. Fortran 90 extends Fortran 77 [32] with array-sections, reductions, parallel-prefixes, and other array manipulation functions, that can all execute in parallel. HPF extends Fortran 90 with additional array manipulation functions, including parallel suffix functions, with a data-parallel `FORALL` loop construct, and with data distribution directives. We compare HPF's data distribution directives to our DDL types in the next section. The remaining features can all be mapped onto Flocc combinators, apart from `FORALL` loops whose array references are not all index translations, scalings, or reflections. More combinators would be required to implement other array index transformations. ZPL [132] is another language for array-based data-parallelism. Its features closely resemble those of Fortran 90, and can all be mapped onto Flocc combinators.

Next we compare Flocc to two map-based programming models, MapReduce [66] and relational algebra [58]. MapReduce jobs consist of two parallel operations, a *map* which maps disk-backed input data into key-value pairs, and a *reduce* which aggregates sorted lists of these values grouped by key. These operations can be mapped onto `map` and `groupReduce`, where the group's reduction function uses `concatList` and `sortList`. Relational algebra consists of standard set operations (e.g., union and intersection), projection (π), selection (σ), column renaming (ρ), and various joins (e.g., natural joins

and equi-joins). These can be supplemented with grouped aggregations and sorting. All of these operations can be mapped onto Flocc combinators, and since these operations are the basis of SQL [45], most SQL queries should be expressible in Flocc as well.

Finally, we compare Flocc to two programming languages which support list-based data-parallelism, Data-Parallel Haskell (DPH) [159] and SISAL [82]. DPH supports parallel versions of a large number of the standard combinators for lists. SISAL supports 1D vectors (which can be considered lists or arrays), and various operations on them including reductions and concatenations, etc. It also features a data-parallel `for` loop that iterates over integer ranges, accessing and aggregating vectors. The features can also all be mapped onto Flocc's list combinators.

Flocc's combinators are therefore as expressive as Fortran 90, MapReduce, relational algebra, DPH, SISAL, and the majority of HPF. The only limitation is that some of the arbitrary combinations of integer array references possible in HPF's `FORALL` loop are not currently possible in Flocc. This demonstrates the flexibility of our approach, in that three usually disjoint data-parallel programming models can all be expressed in our language, and handled by our approach.

8.4.2 Data distributions supported

In this section we compare the data distributions supported by MySQL 5 [147], HPF [135], SISAL [82], and DPH [159], to those supported by DDL types in Flocc. The details of this comparison are listed in Table 8.16.

MySQL 5 features an SQL query planner that supports distributed schemas. Possible schema distributions include hash, key, range, and list partitioning, replication, and sub-partitioning. These must be specified manually by the database designer. DDL types support replication, and hash, key, list, and range partitioning, although the ranges and values for Flocc's range and list partitionings are chosen automatically at runtime, rather than manually. `LINEAR HASH` partitioning is not currently supported, although could easily be added as an additional partition mode, and sub-partitioning using a different scheme is not yet supported, although could be added by changing `DMap`'s partition modes, to a tuple of modes.

HPF's data distribution directives allow array distributions to be specified by the programmer. The `Align` directive specifies how array indices should be aligned, and the `Distribute` directive specifies how to map array indices onto node topologies (i.e., cyclic, block, block(n), and *). The `DArr` types in Chapter 4 are sufficient to implement `Align` directives for simple alignment, axis-collapse, axis-transpose, and replication. The extended `DArr` DDL type in Chapter 6 also supports `Align` offsets and axis-reversal, via the offset and direction DDL type parameters respectively, and in addition support ghosted fringes, which is not supported explicitly by HPF, though can be implemented internally

by HPF compilers. This extended type also supports all the possible `Distribute` directive modes (i.e., cyclic, block, block(n), and *), and redistributions can all be performed via Flocc redistribution functions. These extended `DArr` types can therefore express all of the array distributions possible in HPF.

Finally, SISAL is the only functional language we are aware of with a compiler that targets distributed memory architectures [178]. This compiler partitions `Forall` loops into blocks, and is therefore equivalent to a `DArr` or `DList` type with blocked partitioning. DPH only supports multicores but we include it to provide another list-based comparison. DPH supports blocked list partitioning, which is also supported by our `DList` types, however it can also flatten nested lists for auto-vectorization— a feature we do not currently support. In addition to blocked list distributions, the `DList` type also supports cyclic list distributions, which is an efficient distribution mode for zipping distributed lists etc.

Flocc's DDL types can therefore express the majority of data distributions possible in MySQL, HPF, and SISAL, as well as some additional ones. This again illustrates the flexibility of our approach, since our automatic distributed data layout mechanism is able to find data layouts for three diverse application domains, in a single framework.

Language	Collection	Distribution feature	Sub-feature	Supported	Using
MySQL	Map	Partitioning	HASH	Yes	Hash mode
			LINEAR HASH	Future	LinearHash mode
			KEY	Yes	Hash mode
			RANGE	Yes	DynRange mode
			LIST	Yes	DynDiscrete
		Sub-partitioning		Future	Tuples of modes
		Replicas		Yes	Mirror dimension set
HPF	Array	Align directive	Simple	Yes	Partition function
			Offset	Yes	Extended: offset param
			Axis-collapse	Yes	Partition function (subset)
			Axis-transpose	Yes	Partition function (permute)
			Axis-reversal	Yes	Extended: direction param
			Replication	Yes	Mirror dimension set
		Distribute directive	Cyclic	Yes	Extended: blocksize=1
			Block	Yes	Extended: blocksize=0
			Block(n)	Yes	Extended: blocksize=n
			*	Yes	Extended: blocksize=MAX_INT
		Re-align		Yes	Redistribution functions
		Re-distribute		Yes	Redistribution functions
SISAL	List	Forall partitioning		Yes	Partitoned crossList mapList intRangeList eqJoinList
DPH	List	Shared mem partitions	splitD	Yes	Blk mode
			joinD	Yes	Blk mode
			mapD	Yes	Blk mode
		Vectorization		No	Doesn't flatten nested lists

FIGURE 8.16: Comparison of DDL types to the data distribution features of other languages.

8.4.3 Conceptual benefits of approach

In this section we discuss the key conceptual benefits and limitations of our approach. Table 8.17 supplements this discussion, by showing how the main data-parallel programming paradigms mentioned in Chapter 2, compare to Flocc along several key criteria. These criteria are, whether distributed-memory architectures are supported, whether the model allows data distributions to be customized, whether this customization can be automated, what collections and application areas are supported, and whether the language can be extended with more data-parallel operations, data layouts, and collections. These criteria correspond to the key benefits of our approach. In the rest of this section we elaborate on these benefits, and on the primary limitations.

Distributed-memory support Not all of the languages in Table 8.17 target distributed-memory architectures, and such support is particularly lacking for some of the most high-level and functional programming languages (e.g., SAC, NESL, DPH). Flocc does target such architectures due to its DDL type-inference technique and corresponding back-end.

Customizable data distributions Of those programming languages that do target distributed-memory architectures, two, ZPL and MapReduce, only support a fixed data distribution. In contrast, high-level Flocc programs can be implemented using many different data distributions and communication patterns.

Automatic data distribution optimization Half (i.e., six) of those frameworks that support customizable distributed data layouts for distributed-memory architectures, only support this customization manually. The other six (excluding Flocc) support some sort of automatic distributed data layout optimization, although this varies in quality. The combination of DDL type inference, automatic redistribution insertion, and performance-feedback-based search, used in this thesis can automatically synthesize and optimize DDLs for data-parallel programs without intervention. Furthermore, the use of a performance-feedback-based search gives our approach the potential to find fast DDL solutions, that may seem obscure and unexpected, since it is guided by a true empirical reflection of the system's performance and the program's behavior, rather than simplified models of these.

Collections and application areas supported Apart from the technical and theoretic novelty of our DDL-type-based code synthesis technique, the main criteria that distinguishes it from other similar approaches, is the range of collections and application areas it supports. The majority of data-parallel programming languages only support a single collection type and related operations. The only exceptions are LINQ-based

languages (i.e., PLINQ and DryadLINQ), SISAL, and Chapel, and LINQ only supports inefficient map-based operations on lists, and SISAL only supports vectors, which could be viewed as arrays or lists. Chapel is therefore the only other language that really supports multiple collections, and these do not include lists or disk-backed collections. Furthermore, Chapel requires manual data distribution selection and optimization. Flocc and the distributed-memory implementation synthesis technique described in this thesis, is therefore unique in its support for maps, lists, and arrays. This support is both conceptually interesting, and practically beneficial.

Extensibility The final key conceptual benefit of our framework, is its extensibility. None of the languages in Table 8.17 can be extended with additional data-parallel constructs or combinators apart from DPH. In contrast, Flocc can be extended with additional combinators by just adding code-generation templates. Furthermore, unlike any of the other approaches, we can add additional collection types and DDL types to our framework, without modifying the underlying implementation. Finally, by using a performance-feedback-based optimization strategy, we free ourselves from the need to implement or extend internal cost-models to reflect new architectures or collection types (apart from the simple redistribution cost estimates). Overall, our approach therefore shows much greater potential for extensibility than existing languages.

Synthesis speed Although using a performance-feedback-based search aids portability and allows unusual global optimums to be found, it is significantly slower than traditional compilers. This is unavoidable, since by definition a performance-feedback-based search must invoke a traditional compiler multiple times, to compile candidate implementations. However, we do not see this as a serious flaw, since similar approaches have experienced success in other areas, and since though slower than simple compilation, search times will still be significantly less than the time required to implement even one solution manually, let-alone find the fastest. Furthermore, unlike traditional compilers, the performance-feedback-based search requires that the programmer provides some suitable input data, so that the search can be performed.

Simplicity of back-end Another difficulty with our approach is the need to generate imperative C++ code from a functional programming language. This problem is not critical, since we have managed to generate code that rivals the performance of manually coded C++ equivalents, and since other functional languages like SISAL and SAC have shown very impressive performances. However, it is less straightforward than a simple syntactic translation, and issues like copy-avoidance, collection-preallocation, and memory de-allocation, must all be addressed.

In this section we have briefly outlined the key conceptual benefits and limitations of our distributed-memory implementation synthesis technique. Overall, our approach is one of very few systems that target distributed-memory architectures and synthesizes appropriate DDLs automatically, and is unique in its flexibility and unified support for three different distributed collection types and associated operators.

Approach	Dist mem	Dists	Dist selection	Maps	Lists	Arrays	Mem	Disk	Extensible
Flocc	✓	Many	Automatic (compile time)	✓	✓	✓	✓	Future	✓
Loop-par	✓	Many	Automatic			✓	✓		
DryadLINQ	✓	Many	Automatic	Inefficient	✓		✓	✓	
Array-sections	✓	Many	Automatic			✓	✓		
Pig Latin	✓	Many	Automatic	✓				✓	
SQL	✓	Many	Automatic (schema manual)	✓			✓	✓	
SISAL	✓	Dynamic	Automatic		✓	1D	✓		
ZPL	✓	Fixed	Fixed			✓	✓		
MapReduce	✓	Fixed	Fixed	✓			*	✓	
Chapel	✓	Many	Manual	✓		✓	✓		
X10	✓	Many	Manual			✓	✓		
UPC	✓	Some	Manual			✓	✓		
HPF	✓	Many	Manual (automatic tool)			✓	✓		
Co-Array Fortran	✓	Manual	Manual (hard coded)			✓	✓		
Titanium	✓	Manual	Manual (hard coded)	Manual	Manual	Manual	✓		?
SAC		N/A	N/A			✓	✓		
PLINQ		N/A	N/A	Inefficient	✓		✓		
PQL		N/A	N/A	✓	?	?	✓		
NESL		N/A	N/A		✓	1D	✓		
DPH		N/A	N/A		✓	1D	✓		✓

Figure 8.17: Comparison between Flocc and related approaches.

8.5 Concluding remarks

In this chapter we have presented some experimental and conceptual evaluation of our new approach to automatically synthesize distributed-memory (cluster) implementations of data-parallel programs. We have used our Flocc compiler proof-of-concept to generate MPI implementations of several example programs in C++, and seen that they perform better than PLINQ equivalents, and close to (i.e., within a factor of two of) the performance of hand-coded MPI versions. We have also successfully used our compiler to search for and generate, implementations of four larger map-based programs, out of 170 automatically generated candidate programs. This has demonstrated that our performance-feedback-based search works in practice for some useful example programs. It has also shown that genetic search heuristics are very promising at reducing the overall duration of the performance-feedback-based searches, reducing the total runtime spent to just below 18% of the total runtime required to search the whole state-space, on average. The main limitations of this experimental evaluation are that these larger examples are all map-based, and we do not investigate how the specific input data used effects which solution is returned.

In Section 8.4, we have evaluated the capabilities of our approach. This includes showing that almost all the data-parallel features of six programming languages, from three traditionally disjoint programming models, can be mapped straightforwardly onto Flocc combinators, and that almost all their respective data-distributions can be mapped onto Flocc DDL types. Finally, in Section 8.4.3 we have analyzed the key conceptual benefits, which are most notably the automation of DDL choices and code generation, and doing this for a programming model that spans three usually disjoint data-parallel programming models, and shows significant potential for extensibility.

In the next chapter (cf. Chapter 9) we conclude this thesis, and suggest some directions for future work.

Chapter 9

Conclusion and Future work

In this thesis we have presented a new technique to automatically synthesize distributed-memory implementations of data-parallel programs. This technique takes a data-parallel program, automatically generates possible distribution plans for it, and generates implementations from these plans to run on a cluster. This technique works on programs written in a core functional language called Flocc that supports multiple collection types, in particular maps, arrays, and lists. It encodes information about the data distributions of various data-parallel combinators in dependent types, and uses type inference to automatically infer data distribution information for different concrete plans of these programs from these types. It then uses a performance-feedback-based search, to search for optimal plans, and generates code in C++ using MPI to implement them on the cluster. This chapter summaries our achievements and directions for further work.

9.1 Main contributions

Our main contribution is a technique for automatic synthesis of distributed memory implementations of high-level data-parallel programs that uses dependent types and supports *multiple collection types*, arrays, maps, and lists. Our key insight is that distributed data layout information can be embedded in types, and recovered using a type inference algorithm, in a way that works for multiple collection types, and is thus much more general than existing approaches. This overall contribution involves several subsidiary contributions, which collectively fulfill the research objectives in Section 1.2.

Data distributions and layout inference, as types and type inference We believe that our work is the first to formalize distributed-memory data layouts as *types*. In Chapter 4 and Chapter 6 we have demonstrated that both local and distributed data layouts of maps, arrays, and lists can be encoded as dependent types. We have also

demonstrated in Chapter 4 and Chapter 6 that these types, which use embedded functions and parametric polymorphism, can capture quite nuanced relationships between data layouts, that are sufficient to express the data distribution behaviors of many distributed-memory combinator implementations. In fact, in Section 8.4.2 we show that our extended DDL types express all of the data distributions that are possible in HPF, and most of those possible in MySQL. Chapter 6 and our implementation also demonstrate that this approach works for concrete data structure selection and local data layouts as well. This work fulfills research objective 2.

Then in Chapter 5 we present a novel type inference algorithm (extending standard Damas-Milner) that uses these types to automatically infer data layout information about candidate programs. We believe that we are the first to show that distributed-memory data layouts can be inferred for functional programs using such an algorithm. Chapter 6 also shows how our type inference algorithm can be extended to support more complex constraints between functions, by leveraging a theorem prover and using it to embed various equational theories within our unification algorithm. This fulfills research objective 3.

Improved expressiveness and collection support To our knowledge, ours is currently the only high-level approach to program clusters that supports multiple collection types, specifically maps, arrays, and lists. We have demonstrated this via the types and example derivations in Chapter 4 and Chapter 6, and using our implementation. We support these collections by encoding the data layout information about them using types, and then automatically inferring layout information for programs using these types. We have also found in Section 8.4.1 that Flocc is as expressive as ZPL, relational algebra (and therefore SQL queries), MapReduce, and DPH. Furthermore, it can express all the data-parallel features of Fortran 90 and HPF, apart from `Forall` loops with array references other than translations, reflections, and scalings. This fulfills research objective 1. The approach is also extensible, and we have shown in Chapter 6 that extra type parameters can be added to encode more distribution and layout information, in various useful ways.

Automatic redistribution insertion In addition to finding combinations of combinator implementations that yield valid DDL types (and therefore data layouts), we have devised an algorithm to automatically insert redistribution functions at suitable points in candidate programs. The algorithm is presented in Section 5.2 and used in the experiments discussed in Section 8.3. This allows our approach to synthesize implementations that transform the data distributions, data structures, and local layouts of collections during execution, such that more efficient algorithms can be used, without having to consider all possible redistribution insertions. This works by augmenting our type inference algorithm to identify locations in programs that break the types, and

then synthesizing appropriate chains of re-distribution and re-layout functions to fix these types. This means that we only consider data redistributions when they could be beneficial by allowing us to use faster combinator implementations with more restrictive types. This fulfills research objective 4.

Implementation The final contribution is the implementation and evaluation of our proof-of-concept compiler. This corresponds to research objective 6.

Firstly, to generate MPI implementations from Flocc plans, we have developed a code generation mechanism which we describe in Chapter 7. We have implemented this technique in the back-end of our compiler, and use it in the experiments listed in Chapter 8. To our knowledge this is the first code generation technique to use meta-data inferred by type inference (other than basic data types) to parameterize code generation templates (cf. Section 7.4.6). This fulfills research objective 5, and contributes to objective 6.

Secondly, although we do not consider Flocc to be a contribution per se, it is novel, and we believe that the combination of features that allow us to generate efficient standalone imperative code from a (semi-)functional language is unique (cf. Chapter 3 and Section 7.4). For example, Flocc supports the definition of higher-order functions with polymorphic types, but requires that function parameters to such functions are statically resolvable. This constraint allows function parameters to be lifted into the types during type inference, and allows much faster implementations to be generated, since it means that implementations can use basic data types and inlined operations rather than pointers and reduction engines. The definition of Flocc (cf Chapter 3) and its combinators for maps, arrays, and lists (cf Appendix B), fulfills research objective 1, and the implementation of the corresponding compiler front-end, contributes to objective 6.

Thirdly we have implemented a performance-feedback search within the compiler, which generates, compiles, and executes multiple candidate solutions, to find the one with the best performance. It includes implementations of our DDL type inference algorithm, redistribution insertion algorithm, and various search algorithms. This implementation uses a performance-feedback-based code synthesis search in a new context (cf. Section 7.5). Although the benefits of performance-feedback-based auto-tuning for certain classes of algorithms are well known [20, 87, 166, 209], we are the first to apply such techniques to the data distributions and layouts of programs designed for clusters. Furthermore, we apply it to a much more expressive input language than most current work. The implementation of this search-based compiler, fulfills the implementation part of research objective 6. This shows that it is possible to build a tool that uses the techniques described in this thesis to automatically generate distributed-memory implementations of data-parallel programs written in a functional language.

Finally, in Chapter 8 we present some experimental and conceptual evaluation which shows that we have fulfilled research objective 6 (i.e., that our prototype compiler works

in practice). In Section 8.2 we show that it is possible to generate code that gives similar performance to straightforward MPI equivalents, of several example programs. Then in Section 8.3 we use the full implementation to synthesize cluster implementations of four map-based example programs. This demonstrates that our performance-feedback-based search technique works for some useful example programs. The `mandel` program in particular yielded an implementation with better load balancing than a naïve implementation, that would not have been found using static cost estimates. We also evaluated different search heuristics using these four example programs, and found that genetic searches worked well, converging to the optimal (or near-optimal) solutions in just under 18% of the total runtime, on average. Additional experimental evaluation could be performed to synthesize implementations for non map-based programs, and investigate how the specific input data effects the solution chosen, but these experiments are sufficient to show that our overall code synthesis approach works in practice. In Section 8.4 we evaluate the conceptual capabilities of our approach, by comparing the input language and DDL types to other data-parallel languages and their data distributions, and list the key benefits and limitations of our approach.

These contributions achieve all of the research objectives listed in Section 1.2. Furthermore, we have done this in a way that successfully fulfills the criteria listed in our problem statement (cf. Section 1.1). That is, our approach abstracts away from manual communication, targets distributed-memory architectures (i.e., clusters), automatically selects data layouts, supports multiple collections, and shows potential for extensibility, whilst maintaining reasonable runtime performance. The main caveat is that although our overall approach works for multiple collection types (objective 1), our current prototype back-end's has limited support for array-based combinator implementations. This still fulfills objective 6, but slightly reduces its scope. The future work directions in the next section show how we could improve our implementation, and strengthen these claims.

9.2 Future work directions

There are several interesting directions for further work. This section describes some of these, and their possible outcomes.

Full implementation of array support In order to get an full end-to-end compiler prototype working, we have restricted our back-end to only implement map and list combinator templates, apart from the array-based ones needed for our dense matrix multiply. Implementing more array templates would allow us to extend our evaluation to include array-based input programs as well as those based on maps and lists.

Faster redistribution insertion One limitation of our current implementation is the time complexity of its redistribution insertion algorithm. Our prototype currently uses Algorithm 1 in Figure 5.12, with time complexity exponential in the number of combinator applications. We have presented an improved redistribution insertion algorithm, Algorithm 2 in Figure 5.13, and future work would implement this algorithm in our compiler. This should allow our technique to scale to handle programs with more combinator applications.

Further implementation improvements After improving our prototype's redistribution insertion algorithm and array support, there are a number of other extensions that would improve its search speed, and the quality of the implementations it synthesizes. Firstly, the implementations synthesized for our experiments in Section 8.3 differed in performance by up to 56× so that in order to get 1 second of performance data for the best solution, we need to run the worst solutions for almost 1 minute each. One way to accelerate the search and improve accuracy would therefore be to start the search using a small test data set (or problem size), until the candidates reach a certain mean performance, at which point a larger data set (or problem size) could be used.

Then, to improve the quality of synthesized implementations, and especially array-based ones, we could extend our prototype to support the extended DDL types in Chapter 6. The compiler already supports local data layouts, but does not yet support extended array distributions (cf. Section 6.3) or more flexible DDL types (cf. Section 6.2). To support these we would also have to change our type inference algorithm to use the equational theories in Section 6.4 by calling the E-theorem prover to unify equations involving functions.

To improve the compilers flexibility and extensibility, we could change how our back-end templates are declared, to use a scripting language rather than native Haskell, so that adding templates does not require a fresh build of the compiler. This should be relatively straightforward since the current templates just call monadic actions which could still be called by the interpreter of such a scripting language. Furthermore, we could extend Flocc's syntax, so that new DDL types, combinators, and replacement rules could all be defined in Flocc programs, rather than separate configuration files.

A final way to improve the quality of the generated implementations, would be to implement various back-end optimizations. This would include using a better data deallocation technique instead of our current reference counting policy, and performing copy-avoidance (like SISAL uses [82]) to remove unnecessary allocations and ideally make some changes in-place rather than by duplication.

Other architectures Our work so far has focused on supporting clusters, but we believe that the technique presented in this thesis could be extended to work with

other parallel architectures. In particular, it would be interesting to see whether we could implement support for multi-cores and GPUs, and especially whether we could implement them together (like our local layouts and distribution layouts) so that clusters with GPU equipped multi-core nodes could be supported. This would involve extending our DDL types and back-end templates to support these architectures, but should not require any change to our DDL type inference algorithm or search algorithm.

Static cost estimates Our current prototype relies on performance-feedback to search for good (i.e., fast) distributed-memory implementations of Flocc programs. Another possible approach would be to use static cost estimates to evaluate possible solutions without generating, compiling, and running them. This would be likely to yield poorer solutions, because the performance characteristics of modern architectures are very difficult to predict [87, 5] (e.g., the `mandel` example in Section 8.3), but could make the search much faster, and therefore be useful for fast prototyping of data-parallel programs.

Language extensions and optimizations There are many language extensions that could be implemented to improve usability and expressiveness. Firstly, in addition to tuples we could support records and even classes to make Flocc more accessible to object oriented programmers. Support for records could be implemented as a syntactic sugar, which translates records to tuples without changing the core synthesis technique. Otherwise, we would have to investigate extending our front-end and DDL type systems to support records, perhaps via sub-typing.

An alternative research direction would be to see whether we can implement our technique within an existing language and compiler, like Haskell and GHC [196], or C# and Mono [73] (since these already support some kind of type inference). This would make our technique much more accessible and enable it to leverage the features of such existing languages.

Finally, there are various high-level optimizations that could be implemented within Flocc's pre-processor. For example, logical SQL optimizations like pushing down selections (i.e., `filter` applications) could be implemented for array and map combinators. Furthermore, we have considered the possibility of using another type inference phase to perform a data-flow analysis of map combinators, to identify maps with dense integer domains, that could be transformed to use array combinators. This would let programmers program map and array-based programs using a single set of operators, by enabling the compiler to spot where these can be optimized by storing them as arrays.

9.3 Concluding remarks

Distributed memory architectures seem set to become more and more common, and yet the current techniques to programming them often leave many complex decisions to the programmer, that many will be ill-equipped to make. Considerations include how to partition and distribute programs and their data structures to scale efficiently, what concrete communication mechanisms to use, and how to avoid problems like distributed deadlock.

Existing languages for data-parallel programming rarely target distributed-memory architectures, and those that do are restricted to a fixed distribution model (MapReduce), and only support a limited set of operators (SQL/LINQ/HPF). In this thesis we have presented a more general approach, where distributed-memory implementations are automatically synthesized from data-parallel programs written in Flocc, a high-level functional core language. To our knowledge this is the first approach that captures distributed-memory data layout as a typing problem. In particular, we formalized distributed data layouts by polymorphic dependent type schemes and used a type inference algorithm and performance-feedback-based search to search for optimal DDL plans and implementations.

Unlike similar work, our approach supports multiple collection types (i.e., arrays, maps, and lists) and thus works for a wide variety of programs (cf. Section 8.4), and can be extended with more data types, data distributions, and data-parallel operators. Our approach can boost programmer productivity and program reliability through the conciseness of input programs (cf. Figure 8.1), fully automatic generation of distribution plans and code, and the reduced number of possible bugs compared to low level languages (i.e., no pointers/explicit message passing). Finally, initial performance results (cf. Chapter 8) are substantially better than PLINQ, a similar tool for multi-cores, and are close to manual MPI implementations, indicating that the approach is viable in practice.

Appendix A

Matrix multiply implementations

This appendix compares different implementations of a simple dense matrix multiplication. The conventional algebraic form is:

$$C = AB \tag{A.1}$$

$$C_{(r,c)} = \sum_{i=n}^{i=1} A_{(r,i)} B_{(i,c)} \tag{A.2}$$

```
SELECT A.i as i, B.j as j,
       sum (A.v * B.v) as v
FROM A JOIN B ON A.j = B.i
GROUP BY A.i, B.j;
```

(A) SQL

$$R_1 = A \bowtie_{A.j=B.i} B$$
$$R_2 = \rho_{A.v*B.v/v}(R_1)$$
$$C = \mathrm{G}_{\langle A.i,B.j\rangle,\mathrm{sum}(v)}(R_2)$$

(B) Relational algebra

FIGURE A.1: Applicative matrix multiply implementations

```
1 // A row-major, B col-major
2 double A[M][N], B[P][N], C[M][P];
3 for (int i = 0; i < M; i++) {
4     for (int j = 0; j < P; j++) {
5         C[i][j] = 0.0;
6         for (int k = 0; k < N; k++) {
7             C[i][j] += A[i][k] * B[j][k];
8 }}}
```

LISTING A.1: Dense matrix-matrix multiply in C

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>

// global sizes (A: NxM, B: MxP, C: NxP)
const int N = 5000, M = 5000, P = 5000;

int main(int argc, char **argv)
{
  // vars
  int rank, size;
  float *A, *B, *C;
  int r, c, k, rp, cp;
  double pt, jt;

  // init mpi
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  // work out which partition we're in
  int nparts = sqrt(size);
  if (nparts*nparts < size) {
    // only uses a square number of nodes. others unused.
    size = nparts*nparts;
    if (rank >= nparts*nparts) {
      MPI_Finalize();
      exit(0);
    }
  }

  // ap corresponds to the row, bp to the col, in result C
  int anp = nparts, bnp = nparts;
  int ap = rank / anp, bp = rank % bnp;

  // sizes & offsets of local partitions
  int ah, aw, bh, bw, ch, cw;
  int ar, ac, br, bc, cr, cc;
  ah = N/anp; aw = M;
  bh = M; bw = P/bnp;
  ch = ah; cw = bw;
  ar = ap*ah; ac = 0;
  br = 0; bc = bp*bw;
  cr = ar; cc = bc;

  // init matrices
  if (rank == 0) {
    initMatrices(&A, &B, &C);
  } else {
    initMatrixPartitions(&A, ah, aw, &B, bh, bw, &C, ch, cw);
  }

  // create communicators for partitions
  MPI_Comm nullcomm, rowcomms[anp], colcomms[bnp];
  if (rank == 0) {
    // include rank 0 in all communicators
    for (r = 0; r < anp; r++)
      MPI_Comm_split(MPI_COMM_WORLD, r, 0, &rowcomms[r]);
    for (c = 0; c < bnp; c++)
      MPI_Comm_split(MPI_COMM_WORLD, c, 0, &colcomms[c]);
  } else {
    // include this node in the two communicators that
    // for the two partitions it corresponds to
    for (r = 0; r < anp; r++) {
      if (r == ap)
        MPI_Comm_split(MPI_COMM_WORLD, ap,
                  rank, &rowcomms[ap]);
      else
        MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED,
                  rank, &nullcomm);
    }
    for (c = 0; c < bnp; c++) {
      if (c == bp)
        MPI_Comm_split(MPI_COMM_WORLD, bp,
                  rank, &colcomms[bp]);
      else
        MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED,
                  rank, &nullcomm);
    }
  }

  // send & recv partitions
  if (rank == 0) {
    // broadcast from 0 to various consumers
    for (r = 0; r < anp; r++)
      MPI_Bcast(&A[aw*ah*r], ah*aw, MPI_FLOAT,
            0, rowcomms[r]);
    LOG("sent broadcasts of A");
    for (c = 0; c < bnp; c++)
      MPI_Bcast(&B[bh*bw*c], bh*bw, MPI_FLOAT,
            0, colcomms[c]);
    LOG("sent broadcasts of B");
  } else {
    // receive chunk needed
    MPI_Bcast(A, ah*aw, MPI_FLOAT, 0, rowcomms[ap]);
```

```
97      LOG("received broadcast of A");
98      MPI_Bcast(B, bh*bw, MPI_FLOAT, 0, colcomms[bp]);
99      LOG("received broadcast of B");
100   }
101
102   // multiply partitions
103   LOG("multiplying partitions...");
104   TBEGIN(pt);
105   for (pc = C, r = 0; r < ch; r++) for (c = 0; c < cw; c++) {
106     float v = 0.0f;
107     pa = A+(r*M); pb = B+(c*M);
108     for (k = 0; k < M; k++) {
109       v += (*pa) * (*pb);
110       pa++; pb++;
111     }
112     *pc = v;
113     pc++;
114   }
115   TEND(pt, "mul");
116   LOG("multiplied partitions");
117
118   // send & recv partitions
119   int nwrkers = anp*bnp-1;
120   MPI_Request ireq[nwrkers];
121   MPI_Status istat[nwrkers];
122   if (rank == 0) {
123     // start receives from workers
124         for (k = 1; k < anp*bnp; k++) {
125           rp = k / anp; cp = k % bnp;
126           pc = &C[(cw*ch)*((rp*bnp)+cp)];
127           MPI_Irecv(pc, cw*ch, MPI_FLOAT, k, k, MPI_COMM_WORLD, &ireq[k-1]);
128       }
129       // wait on end of receives
130       MPI_Waitall(nwrkers, ireq, istat);
131       LOG("received all partitions of C");
132     } else {
133       // send local part of C to 0
134       MPI_Send(C, cw*ch, MPI_FLOAT, 0, rank, MPI_COMM_WORLD);
135       LOG("sent partition of C");
136     }
137
138     // free matrices
139     free(A);
140     free(B);
141     free(C);
142     LOG("freed arrays");
143
144     // finalize mpi
145     MPI_Finalize();
146     TEND(jt, "job");
147     return 0;
148   }
```

Listing A.2: Matrix multiplication in C and MPI

Appendix B

Flocc library functions

This appendix contains brief descriptions of all the Flocc map, list, and array combinators with their types.

```
-- |map (f,f1,m). Applies f to every entry in m. f must be
-- |a bijection with respect to the map keys, since otherwise-
-- |keys might be duplicated.
map :: ((i,v)->(j,w), Map i v) -> Map j w

-- |mapInv (f,f1,m). Applies f to every entry in m. f must be
-- |a bijection with respect to the map keys, and f1 must be
-- |the inverse of f. f1 is used in data layout planning.
mapInv :: ((i,v)->(j,w), (j,w)->(i,v), Map i v) -> Map j w

-- |cross (m1,m2). Returns the Cartesian product of m1 and m2.
cross :: (Map i v, Map j w) -> Map (i,j) (v,w)

-- |eqJoin (f1,f2,m1,m2). Returns the Cartesian product of m1 and m2
-- |restricted to where the result of f1 applied to m1 equals
-- |the result of f2 applied to m2.
eqJoin :: ((i,v)->k, (j,w)->k, Map i v, Map j w) -> Map (i,j) (v,w)

-- |ltJoin (f1,f2,m1,m2). Returns the Cartesian product of m1 and m2
-- |restricted to where f1 applied to the m1 is lexicographically
-- |before (less than) f2 applied to m2.
ltJoin :: ((i,v)->k, (j,w)->k, Map i v, Map j w) -> Map (i,j) (v,w)

-- |allPairs (f,m). Groups the elements of m into groups using the
-- |keys returned by f, and then returns all pair combinations of
-- |values from each group.
allPairs :: ((i,v)->k, Map i v) -> Map (i,i) (v,v)

-- |reduce (f,g,v0,m). Applies f to each element in m, and then
-- |aggregates these values using g and the null element v0.
reduce :: ((i,v)->s, (s,s)->s, s, Map i v) -> s
```

```
-- |groupReduce (kf,vf,ff,v0,m). Applies kf and vf to every entry
-- |in m, grouping the values returned by vf using the keys returned by kf.
-- |Each group is aggregated using the associative binary operator ff,
-- |and the 0-element v0.
groupReduce :: ((i,v)->j, (i,v)->w, (w,w)->w, Map i v) -> Map j w

-- |filter (pred,m). Returns the entries in m restricted to those
-- |that satisfy the predicate function pred.
filter :: ((i,v)->Bool, Map i v) -> Map i v

-- |union (m1,m2). Returns the left-biased union of m1 and m2.
union :: (Map i v, Map i v) -> Map i v

-- |intersect (m1,m2). Returns the left-biased union of m1 and m2.
-- |i.e. returns the values from m1.
intersect :: (Map i v, Map i w) -> Map i v

-- |diff (m1,m2). Set difference. Returns the entries in m1
-- |whose keys don't occur in m2.
diff :: (Map i v, Map i w) -> Map i v

-- |countMap m. Returns the number of entries in m.
countMap :: Map i v -> Int

-- |intRangeMap (begin,end,stride). Returns a map with keys between
-- |begin and end in steps of stride.
intRangeMap :: (Int,Int,Int) -> Map Int ()

-- |emptyMap (). Returns an empty map of any type.
emptyMap :: () -> Map i v
```

Listing B.1: Map library functions

```
-- |filterList (f,l). Returns list of all elements in
-- |l where f returns True.
filterList :: (v -> Bool, List v) -> List v

-- |expandList (l,v0,mask). Returns a new list with the
-- |same length as mask, where each element is either v0
-- |if that element in mask is False, or is the next
-- |element in l if True.
expandList :: (List v, v, List Bool) -> List v

-- |concatList (l1,l2). Returns a new list with
-- |l2 concatenated after l1.
concatList :: (List v, List v) -> List v

-- |concatLists l. Returns a new list where all the
-- |lists in l have been concatenated one after the other.
concatLists :: List (List v) -> List v

-- |unconcatList (lens,l). Returns a new list where
```

```
-- |the l is decomposed into sublists. The length
-- |of each sublist is in len.
unconcatList :: (List Int, List v) -> List (List v)

-- |transposeList l. Transposes l.
transposeList :: List (List v) -> List (List v)

-- |zip (l1,l2). Returns a new list where
-- |l1 is aligned elementwise with l2. The length
-- |of the result is min(length l1, length l2).
zip :: (List v, List w) -> List (v,w)

-- |unzipList l. Returns (l1,l2) where l1 = mapList fst l
-- |and l2 = mapList snd l.
unzipList :: List (v,w) -> (List v, List w)

-- |mapList (f,l). Returns a new list formed by applying
-- |f to every element of l.
mapList :: (v -> w, List v) -> List w

-- |findInList (f,f0,notFound,l). Applied f to every element
-- |in l, returning the first element that equals f0. If
-- |no element matches then returns notFound.
findInList :: (v -> w, w, (v,w), List v) -> (Int,(v,w))

-- |reduceList (f,v0,l). Aggregates the elements in l
-- |using f, where v0 is the null element. f must be associative.
reduceList :: ((v,v) -> v, v, List v) -> v

-- |prefixList (f,l). Parallel prefix of v using
-- |the binary (associative) operator f.
prefixList :: ((v,v) -> v, List v) -> List v

-- |foldListL (f,v0,l). Performs a fold left to right using
-- |f as the binary operator, and v0 the null element.
foldListL :: ((v,w)->v, w, List w) -> v

-- |subList (begin,end,l). Returns the elements of
-- |l between index begin and end inclusive.
subList :: (Int, Int, List v) -> List v

-- |head l. Returns the first element of l, or fails if l is [].
head :: List v -> v

-- |tail l. Returns all elements of l apart from the first.
tail :: List v -> List v

-- |take (count,l). Returns the first count element of l.
take :: (Int, List v) -> List v

-- |sortList (f,l). Sorts l using the values projected by f.
```

```
sortList :: (v -> w, List v) -> List v

-- |reverseList l. Returns all the elements of l in the opposite order.
reverseList :: List v -> List v

-- |listLength l. Returns the number of elements in l.
length :: List v -> Int

-- |intRangeList (begin,end,stride). Returns a list of the
-- |integers between begin and end inclusive in steps of stride.
intRangeList :: (Int, Int, Int) -> List Int
```

Listing B.2: List library functions

```
-- |mapArrInv (f,f1,g,a). Maps f and g over a. f must be a projection function,
-- |and f1 its inverse.
mapArrInv :: (i->j, j->i, (i,v)->w, Arr i v) -> Arr j w

-- |reduceArr (f,g,v0,a). Maps f over a and then aggregates values using g.
-- |v0 is the null element.
reduceArr :: ((i,v)->s, (s,s)->s, s, Arr i v) -> s

-- |groupReduceArr (kf,vf,g,v0,a). Maps kf and vf over a, grouping by
-- |kf's result, and aggregating vf's results using g with v0 the null element.
groupReduceArr :: ((i,v)->j, (i,v)->w, (w,w)->w, w, Arr i v) -> Arr j w

-- |groupPrefixArr (kf,vf,g,v0,a). Performs parallel prefix of groups.
-- |kf projects out group indices, vf values, and g is the fold function.
-- |kf must be a projection function.
groupPrefixArr :: ((i,v)->j, (i,v)->w, (w,w)->w, w, Arr i v) -> Arr j w

-- |findInArr (vf,v,notFound,a). Maps vf over a, returning the first
-- |element that matches v, or notFound if it's not found.
findInArr :: ((i,v) -> s, s, (i,v,s), Arr i v) -> (i,v,s)

-- |crossArr (a,b). Returns the Cartesian product of a and b.
crossArr :: (Arr i v, Arr j w) -> Arr (i,j) (v,w)

-- |eqJoinArr (f1,f2,a1,a2). Returns the Cartesian product of
-- |a1 and a2 restricted to where the dimensions returned by
-- |f1 from a1 equal those returned by f2 from a2.
eqJoinArr :: (i->k, j->k, Arr i v, Arr j w) -> Arr (i,j) (v,w)

-- |unionArrWith (f,f0,a,b). Returns a unioned with b, where
-- |elements that occur in a and b and combined using f. The
-- |bounds of the result are min(a,b) to max(a,b), where the
-- |values of any undefined elements are computed using f0.
unionArrWith :: ((v,v)->v, i->v, Arr i v, Arr i v) -> Arr i v

-- |reshapeArr (begin,end,stride,a). Returns a new array containing
-- |the elements of a, with bound begin-end:stride. The new bounds
-- |must contain the same number of elements as a.
```

```
reshapeArr :: (j, j, j, Arr i v) -> Arr j v

-- |arrBound a. Returns bounds (begin,end,stride) of a.
arrBounds :: Arr i v -> (i,i,i)

-- |subArr (begin,end,stride,a). Returns subset of array
-- |in bounds begin-end:stride.
subArr :: (i, i, i, Arr i v) -> Arr i v

-- |shiftArrL (deltaL, a). Returns array a, with indices
-- |moved left (decremented) by i. All values in i must be
-- |0 or positive.
shiftArrL :: (i, Arr i v) -> Arr i v

-- |shiftArrR (deltaR, a). Returns a with indices moved
-- |right (incremented) by i. All values in i must be 0
-- |or positive.
shiftArrR :: (i, Arr i v) -> Arr i v

-- |scaleArr (delta, a). Returns a with all indices multiplied
-- |by delta. All values in delta must be positive.
scaleArr :: (i, Arr i v) -> Arr i v

-- |descaleArr (delta, a). Returns a with all indices divided
-- |by delta. All values in delta must be positive.
descaleArr :: (i, Arr i v) -> Arr i v

-- |reflectArr (i,a). Reflects a along all dimensions for
-- |which i is -1. All values in i must be 1 or -1.
reflectArr :: (i, Arr i v) -> Arr i v

-- |intRangeArr (begin,end,stride). Returns an array with
-- |bounds begin-end:stride.
intRangeArr :: (Int, Int, Int) -> Arr Int ()
```

Listing B.3: Array library functions

```
-- |arrToMap a. Returns a map of all the elements in
-- |a with their indices.
arrToMap :: Arr i v -> Map i v

-- |arrToList a. Returns a list of all the elements
-- |in a with their indices.
arrToList :: Arr i v -> List (i,v)

-- |listToArr (begin,end,stride,v0,l). Returns a new
-- |array with indices between begin and end in steps
-- |of stride. Elements are taken from l, or are
-- |initialized using v0 otherwise.
listToArr :: (i, i, i, i->v, List (i,v)) -> Arr i v

-- |mapToList m. Returns all the (key,value) pairs in m.
```

```
-- |The ordering is arbitrary.
mapToList :: Map i v -> List (i,v)

-- |listToMap l. Returns a new map formed from the
-- |(key,value) pairs in l. If there are multiple
-- |occurances of a key, the last value is the one
-- |used in the map
listToMap :: List (i,v) -> Map i v
```

Listing B.4: Conversion functions

Appendix C

Flocc DDL types

This appendix lists DDL types for array, map, and list combinator implementations. This list of DDL types supplements those in Chapter 4, but is not exhaustive.

`DArr` combinators Figure C.1 lists the DDL types of various array-combinator implementations. `mapArrInv1` and `mapArrInv2` are structure preserving combinators that apply their first two parameter functions to the indices and values, respectively, of a distributed array. The index transformer function f is constrained to be a permutation function, though this is not required in the types. Both `mapArrInv1` and `mapArrInv2` have the same implementation, which acts in-place (i.e., without communication), but they infer DDL information in different directions. In `mapArrInv1` the output partition function g is known, so we construct an input partition function that will provide this output partitioning by applying g after the index transformer function f. `mapArrInv2` does the opposite. If the input is partitioned by g then the output is partitioned by g applied after the inverse index transformer function f_{inv}, since f_{inv} returns the indices from the output elements, that were returned for the same elements in the input array.

`eqJoin1A` to `eqJoin3` perform equi-joins on pairs of distributed arrays (i.e., Cartesian products restricted to where the join-keys are equal), where the two function parameters emit join-key indices from the elements in the first and second arrays respectively. `eqJoin1A` and `eqJoin1B` can do this in-place without communication because their input arrays are partitioned by their join-key emitters f and g along the same nodes in the topology d, and are therefore aligned by join-key index, such that all elements for a given key in both arrays is on the same node. The output combines the indices and elements from the inputs pairwise, and so the outputs are partitioned by the join-keys from the first array $f \cdot$ `fst` in `eqJoin1A`, and those from the second array $g \cdot$ `snd` in `1B`. `eqJoinArr2` accepts any output partition function f, and therefore requires all combinations of elements (the Cartesian product) to be enumerable. Thus we partition the first array by `fstFun` f along `d1`, and mirror the second (hence the `nullF`) along `d1`.

```
mapArrInv1 :: Π(f,_,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v (g · f) d m) -> DArr j w g d m
mapArrInv2 :: Π(_,f_inv,_,_) : (i->j, j->i, (i,v)->w,
  DArr i v g d m) -> DArr j w (g · f_inv) d m

eqJoinArr1A :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (f · fst) d m
eqJoinArr1B :: Π(f,g,_,_) : (i->k, j->k, DArr i v f d m,
  DArr j w g d m) -> DArr (i,j) (v,w) (g · snd) d m
eqJoinArr2 :: (i->k, j->k, DArr i v fstFun(f) d1 m,
  DArr j w nullF d2 (d1,m)) -> DArr (i,j) (v,w) f d1 m
eqJoinArr3 :: (i->k, j->k, DArr i v fstFun(f) d1 (d2, m),
  DArr j w sndFun(f) d2 (d1, m)) -> DArr (i,j) (v,w) f (d1,d2) m

groupReduceArr1 :: (i->j, (i,v)->w, (w,w)->w, w,
  DArr i v f d1 m1) -> DArr j w id d2 m2
groupReduceArr2 :: Π(pf,_,_,_,_) : (i->j, (i,v)->w, (w,w)->w, w,
  DArr i v pf d m) -> DArr j w id d m

subArr :: (i, i,  DArr i v pf d m) -> DArr i v pf d m
shiftArrL :: (i, DArr i v pf d m) -> DArr i v pf d m
shiftArrR :: (i, DArr i v pf d m) -> DArr i v pf d m
scaleArr :: (i, DArr i v pf d m) -> DArr i v pf d m
reflectArr :: (i, DArr i v pf d m) -> DArr i v pf d m
```

FIGURE C.1: DDL types for array combinator implementations.

`fstFun` f ensures that the output is partitioned by f, and all combinations are enumerable without communication. Finally, `eqJoinArr3` is similar, but it partitions and mirrors both input arrays— the first by `fstFun` f along `d1` mirrored along `d2` and `m`, and the second by `sndFun` f along `d2` mirrored along `d1` and `m`. Here `d1` and `d2` are orthogonal dimensions of a Cartesian node toplology, and so the Cartesian product of partitions will be enumerated across `d1` × `d2`.

`groupReduceArr1` and `groupRedueArr2` group array elements by the index returned by the first function parameter— an index projection function that returns a subset of the array's indices, applies the second function parameter to their indices and values, and

then aggregates them using the third function parameter. `groupReduceArr2` can do this in place since its input is partitioned by the index projection function parameter pf, where as `groupReduceArr1` has to exchange intermediate values between nodes to co-locate groups before aggregation.

`subArr` returns the sub-array between the indices in the first and second parameters. This performs no communication. `shiftArrL` and `shiftArrR` increase and decrease the indices of an array by the offsets in the first parameter, respectively. These send the appropriate number of elements from the array edges to neighboring nodes. `scaleArr` multiplies the indices in an array by the coefficients in the first parameter, and performs the necessary communication to do this on the cluster. `reflectArr` returns an array with any of the indices for which the first parameter contains a negative number flipped/reversed, and the rest left the same. This works by exchanging partitions on opposite sides of the center lines over the network, for those indices that are to be reflected, and reversing the order of the elements locally.

`DMap` combinators Figure C.2 and Figure C.3 list the DDL types of various map-combinator implementations. `intRangeMap` creates a map with integer keys in the range (and with the stride) specified. `intRangeMap1`, `intRangeMap2`, `intRangeMap3`, all generate these keys for each partition in parallel, and use hash, dynamic-discrete, and dynamic-range partitioning modes, respectively. `intRangeMapMirr` returns a mirrored map, where the whole map is replicated across the dimensions in `m`.

`countMap` returns the cardinality of a partitioned map, by summing the cardinalities of each partition, and `countMapMirr` calculates the cardinality of a mirrored map, in-place.

`map` is a natural transformation for maps, similar to `mapArrInv1`. `mapInv1` and `mapInv2` are the same, except they also require an inverse map function like with `mapArrInv`. `eqJoin1A` to `eqJoin4` are map equivalents of the `eqJoinArr` functions, with equivalent distributions. The difference here is the join-key emitter functions apply to the map key and value, and so the types use `lft` and `rht` instead of `fst` and `snd` to project from the pairs of key-pair and value-pairs. (`eqJoin4` is just the dual of `eqJoin2`.) `allPairsA` and `allPairsB` perform self-joins using their join-key emitter function parameters. Like `eqJoin1A` and `eqJoin1B` these occur in place, since their input maps are partitioned by their join-key emitter function f. Again `groupReduce1` and `groupReduce2` are map equivalents of `groupReduceArr1` and `groupReduceArr2`. `reduce` accepts maps partitioned in any way, projects values from their key-value pairs, and aggregates them using a binary function. This performs a small amount of communication to gather the intermediate reductions, and broadcast the result to all nodes.

`union`, `intersect`, and `diff` are left-biased set operations. `union` acts in place since its inputs and output are partitioned by the map key via `fst`, and so equivalent keys are co-located. `intersect1` and `diff1` work in the same way. `intersect2` and `diff2` can

```
intRangeMap1    :: (Int,Int,Int) -> DMap Int () Hash fst d m
intRangeMap2    :: (Int,Int,Int) -> DMap Int () DynDiscrete fst d m
intRangeMap3    :: (Int,Int,Int) -> DMap Int () DynRange fst d m
intRangeMapMirr :: (Int,Int,Int) -> DMap Int () p nullF () m

countMap     :: DMap k v p f d m -> Int
countMapMirr :: DMap k v p nullF () m -> Int

map     :: Π(f,_) : ((i,v)->(j,w),
  DMap i v p (g · f) d m) -> DMap j w p g d m
mapInv1 :: Π(f,f_inv,_) : ((i,v)->(j,w), (j,w)->(i,v),
  DMap i v p (g · f) d m) -> DMap j w p g d m
mapInv2 :: Π(f,f_inv,_) : ((i,v)->(j,w), (j,w)->(i,v),
  DMap i v p g d m) -> DMap j w p (g · f_inv) d m

eqJoin1A :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
  DMap j w p g d m) -> DMap (i,j) (v,w) p (f · lft) d m
eqJoin1B :: Π(f,g,_,_) : ((i,v)->k, (j,w)->k, DMap i v p f d m,
  DMap j w p g d m) -> DMap (i,j) (v,w) p (g · rht) d m
eqJoin2  :: ((i,v)->k, (j,w)->k, DMap i v p (lftFun f) d1 m,
  DMap j w p nullF d2 (d1,m)) -> DMap (i,j) (v,w) p f d1 m
eqJoin3  :: ((i,v)->k, (j,v)->k, DMap i v p (lftFun f) d1 (d2, m),
  DMap j w p (rhtFun f) d2 (d1,m)) -> DMap (i,j) (v,w) p f (d1,d2) m
eqJoin4  :: ((i,v)->k, (j,w)->k, DMap i v p nullF d1 m,
  DMap j w p (rhtFun f) d2 (d1,m)) -> DMap (i,j) (v,w) p f d2 m

crossMaps1 :: (DMap i v p f d1 m, DMap j w p nullF d2 (d1,m)) ->
  DMap (i,j) (v,w) p f·lft d1 m
crossMaps2 :: (DMap i v p (lftFun f) d1 m, DMap j w p nullF d2 (d1,m)) ->
  DMap (i,j) (v,w) p f d1 m
crossMaps3 :: (DMap i v p f d1 (d2,m), DMap j w p g d2 (d1,m)) ->
  DMap (i,j) (v,w) p ((f·lft)⊗(g·rht))·Δ (d1,d2) m
crossMaps4 :: (DMap i v p (lftFun f) d1 (d2,m),
               DMap j w p (rhtFun f) d2 (d1,m)) ->
               DMap (i,j) (v,w) p f (d1,d2) m
```

FIGURE C.2: DDL types for map combinator implementations.

```
allPairsA :: Π(f,_) : ((i,v)->k, DMap i v p f d m) ->
  DMap (i,i) (v,v) p (f · lft) d m
allPairsB :: Π(f,_) : ((i,v)->k, DMap i v p f d m) ->
  DMap (i,i) (v,v) p (f · rht) d m

groupReduce1 :: ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d1 m1) -> DMap j w p fst d2 m2
groupReduce2 :: Π(f,_,_,_) : ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap i v p f d m) -> DMap j w p fst d m
reduce       :: ((k,v)->s, (s,s)->s, s, DMap k v p f d m) -> s

union      :: (DMap k v p fst d m, DMap k v p fst d m) ->
  DMap k v p fst d m
intersect1 :: (DMap k v p fst d m, DMap k w p fst d m) ->
  DMap k v p fst d m
intersect2 :: (DMap k v p (lftFun f) d1 m,
  DMap k w p nullF d2 (d1,m)) -> DMap k v p f d1 m
intersect3 :: (DMap k v p nullF d1 m,
  DMap k w p (rhtFun f) d2 (d1,m)) -> DMap k v p f d1 m
intersect4 :: (DMap k v p (lftFun f) d1 (d2, m),
  DMap k w p (rhtFun f) d2 (d1,m)) -> DMap k v p f (d1,d2) m
diff1 :: (DMap k v p fst d m, DMap k w p fst d m) -> DMap k v p fst d m
diff2 :: (DMap k v p (lftFun f) d1 m, DMap k w p nullF d2 (d1,m)) ->
  DMap k v p f d1 m
```

FIGURE C.3: DDL types for map combinator implementations.

partition their outputs by any function f, by partitioning their first inputs by `lftFun` f, and mirroring their second inputs. This sort of distribution would not work for `union` because it must test on the same node whether there is a given key value in either map. `intersect4` is the dual of `intersect3`, i.e., it mirrors its first map, and partitions its second. `diff` cannot be distributed in this way, since it needs to test locally for every element in the first map's partition, whether the respective key exists in the whole second map. The second map must therefore be mirrored, if the first map is arbitrarily partitioned.

```
zip        :: (DList v cyc d m, DList w cyc d m) -> DList (v,w) cyc d m
mapList    :: (v->w, DList v p d m) -> DList w p v m
reduceList :: ((v,v)->v, v, DList v p d m) -> v
filterList :: (v->Bool, DList v blk d m) -> DList v blk d m
crossList  :: (DList v blk d1 m, DList w p d2 (m,d1)) ->
  DList (v,w) blk d1 m
```

FIGURE C.4: DDL types for list combinator implementations.

```
mirrMap :: DMap k v p f d m -> DMap k v p f d (m,m')
sieveMap :: DMap k v p f d (m,m') -> DMap k v p f d m
repartMap :: DMap k v p1 f1 d1 m -> DMap k v p2 f2 d2 m

redistList :: DList v p1 d1 m1 -> DList v p2 d2 m2
distListLit :: List v -> DList v p d m
```

FIGURE C.5: Distributed data layout (DDL) types for some redistribution functions.

`DList` combinators Figure C.4 lists the DDL types of various list-combinator implementations. The `zip` combinator joins together two lists pairwise, up to the length of the shorter list. Its lists must use a cyclic distribution `Cyc` along the same dimensions of the node topology `d`. `mapList` is the list natural transformation combinator, and operates in-place for any input distribution. `reduceList` acts like `reduce` and works on any input distribution. `filterList` takes a list and returns a list of all the elements for which the predicate function parameter returns true. It works in place and so its input and output lists are block distributed, because cyclic distributions require the same number of elements on each node, and removing elements would very likely lead to different numbers of elements per node, and in the wrong order. `crossList` performs the Cartesian product of two distributed lists in-place, where the first list must be block distributed, the second must be mirrored along the same dimension, and the output is block distributed. The first list cannot be cyclic because crossing a cyclic distributed list with a mirrored one, does not give a cyclic (or block) distributed output, unless the number of elements in the second list is the same as the number of nodes in `d1`.

Redistribution functions `mirrMap` to `distListLit` are example redistribution functions. These functions do not modify the data in their argument collections, but change how they are distributed. They are therefore implementations of `id` (i.e., the identity function) on the high level. They can also be thought of as type-casts. `mirrMap` takes any distributed map and mirrors its partitions along another dimension of the global

topology. `sieveMap` acts the opposite way— removing replicated partitions from dimension `m'`. `repartMap` re-partitions a `DMap` using a different partition function over different dimension(s). `redistList` does the same for lists, and optionally mirrors over different dimensions too, and `distListLit` distributes a globally mirrored list in any way on the cluster.

Appendix D

Equational theory proofs

This appendix proves that each equation in the equational theories of projection functions, permutation functions, and index functions in Chapter 6 is sound. It does that by converting them from point-free into pointed form, and then applying the reduction rules in Flocc's operational semantics in Chapter 3.

$$
\begin{aligned}
(\Pi_2 \otimes \Pi_1) \cdot \Delta &= \backslash(x, y) \to (\backslash((a, b), (c, d)) \to (\Pi_2\ (a, b), \Pi_1\ (c, d)))\ ((\backslash u \to (u, u))\ (x, y)) \\
&= \backslash(x, y) \to (\backslash((a, b), (c, d)) \to (\Pi_2\ (a, b), \Pi_1\ (c, d)))\ ((x, y), (x, y)) \\
&= \backslash(x, y) \to (\Pi_2\ (x, y), \Pi_1\ (x, y)) \\
&= \backslash(x, y) \to ((\backslash(u, v) \to v)\ (x, y), (\backslash(u, v) \to u)\ (x, y)) \\
&= \backslash(x, y) \to (y, x)
\end{aligned}
$$

FIGURE D.1: Projection function identities.

$$f \cdot \mathrm{id} = \backslash x \to f\ ((\backslash u \to u)\ x) = \backslash x \to f\ x = f \tag{6.1}$$

$$\mathrm{id} \cdot f = \backslash x \to (\backslash u \to u)\ (f\ x) = \backslash x \to f\ x = f \tag{6.2}$$

$$\begin{aligned} \Pi_1 \cdot \Delta &= \backslash x \to (\backslash (u, v) \to u)\ ((\backslash u \to (u, u))\ x) \\ &= \backslash x \to (\backslash (u, v) \to u)\ (x, x) = \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.3}$$

$$\begin{aligned} \Pi_2 \cdot \Delta &= \backslash x \to (\backslash (u, v) \to v)\ ((\backslash u \to (u, u))\ x) \\ &= \backslash x \to (\backslash (u, v) \to v)\ (x, x) = \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.4}$$

$$\begin{aligned} (\Pi_1 \otimes \Pi_2) \cdot \Delta &= \backslash (x, y) \to (\Pi_1 \otimes \Pi_2)((\backslash u \to (u, u))\ (x, y)) \\ &= \backslash (x, y) \to (\Pi_1 \otimes \Pi_2)\ ((x, y), (x, y)) \\ &= \backslash (x, y) \to (\Pi_1\ (x, y), \Pi_2\ (x, y)) \\ &= \backslash (x, y) \to ((\backslash (u, v) \to u)\ (x, y), (\backslash (u, v) \to v)\ (x, y)) \\ &= \backslash (x, y) \to (x, y) = \mathrm{id} \otimes \mathrm{id} \end{aligned} \tag{6.5}$$

$$\begin{aligned} (\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta &= \backslash (x, y) \to (\backslash (x, y) \to (y, x))\ ((\backslash (x, y) \to (y, x))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (x, y) \to (y, x))\ (y, x) \\ &= \backslash (x, y) \to (x, y) = \mathrm{id} \otimes \mathrm{id} \end{aligned} \tag{6.6}$$

$$\begin{aligned} (f_1 \otimes f_2) \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta &= \backslash (x, y) \to (\backslash (v, w) \to (f_1\ v, f_2\ w))\ ((\backslash (t, u) \to (u, t))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (v, w) \to (f_1\ v, f_2\ w))\ (y, x) \\ &= \backslash (x, y) \to (f_1\ y, f_2\ x) \\ (\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (f_2 \otimes f_1) &= \backslash (x, y) \to (\backslash (t, u) \to (u, t))\ ((\backslash (v, w) \to (f_2\ v, f_1\ w))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (t, u) \to (u, t))\ (f_2\ x, f_1\ y) \\ &= \backslash (x, y) \to (f_1\ y, f_2\ x) \end{aligned} \tag{6.7}$$

$$\begin{aligned} \Pi_1 \cdot (f_1 \otimes f_2) &= \backslash (x, y) \to (\backslash (u, v) \to u)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (u, v) \to u)\ (f_1\ x, f_2\ y) \\ &= \backslash (x, y) \to f_1\ x \\ f_1 \cdot \Pi_1 &= \backslash (x, y) \to f_1\ ((\backslash (u, v) \to u)\ (x, y)) \\ &= \backslash (x, y) \to f_1\ x \end{aligned} \tag{6.8}$$

$$\begin{aligned} \Pi_2 \cdot (f_1 \otimes f_2) &= \backslash (x, y) \to (\backslash (u, v) \to v)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (u, v) \to v)\ (f_1\ x, f_2\ y) \\ &= \backslash (x, y) \to f_2\ y \\ f_2 \cdot \Pi_2 &= \backslash (x, y) \to f_2\ ((\backslash (u, v) \to v)\ (x, y)) \\ &= \backslash (x, y) \to f_2\ y \end{aligned} \tag{6.9}$$

FIGURE D.2: Equations showing soundness of equational theory of projection functions.

$$\begin{aligned}
\Pi_1 \cdot (f_1 \otimes f_2) \cdot \Delta &= \backslash x \to (\backslash (u, v) \to u)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ ((\backslash u \to (u, u))\ x)) \\
&= \backslash x \to (\backslash (u, v) \to u)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ (x,\ x)) \\
&= \backslash x \to (\backslash (u, v) \to u)\ (f_1\ x,\ f_2\ x) \\
&= \backslash x \to f_1\ x = f_1
\end{aligned} \tag{6.10}$$

$$\begin{aligned}
\Pi_2 \cdot (f_1 \otimes f_2) \cdot \Delta &= \backslash x \to (\backslash (u, v) \to v)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ ((\backslash u \to (u, u))\ x)) \\
&= \backslash x \to (\backslash (u, v) \to v)\ ((\backslash (t, w) \to (f_1\ t, f_2\ w))\ (x,\ x)) \\
&= \backslash x \to (\backslash (u, v) \to v)\ (f_1\ x,\ f_2\ x) \\
&= \backslash x \to f_2\ x = f_2
\end{aligned} \tag{6.11}$$

$$\begin{aligned}
\Delta \cdot f &= \backslash x \to (\backslash u \to (u, u))\ (f\ x) \\
&= \backslash x \to (f\ x, f\ x) \\
(f \otimes f) \cdot \Delta &= \backslash x \to \backslash (w, t) \to (f\ w, f\ t)\ ((\backslash u \to (u, u))\ x) \\
&= \backslash x \to \backslash (w, t) \to (f\ w, f\ t)\ (x, x) \\
&= \backslash x \to (f\ x, f\ x)
\end{aligned} \tag{6.12}$$

$$\begin{aligned}
(f_1 \cdot f_2) \cdot f_3 &= \backslash x \to (\backslash y \to f_1\ (f_2\ y))\ (f_3\ x) \\
&= \backslash x \to f_1\ (f_2\ (f_3\ x)) \\
f_1 \cdot (f_2 \cdot f_3) &= \backslash x \to f_1\ ((\backslash y \to f_2\ (f_3\ y))\ x) \\
&= \backslash x \to f_1\ (f_2\ (f_3\ x))
\end{aligned} \tag{6.13}$$

$$\begin{aligned}
(f_1 \otimes f_2) \cdot (f_3 \otimes f_4) &= \backslash (x, y) \to (\backslash (a, b) \to (f_1\ a, f_2\ b))\ ((\backslash (c, d) \to (f_3\ c, f_4\ d))\ (x, y)) \\
&= \backslash (x, y) \to (\backslash (a, b) \to (f_1\ a, f_2\ b))\ (f_3\ x,\ f_4\ y) \\
&= \backslash (x, y) \to (f_1\ (f_3\ x),\ f_2\ (f_4\ y)) \\
(f_1 \cdot f_3) \otimes (f_2 \cdot f_4) &= \backslash (x, y) \to ((\backslash z \to f_1\ (f_3\ z))\ x, (\backslash w \to f_2\ (f_4\ w))\ y) \\
&= \backslash (x, y) \to (f_1\ (f_3\ x), f_2\ (f_4\ y))
\end{aligned} \tag{6.14}$$

FIGURE D.3: Equations 2 showing soundness of equational theory of projection functions.

$$h \cdot \mathrm{id} = \backslash x \to h\ ((\backslash u \to u)\ x) = \backslash x \to h\ x = h \tag{6.15}$$

$$\mathrm{id} \cdot h = \backslash x \to (\backslash u \to u)\ (h\ x) = \backslash x \to h\ x = h \tag{6.16}$$

$$\begin{aligned} h \odot \mathrm{null} &= \backslash x \to (\backslash y \to (h\ y, \backslash_{\to}()\ y)) \\ &= \backslash x \to (h\ x, ()) \\ &= \backslash x \to h\ x = h \end{aligned} \tag{6.17}$$

$$\begin{aligned} \mathrm{null} \odot h &= \backslash x \to (\backslash y \to (\backslash_{\to}()\ y, h\ y)) \\ &= \backslash x \to ((), h\ x) \\ &= \backslash x \to h\ x = h \end{aligned} \tag{6.18}$$

$$\begin{aligned} (h_1 \cdot h_2) \cdot h_3 &= \backslash x \to (\backslash y \to f_1\ (f_2\ y))\ (f_3\ x) \\ &= \backslash x \to f_1\ (f_2\ (f_3\ x)) \\ h_1 \cdot (h_2 \cdot h_3) &= \backslash x \to f_1\ ((\backslash y \to f_2\ (f_3\ y))\ x) \\ &= \backslash x \to f_1\ (f_2\ (f_3\ x)) \end{aligned} \tag{6.19}$$

$$\begin{aligned} (h_1 \odot h_2) \odot h_3 &= \backslash x \to ((\backslash y \to (f_1\ y, f_2\ y))\ x, f_3\ x) \\ &= \backslash x \to ((f_1\ x, f_2\ x), f_3\ x)) \\ &= \backslash x \to (f_1\ x, f_2\ x, f_3\ x) \\ h_1 \odot (h_2 \odot h_3) &= \backslash x \to (f_1\ x, (\backslash y \to (f_2\ y, f_3\ y))\ x) \\ &= \backslash x \to (f_1\ x, (f_2\ x, f_3\ x)) \\ &= \backslash x \to (f_1\ x, f_2\ x, f_3\ x) \end{aligned} \tag{6.20}$$

$$\begin{aligned} (h_1 \odot h_2) \cdot h_3 &= \backslash x \to (\backslash y \to (h_1\ y, h_2\ y))\ (h_3\ x) \\ &= \backslash x \to (h_1\ (h_3\ x), h_2\ (h_3\ x)) \\ ((h_1 \cdot h_3) \odot (h_2 \cdot h_3)) &= \backslash x \to ((\backslash y \to h_1\ (h_3\ y))\ x, (\backslash z \to h_2\ (h_3\ z))\ x) \\ &= \backslash x \to (h_1\ (h_3\ x), h_2\ (h_3\ x)) \end{aligned} \tag{6.21}$$

FIGURE D.4: Equations showing soundness of equational theory of permutation functions.

$$i^{-1^{-1}} = 1/(1/i) = i \tag{6.22}$$

$$\mathbf{1}^{-1} = 1/\mathbf{1} = \mathbf{1} \tag{6.23}$$

$$\begin{aligned} +(i) \cdot -(i) &= \backslash x \to (\backslash x \to (x + i))\ ((\backslash x \to (x + -i))\ x) \\ &= \backslash x \to (\backslash x \to (x + i))\ (x + -i) \\ &= \backslash x \to ((x + -i) + i) \\ &= \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.24}$$

$$\begin{aligned} -(i) \cdot +(i) &= \backslash x \to (\backslash x \to (x + -i))\ ((\backslash x \to (x + i))\ x) \\ &= \backslash x \to (\backslash x \to (x + -i))\ (x + i) \\ &= \backslash x \to ((x + i) + -i) \\ &= \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.25}$$

$$\begin{aligned} \times(i) \cdot \times(i^{-1}) &= \backslash x \to (\backslash x \to (x \times i))\ ((\backslash x \to (x \times 1/i))\ x) \\ &= \backslash x \to (\backslash x \to (x \times i))\ (x \times 1/i) \\ &= \backslash x \to ((x \times 1/i) \times i) \\ &= \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.26}$$

$$\begin{aligned} \times(i^{-1}) \cdot \times(i) &= \backslash x \to (\backslash x \to (x \times 1/i))\ ((\backslash x \to (x \times i))\ x) \\ &= \backslash x \to (\backslash x \to (x \times 1/i))\ (x \times i) \\ &= \backslash x \to ((x \times i) \times 1/i) \\ &= \backslash x \to x = \mathrm{id} \end{aligned} \tag{6.27}$$

$$g \cdot \mathrm{id} = \backslash x \to g\ ((\backslash u \to u)\ x) = \backslash x \to g\ x = g \tag{6.28}$$

$$\mathrm{id} \cdot g = \backslash x \to (\backslash u \to u)\ (g\ x) = \backslash x \to g\ x = g \tag{6.29}$$

$$\begin{aligned} (g_1 \cdot g_2) \cdot g_3 &= \backslash x \to (\backslash y \to f_1\ (f_2\ y))\ (f_3\ x) \\ &= \backslash x \to f_1\ (f_2\ (f_3\ x)) \\ g_1 \cdot (g_2 \cdot g_3) &= \backslash x \to f_1\ ((\backslash y \to f_2\ (f_3\ y))\ x) \\ &= \backslash x \to f_1\ (f_2\ (f_3\ x)) \end{aligned} \tag{6.30}$$

$$\begin{aligned} (g_1 \otimes g_2) \cdot (g_3 \otimes g_4) &= \backslash (x, y) \to (\backslash (a, b) \to (f_1\ a, f_2\ b))\ ((\backslash (c, d) \to (f_3\ c, f_4\ d))\ (x, y)) \\ &= \backslash (x, y) \to (\backslash (a, b) \to (f_1\ a, f_2\ b))\ (f_3\ x,\ f_4\ y) \\ &= \backslash (x, y) \to (f_1\ (f_3\ x),\ f_2\ (f_4\ y)) \\ (g_1 \cdot g_3) \otimes (g_2 \cdot g_4) &= \backslash (x, y) \to ((\backslash z \to f_1\ (f_3\ z))\ x, (\backslash w \to f_2\ (f_4\ w))\ y) \\ &= \backslash (x, y) \to (f_1\ (f_3\ x), f_2\ (f_4\ y)) \end{aligned} \tag{6.31}$$

FIGURE D.5: Equations showing soundness of equational theory of indexing functions.

Appendix E

Flocc language feature evaluation

This appendix compares the language features of Fortran 90, HPF, ZPL, Relational algebra, MapReduce, DPH, and SISAL to Flocc. It demonstrates that Flocc's data parallel combinators (listed in Appendix B) are equivalent to the data-parallel features all of these, apart from HPF `Forall`-loops with array references that are not translations, scalings, and reflections.

Language	Feature	Sub-feature	Flocc combinators
Fortran 90	Constructors	Scalars	list expression; listToArr
(Arrays)		Arrays	concatList; listToArr
		Implied do	intRangeArr; mapArrInv
	Get shape		arrBounds
	Reshape		reshapeArr
	Sections		subArr; shiftArrL; shiftArrR
	Assignment	Whole	mapArrInv; cross; intRange; eqJoinArr; unionArrWith
		Mask	mapArrInv(if)
	Reductions	Whole	reduceArr; findInArr
		Partial	groupReduceArr
	Parallel prefix	Whole	prefixArr
		Partial	groupPrefixArr
	Construction	Merge	eqJoinArr; mapArrInv(if)
		Pack	arrToList; filterList
		Spread	crossArr; intRangeArr; mapArrInv

Language	Feature	Sub-feature	Flocc combinators
		Unpack	expandList; listToArr; reshapeArr
	Manipulation	Cshift	shiftArrL; shiftArrR; subArr; unionArrWith
		EoShift	shiftArrL; shiftArrR; subArr; unionArrWith
		Transpose	mapArrInv(swap)
	Location	Maxloc	reduceArr
		Minloc	reduceArr
HPF	Forall	Index translation	shiftArrL; shiftArrR
(Arrays)		Index scaling	scaleArr
		Index reflection	reflectArr
		Masked	mapArrInv(if)
		Other arr refs	Not supported
	Scatters		mapArrInv(if); eqJoinArr; arrToMap; groupReduce; mapToArr
	Reductions		reduceArr/findInArr
	Prefixes		prefixArr
	Suffixes		prefixArr; reflectArr
	Sorting	Grade_up	arrToList; sortList
		Grade_down	arrToList; sortList; reverseList
ZPL	Assignment		mapArrInv; eqJoinArr
(Arrays)	Regions		shiftArrL; shiftArrR; subArr; unionArrWith
	Creation		crossArr; intRangeArr
	Directions; @		shiftArrL; shiftArrR
	Reductions	Whole	reduceArr; findInArr
		Partial	groupReduceArr
	Parallel prefix	Whole	prefixArr
		Partial	groupPrefixArr
	WHERE		mapArrInv(if)

Language	Feature	Sub-feature	Flocc combinators
	Wrap op		shiftArrL; shiftArrR; subArr; unionArrWith
	Reflect op		reflectArr; shiftArrL; shiftArrR; subArr; unionArrWith
Relational	Set operators	union	union
algebra		intersection	intersect
(Maps)		difference	diff
		Cartesian product	cross
	Projection		mapInv; map; groupReduce
	Selection		filterMap
	Rename		mapInv; map
	Joins	Eq-join	eqJoin
		InEq-Join	ltJoin
		Natural join	mapInv; cross; groupReduce
		Theta join	cross; filterMap
		Semijoin	mapInv; cross; groupReduce
		Antijoin	diff; mapInv; cross; groupReduce
		Division	groupReduce; diff; cross
		Outer join	mapInv; cross; groupReduce; union; diff; listToMap
	Aggregation		reduce; groupReduce
	Sorting		mapToList; sortList
MR			map; mapInv; groupReduce;
(Maps)			sortList; concatList
DPH	GHC.PArr	emptyP	list expression
(List)		singletonP	list expression
		replicateP	intRangeList; mapList
		appendP	concatList
		concatP	reduce; concatList; concatLists
		lengthP	listLength

Language	Feature	Sub-feature	Flocc combinators
		indexP	subList; head
		sliceP	subList
		mapP	mapList
		zipWithP	zip; mapList
		crossP	crossList
		crossMapP	crossList; mapList
		filterP	filterList
		zipP	zip; mapList
		unzipP	unzip
		foldP	reduceList
		permuteP	listToMap; eqJoin; mapToList
		unconcatP	unconcatList
		transposeP	transposeList
		expandP	listToMap; eqJoin; mapInv; mapToList
		combineP	zip; mapList; expandList
		splitP	findInList; subList
		scanlP	foldListL; concatList
		foldlP	foldListL
		scanrP	foldListR; concatList
		foldrP	foldListR
		takeP	findInList; subList
		dropP	findInList; subList
		splitAtP	findInList; subList
		takeWhileP	findInList; subList
		dropWhileP	findInList; subList
		spanP	findInList; subList
		breakP	findInList; subList
		andP	reduceList
		orP	findInList
		anyP	findInList

Language	Feature	Sub-feature	Flocc combinators
		allP	reduceList
		elemP	findInList
		notElemP	findInList
		lookupP	findInList
		sumP	reduceList
		productP	reduceList
		maximumP	reduceList
		minimumP	reduceList
SISAL	Ranges	for x in A	mapList
(Lists)		for I in l n	intRangeList
		for x in A dot y in B	zipList; mapList
		for I in l n cross j in ?	crossList; mapList; intRangeList
	Array creation	literal	list expression
		array_fill(1; N; 0)	mapList
		A —— B	concatList
		A[1: v]	mapList/filterList;expandList/zip
			mapList
	Reductions	array of	reduceList(concatList)
		stream of	N/A
		catenate	reduceList(concatList)
		sum	reduceList(add)
		product	reduceList(mul)
		least	reduceList(min)
		greatest	reduceList(max)

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Pearson Education, 2004.

[2] S. Adachi, H. Iwasaki, and Zh. Hu. Diff: A Powerful Parallel Skeleton. In *International Conference on Parallel and Distributed Processing Techniques and Application (PDPTA'00). Las Vegas, CSREA*, volume 4, pages 525–527, 2000.

[3] J. C. Adams, International Standard Organisation, and American National Standard Institute. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications, 1992.

[4] G. Agha. Actors: A model of Concurrent Computation in Distributed Systems. Technical Report 844, June 1986.

[5] A. Ali, L. Johnsson, and D. Mirkovic. Empirical auto-tuning code generator for FFT and trigonometric transforms. In *5th Workshop on Optimizations for DSP and Embedded Systems (ODES'07), in conjunction with International Symposium on Code Generation and Optimization (CGO)*, 2007.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, and Others. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

[7] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 126–138. ACM, 1993.

[8] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 166–178. ACM, 1995.

[9] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 112–125. ACM, 1993.

[10] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 38–49. ACM, 2009.

[11] ANSI. ISO/IEC 9899:1999 Programming languages C, 1999.

[12] J. L. Armstrong and S. R. Virding. Erlang-an experimental telephony programming language. In *Proc. XIII International Switching Symposium (ISS'90)*, 1990.

[13] T. Aubrey-Jones and B. Fischer. Synthesizing MPI Implementations from Functional Data-Parallel Programs. In *Proc. 7th International Symposium on High-level Parallel Programming and Applications (HLPP'14)*, 2014.

[14] T. Aubrey-Jones and B. Fischer. Synthesizing MPI Implementations from Functional Data-Parallel Programs. *International Journal of Parallel Programming*, to be published. Springer, 2015.

[15] H. E. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Trans. Software Engineering*, 18(3): 190–205, Mar 1992.

[16] R. Barriuso and A. Knies. SHMEM users guide for C. Technical report, June 1994.

[17] C. Barton, C. Caşcaval, G. Almasi, R. Garg, J. Amaral, and M. Farreras. Multidimensional blocking in UPC. *Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS 5335, pages 47–62, Springer, 2008.

[18] R Bellman. On the Theory of Dynamic Programming. *Proc. National Academy of Sciences of the United States of America*, 38(8):716–719, August 1952.

[19] R. E. Bellman and S. E. Dreyfus. *Applied dynamic programming*. Princeton Univ. Press, 1962.

[20] J. Bilmes, K. Asanovic, Ch.-Wh. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *Proc. 11th International Conference on Supercomputing*, pages 340–347. ACM, 1997.

[21] R. E. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. *IFIP Transactions*, 50:111–122, 1994.

[22] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *Proc. ACM SIGMOD International Conference on Management of Data (ICMD'10)*, pages 975–986. ACM, 2010.

[23] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 102–111. ACM, 1993.

[24] G. E. Blelloch. NESL: A nested data-parallel language. (Version 2.6). Technical Report CMU-CS-93-129, 1993.

[25] G. E. Blelloch. NESL: A Nested Data-Parallel Language. (Version 3.1). Technical Report CMU-CS-95-170, 1995.

[26] D. Bonachea. GASNet Specification, v1. Technical report, 2002.

[27] U. Bondhugula. Compiling Affine Loop Nests for Distributed-memory Parallel Architectures. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 33:1–33:12. ACM, 2013.

[28] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction (CC'08)*, LNCS 4959, pages 132–146. Springer, 2008.

[29] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 101–113. ACM, 2008.

[30] A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and Ch. Ueberhuber. Automatic Performance Optimization of the Discrete Fourier Transform on Distributed Memory Computers. In *Parallel and Distributed Processing and Applications (PDPA'06)*, LNCS 4330, pages 818–832. Springer, 2006.

[31] G. Bracha. Generics in the Java programming language. *Sun Microsystems*, pages 1–23, 2004.

[32] W. Brainerd. Fortran 77. *Commun. ACM*, 21(10):806–820, 1978.

[33] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, 2010.

[34] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: array-based query processing in Hadoop. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 66:1–66:11. ACM, 2011.

[35] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[36] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.

[37] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Center for Computing Sciences, Institute for Defense Analyses, Technical report, 1999.

[38] D. Chakrabarti, Y. Zhan, and Ch. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SIAM Data Mining (SDM'04)*, volume 4, pages 442–446. SIAM, 2004.

[39] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *Proc. Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, pages 10–18. ACM, 2007.

[40] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007.

[41] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov. Authoring user-defined domain maps in Chapel. *Cray user group*, 2011.

[42] B. L. Chamberlain, S.-E. Choi, M. Dumler, Th. Hildebrandt, D. Iten, V. Litvinov, and G. Titus. The State of the Chapel Union. *Proc. Cray User Group conference (CUG'13)*, 2013.

[43] B. L. Chamberlain, S. E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin. The case for high-level parallel programming in ZPL. *Computational Science Engineering, IEEE*, 5(3):76 –86, July 1998.

[44] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi. User-defined distributions and layouts in chapel: philosophy and framework. In *Proc. 2nd USENIX conference on Hot topics in parallelism*, pages 12–12. USENIX Association, 2010.

[45] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured English query language. In *Proc. 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[46] C. Chan, J. Ansel, Yee Lok Wong, S. Amarasinghe, and A. Edelman. Autotuning multigrid with PetaBricks. In *Proc. Conference on High Performance Computing Networking, Storage and Analysis, (SC'09)*, pages 1–12, Nov 2009.

[47] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[48] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and Sh.-H. Teng. Generating Local Addresses and Communication Sets for Data-parallel Programs. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 149–158. ACM, 1993.

[49] S. Chatterjee, J. R. Gilbert, R. Schreiber, and Sh.-H. Teng. Automatic Array Alignment in Data-parallel Programs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PPoPP'93)*, pages 16–28. ACM, 1993.

[50] S. Chellappa, F. Franchetti, and M. Püschel. High performance linear transform program generation for the Cell BE. *High Performance Embedded Computing (HPEC)*, 2009.

[51] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization (CGO'05)*, pages 111–122, March 2005.

[52] Ch. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations, University of Southern California, California, USA. Technical report, pages 08–897, 2008.

[53] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*, pages 523–534. ACM, 2010.

[54] S. Chen and S. W. Schlosser. Map-reduce meets wider varieties of applications, Intel Research Pittsburgh. Technical Report IRP-TR-08-05, 2008.

[55] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):pp. 345–363, 1936.

[56] A. Church. The Calculi of Lambda Conversion. *Annals of Mathematics Studies (AM-6)* Princeton University Press, 1985.

[57] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *20th International Parallel and Distributed Processing Symposium, 2006.*, page 7, April 2006.

[58] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[59] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1991.

[60] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proc. ACM/IEEE Conference on Supercomputing (SC'02)*, pages 1–11. IEEE Computer Society Press, 2002.

[61] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[62] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[63] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *Proc. Parallel Architectures and Languages Europe (PARLE'93)*, LNCS 694, pages 146–160. Springer, 1993.

[64] L. Davis. *Genetic Algorithms and Simulated Annealing*. Pitman, 1987.

[65] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-based Optimization on Graphs with Application to Library Performance Tuning. In *Proc. 26th Annual International Conference on Machine Learning*, pages 729–736. ACM, 2009.

[66] J. Dean, S. Ghemawat, P. Kelly, D. Sharp, Q. Wu, and R. While. MapReduce: simplified data processing on large clusters. In *Proc. 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI04)*. USENIX Association, 2004.

[67] R. Dementiev, L. Kettner, and P. Sanders. *STXXL: Standard template library for XXL data sets*. Springer, 2005.

[68] A. Donaldson, C. Riley, A. Lokhmotov, and A. Cook. Auto-parallelisation of Sieve C++ programs. In *Proc. 2007 conference on Parallel processing*, pages 18–27. Springer-Verlag, 2007.

[69] L. DOrazio and S. Bimonte. Multidimensional Arrays for Warehousing Data on Clouds. In *Data Management in Grid and Peer-to-Peer Systems*, LNCS 6265, pages 26–37. Springer, 2010.

[70] Ch. Dornan and I. Jones. Alex User Guide, 2003.

[71] T.B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., 1st edition, 1998.

[72] J. Duffy and E. Essey. Parallel LINQ: Running Queries On Multi-Core Processors. *MSDN Magazine*, pages 70–78, October 2007.

[73] E. Dumbill and N. M. Bornstein. *Mono: a developer's notebook*. O'Reilly Media, Inc., 2004.

[74] J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, and R. Barga. Dryadlinq for scientific analyses. In *2009 Fifth IEEE International Conference on e-Science*, pages 329–336. IEEE, 2009.

[75] J. Ekanayake, H. Li, B. Zhang, Th. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proc. 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[76] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *IEEE Fourth International Conference on eScience, 2008.*, pages 277–284, 2008.

[77] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[78] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proc. 4th International Workshop on High-level Parallel Programming and Applications*, pages 5–14. ACM, 2010.

[79] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. MARIANE: MApReduce Implementation Adapted for HPC Environments. In *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, pages 82–89, 2011.

[80] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(78):604 – 615, 2006.

[81] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[82] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349 – 366, 1990.

[83] M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, September 1972.

[84] F. Franchetti, F. Mesmay, D. McFarlin, and M. Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

[85] A.S. Fraser. Simulation of Genetic Systems by Automatic Digital Computers VI. Epistasis. *Australian Journal of Biological Sciences*, 13(2):150–162, January 1960.

[86] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, 1998.*, volume 3, pages 1381–1384, May 1998.

[87] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[88] D. Adler G. Colvin, B. Dawes and P. Dimov. The boost shared_ptr class template.

[89] J. Garcia, E. Ayguade, and J. Labarta. A Novel Approach Towards Automatic Data Distribution. In *Proc. IEEE/ACM Supercomputing Conference, 1995.*, pages 78–78, 1995.

[90] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proc. VLDB Endow.*, 2:1414–1425, August 2009.

[91] W. Gehrke. *Fortran 95 language guide*. Springer-Verlag New York, Inc., 1996.

[92] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[93] A. Ghuloum, A. Sharp, N. Clemons, S. D. Toit, R. Malladi, M. Gangadhar, M. McCool, and H. Pabst. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures, September 2010.

[94] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.

[95] J. Gosling. *The Java language specification*. Prentice Hall, 2000.

[96] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Message-passing code generation for non-rectangular tiling transformations. *Parallel Computing*, 32(10): 711 – 732, 2006.

[97] C. Grelck. Single Assignment C (SAC) High Productivity Meets High Performance. In *Central European Functional Programming School*, pages 207–278. Springer, 2012.

[98] C. Grleck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming (JFP'05)*, 15(03):353–401, 2005.

[99] W. Gropp, E. L. Lusk, and A. Skjellum. *Using Mpi: Portable Parallel Programming With the Message-Passing Interface*. Number v. 1. University Press Group Limited, 1999.

[100] M. Gupta and P. Banerjee. PARADIGM: a compiler for automatic data distribution on multicomputers. In *Proc. 7th international conference on Supercomputing*, pages 87–96. ACM, 1993.

[101] M. Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[102] M. Harman. Software Engineering: An Ideal Set of Challenges for Evolutionary Computation. In *Proceeding of the Fifteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion*, pages 1759–1760. ACM, 2013.

[103] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[104] R. Harper, R. Milner, and M. Tofte. The Definition of Standard ML Version 2. Department of Computer Science, University of Edinburgh, UK. Technical Report ECS-LFCS-88-62, August 1988.

[105] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[106] P. N. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, A. Kamil, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual version 2.22. Technical Report UCB/EECS-2005-15.4, August 2006.

[107] W.D. Hillis. *The connection machine*. MIT Press, 1989.

[108] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[109] A. Homaifar, Ch. X. Qi, and S. H. Lai. Constrained Optimization Via Genetic Algorithms. *SIMULATION*, 62(4):242–253, 1994.

[110] R. Hooke. Direct search solution of numerical and statistical problems. *Journal of the Association for Computing Machinery (ACM)*, pages 212–239, 1961.

[111] G. P. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257 – 267, 1973.

[112] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proc. 35th SIGMOD international conference on Management of data*, pages 987–994. Microsoft, ACM, 2009.

[113] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, pages 84 –93. Department of Computer Science and Engineering, The Ohio State University, Columbis, OH 43210, 2010.

[114] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proc. 3rd ACM SIGPLAN conference on History of programming languages*, pages 7:1–7:22. ACM, 2007.

[115] K. Kennedy and U. Kremer. Automatic data layout for high performance Fortran. In *Proc. 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM, 1995.

[116] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems.*, 20(4):869–916, July 1998.

[117] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, December 2000.

[118] H. Kuchen and J. Striegnitz. Higher-order Functions and Partial Applications for a C++ Skeleton Library. In *Proc. 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 122–130. ACM, 2002.

[119] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[120] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *Proc. IEEE Congress on Evolutionary Computation (CEC'10)*, pages 1–8, 2010.

[121] M. Le Fur, J.-L. Pazat, and F. C. André. Static domain analysis for compiling commutative loop nests. Technical Report RR-2067, 1993.

[122] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[123] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*, pages 398–416. Springer , 1993.

[124] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25 (3):63–79, March 1992.

[125] D. Levine, D. Callahan, and J. Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17(10-11):1223 – 1244, 1991.

[126] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010*, pages 289–296, Febuary 2010.

[127] J. Li and M. Chen. Generating Explicit Communication from Shared-memory Program References. In *Proc. 1990 ACM/IEEE Conference on Supercomputing*, pages 865–876. IEEE Computer Society Press, 1990.

[128] J. Li and M. Chen. Index domain alignment: minimizing cost of cross-referencing between distributed arrays. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation, 1990.*, pages 424–433, Oct 1990.

[129] G. Liao, K. Datta, and Th. Willke. Gunther: Search-Based Auto-Tuning of MapReduce. In *Euro-Par 2013 Parallel Processing*, pages 406–419. Springer Berlin Heidelberg, 2013.

[130] A. W. Lim, G. I. Cheong, and M. S. Lam. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *Proc. 13th International Conference on Supercomputing*, pages 228–237. ACM, 1999.

[131] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214. ACM, 1997.

[132] C. Lin and L. Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, pages 96–114. Springer , 1994.

[133] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[134] R. Loogen, Y. Ortega-Mallén, and R. Peña marÍ. Parallel functional programming in Eden. *Journal of Functional Programming*, 15:431–475, 5 2005.

[135] D.B. Loveman. High performance Fortran. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(1):25 –42, 1993.

[136] C. Lucchesi. *The Undecidability of the Unification Problem for 3rd Order Languages*. Department of Applied Analysis and Computer Science, Faculty of Mathematics, University of Waterloo, 1972.

[137] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[138] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5, April 1999.

[139] B. Mandelbrot. *Fractals and chaos: the Mandelbrot set and beyond*, volume 3. Springer, 2004.

[140] S. Marlow and A. Gill. Happy: The parser generator for haskell, 2004.

[141] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[142] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[143] J. Melton. SQL language summary. *ACM Comput. Surv.*, 28(1):141–143, 1996.

[144] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[145] R. Milner. The Polyadic -Calculus: a Tutorial. In *Logic and Algebra of Specification*, volume 94, pages 203–246. Springer Berlin Heidelberg, 1993.

[146] R. Mistry and S. Misner. *Introducing Microsoft SQL Server 2008 R2*. Microsoft Press, 2010.

[147] MySQL. Overview of Partitioning in MySQL, 1997.

[148] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly Media, 1996.

[149] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6:40–53, March 2008.

[150] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, pages 533–546. Springer , 1999.

[151] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[152] NVIDIA. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.

[153] M. Odersky. The Scala Language Specification version 2.9. May 2011.

[154] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific Computations on Modern Parallel Vector Systems. In *Proc. 2004 ACM/IEEE Conference on Supercomputing*, page 10. IEEE Computer Society, 2004.

[155] D. M. Olsson and L. S. Nelson. The Nelder-Mead Simplex Procedure for Function Minimization. *Technometrics*, 17(1):45–51, 1975.

[156] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. Yahoo, ACM, 2008.

[157] S. Papadomanolakis and A. Ailamaki. AutoPart: automating schema design for large scientific databases using data partitioning. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 383–392, 2004.

[158] S. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[159] S. Peyton Jones. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Programming Languages and Systems*, pages 138–138. Springer, 2008.

[160] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *Proc. 6th international conference on Advanced functional programming*, pages 267–305. Springer-Verlag, 2009.

[161] B.C. Pierce. *Types and Programming Languages*. Mit Press, 2002.

[162] B.C. Pierce. *Advanced Topics In Types And Programming Languages*. Mit Press, 2005.

[163] S. J. Plimpton and K. D. Devine. MapReduce in MPI for Large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[164] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, 2010.

[165] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[166] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[167] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel and Distributed Systems*, 2(4):472–482, Oct 1991.

[168] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'13)*, pages 13–24, 2007.

[169] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proc. IEEE*, 76(3):259–269, Mar 1988.

[170] C. Reichenbach, Y. Smaragdakis, and N. Immerman. PQL: A Purely-Declarative Java Extension for Parallel Programming. In *ECOOP 2012 - Object-Oriented Programming*, pages 53–78. Springer Berlin Heidelberg, 2012.

[171] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[172] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: harnessing the power of parallelism in a pile-of-PCs. In *Proc. IEEE Aerospace Conference, 1997.*, volume 2, pages 79–91, February 1997.

[173] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[174] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[175] T. Saidani, J. Falcou, C. Tadonki, L. Lacassagne, and D. Etiemble. Algorithmic Skeletons within an Embedded Domain Specific Language for the CELL Processor. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09).*, pages 67–76, September 2009.

[176] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, Sung-Eun Choi, and B. L. Chamberlain. Global Data Re-allocation via Communication Aggregation in Chapel. In *IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12)*, pages 235–242, 2012.

[177] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 language specification. Technical report, May 2011.

[178] V. Sarkar and D. Cann. POSC - a partitioning and optimizing SISAL compiler. In *Proc. 4th international conference on Supercomputing*, pages 148–164. ACM, 1990.

[179] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proc. 1986 SIGPLAN symposium on Compiler construction (CC'86)*, pages 17–26. ACM, 1986.

[180] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, 15:615–650, 7 2005.

[181] R. G. Scarborough and H. G. Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, 30(2):163 –171, march 1986.

[182] S. B. Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(06):1005–1059, November 2003.

[183] S. Schulz. System Description: E 1.8. In *Proc. 19th LPAR, Stellenbosch*, volume 8312, pages 735–743. Springer, 2013.

[184] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., 1986.

[185] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 23–34, 1979.

[186] J. Shin, M. W. Hall, J. Chame, Ch. Chen, P. F. Fischer, and P. D. Hovland. Speeding Up Nek5000 with Autotuning and Specialization. In *Proc. 24th ACM International Conference on Supercomputing*, pages 253–262. ACM, 2010.

[187] B. Singer and M. Veloso. Stochastic Search for Signal Processing Algorithm Optimization. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 41–41, 2001.

[188] A. Sinkarovs and S.-B. Scholz. Semantics-preserving data layout transformations for improved vectorisation. In *Proc. 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 59–70. ACM, 2013.

[189] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–the complete reference*. MIT Press (Cambridge, Mass.), Feb 1996.

[190] S. Srinivasan. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Technical Report UCAM-CL-TR-769, February 2010.

[191] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 Object-Oriented Programming*, pages 104–128. Springer, 2008.

[192] J. Srot, D. Ginhac, and J.-P. Drutin. SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In *Parallel Computing Technologies*, pages 296–305. Springer, 1999.

[193] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[194] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *Proc. 2005 ACM/IEEE Conference on Supercomputing*, page 57. IEEE Computer Society, 2005.

[195] P. Tang and J. N. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *Proc. 8th International Conference on Supercomputing*, pages 44–53. ACM, 1994.

[196] The glorious Glasgow Haskell compilation system users guide version 7.6.1, 2007.

[197] A. Tiwari, Chun Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*, pages 1–12, 2009.

[198] A. Tiwari, V. Tabatabaee, and J. K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Computing*, 35(89):475 – 492, 2009.

[199] T. Tsuda and Y. Kunieda. V-Pascal: an automatic vectorizing compiler for Pascal with no language extensions. In *Proc. 1988 ACM/IEEE conference on Supercomputing*, pages 182–189. IEEE Computer Society Press, 1988.

[200] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Mathematical Society*, s2-42(1):230–265, 1937.

[201] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

[202] University of Southampton. The Iridis Compute Cluster, 2014.

[203] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[204] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36:20–34, December 2001.

[205] T. L. Veldhuizen. C++ templates as partial evaluation. *CoRR*, cs.PL/9810010, 1998.

[206] M. Weiland. Chapel, Fortress and X10: novel languages for HPC, HPCx Consortium. Technical report 0706, October 2007.

[207] R. C. Whaley. ATLAS Version 3.9: Overview and Status. In *Software Automatic Tuning*, pages 19–32. Springer New York, 2010.

[208] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proc. 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[209] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(12):3 – 35, 2001.

[210] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[211] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Distributed and Parallel Databases*, pages 68–77, 1991.

[212] N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 1971.

[213] H. Xi. Dependent ML An approach to practical programming with dependent types. *Journal of Functional Programming*, 17:215–286, 3 2007.

[214] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. (1111.2249), Nov 2011.

[215] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1029–1040. ACM, 2007.

[216] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and Y. M. Others. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13): 825–836, 1998.

[217] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. ACM SIGOPS 22nd symposium on Operating systems principles*, pages 247–260. ACM, 2009.

"Better is the end of a thing than its beginning, and the patient in spirit is better than the proud in spirit." Ecclesiastes 7:8