SUPPORTING HIGH-LEVEL, HIGH-PERFORMANCE PARALLEL PROGRAMMING WITH LIBRARY-DRIVEN OPTIMIZATION

BY

CHRISTOPHER RODRIGUES

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei W. Hwu, Chair
Professor Vikram Adve
Adjunct Assistant Professor Matthew Frank
Associate Professor Steven Lumetta

ABSTRACT

Parallel programming is a demanding task for developers partly because achieving scalable parallel speedup requires drawing upon a repertoire of complex, algorithm-specific, architecture-aware programming techniques. Ideally, developers of programming tools would be able to build algorithm-specific, high-level programming interfaces that hide the complex architecture-aware details. However, it is a monumental undertaking to develop such tools from scratch, and it is challenging to provide reusable functionality for developing such tools without sacrificing the hosted interface's performance or ease of use. In particular, to get high performance on a cluster of multicore computers without requiring developers to manually place data and computation onto processors, it is necessary to combine prior methods for shared memory parallelism with new methods for algorithm-aware distribution of computation and data across the cluster.

This dissertation presents Triolet, a programming language and compiler for high-level programming of parallel loops for high-performance execution on clusters of multicore computers. Triolet adopts a simple, familiar programming interface based on traversing collections of data. By incorporating semantic knowledge of how traversals behave, Triolet achieves efficient parallel execution and communication. Moreover, Triolet's performance on sequential loops is comparable to that of low-level C code, ranging from seven percent slower to 2.8× slower on tested benchmarks. Triolet's design demonstrates that it is possible to decouple the design of a compiler from the implementation of parallelism without sacrificing performance or ease of use: parallel and sequential loops are implemented as library code and compiled to efficient code by an optimizing compiler that is unaware of parallelism beyond

the scope of a single thread. All handling of parallel work partitioning, data partitioning, and scheduling is embodied in library code. During compilation, library code is inlined into a program and specialized to yield customized parallel loops. Experimental results from a 128-core cluster (with 8 nodes and 16 cores per node) show that loops in Triolet outperform loops in Eden, a similar high-level language. Triolet achieves significant parallel speedup over sequential C code, with performance ranging from slightly faster to 4.3× slower than manually parallelized C code on compute-intensive loops. Thus, Triolet demonstrates that a library of container traversal functions can deliver cluster-parallel performance comparable to manually parallelized C code without requiring programmers to manage parallelism. This programming approach opens the potential for future research into parallel programming frameworks.

TABLE OF CONTENTS

CHAPTER 1 Introduction 1
1.1 Container Traversal as a Programming Abstraction 4
1.2 Library Functions as an Implementation Mechanism for Parallel Programming Abstractions 6
1.3 High-Level Abstractions of Distributed Memory 12
1.4 Challenges in Producing High-Performance Parallel Code 14
1.5 Summary of Contributions 24
1.6 Organization of This Dissertation 26

CHAPTER 2 Introduction to Functional Programming 28
2.1 Conventions 28
2.2 Reduction Semantics 30
2.3 Loops and Higher-Order Functions 33
2.4 Data Structures 35
2.5 Divergence 38
2.6 Types 39
2.7 Polymorphism 42

CHAPTER 3 The Triolet Programming Language 44
3.1 Types 45
3.2 Statements 46
3.3 Expressions 48
3.4 Introduction to Container Functions 50
3.5 Containers 55

CHAPTER 4 Compiling Triolet Code 59
4.1 Core Language 61
4.2 High-Level Optimizations 69
4.3 Backend 73

CHAPTER 5 Optimization-Friendly Container Library Design 77
5.1 Loop Fusion Background 78
5.2 Hybrid Iterators 85
5.3 Imperative Algorithms 89
5.4 Multidimensional Iterators 89

5.5 Parallelism 91
5.6 Array Partitioning 92

CHAPTER 6 Evaluation 95
6.1 Eden Overview 96
6.2 Source Code Size 97
6.3 Sequential Execution Time 98
6.4 Parallel Speedup and Scalability 99
6.5 Breakdown of Parallel Execution Time 106

CHAPTER 7 Related Work 109
7.1 Parallel Loop Languages 113
7.2 Vector Languages 114
7.3 Collection Libraries 115
7.4 Query Languages 115
7.5 Functional Data-Parallel Skeletons 116
7.6 Other Data-Parallel Skeletons 117

CHAPTER 8 Conclusion 118
8.1 Future Directions 120

REFERENCES 122

CHAPTER 1

Introduction

Software developers want more performance. Hardware improvements have historically driven a steady increase in processor speed that enabled users to speed up their software simply by migrating to newer, faster processors. Recently, however, processor speeds have stagnated due to physical limits on power consumption and dissipation. Meanwhile, multi-core processors have proliferated, and networked computers have become commonplace. As software developers can no longer wait for faster hardware, many turn instead to parallel processing to meet their performance goals.

Many programs contain performance-critical loops that can be converted to parallel form. A parallel loop executes many instances of the same task concurrently. While parallel execution speeds up a workload by partitioning it across many processors, it also incurs overhead to launch tasks, move data, and combine results. Some components of the overhead do not get faster as the number of processors is scaled up, limiting the maximum speedup that a program can achieve through parallel execution. The need to mitigate overhead is one factor that makes parallel programming harder than sequential programming.

A large repertoire of parallelization techniques has been catalogued [1, 2]. Parallel associative reduction serves as one example. In sequential imperative languages, reductions are typically written as a loop that updates an accumulator in each iteration. Such a loop can be parallelized. To produce correct results, the parallelized loop must prevent interleaving of accumulator updates. For instance, decoupled software pipelining can be used to parallelize while keeping accumulator updates in their original order [3]. However, scalability is limited because multiple accumulator updates cannot happen simultaneously. Reduction trees are

a well-known scalable way of implementing reductions. In a (two-level) reduction tree, each processor performs a sequential reduction using its own, locally stored accumulator. The results from all processors are sent to the main processor, which performs another reduction to combine the processors' results. A reduction tree computes parts of the reduction in parallel, enabling greater speedup than the simpler loop with a single accumulator. The best reduction tree structure depends on the architecture of the system where it executes: while a two-level reduction tree works well on a moderately sized multicore processor, larger systems typically use deeper reduction trees to avoid a bottleneck at the final step. Reduction trees exemplify common properties of parallelization techniques: they apply to a specific class of algorithms, their implementation depends on hardware organization, and they reduce the overhead of naïve parallelization at the expense of additional complexity.

A commonly used low-level, high-performance parallel software development methodology is for a developer to manually implement efficient parallelization techniques, starting from a sequential C loop, as needed to meet his or her performance goal. This methodology sets a reference point against which to evaluate a high-level parallel programming language. A high-level language should provide an easy-to-use interface to high-level parallelization techniques, while still providing substantial parallel speedup over sequential C code. Many high-level languages and libraries satisfy this goal for shared memory environments by efficiently using threads and shared-memory communication. Fewer solutions exist for distributed memory environments, because the communication latency of distributed hardware is not easily hidden behind a high-level abstraction. Fewer still exploit the hybrid of shared and distributed memory found in clusters of multicore computers. A simple high-performance programming model would make it easier to speed up programs using clusters.

The system presented in this dissertation, Triolet, demonstrates a high-level and high-performance approach to implementing distributed parallelization techniques. Triolet consists of a programming language, compiler, and library. The programming language and library adopt a high-level programming style based on container traversals, a common programming abstraction. Internally, the library uses high-level knowledge of data structure

traversal operations to assemble efficient sequential and parallel loops. Efficiency comes from distributing work in a way that reduces communication requirements, avoiding movement of unused data, traversing data in ways that exploit cache locality, and combining results in ways that mitigate communication and computation latency. On a 128-core cluster, computationally intensive loops written in Triolet were found to yield a speedup of 9.6–99× relative to simple loops in sequential C. The Triolet code is shorter and simpler than manually parallelized C code. While manually parallelized C is sometimes faster (ranging from slightly slower to 4.3× faster than Triolet), Triolet delivers much of the achievable speedup without requiring a developer to invest time implementing hand-optimized parallel algorithms.

Triolet advances beyond prior systems in two ways. First, it demonstrates that distributed container traversals can be implemented efficiently as library functions. In doing so, it shows that high-level parallel operations can compile to efficient code without compiler support for parallelism. Instead of generating parallel code within the compiler, parallel execution strategies are implemented in library code. The compiler inlines and specializes the library code by applying general-purpose optimizations. Such specialization is possible because all the information needed to determine a typical program's loop structure is available at compile time. The net effect is similar to a parallelizing compiler that performs loop transformations and inserts communication code. Decoupling the implementation of parallelism from the design of the compiler makes it easier to develop and explore new parallelization techniques: changes are made by writing or modifying library code.

Second, Triolet's library uses a flexible mechanism for data and work distribution that reduces communication overhead for some distributed algorithms expressed as data structure traversals. In prior implementations of distributed traversal, a given loop would use a single partitioning strategy for both work and data. To express algorithms that traverse multiple data structures in different ways (as in matrix multiplication) or that dynamically generate inputs from some initial set (as in modeling physical interactions between nearby objects), a programmer would have to restructure loops and data structures, which adds overhead and complexity, or else explicitly manage data and work distribution in a lower-level pro-

gramming model. Triolet's container traversal functions internally build data distribution strategies and nested loops instead of relying only on pre-built partitioning strategies. Complex traversal patterns and loop nests can be assembled through compositions of library function calls.

The next four sections discuss data structure traversals (Section 1.1), the design of libraries for optimizations (Section 1.2), prior approaches to data distribution (Section 1.3), and elements of Triolet's implementation that result in high parallel performance (Section 1.4).

1.1 Container Traversal as a Programming Abstraction

Triolet adopts a high-level programming style that uses container traversals as abstractions of parallelizable loops. Container traversal interfaces have appeared many times in programming languages and libraries. This section presents major categories of container traversal interfaces and discusses their common properties.

Traversals embody common patterns of computation over collections of data. Sipelstein et al. [4] describe and compare traversal functionality in various programming languages. As an example, it is common to perform an operation independently on each element of a collection, yielding a collection of computed results. Given a collection and an operation to perform on its elements, the library function *map* carries out this task in many programming languages. Each input is retrieved and each output stored by `map`. It applies the given operation to each input to generate outputs. Traversals are suitable for expressing parallelism because the semantics of common traversals do not imply an evaluation order. Since each result of a map is independent of the others, its meaning is not affected by whether elements are processed sequentially or in parallel.

The development of large vector-parallel processors fueled interest in *data-parallel* programming [5]. The term "data-parallel" is commonly associated with the execution on vector processors but is sometimes used independently of the execution model. Many data-parallel programming languages use array traversal to express parallelism [6, 7, 8, 9]. When array

traversals are executed on vector hardware, individual array elements are processed concurrently on different scalar processing units. Vector processors are SIMD, or single instruction, multiple data, meaning that the same operation is executed concurrently on all scalar processing units. If the processing of an array element involves control flow, it can be translated into SIMD form for execution on vector processors [6]. Recent vector languages have targeted GPUs and multicore CPUs [7, 8, 9, 10]. However, conforming to a SIMD execution model often requires extra computation for storing and reorganizing arrays of data. This overhead was acceptable on vector processors, but can be severely limited by memory bandwidth bottlenecks on modern CPU and GPU architectures. Recent research has begun to address this issue [7, 11].

A related line of parallel work uses array traversal to express parallel loops [12, 13, 14]. Individual array elements may be processed concurrently by different threads. To reduce overhead, a thread typically processes a section of an array. The primary difference between vector and loop parallelism is that the latter permits arbitrary control flow in loop bodies.

Container traversals embody sequential loops when applied to lazily generated collections of values, called *iterators* in this dissertation. Iterators are also known as streams or lazy lists. In functional languages, the distinction between iterators, linked lists, and vectors is blurred by optimizations that eliminate temporary data structures [8, 10, 13, 15, 16, 17, 18], the use of lazy lists to interleave execution of loops, and implicit conversions between representations [16, 17]. These techniques aim to reduce memory usage and/or execution time by selecting the representation with the most suitable evaluation behavior and data storage format for the context where it is used. Language INtegrated Queries (LINQ) is an iterator library designed to subsume database queries, which would conventionally be written in the domain-specific Structured Query Language [19]. CLU (short for "cluster") introduced iterators as a mechanism for writing imperative loops over containers [20]. Iterators in C++ and Java are a more general class of programming interfaces derived from CLU iterators. Iterators have also been used as a representation of parallel loops [19, 21].

Arrays, vectors, lists, and iterators can be generalized into a common *container* abstrac-

tion. A container is a homogeneous collection of values organized into a logical shape, such as a 2D array of a particular size. The notion of a container abstracts over differences in storage format and evaluation semantics. At minimum, a container can be accessed by traversing it, which reads each of its elements in a loop. Traversals also abstract over evaluation semantics. An implementation of `map`, for instance, may execute sequentially or in parallel, and it may generate all its results immediately or individual results when demanded. The result value computed for each output position is the same regardless of these choices, and the notion of traversal captures this common meaning. The notion of containers makes it possible to reason about containers (including arrays, vectors, lists, and iterators) in a common way.

Container traversals are a broadly applicable way of writing loops. They express potential parallelism and convey useful information about access patterns, both properties being useful for implementing parallel loops. The design of Triolet uses a custom iterator design as its basic representation of parallelizable loops.

1.2 Library Functions as an Implementation Mechanism for Parallel Programming Abstractions

How should tool developers provide high-level interfaces to parallelization techniques? To parallelize code efficiently, it is necessary to transform code to suit the hardware on which it will run, and this has traditionally been the domain of compilers. However, compilers are complex pieces of software that are difficult to develop and extend. Integrating support for new parallel abstractions into an existing compiler is a slow process, as evidenced by the decades-long evolution of parallel features in long-lived languages or language extensions such as Fortran, C++, Sisal, and OpenMP. This difficulty has motivated several lightweight approaches illustrated in parts (b), (c), and (d) of Figure 1.1.

Lightweight interfaces consolidate the implementation of high-level parallel features into a component of the compile-time toolchain or run-time library, relying on a preexisting compiler infrastructure to do most of the work. Many parallelization techniques need only a

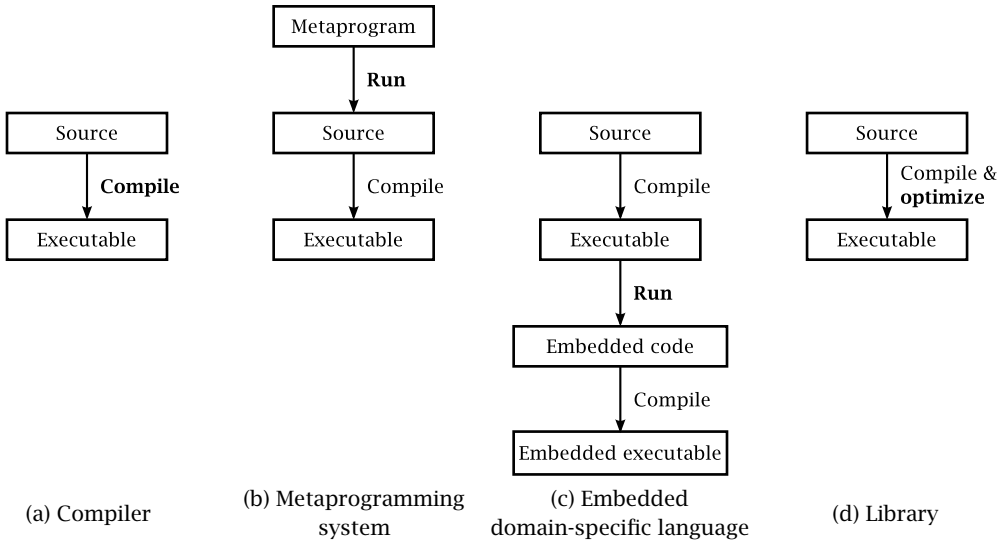


Figure 1.1: Steps of compilation and execution in different parallel programming approaches. Each arrow is a step of compilation or execution. The step labeled in bold is where high-level parallel abstractions are transformed into low-level code. Approaches (b), (c), and (d) separate parallel software abstractions from the compiler.

limited degree of control over the program code. In such cases, decoupling high-level parallel features from other aspects of compilation makes them easier to develop. Classes of lightweight interfaces are discussed below.

Embedded domain-specific languages (EDSLs) [22] (Figure 1.1(c)) are programming languages whose interfaces resemble libraries. To a user, an EDSL exposes abstract data types and operations on them. Operations do not compute their results immediately, but generate embedded code for computing results. This code is executed when a result needs to be extracted for use in the host language. For instance, an "array of integers" object may comprise code that computes the desired array. An "array summation" function that returns the sum as a host-language numeric value may compute the sum by extracting the code from its array argument, combining it with reduction tree code that was written by the developer of the EDSL, compiling it, and executing it [23]. EDSLs have been developed to help developers use restricted programming models including GPU shaders [24, 25, 26], vector processors [24, 25, 26, 27], and database queries [19, 28, 29]. Delite [30] is a framework for building parallel EDSLs.

Metaprogramming systems are programs that generate source code in the form of text or abstract syntax trees. The code they generate is compiled and linked along with the remainder of a program as shown in Figure 1.1(b). A metaprogramming library may assemble a reduction tree, for instance, by injecting problem-specific value-generating and result-combining code into a parallel algorithm template supplied by the library. Owing to the popularity of C++, a number of parallel loop abstractions have been developed using C++ template metaprogramming [31, 32, 33].

High-level parallel features can be packaged into *library functions*, which is the approach taken in this work. Cole popularized this approach, calling such libraries "algorithmic skeletons" [34]. In this approach, a handwritten implementation of a reduction tree would be provided as a library function. A user would customize the reduction to a specific problem by passing value-generating and result-combining functions as arguments to the reduction function. These functions would be called during the execution of the reduction tree. Note that this functional programming style is different from procedural libraries such as MPI (an abbreviation of "Message Passing Interface"), which require users to control the parallel execution of all non-library code.

For creators of high-level parallel programming abstractions, libraries are arguably the most constrained approach. The power of compilers, metaprogramming systems, and EDSLs comes from their ability to manipulate programs prior to generating executable code. A library function cannot manipulate the program it is part of, seemingly making it inadequate for high-performance parallel programming abstractions [30]. Nevertheless, libraries can be written so as to induce an optimizing compiler to perform static program transformations. One of the ways that compilers speed up programs is by precomputing values that can be determined at compile time. This strategy of evaluating the known parts of a program in advance is a form of *partial evaluation* [35]. Such compile-time computation of programs can be used like a metaprogramming system to statically transform code [36]. The idea of designing library code to perform well when a compiler optimizes it together with code that uses it exists in programming folklore but has not, to the best of my knowledge, been

named. I call it *library-driven optimization*. Library-driven optimization has been used for generating loops from container traversals in Haskell libraries [14, 17, 36, 37]. The side effect free, statically typed nature of Haskell code facilitates such optimizations by removing hard-to-analyze nonlocal effects and ensuring that optimizations will complete when run exhaustively [38].

For library-driven optimization to be effective, library and user code must cooperate to enable compile-time optimization. Compilers restrict the scope of what they will optimize to limit code size and compilation time. If a library author intends a particular piece of code to be statically evaluated, it must be written in a language subset that the compiler statically evaluates. In Triolet, the subset excludes arrays, externally defined functions, mutable data, and general recursion. Furthermore, code intended for static evaluation should not depend on run-time input, because such data is unknown and prevents partial evaluation. Programs that do not follow these restrictions will optimize poorly and run inefficiently. For instance, the following Triolet code assigns different values to `y` depending on the value of an unknown Boolean `run_time_condition`. Since the value of `y` following the `if` expression may come from either branch, it is statically unknown, which inhibits optimization of `sum`. On the other hand, if the compiler can simplify `run_time_condition` to `True` or `False`, optimization is not inhibited.

```
if (run_time_condition): y = map(foo, x)
else:                    y = map(bar, x)
z = sum(y)
```

These restrictions are not onerous in practice. Many transformations can be expressed in a non-looping, array-free functional language subset, and the loop structure of typical programs does not depend on run-time information.

In library-driven optimization, library functions make optimization decisions based on statically known data in their arguments. Similarly to an EDSL, library functions can embed code into their output that will be extracted by subsequent library function calls. However, unlike an EDSL, a library cannot inspect or modify code. (Libraries can construct, but not examine, functions.) Consequently, library-driven optimization cannot analyze or transform

loop bodies. Many conventional loop transformations are guided by analysis of memory references in a function body, but library code cannot examine the memory references performed by a function. The inability to modify a function body's code rules out loop transformations such as loop distribution, tiling of nested loops, and vectorization of scalar code. Horizontal fusion, which fuses loops that do not interact but have the same control flow, is not possible with library-driven optimization since neither loop can "see" the other. These limitations make sense in light of the library-driven optimization approach. Whereas loop transformations analyze low-level unoptimized loops and convert them into low-level optimized loops, library-driven loop transformations *assemble* low-level optimized loops from high-level specifications.

The EDSL approach can be more powerful than library-driven optimization, but it achieves this at the cost of replicating a compiler's functionality. Whereas library-driven optimization uses the same general-purpose optimization engine to simplify both the static and dynamic parts of a program, embedded domain-specific languages usually contain their own optimization engine [24, 25, 27, 30]. For instance, Delite optimizes its own internal high-level representation of loops, called Delite ops. Delite's optimizer performs constant subexpression elimination, constant propagation, dead code elimination, and code motion in addition to domain-specific optimizations [30]. Delite does these optimizations even though it passes its output to a Scala, C++, or CUDA backend compiler that is likely to perform similar optimizations, presumably because these general-purpose optimizations improve the quality of its domain-specific optimizations.

Metaprogramming engines perform compile-time evaluation like library-driven optimization, but with an explicit distinction between statically and dynamically executed code. The separation of static and dynamic execution allows a developer to control what is computed at compile time instead of relying on a compiler to statically evaluate the right things. However, the control comes at the cost of a more complicated programming model. In the case of metaprogramming iterator libraries, which are used for statically generating loops, iterators are more complicate than the usual notion of lazily generated collections of values

(Section 1.1). In metaprogramming libraries, iterators consist of two components that users manipulate separately. An iterator's static data consists of code that is injected into loop bodies. Its dynamic data consists of the remaining information, which is run-time parameters and inputs to loops. While metaprogramming is an invaluable approach to escape the unpredictability of fickle optimization heuristics, the benefit of introducing guaranteed static evaluation diminishes when compiler optimizations reliably evaluate a large subset of a programming language.

A comparison between user-defined generic functions in an ordinary iterator library and a metaprogramming library demonstrates the extra complexity of metaprogramming. In Haskell, lists serve as iterators. A generic function `augment` for adding a value `x` to each element of a list `ys` can be defined as follows. The type signature on the first line indicates that the function operates on an arbitrary numeric type `a`, and that its parameter and return iterators are both lists of `a` (written `[a]`). In the function body, the call of `map` applies `(x+)`, which adds `x` to a number, to each element of `ys`.

```
augment :: Num a => a -> [a] -> [a]
augment x ys = map (x+) ys
```

Analogous C++ code, using the Thrust iterator library, tracks the code of iterators through C++'s static type system. Iterator code is exposed to users in the form of elaborate types. The C++ code shown below is much more verbose than the Haskell code; however, its verbosity is an effect of C++'s syntax rather than metaprogramming in general. In the code, the definition of `augment` uses the iterator class `transform_iterator` and factory function `make_transform_iterator` from the Thrust library. Two global type definitions are used in writing the return type of `augment`. `AugmentFunc(x)` is the equivalent of the Haskell function `(x+)`. The type alias `Elem<T>` is defined as a shorthand for the type of `T`'s elements. At the metaprogramming level, `augment` operates on iterator code by relating an arbitrary type `Iterator`, which is also the type of the function parameter `ys`, to a more complex type `transform_iterator<AugmentFunc<Elem<Iterator> >, Iterator>`, which is the type of its return value. The return type specifies how the returned iterator's code is composed

from code associated with `transform_iterator`, `AugmentFunc`, and `Iterator`. At run time, `augment` inserts the values of `x` and `ys` into the returned iterator. These values contain run-time information such as the loop bound that is needed when the loop executes.

```
template<typename Elem>
struct AugmentFunc : public unary_function<Elem, Elem> {
  Elem x;
  AugmentFunc(Elem _x) : x(_x) {}
  Elem operator()(Elem y) {return x + y;}
};

template<typename Iterator>
using Elem = typename Iterator::value_type;

template<typename Iterator>
transform_iterator<AugmentFunc<Elem<Iterator> >, Iterator>
augment(Elem<Iterator> x, Iterator ys) {
  return make_transform_iterator(ys, AugmentFunc<Elem<Iterator> >(x));
}
```

The elaborate iterator types in the C++ code express how to generate code when an iterator is used. These details are irrelevant to users who, following the container abstraction, simply want a dynamically generated collection of `Elem`s. More generally, explicit separation of static and dynamic program components is a burden on users.

Libraries, with the aid of compile-time partial evaluation, can perform high-level optimizations similar to what a compiler, EDSL, or metaprogramming system can do. Library code is decoupled from compiler code, which makes libraries more constrained than compilers or EDSLs but also easier to develop. Since library-driven optimizations are expressed as program code rather than metaprogram code, libraries do not require programmers to work with an extra stage of execution as metaprogramming systems do. For these reasons, Triolet uses library-driven optimization for implementing parallelization techniques.

1.3 High-Level Abstractions of Distributed Memory

In a distributed parallel program on a cluster, the latency of network communication is likely to contribute to the program's total running time. The negative effect of latency can be

reduced by transferring needed data as soon as possible and overlapping communication with computation. In a message-passing programming model such as MPI, programmers perform these optimizations by controlling when and how messages are sent. Most higher-level programming models offer additional convenience but still task programmers with deciding when to send messages and where to perform computation, largely because all automatic message-passing schemes trade off some generality and/or efficiency.

The partitioned global address space (PGAS) programming model is popular in general-purpose distributed programming languages [39, 40, 41, 42, 43]. It provides the convenience of a single address space for referencing and accessing data, as in a shared memory system, without hiding the critical performance difference between fast local memory access and slow remote memory access [44]. Programmers explicitly place data in memory domains and execute computation on memory domains. Programmers are expected to optimize communication by placing computation together with data that it uses frequently. Thus, PGAS models simplify, but do not eliminate, the management of data and work distribution.

Some algorithmic skeleton libraries provide a view of distributed data that is reminiscent of the PGAS programming model [45, 32]. As in the PGAS model, code can access the entire distributed data structure through the library API. Local accesses directly access memory, while global accesses may perform network communication.

Distributed shared memory provides the simplicity of shared memory programming on distributed hardware. The best-known approach, called software distributed shared memory (software DSM), resembles a software implementation of cache coherence [46]. Software monitors memory accesses and forwards the contents of written memory regions to processors that use them. However, software DSM has not seen widespread adoption, in part because it aims to be a nearly-invisible system software layer but performs poorly with some usage patterns and does not integrate smoothly with existing software environments [47]. There is room for simpler, domain-specific implementations of shared memory that do not expose these issues.

A much simpler implementation of the shared memory model is possible for parallel

traversals of immutable data structures. Since a traversal expresses a loop with a known access pattern, the data that it will access is known in advance and it can be eagerly copied to the destination. Since data is immutable, there is no need to monitor memory accesses and forward updated data as in software DSM. Inputs are simply copied over the network when a task is launched and results are copied back when it completes. This copying-based shared memory model has been used in several algorithmic skeleton implementations [48, 49, 50, 28]. While these projects demonstrate the feasibility of a copying-based shared memory model, they emphasize ease of use and scalable performance rather than absolute performance. They do not attempt to deliver performance competitive with manually parallelized MPI code.

1.4 Challenges in Producing High-Performance Parallel Code

The performance of manually optimized code in a low-level parallel programming model, such as C with MPI and OpenMP, comes from the combined use of many programming techniques. Inner loops sequentially compute results using mutable private storage. Cores on a cluster node share input and output storage, partition work for cache locality, and avoid communication by computing partial results locally. Cluster nodes partition work to minimize communication, send only necessary data, and avoid communication by computing partial results locally.

A high-performance distributed implementation of container traversals should employ these techniques as well. Armed with high-level information about units of parallel work, access patterns to input data, and result collection strategies, an implementation needs to find efficient parallelization strategies for loops that a programmer has written in terms of traversals. Even with high-level information, it is challenging to reproduce enough low-level programming techniques in a library to approach the performance of manually parallelized code.

Some implementation decisions can be illustrated with a function for evaluating the qual-

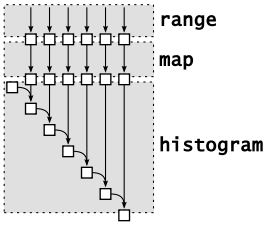
ity of hash functions. A simple way to test the quality of a hash function f is to take a range of input values $x \in \{0 \ldots n-1\}$ and compute the low-order bits of the hash, $f\ x \mod 256$, for each x. Histogramming hash values helps to find undesirable biases in a hash function's output. In Triolet, a function can be defined to perform these operations as follows.

```
def hashDistribution(f, n):
  inputs = range(n)                      # Range from 0 to n-1
  hashed = (f(x) % 256 for x in inputs) # Hash each number modulo 256
  return histogram(256, par(hashed))    # Histogram hashed values
```

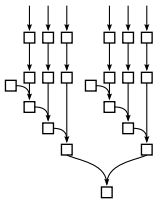
The three steps of the computation are performed by the three lines of the function body. Each line calls a library function that represents a parallelizable loop. The functions can be understood as loops with `n` iterations, though all but the last actually consist of non-looping code that constructs an iterator. The comprehension on the second line is syntactic sugar for a call to `map`. The call to `par` on the last line indicates that the histogram should be parallelized.

Finding an efficient parallel implementation can be framed as a scheduling problem. Each iteration of each loop is a task to be scheduled. An example execution of `hashDistribution` is shown in Figure 1.2(a). Tasks are shown as arrows and data are shown as boxes. Histogram tasks, which have two inputs, appear as two-tailed arrows. While `hashDistribution` executes directly as an optimized parallel loop, it is instructive to describe the intended behavior in terms of a sequence of transformations from simple to optimized code.

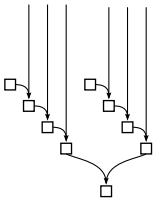
Potential parallelism is limited in Figure 1.2(a) because each histogram task modifies the data produced by the previous task, producing a sequence of dependent tasks. For parallel execution on two cores, input is partitioned into two pieces and a partial histogram is computed on each piece as shown in Figure 1.2(b). This partitioning is chosen by `histogram` using information about loop structure from the `range`, `map`, and `par` library calls. On a cluster, work would be partitioned across cluster nodes in the same way. Because the flow of data through these calls is statically known, compiler optimizations resolve the loop structure statically. The number of processor cores and loop iterations are found dynamically. Compile-time optimizations inline and specialize the relevant library code, consisting in part



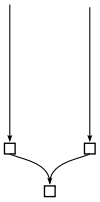
(a) Task graph from source code. Boxes are drawn around tasks created by each skeleton.



(b) After histogram privatization. Histogram tasks are reassociated to yield two groups of independent tasks.



(c) After fusion. Tasks from calls to `histogram` are merged with the `range` and `map` tasks that produce their input.



(d) After blocking. Groups of tasks are merged into coarse-grained tasks.

Figure 1.2: A sample execution of `hashDistribution`, shown as a task graph, as successive optimizations are applied.

of an outer parallel loop over cores and an inner sequential loop over values of `x`. This transformation is analogous to histogram privatization.

Because each `range` and `histogram` task comprises only a few instructions, tasks should be grouped into larger units of work to amortize run-time scheduling overhead and communication costs. Loop fusion groups iterations of one loop with iterations of another (Figure 1.2(c)), reducing the amount of data transferred from one task to another and thereby reducing communication overhead. Again, fusion occurs statically because the loop structure is statically known.

To amortize the overhead of task creation, multiple tasks can be aggregated into one coarse-grained task per core (Figure 1.2(d)). The library does this by executing a core's work in a sequential inner loop. Inlining the loop into the surrounding code enables further optimizations to produce a C-like inner loop containing only arithmetic, data structure accesses, and a call to `f`.

Library-driven optimization relies on the ability to compose optimized loops out of simpler components. On shared memory, the primary concern is work distribution—specifically,

loop fusion and blocking. General approaches in this area separate the creation of parallel work from the decision of how to fuse and block loops by representing loop bodies as indexed computations. That is, a loop is a computation parameterized over an index space (called an iteration space). Examples of indexed loop representations are Single Assignment C's with-loops [13] and Repa's delayed arrays [14]. An indexed representation represents a collection of tasks with an unspecified schedule. Fusion, then, builds a loop whose body does the work of multiple loop bodies. Work is distributed by executing the loop body at each element of its index space and collecting the results.

A cluster implementation also needs to manage data distribution. The most efficient parallelization strategy shares storage within a node and passes messages across nodes. Triolet uses sharing and message passing together (Section 1.4.1). In prior distributed implementations of container traversals, work and data distribution are the same [48, 49, 50, 28]. A single data structure would be partitioned across processors and work would be assigned to the processor that has its input data. More complex data movement schemes, such as sorting by key, still tie work and data distribution together. Some parallel loops are not efficient to execute in this manner, either because tasks use multiple inputs or because a unit of data produces a variable number of subtasks. Triolet's iterator design separates data decomposition from work decomposition (Section 1.4.2) and permits subtask creation (Section 1.4.3) to support writing a broader variety of loops in terms of container traversals. Each of these features generalizes prior container traversal implementations so as to confer high performance onto a wider range of programs.

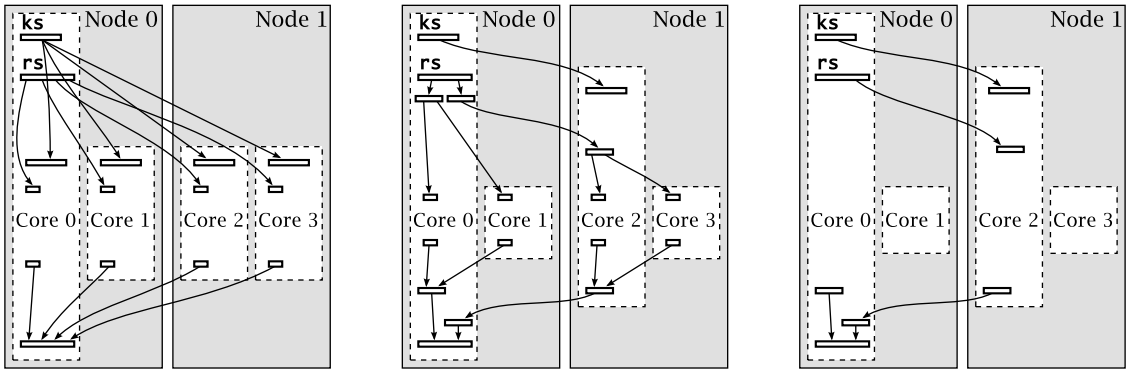
1.4.1 Hybrid Shared and Distributed Parallelism

Few container traversal interfaces exploit the memory organization of a cluster of multiprocessors. Either the implementation uses shared memory, in which case parallelism is limited to one cluster node, or it uses message passing, in which case it redundantly copies data between cores on the same node. This is unfortunate, as extra copying takes time and the replicated data increases cache pressure. Of prior implementations, only Muesli uses

shared-memory and distributed-memory parallelism together [45]. Figure 1.3 illustrates how a well-designed data distribution mechanism avoids redundant copying in the parallel map from the benchmark `mri-q`. The data distribution mechanism is responsible for supplying a part of `xs` and all of `ks` to each core and collecting the outputs into a single array. The runtime behavior of a purely message-passing framework is shown in Figure 1.3(a). Because data is always copied between cores, even cores on the same node, this system creates extra copies of `ks` and sends them over the network, adding communication overhead and cache pressure.

Efficient parallel execution is possible with a two-level parallel work decomposition that partitions data first across nodes and then across cores within a node. A typical divide-and-conquer framework, adapted to this two-level work decomposition, would behave as shown in Figure 1.3(b). The shared array `ks` is distributed to all nodes, not all cores. The input array `xs` and the output array are subdivided across nodes and further subdivided across cores. To separate the responsibilities of work decomposition and data partitioning, divide-and-conquer frameworks typically express division and combining as functions. For arrays, the functions would construct a new array at each divide or combine step, resulting in the unnecessary copying shown in the figure. HDC (the Higher-Order Divide and Conquer skeleton library) gives up flexibility for performance by providing skeletons with built-in, optimized partitioning for 1D arrays [48].

Triolet's work-partitioning strategy logically subdivides arrays without copying them, eliminating redundant per-core and per-node copies of input data (Figure 1.3(c)). Subarrays are represented by *slice objects*, which are an array-like container. A slice object holds a reference to the original array and the subarray bounds. Its interface imitates the behavior of a new array extracted from the original one, while avoiding the overhead of creating a copy. When using a slice object on the node where it is created, the original array is accessed. When copying a slice object from one node to another, only the subarray is copied. Since slice objects for data distribution are created and used over the scope of a parallel loop, compile-time optimizations can normally extract their fields and inline their methods. Inner loops



(a) Flat distributed parallelism. Data is copied between cores whether or not they inhabit the same node.

(b) Generic divide-and-conquer parallelism. Data is shared on a node. Array contents are copied when dividing and combining.

(c) Triolet's divide-and-conquer parallelism. Array contents are not copied when dividing. The output array is shared at the node level.

Figure 1.3: Data movement during the execution of a parallel map with different runtime data distribution mechanisms. Time evolves downward. Arrows show data copying events.

after optimization directly access the slice object's underlying array. Additionally, parallel map creates one output array per node, with each core writing part of the array. Per-core output arrays are not created. Other algorithms, such as reductions, still create per-core output data. The resulting pattern of copying is close to what a programmer would produce using C with OpenMP and MPI.

The Triolet runtime serializes data in order to permit communication of arbitrary data. Serialization packs data into a single block of bytes that can be sent as an MPI message. Serialization introduces an extra copying step into inter-node communication, which can impact the performance of communication-bound loops. Triolet shares this overhead with other systems that provide a shared memory view by copying objects.

1.4.2 Composing Data Distributions

When writing algorithms as data structure traversals, it is common to include data-rearranging transformations on input data. Perhaps the most frequent instance of this is zipping arrays together. For example, the benchmark `mri-q` contains a loop `(sqrt(x*x + y*y) for (x, y) in zip(reals, imags))` that computes the magnitudes of complex numbers

whose real and imaginary parts are stored in separate arrays `reals` and `imags`. The real and imaginary arrays are traversed together by first zipping them into one container. The function `zip` combines its two arguments elementwise, producing a container holding pairs of values. In this case, real and imaginary values are combined into complex numbers. Actually building an array of complex numbers would involve unnecessary work. This work is eliminated in the shared memory domain by lazily rearranging data at the scale of individual elements [17]. Instead of building an array, `zip` returns an iterator containing code that reads from each input and constructs a pair of values. This code is executed on demand to read complex numbers without rearranging stored data.

To avoid rearranging data on distributed memory hardware in a copying-based shared memory model, data-rearranging functions like `zip` should also customize how data is copied between distributed memory domains. In the case of two zipped arrays, the input data is not a single array, but two. It should be distributed by partitioning both arrays across nodes. Triolet generalizes this idea by representing a partitioning strategy as a function that extracts the input data needed by a given subset of a loop's iteration space. A subset of a loop's iteration space is represented by an index range. Triolet's iterators contain partitioning functions, and some traversal functions build new partitioning functions from old ones as they reorganize data.

Parallel, 2D blocked dense matrix multiplication composes several data distribution functions to efficiently distribute a block of each input array to each cluster node using two lines of code. The matrix product AB is computed (after transposing B for faster memory access) by evaluating dot products of rows of A with rows of B^T:

```
zipped_AB = outerproduct(rows(A), rows(BT))
AB = [dot(u, v) for (u, v) in par(zipped_AB)]
```

Here, `dot` is taken to be defined as sequential code so that the computation of a single output block is sequential. The block-based data partitioning is composed from several library function calls that change the logical arrangement of array data. The calls to the library function `rows` reinterpret the 2D arrays A and B^T as 1D iterators over array rows, where each array

row is a 1D iterator over elements. The zip-like library function `outerproduct` creates a 2D iterator paring rows of A with rows of B^T. Together, the functions on the first line determine a block distribution of input data. Iterators returned by `rows` associate each task with the corresponding array row. From these iterators, `outerproduct` associates each 2D matrix block with the rows of A and B corresponding, respectively, to the block's vertical and horizontal extent. When parallel tasks are launched by the comprehension on the second line of code, each task will be sent only the array rows that it needs to compute its output.

In the absence of support for general data partitioning strategies, a programmer can still write additional code to partition data by extracting the data needed by each task. However, manual partitioning entails redundant copying reminiscent of that discussed in the previous section. A programmer must also manually tile loops in order to have coarse-grained units of work that take blocks of data as input. Since manually written partitioning occurs outside a parallel loop, it limits parallel scalability. For traversals using regular data reorganization patterns such as those of `rows`, `zip`, and `outerproduct`, Triolet's use of data partitioning functions avoids copying in the same way as for a simple array traversal.

1.4.3 Nested Parallel Loops

In many algorithms expressed as container traversals, a single container element may produce a variable number of units of work, each of which produces an output value. In terms of loops, such algorithms are nested loops where the inner and outer loops cooperate to combine results. This cooperative behavior distinguishes nested traversals from the simpler situation where an inner loop merely produces a result value. The difficulty of executing this form of container traversal efficiently has motivated research in sequential compilation techniques [18]. Prior approaches could not execute such nontrivial work distributions as a single parallel loop. They would run multiple parallel phases, redistributing intermediate data so that each phase performs one unit of work per container element. For some algorithms, this can produce excessively fine-grained runtime tasks and/or increase a program's asymptotic memory requirements.

An example of a loop with variable-length outputs comes from particle-based simulation. Some simulation algorithms build a *Verlet list* or *neighbor list* holding all pairs of particles that are separated by less than a given distance [51]. Assuming that a `distance` function has been defined, particle pairs can be computed by visiting each pair of particles and, if the distance is small, adding the pair to a list.

```
def neighbors(xs):
  xs_indexed = zip(xs, range(len(xs)))
  return [(x, y)
          for (x, i) in par(xs_indexed)
          for y      in xs[i+1:]
          if distance(x, y) < 2.0)]
```

Triolet uses a conventional translation of comprehensions into library function calls [52]. Each `for` and `if` clause is a separate operation. The list comprehension following the `return` keyword can be read as a loop nest containing two loops and a guard. The outer `for` clause (`for (x, i) in par(xs_indexed)`) loops over every element `x` of `xs` along with its index `i`. The inner `for` clause (`for y in xs[i+1:]`) loops over every element of `xs` that is at index `i+1` or greater; or, in other words, it transforms every pair $\langle x, i \rangle$ from `xs` into a sequence of triples $\langle x, i, y_1 \rangle, \langle x, i, y_2 \rangle, \ldots$, where each y_k is taken from `xs`. Visiting only higher indices ensures that each pair of particles is visited only once. The `if` clause, or guard, skips loop iterations for which the distance test evaluates to `False`. In the remaining iterations the tuple `(x, y)` is put into an output list. Loop fusion is a necessary first step in order to avoid the overhead of actually constructing the $\langle x, i, y \rangle$ triples. The fused loop, when executed sequentially, is similar to the following Python pseudocode.

```
r = []
for (x, i) in xs_indexed:
  for y in xs[i+1:]:
    if distance(x, y) < 2.0:
      r.append((x, y))
return r
```

Triolet's loop representation is extended to support nested loops. A unit of work can produce either a single result or an iterator over multiple results. These "inner" iterators become nested loops after compile-time optimizations.

While several frameworks support parallel execution and/or fusion of skeletons with variable-length outputs, no prior framework produces parallelized and fused loops with low overhead and without requiring custom compiler-based loop transformations. Some frameworks employ rewrite-based fusion that replaces known combinations of functions with prebuilt, optimized functions [6, 8, 9, 53, 54]; however, this approach is limited by a library implementor's ability to anticipate and manually optimize useful patterns. NESL parallelizes but cannot fuse these loops [6]. Scala's parallel collections library uses lazily executed iterators to achieve the same execution order as a fused loop [21]. However, its iterators are not statically simplified to loops, resulting in significant overhead relative to when domain-specific optimizations are applied [55, 56]. Traversals with variable-length outputs have been fused by extending an EDSL with a new loop representation and new optimizations [56].

1.4.4 Irregular and Multidimensional Loops

Container traversal APIs for 1D containers and multidimensional arrays are both useful, but are rarely found together. The division between interfaces arises partly because some operations are only meaningful in one setting and, consequently, some techniques for implementing traversals are primarily used in one setting. Triolet supports both sets of functionality through the same iterator interface. Though not directly related to distributed parallelism, integrating both sets of functionality enables both 1D and multidimensional loops to take advantage of the same parallel library code.

Traversal functions supporting variable-length outputs, and the issues involved in executing them efficiently, are only relevant to 1D containers. The `filter` library function is an example that conditionally discards elements of a container. Whereas discarding an arbitrary subset of a list's elements yields a list (by packing the remaining elements together), discarding an arbitrary subset of an N-dimensional array does not, in general, yield an N-dimensional array, as an arbitrary subset of array elements cannot be packed into an array while preserving their relative positions. Consequently, functions involving variable-length outputs are generally not provided for multidimensional arrays, and optimizations in multi-

dimensional array interfaces treat such functions as second-class citizens. For instance, they are not considered for fusion with subsequent loops. Internally, multidimensional loops are represented in an indexed form that is not convenient for fusing such loops.

In interfaces designed around 1D data structures, multidimensional arrays are second-class citizens. During optimization, traversal functions are either abstract primitive operations or represent step-by-step iteration through data. These representations support fusion of loops with variable-length outputs, but require ad-hoc mechanisms for handling parallelism and index space transformations. A programmer may work with multidimensional arrays by flattening them to one dimension, but this involves expensive divide and modulus operations to reconstruct the original array indices from flattened indices. Alternatively, a programmer may use nested 1D arrays at the expense of additional levels of indirection to access array elements. Neither choice is attractive for writing multidimensional loops.

Triolet's iterators use both the indexed and step-by-step internal representation of loops. Indexed representations can represent multidimensional loops and some forms of 1D loops. They are converted to step-by-step representations when necessary to represent 1D loops with irregular iteration behavior. Thus, Triolet supports both classes of functionality.

1.5 Summary of Contributions

This dissertation as a whole demonstrates that a container traversal interface is suitable for writing high-level, high-performance distributed parallel loops: high-level because it introduces minimal complexity beyond what arises in sequential programming, and high-performance because it yields performance comparable to manually parallelized C code. This performance comes from a suite of compile-time optimizations that remove abstraction overhead and a set of container library implementation techniques that avoid unnecessary copying. Among high-level container traversal programming interfaces for distributed parallelism where programmers do not manage parallel data decomposition, work decomposition, or result collection, Triolet is the first language implementation designed to reach C-like performance.

This dissertation also demonstrates that library-driven optimization is an effective way to implement distributed container traversals. In particular, it is the first application of library-driven optimization to distributed communication. It thus serves as a counterpoint to the bulk of prior research in distributed parallel programming, which supported communication through language features with semantics facilitating the implementation of a hidden communication layer and through compile-time analysis and transformation of memory references.

Some novel aspects to Triolet's language, library, and compiler support the above broad contributions.

- Copying and communication overhead are reduced by building specialized data decomposition and distribution code for each loop. Data distribution functions are embedded into iterators and are transformed by traversal functions. Custom data distributions are built up as iterators are processed through multiple traversal functions. In contrast, prior distributed container traversal implementations relied on a limited inventory of pre-built communication patterns. To conform to input and output formats expected by pre-built code, it was often necessary to reorganize or replicate data, increasing computation and/or communication time. The extra processing is not compute-intensive, which due to prior implementations' lack of shared memory support made it unprofitable to parallelize, exacerbating its effect as a scalability bottleneck.

- Nested traversals can be executed as a single parallel loop. Prior work could only execute "flat" parallel loops producing one intermediate result for each input container element. Where inner loops were supported, they had to produce a single result. To run nested traversals, it would be necessary either to manually tile loops or to run multiple phases of parallel execution. The latter choice would redistribute a potentially large number of intermediate results between phases. Internally, Triolet's library represents nested traversals with nested iterators. Result-collection code processes nested iterators recursively and gets optimized to nested loops.

- A shared programming interface and implementation is used for list traversal and multidimensional array traversal. These constitute two sets of partially overlapping, partially incompatible functionality. The shared functionality uses the same programming interface and implementation, while the type system guarantees that incompatible functionality is not used in an incorrect way. In particular, nested iterators only arise in a subset of list traversal operations, and they cannot represent multidimensional traversals. While many programming interfaces provide one or the other feature set, Triolet merges the two gracefully.

- Triolet demonstrates a refinement to compiler inlining heuristics that allows recursive functions with structurally decreasing arguments to be used within the library-driven optimization paradigm. Functions may be annotated to indicate that they should be inlined only if the compiler has statically determined the arguments' data constructors to a specified degree. This form of inlining guidance allows a compiler to unroll recursive functions when it aids optimization without the unprofitable code explosion that would result from unrolling recursion indiscriminately.

1.6 Organization of This Dissertation

The main contribution of this dissertation is a system that runs loops expressed using data structure traversal on clusters of multicore processors with performance comparable to manually parallelized C code. Triolet is a statically typed functional programming language. The semantics of typed functional languages play a role in the Triolet language and the design of a suite of compiler optimizations that statically evaluate programs. Chapter 2 presents some background on functional language semantics. Chapter 3 presents the Triolet programming language.

Triolet's library is designed to work together with general-purpose compile-time optimizations to deliver high performance. Triolet's compiler follows in the footsteps of other statically typed functional language compilers [57, 58]. The compiler extends prior methods

for unboxed data layout with support for efficient data layout and automatic generation of serialization code for communicating between remote processors. In particular, the compiler's unboxing support provides efficient low-level array support that is orthogonal to the design of the library. Chapter 4 presents the organization of the compiler. Chapter 5 presents the organization of the library and how compiler optimizations play a role in optimizing library code.

Triolet's design strives to be as efficient as manually parallelized C code, and as high-level as prior functional algorithmic skeletons. Chapter 6 presents an evaluation of Triolet's performance and a discussion of its applicability to some realistic parallel code. The four benchmarks evaluated in that chapter are derived from preexisting GPU benchmarks [59].

Chapter 7 surveys and compares related work. Chapter 8 summarizes and concludes.

CHAPTER 2

Introduction to Functional Programming

The design of the Triolet language, compiler, and libraries has foundations in functional programming. This dissertation uses some conventions of discourse that are commonly adopted in functional programming. For readers unfamiliar with functional programming, this chapter introduces conventions, concepts, and syntax. Topics that are unique to Triolet are avoided in this section in favor of foundational concepts.

2.1 Conventions

When discussing what code means, an expression may be identified with the result of its execution, and two expression may be identified if their execution produces indistinguishable results. For example, the Triolet expression iter $[0, 1, 2, 3, 4]$ may be used in place of the value that this expression returns (which cannot be written directly). Imaginary numbers provide a parallel in mathematics: we identify the expression $1 + 2i$ with the complex number produced by multiplying 2 by i and adding 1, and we neither have nor need a more direct way of writing this number. Continuing the example, the expression range 5 may stand for the same iterator value as iter $[0, 1, 2, 3, 4]$, even though they have different execution behavior, because they can be used interchangeably (producing the same value) in all circumstances where both expressions are valid.

This notion of equivalence between expressions underlies the execution semantics of Triolet code. Triolet's execution semantics is defined in terms of *reduction rules* specifying how to replace expressions with equivalent, "simpler" expressions. When discussing how code executes or what code means, reduction rules may be selectively employed to simplify

an expression into a form that emphasizes a point. Again, this practice has parallels in mathematics. Evaluating $(\sqrt{5}+1)(\sqrt{5}-1)$ while approximating numbers to two decimal places produces a nearly integral value:

$$(\sqrt{5}+1)(\sqrt{5}-1) = (3.24)(1.24) = 4.02$$

The number 4.02 was obtained by repeatedly applying reduction rules to simplify expressions. Is it a coincidence that the number is almost integral? Evaluating in a different order makes it clear how the algebraic numbers cancel out to produce an integral result:

$$(\sqrt{5}+1)(\sqrt{5}-1) = (\sqrt{5})^2 + \sqrt{5} - \sqrt{5} - 1 = 5 - 1 = 4$$

Just as a change of evaluation order illustrated a point here, selective evaluation of functional expressions is a useful tool for explaining how a piece of code works. Selectively evaluating expressions is also useful for understanding how compiler optimizations work: some optimizations *partially evaluate* expressions at compile time, thereby reducing the work a program must do when it executes.

A functional program can be executed by exhaustively applying rewrite rules. However, expression rewriting is inefficient on real computer hardware. After performing high-level optimizations, Triolet's compiler compiles high-level functional code to executable code, following methods developed for other functional languages [60].

While it is instructive to understand the translation from functional expressions to executable code, the translation provides little insight into what happens when a given piece of source code is executed. An apparently complicated and expensive programming language feature can, in fact, be simple and cheap because optimizations reliably replace it by cheaper operations in common cases. Santos's operational semantics for high-level functional code [61] covers the most important performance factors.

Triolet is a typed language. Types are usually hidden from code examples or shown separately as type signatures, because systematically showing all type information tends to

hamper readability. This should be understood as a notational convention. Programs contain enough type information that a compiler and runtime system could, in principle, keep track of the type of every value. One can sweep some details under the rug to read Triolet code while ignoring types; however, once type-based overloading and data type representations are brought into the picture, types are an essential part of Triolet's semantics.

2.2 Reduction Semantics

There is a useful parallel between functional programming and mathematics. In both settings, the meaning of an expression is given by the definitions of the variables and operators used in the expression, rather than by the behavior of a processor. A human calculating the value of an expression proceeds by substituting variables, instantiating formulas, and reducing terms. The expressions produced at every step of this process stand for the same value. Understanding how to do the same with functional code helps in reading and writing functional programs as well as understanding how compilers can optimize them. This section presents an instance of discrete convolution, first as math, then as functional code to clarify the relationship between math and code.

The convolution of a signal A with a kernel K having 3 nonzeros is defined as $(A * K)_i = \sum_{k=-1}^{1} A_{i-k} K_k$. The left-hand-side of this equation names the operator $*$ being defined. The form of the left-hand-side indicates that $*$ is defined as a three-place operator. The right-hand side of the equation gives the value of a convolution in terms of its three arguments. Given a discrete signal x and convolution kernel y with values

$$x_0 = 0.1 \qquad x_1 = -0.2 \qquad x_2 = -0.5$$

$$y_{-1} = 1 \qquad y_0 = 0 \qquad y_1 = -1$$

one can compute $(x * y)_1$, the convolution of x and y at index 1, through the following steps. Multiple steps are condensed into one for brevity.

$$
\begin{aligned}
(x * y)_1 &= \sum_{k=-1}^{1} x_{1-k} y_k && \text{Instantiate convolution formula} \\
&= x_{1-(-1)} y_{-1} + x_{1-0} y_0 + x_{1-1} y_1 && \text{Instantiate summation} \\
&= x_2 y_{-1} + x_1 y_0 + x_0 y_1 && \text{Reduce arithmetic} \\
&= -0.5(1) + -0.2(0) + 0.1(-1) && \text{Instantiate formulas for } x \text{ and } y \\
&= -0.6 && \text{Reduce arithmetic}
\end{aligned}
$$

In the calculation process, the operators $*$ and $\sum$ and the variables x and y are instantiated, and arithmetic terms involving numbers are simplified.

In a functional setting, a programmer could define convolution in a way that closely follows the mathematical definition:

```
convolution A K i = let convAt k = A (i - k) * K k
                    in mathSum (-1) 1 convAt
```

Just as the mathematical definition defined a new operator, $*$, this equation defines a new variable, convolution. The left-hand side of the equation shows convolution called with three arguments, indicating that its value is a three-argument function. The equals sign introduces the function body. It can be read as forming an equation: the function call on the left is equal to the expression on the right for any choice of parameters A, K, and i.

In the function body, **let** ... **in** delimits a group of local variable definitions. One local variable, convAt, is defined here. This variable holds a function that computes the original convolution equation's kth summand. (The $*$ now stands for multiplication.) The call to mathSum carries out the summation (mathSum is defined in Section 2.3). This code is almost a direct translation of the mathematical equation for convolution, yet (with a few syntactic changes) it is also executable in Haskell or ML.

Signals x and y could, similarly, be defined as functions that take an index and return the signal value at that index.

```
x 0 = 0.1
```

$x\ 1 = -0.2$

$x\ 2 = -0.5$

$x\ 3 = -0.4$

$y\ (-1) = 1$

$y\ 0 \quad = 0.1$

$y\ 1 \quad = -1$

These equations utilize *pattern matching*: which equation gives the function's result depends on what argument x or y is called with. The first equation, for instance, says that a call of x with argument 0 evaluates to 0.1. The function x compares its argument against each equation until one matches, like a `switch` statement in C. It is a run-time error if no equation matches.

Functional code, like mathematical formulas, can be evaluated (or reduced) at a source code level. Evaluating code "on paper" is a useful way to understand how functions behave. The evaluation of a call to convolution is traced below. A long arrow connects successive steps of evaluation. As before, multiple steps of evaluation are condensed for brevity.

convolution $x\ y\ 1$	
$\longrightarrow$ **let** convAt $k = x\ (1-k) * y\ k$ **in** mathSum $(-1)\ 1$ convAt	Inline convolution
$\longrightarrow$ **let** convAt $k = x\ (1-k) * y\ k$ **in** $0 +$ convAt $(-1) +$ convAt $0 +$ convAt 1	Inline mathSum
$\longrightarrow$ **let** convAt $k = x\ (1-k) * y\ k$ **in** $0 + (x\ (1-(-1)) * y\ (-1)) +$ $(x\ (1-0) * y\ 0) + (x\ (1-1) * y\ 1)$	Inline convAt
$\longrightarrow\ 0 + (x\ (1-(-1)) * y\ (-1)) +$ $(x\ (1-0) * y\ 0) + (x\ (1-1) * y\ 1)$	Remove unused variable
$\longrightarrow 0 + x\ 2 * y\ (-1) + x\ 1 * y\ 0 + x\ 0 * y\ 1$	Reduce arithmetic

$\longrightarrow 0 + -0.5 * 1 + -0.2 * 0 + 0.1 * -1$ Inline x and y

$\longrightarrow -0.6$ Reduce arithmetic

These evaluation steps parallel the earlier mathematical calculation. Each step proceeds by transforming an expression. Transformation is localized: to transform an expression, one need examine only the expression itself and the definitions of variables that it uses.

2.3 Loops and Higher-Order Functions

Here mathSum is presented in more detail as an example of a loop (summation involves a loop) and a higher-order function (mathSum's third parameter is a function). A function is *higher-order* if any of its parameters or return value must be a function for error-free evaluation. The summation $\sum_{k=l}^{u} f(k)$ is computed by mathSum. The lower bound l, upper bound u, and summand f are passed as parameters. By parameterizing over an arbitrary summand, expressed as a function of k, mathematical summation can be defined as a function once and for all.

```
mathSum l u f =
    let loop i x = if i > u
                   then x
                   else let y = x + f i
                        in loop (i + 1) y
    in loop l 0
```

The local function `loop` decomposes the problem of summing over a range $i \ldots u$ into computing a value at i and summing over a smaller range $i+1 \ldots u$. Parameter x holds the partial sum over $l \ldots i-1$. If the range is empty, the sum is simply x; otherwise, a new partial sum is assigned to y and `loop` is called to solve a new subproblem. With this breakdown, the entire summation is computed by summing the full range with a partial sum of zero,

loop l 0.

We can trace the execution of $\sum_{k=0}^{2} k^2$ by repeatedly inlining and reducing the call of loop. With the help of the auxiliary function definition square $x = x*x$, the call mathSum 0 2 square expands into

```
let loop i x = if i > 2
               then x
               else let y = x + square i in loop (i + 1) y
in loop 0 0
```

The loop is initially called with $i = 0$ and $x = 0$. Inlining and reducing once computes the partial sum at index 0:

```
loop 0 0 ⟶  if 0 > 2
            then 0
            else let y = 0 + square 0 in loop (0 + 1) y
         ⟶ loop 1 0
```

Evaluation yields another call of loop. This process repeats until $i = 3$, at which point the Boolean condition evaluates to True and no further calls are made:

```
         ⟶  if 1 > 2
            then 0
            else let y = 0 + square 1 in loop (1 + 1) y
         ⟶ loop 2 1
         ⟶  if 2 > 2
            then 1
            else let y = 1 + square 2 in loop (2 + 1) y
         ⟶ loop 3 5
```

$$
\begin{aligned}
&\longrightarrow \ \textbf{if } 3 > 2 \\
&\qquad \textbf{then } 5 \\
&\qquad \textbf{else let } y = 5 + \mathsf{square}\ 3 \textbf{ in } \mathsf{loop}\ (3+1)\ y \\
&\longrightarrow 5
\end{aligned}
$$

The administrative work of counting from 0 to 3 makes this tracing process rather tedious. By leaving square unevaluated, we can reduce the uninteresting work to reveal just the three-element summation:

$$
\begin{aligned}
\mathsf{loop}\ 0\ 0 \longrightarrow \ & \textbf{let } y = 0 + \mathsf{square}\ 0 \\
& \textbf{in } \mathsf{loop}\ 1\ y \\
\longrightarrow \ & \textbf{let } y = (0 + \mathsf{square}\ 0) + \mathsf{square}\ 1 \\
& \textbf{in } \mathsf{loop}\ 2\ y \\
\longrightarrow \ & \textbf{let } y = ((0 + \mathsf{square}\ 0) + \mathsf{square}\ 1) + \mathsf{square}\ 2 \\
& \textbf{in } \mathsf{loop}\ 3\ y \\
\longrightarrow \ & 0 + \mathsf{square}\ 0 + \mathsf{square}\ 1 + \mathsf{square}\ 2
\end{aligned}
$$

Selectively evaluating the uninteresting work makes it clearer what this code accomplishes. This trick was applied when inlining mathSum in Section 2.2. The ability to simplify individual expressions is indispensable to understanding larger pieces of code.

2.4 Data Structures

A programming language's built-in data types only serve a limited range of purposes, so most languages also provide a way of constructing new data structures. This section introduces *algebraic* data structures, which were used early on by the influential language ML and subsequently spread into other languages. The power of algebraic data structures is their ability to statically classify data through a type system. Static typing rules out some classes of

programming bugs and enables run-time data representations to be more space-efficient and time-efficient than less-expressive paradigms. This section presents algebraic data structures in an untyped setting to emphasize their interpretation. Section 2.6 completes the picture with algebraic data *types* that classify data.

Algebraic data structures originate from the application of abstract algebra to data structure design. An analogy from mathematics illustrates the central concepts. Given the term $1 + 2(3)$, we can reduce the addition and multiplication to get 7. But the term $1 + 2i$ (where i is the imaginary unit) is already in its simplest form, despite having a multiplication and an addition. The usual interpretation is that the term $1 + 2i$ *does* stand for a particular complex number that is produced by multiplying 2 by i and adding 1, we just have no way to write this number directly. Expressions of the form $a + bi$ (where a and b stand for real numbers) are perfectly serviceable ways of writing complex numbers. Every complex number has a unique decomposition into two real numbers a and b in this form, and every pair of real numbers uniquely identifies a complex number. This uniqueness property means the the addition and multiplication in $a + bi$ are *reversible*: we can deconstruct a complex number into its two constituent real numbers. Effectively this form is a data structure containing two fields, "real part" a and "imaginary part" b.

Functional languages with algebraic data structures impose an operational semantics onto the semantic notion that data structures are reversible functions. Functional languages distinguish *functions* (which can be reduced, but not deconstructed) from *data constructors* (which can be deconstructed, but not reduced). To work with complex numbers in functional code, assume for now that there is a predefined two-parameter data constructor called Complex. The *data expression* Complex 1 2 is the functional equivalent of the term $1 + 2i$. It syntactically resembles a two-argument function call, but with a data constructor in place of a function. A typical operational interpretation for this term is that it creates a new heap object containing references to a type descriptor for Complex and the numbers 1 and 2 (Figure 2.1) and returns the new object's address. The first pointer points to a type descriptor holding statically generated information about the data constructor Complex. Type descriptors

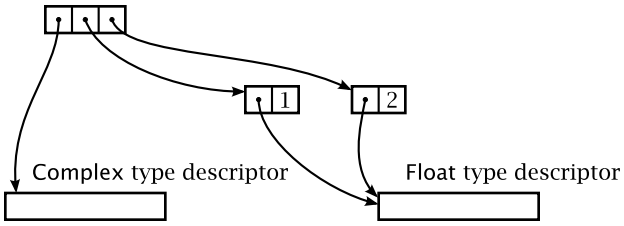


Figure 2.1: Memory organization of an algebraic data structure representing a complex number during the execution of untyped functional code.

hold whatever run-time information may be required about an object, such as the number of fields the object has. The second and third pointers point to data structures representing the floating-point numbers 1 and 2, respectively. The term Complex 1 2 is also used to write the value returned by the expression. That is, this term stands for a piece of data consisting of pointers to the data structures for Complex, 1, and 2 in that sequence.

Data is deconstructed by *case expressions*. A case expression examines an object's data constructor and binds its fields to variables. For example, functions could be defined to get the real part, imaginary part, and magnitude of a complex number:

$$\mathsf{real}\ x = \textbf{case}\ x\ \textbf{of}\ \mathsf{Complex}\ y\ z.\ y$$

$$\mathsf{imag}\ x = \textbf{case}\ x\ \textbf{of}\ \mathsf{Complex}\ y\ z.\ z$$

$$\mathsf{mag}\ x = \textbf{case}\ x\ \textbf{of}\ \mathsf{Complex}\ y\ z.\ \mathsf{sqrt}\ (y * y + z * z)$$

The expression mag (Complex 3 4) is evaluated as follows.

$$\begin{aligned}\mathsf{mag}\ (\mathsf{Complex}\ 3\ 4) &\longrightarrow \textbf{case}\ \mathsf{Complex}\ 3\ 4\ \textbf{of}\ \mathsf{Complex}\ y\ z.\ \mathsf{sqrt}\ (y * y + z * z)\\ &\longrightarrow \mathsf{sqrt}\ (3 * 3 + 4 * 4)\\ &\longrightarrow 5\end{aligned}$$

The case expression deconstructs the value Complex 3 4 into its constituent values 3 and 4, putting them into y and z respectively. Deconstructing the value involves pattern matching: the case expression first checks that the value is a Complex term, and signals a run-time error if it is not. Operationally, the case expression checks the given value's type descriptor, reads

the value's two fields into variables, and executes the subexpression.

Case expressions may supply handlers for several different data constructors, dynamically selecting the one that matches the given value. For instance, Boolean values are data terms consisting of a data constructor, either True or False, and no fields. A function to get the absolute value of an integer could be written

```
abs x = case x ≥ 0
        of True. x
           False. 0 − x
```

The case expression examines the result of $x \geq 0$ and evaluates the matching handler. The more familiar **if** ... **then** ... **else** ... notation is shorthand for **case** ... **of** {True. ...; False. ...}.

Functions and algebraic data structures take over the roles filled by lower-level primitive operations in imperative code. Memory is written by data expressions and read by case expressions. Conditional execution is accomplished through case expressions. Other control flow transfers are accomplished through function calls.

2.5 Divergence

Evaluating an expression may fail to yield a value in one of several ways. A recursive function may loop forever without producing a result. Some operations may produce run-time errors, such as dividing by zero or reading an out-of-bounds array index. Many functional languages have a primitive operation, **error**, that interrupts execution when evaluated. These outcomes of evaluation are called *divergence*.

In Triolet, divergence is treated like a benign side effect. Compiler optimizations are allowed to turn a diverging program into one that diverges for a different reason, or to remove divergence from a function that both diverges and returns a value. The role of divergence is similar to the role of "undefined behavior" in compiling C code [62].

The properties of divergence affect which compiler optimizations are safe and enable some profitable transformations. By knowing that some parts of a function diverge, a compiler can infer additional information to use when optimizing other parts of the function. Error checks, for instance, diverges by interrupting execution if a condition indicating an error is met. The compiler can infer that, if an error check does not diverge, its error condition is not met. Suppose we have a function that requires its Complex argument to have a zero real component:

foo x = **case** x **of** Complex y z. **if** $y \neq 0$ **then error else** cos z

If foo returns, its argument's real component must be zero, and this fact can be used for optimizing subsequent code. For example, in the expression **let** u = foo x **in case** x **of** Complex y z. $y + u$, the real component of x is surely zero after foo returns. Thus, y is known to be zero and the expression $y + u$ can be simplified to u.

2.6 Types

To this point, this chapter has tacitly assumed an *untyped* language, where any term can be used in any context. The term 1 + 2 is permissible, and so is the term 1 + convolution. But the sum of an integer and a function is conventionally meaningless. Such a term, though not useful, could still arise due to a programming error. It would be better for a compiler to reject such code than to turn it into executable code that does nothing useful. Ruling out such meaningless terms is the job of a *type system*.

Type systems aid software engineering and facilitate generation of efficient code. A type annotation written by a programmer expresses a fact about program behavior that a compiler can verify; thus, type annotations are machine-checked documentation. Type checking helps to detect misbehaving code early and rule out some classes of bugs. By constraining the possible values of variables and terms, a type system allows a compiler to create programs that store less information (saving space) and check less information (saving time).

Triolet uses a *static* type system. Static types are a compile-time notion for explaining how a program behaves. Static type information does not exist at run time. Values are classified into types; a type is a name for a set of values. A type is said to be *inhabited* if there are values that have that type; its values are called its inhabitants. The value 1 is said to *have type* Int or *inhabit* Int.

Variables and expressions are ascribed types to indicate what values they may contain or evaluate to. We say, for instance, that a term or variable *has type* Int if its value can only be an integer. (To avoid some subtleties regarding numeric types, in this section all numbers are assumed to be integers.) For instance, the term $1 + 1$ has type Int. Terms that have a type are *well-typed*. Terms such as $1 +$ convolution are *ill-typed*; they cannot have a value, and are rejected by the compiler.

A function's type summarizes how it behaves when called. The type indicates what is required of its arguments and what is guaranteed of its return value. Types allow a function's correctness to be verified in isolation from the rest of a program: requirements on parameters are assumed to be true while checking the function, and verified in each place the function is used. For instance, in the term **let** $f\ x = x * x + 1$ **in** f, the function f takes and returns integers. Its type is written Int → Int, the → operator denoting a function type. Since the typing rules guarantee that f will only be passed Ints, it is not necessary for f to check whether x is an Int before reading its value from memory.

In a language with immutable data and static typing, new data types may be defined *algebraically* by declaring the name of a new type together with all the ways that values of that type may be built [63]. As in the untyped setting, data values are built with data constructors. Now, each data constructor is introduced with a type signature determining what field types it takes and what result type it produces. The following example shows a definition of a binary tree data type.

```
data IntTree : ⋆ where
  IntBranch : Int × Int → IntTree
  IntLeaf : Int → IntTree
```

This definition introduces a type constructor, IntTree, and two data constructors, IntBranch and IntLeaf, for creating trees whose leaves are labeled with an integer. The kind signature IntTree : $\star$ indicates that IntTree is a type. The data constructors' type signatures reflect the fact that they take field values (whose types are given on the left of the arrow) and produce a new piece of data (whose type is given on the right). IntBranch builds a new tree from two trees representing its left and right subtrees. IntLeaf builds a new tree representing a leaf from an Int representing its label.

Since a data type definition determines its constructors, it also determines how case expressions examine values of that type. For IntTrees, a case expression matching any IntTree would have one handler for each of IntBranch and IntLeaf, as demonstrated by the following leaf-counting function.

```
numLeaves t = case t
                of IntBranch ℓ r. numLeaves ℓ + numLeaves r
                   IntLeaf x. 1
```

This function has type IntTree → Int. Since an IntTree can only be built with two data constructors, every possible argument value is handled by this function.

Because an algebraic data type definition lists all the ways that a data type can be built, an implementation can use the definition to choose an optimized representation of the type in memory. Any given Tree is either a branch or a leaf, so a tree's type descriptor can simply be a Boolean flag to distinguish between the two possibilities. Furthermore, the Int field's value does not have to be represented by a pointer to an Int object as it was in untyped code. Since the field must be an integer, its value may be stored in the object as a machine word, eliminating a level of indirection to access the data. Values whose contents are stored directly as part of another value are called *unboxed*. The term "box" in this context refers to the block of heap-allocated storage normally used to hold a value. A data type is called unboxed if its values are always unboxed. Memory layout optimizations, including unboxing, have a significant effect on performance. Unboxing transformations have been studied extensively [64, 65, 66, 67, 68]. Unboxing can dramatically improve the performance of access

to small objects, such as numeric values, and array elements.

A type system statically constrains the possible values held in objects, reducing the possibility of errors and eliminating some dynamic checks and boxing. In this section's examples, each piece of code is fixed to one type. However, a central feature of container libraries is that the same container data types and functions may be reused on collections of values of arbitrary types. The more powerful type system features discussed in the next section are needed to express such code.

2.7 Polymorphism

Many functions and data constructors are *polymorphic*, working "the same way" for an arbitrary choice of types. For example, the function

$\mathsf{select}\ b\ x\ y = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y$

selects one of x or y, depending on the Boolean value b. It can be used to select between two values of any type (both values must have the same type). To select numbers ($\mathsf{select}\ b\ 0\ 1$), select can be given type $\mathsf{Bool} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. To select Booleans ($\mathsf{select}\ b\ \mathsf{False}\ \mathsf{True}$), it can be given type $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$. In fact, for any type τ, select can be given type $\mathsf{Bool} \to \tau \to \tau \to \tau$. This freedom to choose an arbitrary type for some parts of the code is called polymorphism.

The type of a polymorphic function or data constructor expresses the range of all possible types it may be ascribed. In a polymorphic type, type variables stand for the parts that can vary. A polymorphic type for select may be written $\forall \alpha : \star .\ \mathsf{Bool} \to \alpha \to \alpha \to \alpha$. In this type, the quantifier $\forall$ introduces a type variable α standing for an arbitrary type. Each time select is called, a type is chosen to use in place of α. The caller makes this choice by passing a type to use in place of α as an extra argument to select. This passing of type parameters can be interpreted as computing type information while a program runs.

A closely related concept is that of *parametric types*. Just as a polymorphic function works in the same way for arbitrary types, parametric types describe the same data struc-

tures for some arbitrary choice of the types held in the data structure. Linked lists are an example. A value is not a LinkedList, but rather a LinkedList Int if it is a list containing Ints, a LinkedList Bool if it is a list containing Bools, and so forth. Applying the parametric type constructor LinkedList to an argument such as Int or Bool yields a concrete type that can be used to describe some lists.

Parametric data type definitions, like polymorphic types, use type variables to stand for the parts that vary. The trees from the previous section can be generalized to arbitrary label types.

data Tree $(\alpha : \star) : \star$ **where**

Branch : Tree $\alpha \times$ Tree $\alpha \to$ Tree α

Leaf : $\alpha \to$ Tree α

The type constructor Tree takes the label type as a parameter α. Occurrences of IntTree in the previous section are now Tree α. In leaves, the label type Int has been replaced by α.

CHAPTER 3

The Triolet Programming Language

This chapter introduces the Triolet programming language.

The Triolet programming language was designed to offer convenience and readability for programmers. While the development of Triolet was not intended to introduce innovative language features, no existing programming language completely suited Triolet's design goals of usability and high performance. Triolet achieves its goals by drawing elements from two programming languages. Triolet's syntax and control flow are Python-like to make it readable to casual and novice users. Triolet code can usually be read as if it were Python code. Triolet's data structure semantics is Haskell-like to facilitate compile-time optimization. A fortuitous overlap between the language features of Python and Haskell lets Python's expression syntax be reinterpreted in terms of Triolet's data structures.

Triolet was modeled after Python because Python is widely used and its syntax includes list comprehensions and first-class functions, both heavily used in Triolet. The similarity to Python is meant to lower the initial learning barrier by enabling novice programmers to intuit the meaning of Triolet code from their knowledge of Python. The strategy of grafting a functional semantics onto an existing imperative language's syntax follows in the footsteps of Single Assignment C [13] and Copperhead [7]. In particular, Copperhead is also a functional language using Python syntax.

Triolet is intended for accelerating small, compute-intensive loops within an application, similar to how OpenCL is used. In a typical use case, a developer would start by writing an application in C and C++. Upon finding that the application spends much of its time in computationally intensive, parallelizable loops, the developer would rewrite those loops as

Triolet functions. Using Triolet's C++ API, he or she would write C++ code to convert between C++ and Triolet data, and insert calls to Triolet functions.

The following example code shows the structure of a typical Triolet file. This file contains two statements: a definition of the function `minimum`, and a statement that exports this function to C++ (by creating a wrapper function that can be called from C++). The exported function's type is written using the syntax described in Section 3.1. The body of `minimum` defines a function `min` for selecting the least of two values, then calls the library function `reduce1` to find the minimum of all values in `xs`. Section 3.2 describes statements. Section 3.3 describes expressions. Sections 3.4 and 3.5 describe Triolet's library functions.

```
def minimum(xs):
  def min(x, y):
    if x < y: return x
    else:     return y
  return reduce1(min, xs)

export cplusplus minimum : list(float) -> float
```

A C++ file that calls `minimum` might look as follows.

```
#include "minimum_cxx.h" // File generated by Triolet compiler

using namespace Triolet;

// Call the Triolet function on a 100-element list
float minimum100(float *p)
{
  List<Float> l = CreateFloatList(100, p); // Copy to a new Triolet list
  return (float) minimum(l);               // Call the Triolet function
}
```

3.1 Types

The abstract syntax of kinds and types is shown in Figure 3.1. Types are always specified on exported functions. Types may optionally annotated onto function parameters and returns. A `require` statement can optionally be used to specify the type of a local variable. Type

$\alpha \in$ Identifiers

kind ::=	`type` | *kind* `->` *kind*	Kind
type ::=	α	Type variable or constructor
	| *type*`(`[*type*`,`]$^+$`)`	Type application
	| `(`[*type*`,`]*`)`	Tuple type
	| *type*[`*` *type*]*] `->` *type*	Function type

Figure 3.1: Syntax of Triolet types.

annotations are useful for documentation and debugging. They can also constrain the types of terms when there is not enough information for type inference to determine a type.

Triolet has a simple kind system (in the technical sense). There is a single base kind, `type`. Arrow kinds κ_1 `->` κ_2 are inhabited by parametric types.

Type application is written in function syntax, e.g., `list(int)` is the application of `list` to `int`. Tuple types are written in tuple syntax, e.g., `(int, float, bool)` is the application of the 3-tuple type constructor to `int`, `float`, and `bool`. Function types are written with an arrow. If a function takes multiple arguments, the domain is written as a product. For instance, the type of a Boolean NAND function is `bool * bool -> bool`.

3.2 Statements

The syntax of statements is shown in Figure 3.2. Assignment and conditional execution work as in Python. Return statements must be provided with a return value.

A sequence of function definitions without intervening statements acts as a set of mutually recursive function definitions. Because definitions are recursive, such functions can be used to write sequential loops.

An `assert` statement evaluates a Boolean expression and diverges if the result is `False`. Assertions express conditions that must hold during execution. The compiler may use these conditions in optimizing code. For example, an assertion can be used to indicate that the size

stm ::=	*target* = *exp*	Assignment
	| `if` *exp*`:` *suite* [`elif` *exp*`:` *suite*]* [`else:` *suite*]	Conditional execution
	| `return` *exp*	Function return
	| *def*$^+$	Function definition group
	| `assert` *exp*	Dynamic assertion
	| `require` *x* `:` *type*	Static assertion
suite ::=	*stm**	
poly-sig ::=	`@forall(`[*x* [`:`*kind*]`,`]*`)` [`@where(`[*type*`,`]*`)`]	Polymorphic type signature
def ::=	[*poly-sig*] `def` *x*`(`[*x* [`:`*type*]`,`]$^+$`)` [`->` *type*]`:` *suite*	Function definition
export ::=	`export cplusplus` [*str*] *x* `:` *type*	Export declaration
top ::=	*def* | *export*	Top-level statement
module ::=	*top**	Triolet module

Figure 3.2: Syntax of Triolet statements and global definitions.

of a list is known at compile time:

```
def f(A):
  assert len(A) == 4
  return len(A)
```

The second statement will only execute if `len(A)` has the value 4, so the compiler can replace the second `len(A)` with a literal 4.

A `require` statement declares the type of a variable. Declaring types is useful for debugging type errors, and is occasionally necessary to resolve operator overloading. In the following example, the definition and use of `ones` are overloaded such that the type of `ones` is not uniquely determined. The `require` statement fixes the type of `ones`.

```
def size(A):
  ones = [1 for x in A]
  require ones : view(list_dim, int)
  return sum(ones)
```

Function definitions optionally take parameter and return type annotations. The following example restricts `size` to functions that take a `list(int)` parameter and return an `int`.

```
def size(A : list(int)) -> int:
  ones = [1 for x in A]
```

```
  require ones : view(list_dim, int)
  return sum(ones)
```

Function definitions can also be given polymorphic type annotations. Polymorphic type annotations declare type parameters and type class constraints. The following annotates a polymorphic type onto `size`. This is the same type that is inferred if the function definition is not annotated.

```
@forall(t : type -> type, a)
@where(Repr(t(a)), Repr(a), Traversable(t), shape(t) == list_dim)
def size(A : t(a)) -> int:
  ones = [1 for x in A]
  require ones : view(list_dim, int)
  return sum(ones)
```

A Triolet file consists of a sequence of function definitions and export declarations in arbitrary order. An export declaration directs Triolet to generate a wrapper that allows a function to be called from another language. The following statement exports `size` on lists to C++. The C++ function is called `triolet_size`.

```
export cplusplus "triolet_size" size : list(int) -> int
```

3.3 Expressions

The syntax of Triolet expressions is shown in Figure 3.3.

A *target* designates the destination of an assignment. A variable target assigns a variable. For instance, in `[1 for ix in range(10)]`, the target `ix` binds the variable `ix`. The context where the target appears determines what value is bound; in this case, values bound to `ix` come from the iterator returned by `range(10)`. A tuple target deconstructs a tuple and assigns its fields to targets. In `[magnitude * exp(angle) for (magnitude, angle) in A]`, the values taken from `A` must be 2-tuples and their two fields are bound to `x` and `y` respectively.

Most expressions *exp* are analogous to their Python equivalents. Function calls and lambda expressions must take at least one argument. Generators and comprehensions play

x ∈ Identifiers
n ∈ Numbers
$\oplus$ ∈ {`or`, `and`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `|`, `&`, `^`, `<<`, `>>`, `+`, `-`, `*`, `/`, `//`, `%`, `**`, `->`}

target	::= x	Variable binding
	| `(`[x`,`]*`)`	Tuple binding
exp	::= x	Variable
	| n | `True` | `False` | `None`	Literal
	| `(`[*exp*`,`]*`)`	Tuple
	| `[`[*exp*`,`]*`]`	List
	| `-`*exp* | `not` *exp*	Unary operator
	| *exp* $\oplus$ *exp*	Binary operator
	| *exp*`[`*exp*`]`	Subscript
	| *exp*`[`[*exp*]`:`[*exp*][`:`[*exp*]]`]`	Slice
	| `(`*exp gen*`)`	Generator
	| `[`*exp gen*`]`	Comprehension
	| *exp*`(`[*exp*`,`]$^+$`)`	Call
	| *exp* `if` *exp* `else` *exp*	Conditional
	| `lambda` [x`,`]$^+$`:` *exp*	Anonymous function
	| `let` *target* `=` *exp* `in` *exp*	Local binding
gen	::= `for` *target* `in` *exp clause**	Value generator
clause	::= `for` *target* `in` *exp*	Iteration
	| `if` *exp*	Guard
	| `let` *target* `=` *exp*	Local binding

Figure 3.3: Syntax of Triolet expressions.

a more prominent role in Triolet than in Python, since they are used in parallel programming. Triolet introduces let expressions and let clauses as a way to locally bind variables. The expression `let (x, y) = foo(x) in x + y` takes the result of `foo(x)`, which must be a tuple, and adds its two fields together.

A generator expression or comprehension contains a body *exp* followed by a sequence *gen* of generator clauses. The sequence of clauses can be read as a loop nest, with the expression forming the loop body. For example, the expression `[x*y for x in xs for y in ys if x > 0 and y > 0]` is analogous to the following Python loop.

```
result = []
for x in xs:
  for y in ys:
    if x > 0 and y > 0:
      result.append(x*y)
```

A `let` clause, like a `let` expression, can be used to locally bind variables in a generator.

3.4 Introduction to Container Functions

Triolet has a library of functions that operate on collections of values. Most of these functions can exploit parallelism by partitioning a collection and processing each partition in a separate thread. Informally, a *container* is a data structure that holds a collection of values. Formally, a container in Triolet is a value whose type is an application of a traversable type constructor (Section 3.5.3). In Triolet, containers can hold an arbitrary data type and all values in a given container must have the same type.

Triolet's container functions are generalized to work with a variety of container types. For the purposes of this section, container functions are assumed to operate only on lists. This restriction simplifies the description of how the *contents* of a container get transformed. This understanding carries along to Section 3.5, which generalizes the *structure* of containers.

Mapping A *map*, performed by `map`, transforms each element of an array by a given function. The expression `map(f, [x, y, z])` applies the function `f` to the array elements `x`,

y, and z, returning an array equal to `[f(x), f(y), f(z)]`. Many algorithms have communication-free parallel phases, expressed as a mapping of some function over an array.

A list comprehension with a single `for` clause is equivalent to a map. However, the two forms are generalized differently in Section 3.5.

Reduction A *reduction*, performed by `reduce`, combines all the elements of a data structure into one result value by repeatedly applying a given associative or pseudo-associative binary function f. The binary operation's identity value z is used as the initial value of the result. The expression `reduce(f, z, [k, l, m, n])` computes `f(f(f(f(z, k), l), m), n)`. The associativity and identity properties of `f` and `z` admit other ways of computing the result, which can be exploited for parallelism. If the result is computed as `f(f(f(z, k), l), f(f(z, m), n))` instead, then a parallel computer could evaluate the subterms `f(f(z, k), l)` and `f(f(z, m), n)` in parallel.

Summation, performed by `sum`, is a common special case of reduction where the binary function is addition and the identity element is zero. Signal processing and physical modeling often involve computations of the general form $\sum_x f(x)$; these can be written with `map` and `sum`. For example, the norm of a vector $|\vec{v}|$ is $\sqrt{\sum_{v_i \in \vec{v}} v_i^2}$. In Triolet, this is written `sqrt (sum (map (lambda v_i: v_i*v_i, v)))`. The call to `map` computes all squared components v_i^2, the call to `sum` sums them, and the call to `sqrt` takes the square root.

Reductions that are idempotent, but do not have an identity value, can be written using `reduce1`. This function uses an element of its input list like an identity value. That is, `reduce1(f, [x, y, z])` is equivalent to `reduce(f, x, [y,z])` as long as `f` is idempotent. This function diverges when given an empty list. Such reductions typically arise when finding a least or greatest value with respect to some ordering.

Histogramming A *histogram* is an array recording the number of occurrences of discrete values in a set, such as the number of times each integer is found in some array. Elements of a histogram array are called *bins*. Triolet's `histogram` function computes weighted histograms. A weighted histogram computation takes an array of key-weight pairs as input, sums the

weights at each key, and builds an array of the summed values. The array size is given as the first parameter. For instance, the expression `histogram(4, [(0,10), (1,5), (0,9), (2,-5)])` constructs a four-element array and populates it with the given key-weight pairs, producing $[19, 5, -5, 0]$. An unweighted histogram is computed by using 1 for every weight.

Range construction A *range*, computed by `range`, is an array holding the sequence of integers from zero up to a given bound. The expression `range(4)` returns $[0, 1, 2, 3]$. Ranges are the container analogue of counted loops. When a range is used as the input to a container operation, the values in the range play the role of a loop counter. For instance, the Triolet expression `sum(map(f, range(n)))` represents the same computation as the following C code fragment.

```
s = 0;
for (i = 0; i < n; i++) s = s + f(i);
```

Slicing *Slicing* selects array elements whose indices lie in a given range. A slice expression `a[l:u:s]` selects the elements of array `a` at indices from the lower bound `l` up to and excluding the upper bound `u` at intervals of the stride `s`. For example, the expression `[2, 3, 5, 7, 11, 13, 17, 19][1:7:2]` selects the elements at indices 1, 3, and 5, returning $[3, 7, 13]$.

Zipping *Zipping*, performed by `zip`, combines two arrays by tupling together array elements at corresponding indices. The name "zip" recalls how a zipper's two rows of teeth line up when it is closed. The expression `zip([-3,-2,-1], [4,5,6,7])` returns $[\langle -3, 4\rangle, \langle -2, 5\rangle, \langle -1, 6\rangle]$. The tail of the longer argument array is dropped. It is sometimes convenient to store a collection of data in multiple arrays, each holding a different property of the data. This is called "structure-of-arrays" storage. Zipping is used to combine data from multiple arrays as input to a parallel computation. For instance, given some complex numbers stored as an array of real components and an array of imaginary components, the arrays could be zipped together to create an array of real-imaginary pairs for further processing. The resulting array is often called "array-of-structures" since each array element is a data structure with two

fields.

Filtering *Filtering*, performed by `filter`, selects those elements of an array that satisfy a given test. Tests are given as functions that map array elements to a Boolean value. For instance, the function `lambda x: x >= 0` identifies nonnegative array elements. The expression `filter(lambda x: x >= 0, [1, -2, 3, 4])` returns the nonnegative array elements as a new array, $[1, 3, 4]$.

Singleton construction The `unit` function creates a one-element array containing a given value. For instance, `unit(5)` yields $[5]$. This function is a building block of nested loops. List comprehensions, described later in this section, sometimes translate into code that calls `unit`.

Flattening Some algorithms expand a single input value into multiple outputs. The array-creating functions introduced so far cannot do this, as they produce at most one array element per loop iteration. Multiple outputs may be produced by creating an array of arrays, then *flattening* it into an array.

Flattening is employed, for example, in finding all ordered pairs of *neighbors* in a given set of points, where two points are neighbors if the distance between them is less than 1. Consider a set of points s having members s_0, s_1, s_2. One can map over s twice to create all pairs of points.

```
all_pairs = map(lambda x: map(lambda y: (x, y), s), s)
```

This computes $[[\langle s_0, s_0\rangle, \langle s_0, s_1\rangle, \langle s_0, s_2\rangle], [\langle s_1, s_0\rangle, \langle s_1, s_1\rangle, \langle s_1, s_2\rangle], [\langle s_2, s_0\rangle, \langle s_2, s_1\rangle, \langle s_2, s_2\rangle]]$. Supposing a function `is_neighbor` is available to decide whether the points in a pair are neighbors, one can then filter the inner lists to obtain neighbor points.

```
neighbor_pairs =
  map(lambda xys: filter(lambda xy: is_neighbor(xy), xys), all_pairs)
```

This computes a list of lists containing all neighbor pairs, $[[\langle s_0, s_0\rangle, \langle s_0, s_1\rangle], [\langle s_1, s_0\rangle, \langle s_1, s_1\rangle, \langle s_1, s_2\rangle], [\langle s_2, s_1\rangle, \langle s_2, s_2\rangle]]$. The nesting of lists is an artifact of the nested loops that created

this data structure. Typically, further processing would treat all pairs in the same way. The nested list organization provides no useful information compared to a flat list, but adds needless complexity to further processing. One can apply the flattening function `concat` to concatenate the sub-lists, yielding $[\langle s_0, s_0\rangle, \langle s_0, s_1\rangle, \langle s_1, s_0\rangle, \langle s_1, s_1\rangle, \langle s_1, s_2\rangle, \langle s_2, s_1\rangle, \langle s_2, s_2\rangle]$.

Some container libraries, including Triolet's, supply an alternative flattening function `concatMap` that is a combination of `map` and `concat`. Either of `concat` and `concatMap` can be implemented in terms of the other according to the following formulas:

$$\texttt{concat}\ x = \texttt{concatMap}\ (\lambda y.\ y)\ x$$

$$\texttt{concatMap}\ f\ x = \texttt{concat}\ (\texttt{map}\ f\ x)$$

Although `concat` is conceptually simpler, `concatMap` is more amenable to loop fusion and can lead to lower asymptotic space usage.

List comprehensions List comprehensions are a syntactic form for writing mapping, filtering, and concatenation with a loop-like syntax. Derived from mathematical set comprehension notation, they first appeared in the SETL programming language [69] and were subsequently adopted into many other languages. List comprehensions are translated into calls of `map`, `concatMap`, `filter`, and `unit` [52].

The simplest nontrivial list comprehension consists of a body followed by a single generator clause, for example `[x * x for x in range(5)]`. The generator clause `for x in range(5)` loops over `range(5)`, binding each element of the range to x. The body computes `x * x` on each generated value, and the results are collected into a list, producing $[0, 1, 4, 9, 16]$. Looping over array elements and building a result is precisely what `map` does, and this expression could be written without list comprehensions as `map(lambda x: x * x, range(5))`.

List comprehensions may also contain guard clauses, introduced by the `if` keyword. A guard clause in a list comprehension skips loop iterations, just as `filter` skips array

elements. For instance, `[x * x for x in range(5) if x > 0]` does the same thing as `map(lambda x: x * x, filter(lambda x: x > 0, range(5)))`.

A comprehension with multiple `for` clauses translates into a loop nest flattened with `concatMap`. The neighbor list computation presented above can be written more compactly as `[(x, y) for x in s for y in s if is_neighbor((x, y))]`.

3.5 Containers

Lists are awkward at storing many kinds of data. Matrix and image data, for instance, are naturally organized as 2D arrays. Moreover, lists are eagerly evaluated (each container function fully generates its result before the next begins), which can lead to unnecessary storage of temporary values. Triolet generalizes container functions over different logical arrangements of data (Section 3.5.1). Triolet further generalizes over different storage formats, supporting both access to individual elements of a container (Section 3.5.2) and collective access to all elements of a container (Section 3.5.3). In this way, the same library interface can be used to access different data types having different performance characteristics.

3.5.1 Domains

Containers in Triolet denote finite maps with restricted domains. That is, a container denotes a set (its *domain*) drawn from a set of sets (a *domain type*), together with a function associating a value to each of the set's elements. The elements of the set are called *indices*. Domains originated as a programming language abstraction in ZPL [70, 71], where they were called "regions." Chapel [43] and Haskell [72] provide similar abstractions for re-mapping array indices. Whereas ZPL, Chapel, and Haskell use concrete arrays as the underlying storage format, Triolet abstracts over storage formats using the overloaded interfaces in Sections 3.5.2 and 3.5.3.

Programmers use 1D arrays for (at least) two different ways of organizing data, whose indexing semantics differ. A collection of data may represent an *ordered sequence*, such as a

sequence of images ranked from lightest to darkest. An element's position is relative ("x is in fifth place"). A collection may instead represent a *mapping* from integer indices to values, such as the values of an electronic signal sampled at times $0, 1, \ldots$. An element's position is absolute ("x is at position 5"). Removing the element at position 0 means something different in each case: in the former, x becomes fourth among the remaining elements, but in the latter, x's associated sampling time and thus its position is unchanged. In Triolet, these semantics are captured by different domain types. Indices of ordered sequences are described by values of type `list_dim`. Indices of regularly spaced, integer-to-value mappings are described by `dim1`.

A value of type `list_dim` describes the index space of a container, such as a list, whose elements represent an ordered sequence of values. A list's domain is the set $\{n \mid 0 \leq n < l\}$, and the list contains a concrete array holding one value for each element of the domain. For some operations on domains, it is useful to have a distinguished domain that is larger than all other domains. The largest list domain represents the set $\{n \mid 0 \leq n\}$ of all nonnegative integers. In Triolet, the domain type of lists is `list_dim`, which denotes the set of domains $\{\{n \mid 0 \leq n < l\} \mid l \in \mathbb{N} \cup \{\infty\}\}$. A `list_dim` data structure holds the upper bound of a domain, i.e., the value of l. The list $[0.5, 2.5, 1.5]$ denotes the domain $\{0, 1, 2\}$ and mapping $\{0 \mapsto 0.5, 1 \mapsto 2.5, 2 \mapsto 1.5\}$.

A `dim1` describes the index space of a container whose elements represent a one-dimensional array. The counterpart of `list` for this domain type is called `array1`. Like a list, its elements are stored in a concrete array. Like a region in ZPL, a `dim1` can have an arbitrary lower bound and arbitrary positive stride between elements. It is defined in this way so that a slice expression such as `a[l:u:s]` can be understood as restricting `a` to the subdomain given by the slice `l:u:s`. In the common case where the stride s is 1, a `dim1` denotes the set of all integers between a given lower bound l and upper bound u, including l but excluding u. In the general case, a `dim1` additionally has a stride $s > 0$ and alignment $a = l \mod s$, and denotes the set of integers between l and u that are multiples of s offset by a, $\{n \mid l \leq n < u \wedge \exists m.\, n = ms + a\}$.

Higher-dimensional array domains consist of a `dim1` for each dimension, and denote the Cartesian product of the `dim1` sets. Triolet provides domain types `dim2` and `dim3`, and array types `array2` and `array3`, analogous to the 1D types.

3.5.2 Indexable Containers

An *indexable* container type supports random access to its elements. Indexable containers are a generalization of arrays. Each indexable type has three associated operations. First, an indexable object can be subscripted, using the subscript operator, to extract a value at a given index. The expression `a[9]` extracts the element of `a` at index `9`. The expression `b[(0,1,2)]` extracts the element of `b` at index `(0,1,2)`. Second, an indexable object can be sliced to extract a subset of its elements. The slice expression `a[5:50]` extracts the elements of `a` at indices `5` up to and excluding `50` in the form of a new indexable container (whose exact type is hidden and implementation-dependent). In the `list_dim` domain, a slice's elements are shifted so that indices start from zero; in other domains, elements keep their original indices. Third, the domain of an indexable object can be retrieved by calling `domain`. A client that uses an indexable container can access its contents in whatever way is convenient.

The indexable interface is also used as internal abstraction layer within the library to parallelize loops. Divide-and-conquer parallel kernels examine their input's domain when operating on a whole data structure, such as allocating storage for a new output array. Parallel kernel slice a loop's input data structure to partition it across threads. Individual threads repeatedly index into their input data structures. To afford the library some flexibility in how it schedules work, data structures that can be accessed in parallel expose a less general interface, described in the next section.

3.5.3 Traversable Containers

A *traversable* container type is one that provides a method of looping over, or *traversing* its elements. The `iter` function creates an iterator that loops over a container's elements. The

`build` function assembles a new container from the elements of an iterator. These two methods together constitute the traversable interface. These functions are usually not needed in Triolet source code since most container functions call `iter` on their argument. For instance, the expression `[2*x for x in a]` is syntactic sugar for `build(map(lambda x: 2*x, iter(a)))`. It calls `iter` on the input, `a`, to convert it to an iterator, calls `map` to transform the iterator's contents, and calls `build` on the transformed iterator to convert it to a container. The input and output container types are determined by type inference based on the surrounding code.

Triolet's iterators are a fairly complicated data structure because they carry information about potential parallelism, parallel communication, and loop nesting; their internal structure is the subject of Chapter 5. Semantically, iterators are a traversable data structure whose contents are computed lazily. They can be understood as a sort of restricted array that has different performance characteristics.

CHAPTER 4

Compiling Triolet Code

The Triolet compiler is responsible for eliminating the many levels of abstraction in Triolet code, yielding high-performance object code. Triolet source code heavily utilizes overloading, generic functions, first-class functions, and dynamically allocated data. These features, if not dealt with by compile-time optimizations, carry a heavy run-time penalty. Optimized, high-performance Triolet code spends most of its execution time reading and writing pre-allocated arrays, computing with scalar data held in registers or on the stack, and branching to statically known instructions.

The compiler's organization, shown in Figure 4.1, broadly follows the design of other compilers for statically typed, functional languages, such as GHC [57] and TIL [58]. The compiler frontend translates the language's full syntax into a simpler internal language while elaborating type and representation information that is implicit in source code. The output of the frontend is a small functional language based on System F [73]. High-level optimizations are performed at this level. These optimizations are designed to minimize the frequency of function calls and dynamic creation of heap-allocated data structures, both of which are costly in general. After high-level optimization, the internal representation is lowered to a low-level, imperative *backend* language that is a variant of Administrative Normal Form [74]. While the backend language still has first-class functions, its data types and other operations are not far removed from processor-level instructions. The backend's primary responsibility is to generate efficient code from function calls. Low-level optimizations perform additional inlining and simplification. Closure conversion translates first-class functions into global functions, data structure manipulation, and transfers of control flow [60, 75]. Finally, C

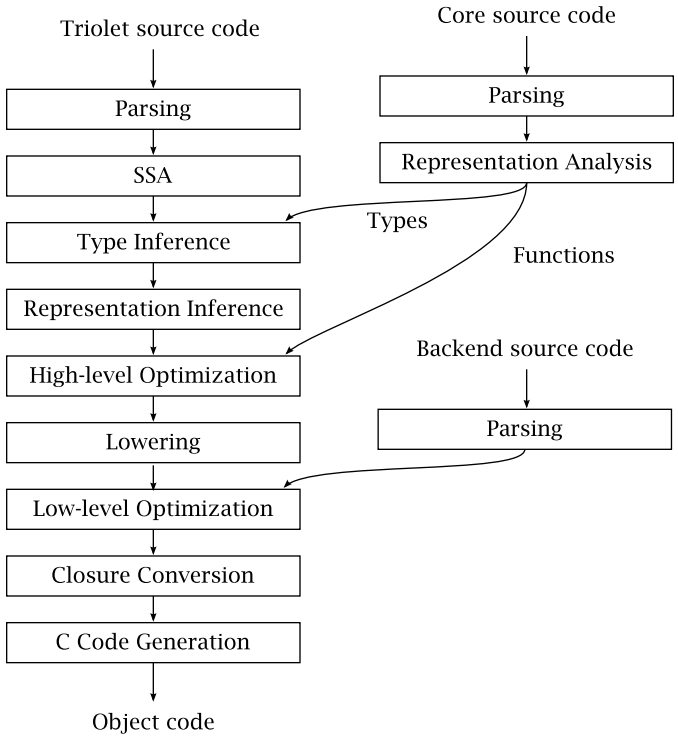


Figure 4.1: Flowchart showing stages of the Triolet compiler.

code is generated and processed through the GCC compiler, which employs a large suite of instruction-level optimizations.

In addition to the compilation path for Triolet source code along the left of Figure 4.1, the compiler processes library code written in a human-readable form of the high-level and low-level intermediate languages. These steps are shown along the right side of the figure. Type declarations in library code are taken into account during type inference of Triolet code. Library functions are compiled to object code, and they may also be inlined during high-level or low-level optimizations.

The compiler is shaped by the need to translate Triolet source to the high-level internal representation, optimize, and then translate to object code. Accordingly, the *core* language used by high-level optimizations is presented first, in Section 4.1. Section 4.2 summarizes the optimizations performed by the compiler. While none of these optimizations are new, some have been adjusted to better handle data structures and optimizations. Section 4.3 presents

the backend of the compiler.

4.1 Core Language

Figure 4.2 presents Triolet's core language, where most optimizations take place. Core is an extension of System F [73]. Like System F, the core language is functional and statically typed. Evaluation is eager. The core language's distinctive feature is its support for unboxed object fields and mutable initialization of fields. These features support efficient initialization and reading of numeric arrays in generic, parallel library code.

Triolet uses one of three storage strategies for run-time objects, depending on their type. *Boxed* objects are dynamically or statically allocated pass-by-reference data. Boxing is a simple storage strategy to implement, but it imposes a modest cost: the dynamic allocator is invoked each time a boxed object is created, metadata is added to each object to support runtime services such as marshaling, and a memory operation is performed to read or write an object field. These costs constitute a significant overhead for small objects such as numbers. This overhead can be reduced by consolidating many objects into fewer, larger boxed objects. Objects are called *unboxed* when their storage is allocated within other objects or program variables. Triolet distinguishes two unboxed storage strategies. *Value* objects are unboxed objects with pass-by-value semantics. They can be decomposed into primitive types and allocated to program variables. *Bare* objects are unboxed objects with pass-by-reference semantics. Since any bare object is represented by a reference, they can be used in code where the object's type is unknown. The two unboxed storage strategies each play a role in high-performance code. Value objects represent small data that can be held directly in registers. Bare objects represent compact aggregations of data, such as arrays, whose elements can be located with pointer arithmetic rather than by dereferencing pointers.

Types are classified into storage strategies using the kind system. This method of tracking storage strategies is adopted from previous work [53]. Kinds are used to classify types based on their storage strategy. An object's type may have kind box, val, or bare. References to

Type constructors	T	Data constructors	C
Type-level integers	N_{type}	Type variables	$\alpha, \beta, \gamma, \ldots$
Integer literals	N	Value variables	$\ldots, x, y, z$

Kinds	$\kappa, \iota ::= \mathsf{box} \mid \mathsf{val} \mid \mathsf{bare} \mid \mathsf{mut} \mid \mathbb{Z} \mid \kappa \to \kappa$
Types	$\tau, \pi ::= \alpha \mid \tau\ \tau \mid \forall \alpha : \kappa.\ \tau \mid \lambda \alpha : \kappa.\ \tau \mid \tau \to \tau$
	$\mid T \mid N_{\text{type}} \mid \mathsf{Int} \mid \mathsf{Arr} \mid \mathsf{Mut} \mid \mathsf{Sto}$
Data types	$data ::= \textbf{data}\ T\ \overline{\alpha : \kappa} : \kappa\ \textbf{where}\ C\ \overline{\alpha : \kappa}\ \overline{\tau}$
Expressions	$c, d, e ::= x \mid N \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{error}\ \tau$
	$\mid e\ e \mid e\ \tau \mid \lambda x : \tau.\ e \mid \Lambda \alpha : \kappa.\ e$
	$\mid \textbf{case}\ e\ \textbf{of}\ \overline{e} \Rightarrow C\ \overline{\alpha}\ \overline{x}.\ e \mid \overline{e} \Rightarrow C\ \overline{\tau}\ \overline{\tau}\ \overline{e}$

Figure 4.2: Syntax of Core.

imperatively accessed bare objects have types of kind mut. The kind $\mathbb{Z}$ classifies type-level integers, used for array sizes.

Types include the usual terms of System F plus built-in and user-defined constants. We write type-level integers with a subscript (e.g., 3_{type}) to distinguish them from integer values. Since we do not perform arithmetic on type-level integers, they are merely symbolic constants as far as the compiler is concerned. Int is an unboxed integer type. Unboxed array types are built with the primitive type constructor Arr. An Arr τ π, for any τ and π, is an unboxed object consisting of τ instances of π laid out consecutively in memory. Unboxed arrays are a building block for user-level array objects, which are heap-allocated and have arbitrary size. A Mut τ is a reference to a mutable object of type τ. A Sto is a store fragment [76, 77] denoting the state of a piece of mutable data at some point in time. Types are erased in the compiler backend and do not exist at run time.

Data type definitions create additional type-level constants. To simplify the presentation, data type definitions are shown for single-constructor data types only. The data type definition **data** $T\ \overline{\alpha : \kappa} : \kappa_T$ **where** $C\ \overline{\beta : \iota}\ \overline{\tau}$ defines a new type constructor T and data constructor C. Objects built with C have field types $\overline{\tau}$, which depend on type parameters $\overline{\alpha}$ determined by the context where the object is used and existential types $\overline{\beta}$ determined when the object is created.

Expressions include variables, application, and abstraction terms as in System F. Integer

literals have type Int. Let expressions bind the result of an expression to a variable. An error expression, which represents an unrecoverable run-time error, interrupts execution. A data expression $\overline{d} \Rightarrow C\ \overline{\tau}\ \overline{\pi}\ \overline{e}$, for some data constructor C with associated type constructor T, creates a new object of type $T\ \overline{\tau}$ containing existential types $\overline{\pi}$ and values computed by $\overline{e}$. A case expression **case** c **of** $\overline{d} \Rightarrow C\ \overline{\alpha}\ \overline{x}.\ e$ reads the contents of the object computed by c into type variables $\overline{\alpha}$ and variables $\overline{x}$ for use in expression e. Both case and data expressions take *size parameters* $\overline{d}$ holding run-time size information. When an expression has no size parameters, we omit the $\Rightarrow$ symbol.

Imperative computation is encoded using store-passing. A store fragment is a value representing a particular state of some part of memory. An imperative update to memory takes a store fragment (representing the old state of memory) as input and produces a new one (representing the new state). Passing store fragments from one imperative operation to the next makes dependences visible to the compiler so that they are preserved during optimization. Multiple store fragments representing disjoint parts of memory can exist simultaneously. For instance, in a parallel loop writing an array, each parallel task access a disjoint piece of memory. Each task operates on an array fragment whose state is represented by a store fragment; concurrent tasks operate independently on separate store fragments. Note that Triolet does not attempt to analyze imperative side effects, nor guarantee their safety, so the compiler and type system do not need special machinery for representing store fragments.

4.1.1 Kind System

Core generalizes System F's kind system to support multiple storage strategies as shown in Figure 4.3. Where System F has only the kind $\star$ of proper types, Core has the four kinds box, val, bare, and mut. The kinding rules statically assign one of these kinds to every variable, expression, and object field. Kind information is used during optimization and code generation.

Functions are boxed and can take and return objects of any proper type. Since universal quantification has no run-time effect, a universally quantified type $\forall \alpha{:}\kappa.\tau$ has the same kind

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \pi : \kappa_1}{\Gamma \vdash \tau\ \pi : \kappa_2}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2 \qquad \kappa_2 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}\}}{\Gamma \vdash \forall \alpha : \kappa_1 . \tau : \kappa_2} \qquad \frac{\begin{array}{l}\Gamma \vdash \tau : \kappa_1 \qquad \kappa_1 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}, \mathsf{mut}\} \\ \Gamma \vdash \pi : \kappa_2 \qquad \kappa_2 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}, \mathsf{mut}\}\end{array}}{\Gamma \vdash \tau \to \pi : \mathsf{box}}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha : \kappa_1 . \tau : \kappa_1 \to \kappa_2}$$

$$\mathsf{Int} : \mathsf{val} \qquad \mathsf{Arr} : \mathbb{Z} \to \mathsf{bare} \to \mathsf{bare} \qquad \mathsf{Mut} : \mathsf{bare} \to \mathsf{mut} \qquad \mathsf{Sto} : \mathsf{val}$$

Figure 4.3: Kinding rules and kinds of constants in Core.

as τ. Objects of kind box, val, and bare can have universally quantified types. Other kinding rules are the same as in System F.

Algebraic data types must have kind box, bare, or val. Consider a data type definition, **data** $T\ \overline{\alpha : \kappa} : \kappa_T$ **where** $C\ \overline{\beta : \iota}\ \overline{\tau}$. The data type's storage strategy κ_T must be one of these three kinds. Furthermore, the field types $\overline{\tau}$ must all have one of these three kinds. Two additional restrictions apply to unboxed data type definitions. First, data types of kind val may only contain value and boxed fields, so that their contents can be allocated to registers. Bare fields cannot be stored in registers. Second, recursively defined types must be boxed, so that unboxed objects of a given type have a bounded size. Since the kind annotation κ_T on a data type definition determines the storage strategy of the type, memory layout decisions are made on a per-type basis.

4.1.2 Data Constructor Type Signatures

Data constructors are assigned type signatures summarizing their behavior. These signatures are used, among other things, to compute memory layouts. Two forms of type signature are used. Initially, a data constructor's type signature is taken directly from its data type definition. A definition **data** $T\ \overline{\alpha : \kappa} : \kappa_T$ **where** $C\ \overline{\beta : \iota}\ \overline{\tau}$ introduces a constructor C with type signature $\forall \overline{\alpha : \kappa} . \forall \overline{\beta : \iota} . \overline{\tau} \to T\ \overline{\alpha}$. The signature relates the type of an object $T\ \overline{\alpha}$, to the types of its fields, $\overline{\tau}$, for all choices of α and β. Type signatures are then refined (Section 4.1.4) by adding a list of size parameter types $\overline{\tau_s}$ to indicate what run-time size

information is required when accessing an object of type T. Refined signatures are written $\forall \overline{\alpha : \kappa}.\, \overline{\tau_s} \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\alpha}$. Refined signatures are used for type checking and lowering.

Sometimes it is necessary to match, or *instantiate*, a constructor signature to a given type $T\ \overline{\pi}$ in order to determine what a value of that type contains. The notation $C \geq \mathit{sig}$ is used to mean that C's signature is instantiated to the signature *sig*. For a constructor $C : \forall \overline{\alpha : \kappa}.\ \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\alpha}$ and any well-kinded substitution $\theta = [\overline{\pi / \alpha}]$, we have $C \geq \forall \overline{\beta : \iota}.\, \overline{\theta(\tau)} \to T\ \overline{\pi}$. When a particular choice of existential types $\overline{\pi_e}$ is also given, we write $\geq_{\overline{\pi_e}}$ for instantiating the fields to those types. Letting $\theta_e = [\overline{\pi_e / \beta}]$, we have $C \geq_{\overline{\pi_e}} \overline{\theta_e(\theta(\tau))} \to T\ \overline{\pi}$.

4.1.3 Memory Layout

An object is represented concretely as a sequence of primitive values. These values may be held in variables or laid out sequentially in memory. To generate code that creates, reads, or writes an object requires knowledge of its structure in terms of primitive values, which we call its *layout*. This section defines layouts and associates core types with layouts.

To simplify the presentation, this section assumes that memory is word-addressed, primitive types occupy one word, and algebraic data types have a single constructor. The Triolet compiler passes run-time size and alignment information and inserts padding bytes for alignment. Multi-constructor unboxed types are tagged unions having the maximum size and alignment of all constructors. While these details entail more complex calculation, they follow the same overall framework.

Layouts are nested sequences of primitive values:

$$\varsigma ::= \mathsf{Int} \mid \mathsf{Ptr} \mid \langle\rangle \mid \varsigma \times \varsigma$$

The primitive layouts Int and Ptr occupy one word of memory. The unit layout $\langle\rangle$ occupies zero words of memory. The product layout $\varsigma_1 \times \varsigma_2$ consists of a ς_1 abutting a ς_2, like a two-member `struct` in C. Nested products $\varsigma_1 \times (\varsigma_2 \times \cdots)$ are abbreviated $\langle \varsigma_1, \varsigma_2, \ldots \rangle$ or $\langle \overline{\varsigma} \rangle$. An unboxed array is a product of N instances of ς, written ς^N. Often, only the number of words

$$\frac{\Gamma \vdash \tau : \mathsf{box}}{\Gamma \vdash \tau \downarrow_{\text{field}} \mathsf{Ptr}} \qquad \frac{\Gamma \vdash \tau : \kappa \qquad \kappa \in \{\mathsf{val}, \mathsf{bare}\} \qquad \Gamma \vdash \tau \downarrow \varsigma}{\Gamma \vdash \tau \downarrow_{\text{field}} \varsigma}$$

$$\frac{N \propto N_{\text{type}} \qquad \Gamma \vdash \tau \downarrow_{\text{field}} \varsigma}{\Gamma \vdash \mathsf{Arr}\ N_{\text{type}}\ \tau \downarrow \varsigma^N} \qquad \frac{}{\Gamma \vdash \mathsf{Int} \downarrow \mathsf{Int}}$$

$$\frac{C \geq \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\pi} \qquad \Gamma, \overline{\beta : \iota} \vdash \overline{\tau \downarrow_{\text{field}} \varsigma}}{\Gamma \vdash T\ \overline{\pi} \downarrow \langle \overline{\varsigma} \rangle} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau \downarrow \varsigma}{\Gamma \vdash \forall \alpha : \kappa.\, \tau \downarrow \varsigma}$$

Figure 4.4: Inference rules for memory layout of Core types.

occupied by a layout matters. The size in words of ς is written $\text{size}(\varsigma)$. The size of Int or Ptr is 1, the size of $\langle\rangle$ is 0, and the size of $\varsigma_1 \times \varsigma_2$ is $\text{size}(\varsigma_1) + \text{size}(\varsigma_2)$.

An object is laid out as a sequence of fields, which we write as a product. Since boxed fields are just pointers, an object with two boxed fields would have layout $\langle \mathsf{Ptr}, \mathsf{Ptr} \rangle$. An object's unboxed fields become part of the object's layout. For example, the layout of a Stored Int is $\langle \mathsf{Int} \rangle$, rather than $\langle \mathsf{Ptr} \rangle$, since its field is unboxed. A Tuple (Stored Int) (Stored Int) has two unboxed fields, and each field has layout $\langle \mathsf{Int} \rangle$, so the tuple's layout is $\langle \langle \mathsf{Int} \rangle, \langle \mathsf{Int} \rangle \rangle$, which is a sequence of two Ints sitting in memory.

Memory layout rules are presented in Figure 4.4. The layout of a type is given by a judgment of the form $\Gamma \vdash \tau \downarrow \varsigma$, meaning that in type environment Γ, an object of type τ has layout ς. The layout rules can be read as an algorithm for computing ς from Γ and τ. As an algorithm, layout computation proceeds by structural recursion on τ and on the fields of objects. The related judgment $\Gamma \vdash \tau \downarrow_{\text{field}} \varsigma$ means that an object field of type τ has layout ς. The first two rules state that the layout of a boxed field is Ptr, while the layout of an unboxed field of type τ is the layout of τ.

The remaining rules give the layouts of types. The layout of an array $\mathsf{Arr}\ N_{\text{type}}\ \tau$ is computed by finding the layout of τ, then creating a product of N instances of that layout. We write $N \propto N_{\text{type}}$ to mean that the type-level integer N_{type} represents the same number as the integer N. The layout of Int is simply Int. The next rule gives the layout of any algebraic data type $T\ \overline{\pi}$. The layout of the type is the product $\langle \overline{\varsigma} \rangle$ of the layouts of its fields. The

field types, found from the type parameters and constructor type signature, determine the field layouts. The final rule gives the layout of a universally quantified type. Since universal quantification has no run-time effect, the layout of $\forall \alpha : \kappa . \tau$ is the layout of τ. Functions are opaque objects and are not assigned a layout.

Some first-class polymorphic types cannot meaningfully be assigned a layout, because an object of that type can represent multiple types having mutually inconsistent layouts. If we attempt to derive the layout of such a type, we will eventually ask for the layout of a $\forall$-bound or existential type variable. None of the rules in Figure 4.4 give the layout of a type variable; after all, an unknown type has an unknown layout. Layouts dependent on an existential variable are reported as errors when data type definitions are analyzed (Section 4.1.4). Layouts dependent on a $\forall$-bound variable are detected in code during lowering. While detecting errors during lowering is sufficient to rule out invalid code, detecting errors before optimization would likely yield more consistent error reports. Early error detection is not implemented. Such errors can be fixed by boxing problematic objects or fields so their layout does not affect the enclosing object's layout.

4.1.4 Using Layouts for Polymorphic Code Generation

The compiler uses layout information for generating code in several instances. Layouts are used when generating code to construct or inspect types, discussed in detail in Section 4.3. Layout information is used to compute a type's size, for example when allocating storage or indexing into an array. It is used for generating serialization and deserialization code for each data type. In each case, the computation proceeds recursively over a data type's layout. Due to polymorphism, some information is not available at compile time. The compiler arranges for this information to be passed as run-time parameters.

To find what type information should be passed at run time, the compiler analyzes each data type definition to determine parameters associated with that type. Parameters associated with data types are reflected in the types of data expressions, case expressions, and compiler-generated functions for computing sizes, serializing values, and deserializing val-

ues. The types of data constructors and compiler-generated functions indicate what parameters they require. When type checking functions, the compiler verifies that the required run-time parameters are passed at each data expression, case expression, and function call.

To identify run-time parameters associated with a given algebraic data type T, the compiler attempts to compute the layout of its type $T\ \overline{\alpha}$. The computation process looks through the known parts of the type's structure to find parts whose size depends on a type parameter, and thus is not statically known. For example, consider the following definition of an unboxed tuple type.

data Tuple $(\alpha, \beta : \mathsf{bare}) : \mathsf{bare}$ **where** tuple : $\alpha \times \beta \to$ Tuple $\alpha\ \beta$

To compute the size of a Tuple $\alpha\ \beta$, the size of the data constructor tuple is computed. It contains fields of type α and β, and so its size depends on the sizes associated with those two type parameters. Therefore, the size of a Tuple $\alpha\ \beta$ is a function of the sizes Sz α and Sz β. After adding size parameters, the type signature of tuple is $\forall \alpha, \beta : \mathsf{bare}.$ $(\mathsf{Sz}\ \alpha, \mathsf{Sz}\ \beta) \Rightarrow \alpha \times \beta \to \mathsf{Tuple}\ \alpha\ \beta$.

Polymorphic code may be reused to manipulate multiple data types having different sizes. In such code, size-computing expressions generated by the compiler will include terms that stand for the size of a type variable's layout. Since such sizes cannot be computed statically, programs are required to supply each unknown size where needed by passing size parameters.

4.1.5 Type Checking

Typing in core is similar to System F in most respects. Unboxing support imposes some extra conditions, as some information must be fixed at compile time and some information must be passed at run time. Extra conditions imposed by unboxing are described here along with the full typing rules for data and case expressions.

In order for the backend to break down value objects into primitive values that fit in registers, all variables and expressions of kind val must have a statically known layout. The

auxiliary judgment $\Gamma \vdash \tau$ ok defined below holds if τ has a statically known layout or if it has a pass-by-reference storage strategy.

$$\frac{\Gamma \vdash \tau : \mathsf{val} \qquad \Gamma \vdash \tau \downarrow \varsigma}{\Gamma \vdash \tau \text{ ok}}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \kappa \in \{\mathsf{bare}, \mathsf{box}, \mathsf{mut}\}}{\Gamma \vdash \tau \text{ ok}}$$

This judgment is added to System F's typing rules to ensure that variables introduced by λ, Λ, and case terms are ok, and that the result of every expression is ok.

The typing rules for case and data expressions use the type signature of a given data constructor to relate the type of an object to the types and kinds of its existential types, fields, and size parameters. Bare objects are typed differently in data expressions since they are initialized as mutable objects. We write $\Gamma \vdash e : \text{init}(\tau)$ to mean that e is a τ or an initializer for τ. That is, it is shorthand for $\Gamma \vdash e : \mathsf{Mut}\ \tau \to \mathsf{Sto}$ if τ has kind bare, or $\Gamma \vdash e : \tau$ otherwise.

$$\frac{\begin{array}{c} C \geq \overline{\tau_s} \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\pi} \qquad \Gamma \vdash e : T\ \overline{\pi} \\ \Gamma, \overline{\beta : \iota}, \overline{x : \tau} \vdash c : \pi_c \qquad \Gamma \vdash \overline{d : \tau_s} \qquad \Gamma \vdash \overline{\tau \text{ ok}} \end{array}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \overline{d} \Rightarrow C\ \overline{\beta}\ \overline{x}.\, c : \pi_c}$$

$$\frac{\begin{array}{c} C \geq \overline{\tau_s} \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\tau_u} \\ \Gamma \vdash \overline{\tau_e : \iota} \qquad \Gamma \vdash \overline{e : \text{init}(\tau[\overline{\tau_e}/\overline{\beta}])} \qquad \Gamma \vdash \overline{d : \tau_s} \end{array}}{\Gamma \vdash \overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e} : \text{init}(T\ \overline{\tau_u})}$$

4.2 High-Level Optimizations

The high-level optimizations in Triolet closely follow those in GHC [61, 57], since the internal languages of both compilers are similar. Triolet's high-level optimization phases are summarized here. Since some optimizations enable others, the phases are run repeatedly.

Expression flattening Expressions are restructured to increase other opportunities for optimization. The general effect of restructuring is to produce a flatter-looking program containing longer sequences of **let** and **case** expressions. Expressions bound to variables are

decomposed into smaller pieces, and variables become visible over larger scopes, both of which aid later optimizations. An example of flattening is to transform **let** z = (**let** y = $f\ x$ **in** $1 + y$) **in** $h\ z$, which assigns the result of a complicated expression to z, into **let** $y = f\ x$ **in** (**let** $z = 1 + y$ **in** $h\ z$), where the right-hand side of every assignment is a simple expression.

Simplification The compiler performs a large number of optimizations in a forward pass over the program [57]. Some of these optimizations directly reflect reduction rules; consequently, they have the combined effect of evaluating parts of the program whose outcome is known statically.

While most of Triolet's simplification steps follow previous methods, Triolet's inlining strategy was refined to support library-driven optimization of recursive functions. During simplification, inlining copies the definition of a function to a site where the function is called. The primary benefit of inlining is that it enables further simplification of the function body using information available at the callsite. Its drawback is that it replicates code, potentially increasing the compilation time and program size. In library-driven optimization, some functions are meant to be inlined and specialized for the parameter values given at their callsites. Programmers can annotate important functions to direct the compiler to aggressively inline them. However, inlining recursive functions tends to result in an unprofitable code explosion. Compilers generally do not inline recursive functions. An early implementation of Triolet's compiler inlined recursive functions up to a fixed depth. While this aggressive strategy did inline performance-critical functions, it also led to code bloat and long compilation times, motivating a targeted approach.

Triolet introduces inlining annotations to profitably inline recursive functions with structurally decreasing arguments. Calls are inlined only if the structurally decreasing components of the function arguments are statically known. This inlining technique can be demonstrated with the tree leaf counting function from Section 2.6, repeated here.

```
numLeaves t = case t
                of IntBranch ℓ r. numLeaves ℓ + numLeaves r
                   IntLeaf x. 1
```

Suppose a program contains the expression numLeaves (IntBranch (IntLeaf 1) (IntLeaf n)). Since numLeaves is recursive, it could be inlined an arbitrary number of times. However, only three instances are useful: inlining the original call enables the IntBranch term to be processed statically, then inlining the two recursive calls enables the IntLeaf terms to be processed statically. On the other hand, if a program contains the expression numLeaves (foo 500), and foo is not inlined, then inlining is not useful at all because it is not known whether the result of foo is an IntBranch or IntLeaf. Inlining is only profitable when the argument's data constructor is statically known.

In Triolet's core language, the function definition can be annotated with an inlining hint, `inline_struct(C)`, to enable recursive inlining. The letter C means that inlining is enabled only if the argument's constructor is known, while T means it does not matter. In general there is one letter per function parameter. When used on structurally decreasing parameters, this inlining method cannot lead to runaway code growth because inlining occurs at most once for each statically known data constructor, and new constructors are not introduced in the inlined code. Triolet does not verify that parameters are structurally decreasing, but such checks are possible.

Triolet's library expresses nested traversals with a recursive data type as discussed in Section 5.2. Loops implemented in the library process this data type recursively and should be inlined. Inlining is practical since typical programs have shallow loop nesting (and thus a shallow recursion depth). In the unusual case where the nesting depth is not statically known, recursion is not expanded, avoiding code explosion.

Occurrence analysis Occurrence analysis [38] detects how often each variable is used after it is bound. The analysis pass annotates variable bindings with information about their uses. This information is used for deciding which optimizations are beneficial. For instance, if a

function is never called, its definition can be deleted (and the occurrence analyzer does so); if it is called in exactly one place, inlining it will eliminate a function call without increasing program size and is always beneficial; if it is called in many places, inlining will increase code size and should only occur if its benefits outweigh this cost.

The analysis also identifies functions that are certain to be called at least once. The "certain to be called" property is similar to postdominance [78]. If a function is definitely called, it is safe to hoist code out of the function body. Otherwise, hoisting is unsafe because it may cause a program to execute code that it did not originally execute, increasing execution time or causing a program to diverge.

Common subexpression elimination The use of overloaded operations translates into code for dynamically constructing and consulting dictionaries of overloaded methods. After other optimizations, a function may end up containing multiple pieces of code that build the same method dictionaries. Triolet employs common subexpression elimination to eliminate this redundant code. Because overloading is always resolved in the same way for any given type, any two dictionaries of the same type must be identical. This observation leads to a simple common subexpression elimination policy that checks whether types are equal, rather than whether expressions are identical. If an expression has a dictionary type, and a variable of the same type is in scope, the expression can be replaced by the variable.

Hoisting Hoisting moves variable bindings closer to the beginning of the function they are defined in and pulls them out of nested functions, as long as they do not depend on locally defined variables. Hoisting increases the scope of a variable binding, improving opportunities for other optimizations. When code is hoisted out of a function, it may also reduce the number of times the code is executed.

Data structure flattening Data structure flattening [64] eliminates boxing of values passed to and returned from functions via direct function calls. The main benefit of this transformation is to remove memory allocation from inner loops. Innermost loops typically compile

Types	$\sigma ::= \mathsf{Int} \mid \mathsf{Ptr} \mid \langle\rangle \mid \sigma \times \sigma$
Values	$v ::= x \mid N \mid \langle\rangle \mid v \times v$
Patterns	$p ::= v \mid \langle\rangle \mid p \times p$
Expressions	$e ::= v \mid e\ e \mid e\ op\ e \mid \lambda x : \sigma.\ e \mid \textbf{let}\ p = e\ \textbf{in}\ e$
	$\mid \textbf{load}_\sigma\ e \mid \textbf{store}_\sigma\ e\ e \mid \textbf{alloc}\ e$

Figure 4.5: Syntax of the backend's intermediate representation.

to a single, recursive function. While other optimizations can typically eliminate memory allocation from the body of the loop, the call to the next iteration cannot be eliminated by inlining, so other optimizations do not get a chance to eliminate the allocation of any values passed to the next iteration. Data structure flattening "unpacks" data structures into their fields, which are passed as multiple arguments or return values.

4.3 Backend

The compiler's backend serves as a bridge from Core to low-level C code. In contrast to Core's data types, the backend language's abstraction of data is close to the physical processor implementation. Local variables hold small, monomorphic structures that can potentially be put into processor registers. Data structures reside either in local variables or in memory. Memory is accessed with load and store operations. The first stage of the compiler's backend translates case and data expressions into memory operations and packing and unpacking of local variables.

The backend language is shown in Figure 4.5. It is monomorphic and does not associate type information with pointers or memory locations. Types σ are defined identically to layouts ς. Products are unpacked using the multi-assignment pattern $\textbf{let}\ x \times y = \ldots\ \textbf{in}\ \ldots$. In types, values, and patterns, we abbreviate nested products $\sigma_1 \times (\sigma_2 \times \cdots)$ as $\langle \sigma_1, \sigma_2, \ldots \rangle$ or $\langle \overline{\sigma} \rangle$. There are the usual primitive operations for integer and pointer arithmetic. A value of type σ is loaded from address e by $\textbf{load}_\sigma\ e$ and stored to a pointer by $\textbf{store}_\sigma\ e$. An e-word block of memory is allocated by $\textbf{alloc}\ e$.

$$\text{read}(\Gamma, p, n, \tau) = \begin{cases} \textbf{load}_{\sigma}\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{val} \text{ and } \Gamma \vdash \tau \downarrow \sigma \\ \textbf{load}_{\mathsf{Ptr}}\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{box} \\ p+n & \text{if } \Gamma \vdash \tau : \mathsf{bare} \end{cases}$$

$$\text{write}(\Gamma, p, n, \tau, e) = \begin{cases} \textbf{store}_{\sigma}\ (p+n)\ e & \text{if } \Gamma \vdash \tau : \mathsf{val} \text{ and } \Gamma \vdash \tau \downarrow \sigma \\ \textbf{store}_{\mathsf{Ptr}}\ (p+n)\ e & \text{if } \Gamma \vdash \tau : \mathsf{box} \\ e\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{bare} \end{cases}$$

Figure 4.6: Code generated for reading and writing object fields. Code generation uses a type environment Γ, object pointer p, field offset n, and field type τ.

When generating address calculation for the fields of an object, the compiler computes a size for each of the object's fields as described in Section 4.1.4. The object's size is the sum of the sizes of all its fields, and the offset of each field is the sum of the preceding fields' sizes. We write $\text{layout}(\Gamma, T\ \overline{\pi}, \overline{e})$ for computing the size and field offsets of $T\ \overline{\pi}$ using type environment Γ. The values of size parameters are supplied by $\overline{e}$. For instance, $\text{layout}(\Gamma, \mathsf{Tuple}\ \alpha\ \beta, (s_\alpha, s_\beta))$ produces the expression $\textbf{let}\ \langle m \rangle = s_\alpha\ \textbf{in let}\ \langle n \rangle = s_\beta\ \textbf{in}\ (m + n) \times \langle 0, m \rangle$, which evaluates to a product containing the size of a polymorphic tuple and the offsets of its two fields.

Individual object fields are read and written differently depending on their storage strategy (Figure 4.6). We write $\text{read}(\Gamma, p, n, \tau)$ for reading an object of type τ from the address $p + n$, and $\text{write}(\Gamma, p, n, \tau, e)$ for using the object or initializer returned by e to write an object of type τ to the address $p + n$. For writing multiple fields, we extend read and write to lists: $\text{read}(\Gamma, p, \overline{n}, \overline{\tau})$ and $\text{write}(\Gamma, p, \overline{n}, \overline{\tau}, \overline{e})$. Boxed and value objects are read and written by transferring data between a variable and memory. The data is either an object or a pointer. Bare fields, on the other hand, cannot be transferred to a variable. Reading a bare field simply returns a pointer to the field. Writing a bare field is accomplished by passing the field's address to a given initializer function.

For example, to lower the data expression $\mathsf{stored}\ 2$, we would first determine that the object being constructed has a field at offset 0 of type Int and kind val. The field is written by $\text{write}(\Gamma, p, 0, \mathsf{Int}, 2)$, which is the store operation $\textbf{store}_{\mathsf{Int}}\ p\ 2$. The rest of the lowering

algorithm wraps this into the initializer $\lambda p : \mathsf{Ptr}.\ \mathbf{store}_{\mathsf{Int}}\ p\ 2$.

With layouts and field-accessing operations, we are now ready to describe the lowering of data and case expressions. We write $\text{lower}[\![e]\!]\Gamma$ for the translation of Core expression e to an equivalent backend expression, using type information from the Core type environment Γ. Most of the time, Γ is simply passed along from one step to another; $\text{lower}[\![e]\!]$ is shorthand for $\text{lower}[\![e]\!]\Gamma$.

An object with the val storage strategy is lowered to a product value. For this storage strategy, lowered data expressions construct product values.

$$\text{lower}[\![C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}]\!] = \langle \overline{\text{lower}[\![e]\!]} \rangle \qquad \text{if } \Gamma \vdash C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e} \text{ has kind val}$$

Lowered case expressions, conversely, unpack product values. In addition to unpacking a value, the lowering algorithm computes the local type environment to use while lowering the body of the case expression. The type $T\ \overline{\pi}$ of the case expression's scrutinee c is used to infer type information for the bound type variables $\overline{\beta}$ and fields $\overline{x}$.

$$\begin{aligned}\text{lower}\left[\!\left[\mathbf{case}\ c\ \mathbf{of}\ C\ \overline{\beta}\ \overline{x}.\ e\right]\!\right] = \mathbf{let}\ \langle \overline{x} \rangle = \text{lower}[\![c]\!]\ \mathbf{in}\ \text{lower}[\![e]\!](\Gamma, \overline{\beta : \iota}, \overline{x : \tau}) \\ \text{where } \Gamma \vdash c : T\ \overline{\pi} \\ \Gamma \vdash C \geq \forall \overline{\beta : \iota}.\ \overline{\tau} \to T\ \overline{\pi} \\ \text{if } \Gamma \vdash T\ \overline{\pi} : \mathsf{val}\end{aligned}$$

Objects with the box or bare storage strategies reside in memory, and operations on them are lowered to memory operations. Lowering a data expression $\overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}$ starts by finding the type $T\ \overline{\tau_u}$ of its result value. Code to compute the value's layout is generated using the type and run-time size information $\overline{d}$. The computed object size and field offsets are used to allocate storage and to write fields. For boxed objects, storage is allocated. For bare objects, the address of the object's storage is passed in as a parameter.

$$\text{lower}\Big[\!\Big[\,\overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}\,\Big]\!\Big] = \begin{array}{l}\textbf{let}\ \langle \overline{n} \rangle \times m = \text{layout}(\Gamma, T\ \overline{\tau_u}, \overline{\text{lower}[\![d]\!]})\ \textbf{in} \\ \textbf{let}\ p = \textbf{alloc}\ m\ \textbf{in} \\ \textbf{let}\ \langle\rangle = \text{write}(\Gamma, p, \overline{n}, \overline{\tau}, \overline{\text{lower}[\![e]\!]})\ \textbf{in}\ p\end{array}$$

$$\text{where}\ \ \Gamma \vdash C \succeq_{\overline{\tau_e}} \overline{\tau} \to T\ \overline{\tau_u}$$

$$\text{if}\ \ \Gamma \vdash T\ \overline{\tau_u} : \mathsf{box}$$

$$\text{lower}\Big[\!\Big[\,\overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}\,\Big]\!\Big] = \begin{array}{l}\textbf{let}\ \langle \overline{n} \rangle \times m = \text{layout}(\Gamma, T\ \overline{\tau_u}, \overline{\text{lower}[\![d]\!]})\ \textbf{in} \\ \lambda p : \mathsf{Ptr}.\ \text{write}(\Gamma, p, \overline{n}, \overline{\tau}, \overline{\text{lower}[\![e]\!]})\end{array}$$

$$\text{where}\ \ \Gamma \vdash C \succeq_{\overline{\tau_e}} \overline{\tau} \to T\ \overline{\tau_u}$$

$$\text{if}\ \ \Gamma \vdash T\ \overline{\tau_u} : \mathsf{bare}$$

Case expressions compute the data type's layout, then read the object's fields. As with value objects, they update the local type environment while processing the body of the expression.

$$\text{lower}\Big[\!\Big[\,\textbf{case}\ e\ \textbf{of}\ \overline{d} \Rightarrow C\ \overline{\beta}\ \overline{x}.\ c\,\Big]\!\Big] = \begin{array}{l}\textbf{let}\ p = \text{lower}[\![e]\!]\ \textbf{in} \\ \textbf{let}\ \langle \overline{n} \rangle \times m = \text{layout}(\Gamma, T\ \overline{\pi}, \overline{\text{lower}[\![d]\!]})\ \textbf{in} \\ \textbf{let}\ \langle \overline{x} \rangle = \text{read}((\Gamma, \overline{\beta : \iota}), p, \overline{n}, \overline{\tau})\ \textbf{in} \\ \text{lower}[\![c]\!](\Gamma, \overline{\beta : \iota}, \overline{x : \tau})\end{array}$$

$$\text{where}\ \ \Gamma \vdash e : T\ \overline{\pi}$$

$$\Gamma \vdash C \succeq \forall \overline{\beta : \iota}.\ \overline{\tau} \to T\ \overline{\pi}$$

$$\text{if}\ \ \Gamma \vdash T\ \overline{\pi} : \mathsf{box}\ \text{or}\ \Gamma \vdash T\ \overline{\pi} : \mathsf{bare}$$

CHAPTER 5

Optimization-Friendly Container Library Design

There is a large semantic gap to be bridged for Triolet's library to deliver high performance. Like other container traversal interfaces, Triolet's functions represent semantically meaningful operations on data rather than mechanisms for controlling work distribution and communication. It is the programmer's job to describe an algorithm and the library's job to execute it efficiently. To economize on memory traffic and parallel communication, the library contains code to distribute parallel work across cluster nodes and cores, to distribute data across nodes, and to gather results efficiently. The library is designed so that much of this decision making is resolved statically by compile-time optimizations. Inner loops, at least, must be constructed statically in order to minimize overhead.

Section 5.1 explains prior approaches to loop fusion. Section 5.2 explains how Triolet's iterators build on prior loop fusion techniques. Section 5.3 describes how mutable data is confined to individual tasks in order to avoid races and missing updates. Section 5.4 generalizes iterators for programming with multidimensional arrays. Section 5.5 describes how Triolet manages parallel tasks and communication. Much of this section pertains to the interaction between library code and compiler optimizations. When discussing interaction with optimizations, code is presented using Triolet's core language from Chapter 4. Notational conventions from Chapter 2 are employed to shorten the code examples: type annotations are elided, function definitions are shown in equational form, some functions are shown as infix operators, and **if** is used as a synonym for Boolean **case**. Additionally, Haskell-style **class** declarations are used to present data type definitions that describe traits of types.

Parts of this chapter appeared in the ACM Symposium on Principles and Practice of Parallel Programming [79]. The material is used with permission.

5.1 Loop Fusion Background

Container traversals express simple container transformations. It may take a composition of several container traversals to write what would be a single parallel loop in a low-level parallel language. Loop fusion merges multiple traversals into a single loop. In a library-driven optimization framework, because loop fusion brings together information from multiple library calls, it is the foundation on top of which other parallel loop implementation techniques are built.

A dot product function, defined as follows, provides an example where performance can benefit from loop fusion.

```
def dot(xs, ys): return sum(x*y for (x, y) in par(zip(xs, ys)))
```

This statement defines dot as a function of xs and ys. The body of dot calls the library functions `zip`, `par`, `map`, and `sum` to sum the elementwise product of `xs` and `ys`. The call of `map` arises from desugaring the list comprehension that performs the elementwise product. Loops execute sequentially by default. The call to `par` designates the loop as parallel, directing the library to use all available parallelism when computing the dot product. Though not computationally intensive enough to benefit from cluster parallelism, dot exhibits the importance of loop fusion. All four library functions are fused into a single loop. The work of `map` and `sum` is parallelized. Data distribution code is built by `zip`.

While `dot` has a relatively simple looping pattern, some algorithms loop over irregular or multidimensional iteration spaces, which adds complexity to loop fusion, work distribution, and result collection. Triolet builds on several existing loop fusion strategies, discussed in this section, to support fusion in a broader variety of cases than any individual loop fusion strategy can do.

At the most basic level, a library may operate only on data structures whose contents are stored in memory, such as arrays. Each function contains a loop that reads its entire input and writes its entire output. Library writers may use parallel algorithms within each function, but they cannot take advantage of loop fusion directly. Compilers may fuse loops by rewrit-

ing known patterns of function calls [6, 8, 56]. For example, any pair of calls to filter (discussed in Section 3.4) matching the pattern filter g (filter f a) can be fused by rewriting it to filter ($\lambda x.\ f\ x$ && $g\ x$) a. That is, extracting the subset of a consisting of values that satisfy predicate f, then extracting the subset of the subset that satisfies predicate g, is the same as extracting the subset of a that satisfies both f and g. The rewritten code does the work of both filter calls in one pass over a. Because this approach involves designing ad-hoc fusion rules for each pattern, its effectiveness is limited by a library implementor's ability to anticipate and make rules for all combinations of function calls. To rewrite the example histogramming code to a single loop, for instance, requires a preexisting histogram-of-doubly-nested-loop function (or a generalization thereof) to be available. It is unlikely that a library would contain such a pattern. A more systematic approach is necessary to fuse a larger inventory of loops.

In an imperative setting, loop fusion interleaves the execution of loops on an iteration-by-iteration basis [80]. Unfortunately, functions with variable-length outputs, such as filter, have dependence patterns that cannot be expressed purely in terms of loop iterations. In the example above, the first iteration of the second call to filter may depend on any iteration of the first call to filter. Consequently, this example cannot be fused by reordering loop iterations.

Triolet uses a relatively simple and robust approach to loop fusion that depends only on general-purpose compile-time optimizations. This approach uses what we call *virtual* data structures in place of some of a program's arrays. Several virtual data structure encodings, listed as the rows of Table 5.1, have been developed to enable loop fusion. In the table, a check mark means the feature can be used or its output is fusible. What they have in common is that they all contain a function that is called to compute the data structure's contents. Use of a function effectively defers computation until results are needed. Compile-time optimizations inline the function at the site where it is used, typically in a loop body, fusing loops. In the example from Section 1.6, map and concatMap return virtual data structures instead of arrays. Unfortunately, none of the encodings is versatile enough to use as a general-purpose,

Table 5.1: Features of fusible virtual data structure encodings.

	Parallel	Zip	Filter	Nested traversal	Mutation
Indexer	✓	✓			
Stepper		✓	✓	✓	
Fold			✓	✓	
Collector			✓	✓	✓

parallelizable fusion mechanism. For instance, none of the encodings can fuse loops to get the desired parallel loop in Section 1.6.

The rest of this section presents the virtual data structures in Table 5.1 and explains why each encoding has limited applicability. The following section introduces a new encoding used by Triolet that builds on these encodings to work around their limitations. We use as an example a list holding consecutive integers $[0, 1, 2]$. When used as the input to a container traversal, this list is analogous to a counted loop with three iterations.

Indexers An indexer encoding consists of a size and a lookup function. The ith element of a data structure is retrieved by calling the lookup function with argument i. The example list would be encoded as the pair $\langle 3, \lambda i.\ i \rangle$: the list's size is 3, and its ith element is i. Mapping a function f over this data structure builds a new virtual list whose lookup function calls the original lookup function, then calls f on the result: $\langle 3, \lambda j.\ f\ ((\lambda i.\ i)\ j) \rangle$, which the compiler simplifies to $\langle 3, f \rangle$. Summing the elements of this data structure proceeds by looping over all indices less than 3, calling the lookup function on each, and summing the results. The `map` function and many other indexer-based functions consist of straight-line code that builds an indexer, rather than a loop that builds an array. Loop fusion becomes a function inlining task, which is typically easier for compilers to accomplish than traditional loop transformations.

Since indexers allow any element to be retrieved independently of the others, indexers can be used in parallel loops. In C++, readable random access iterators fill the role of indexers. Thrust [33] and Repa [14] use indexers internally to generate fused parallel loops. Parallel loop bodies in functional languages [12, 13] resemble indexers, though they are special

syntactic forms rather than ordinary functions.

The random-access nature of indexers makes them unsuitable for fusing loops that produce a variable number of outputs per input, including the functions concatMap for nested traversal and filter for conditionally skipping elements. To retrieve a value at one index, one must compute some information about all lower indices, which wastes work. For instance, to look up the output at index 10, it's necessary to find the producers of all output elements up to index 10. The usual solution is to precompute the necessary index information using a parallel scan, but because parallel scan is a multipass algorithm, fusion is impossible; all temporary values have to be saved to memory at some point.

The values at different indices can be computed independently, enabling parallel execution of indexers. While an indexer's lookup function is independent, loops built from indexers may have cross-thread interactions. In many algorithms, parallel tasks communicate with one another to collect results. Sequential loop iterations may also share data. For instance, in a sequential reduction, the accumulator may be accessed by every loop iteration. Since the value at each index is independent of the others, an indexer cannot represent reductions, variable-length outputs, or other forms of interaction across loop iterations.

Steppers A stepper encoding is a coroutine that returns one data structure element each time it is run, until all elements have been extracted. Steppers are not parallelizable since it is only possible to retrieve the "next" element at any given time. In C++ and other imperative languages, readable forward traversal iterators play the role of steppers. The Haskell vector library uses the fusible stepper encoding presented by Coutts et al. [17]. In this encoding, a stepper contains a suspended state and a function for getting the next value. Getting the next value can either Yield a new state and one data structure element, or signal that traversal is Done.

data Step $(\alpha : \text{box}) : \text{box}$ **where** Step : $s \times (s \to \text{Next } s\ \alpha) \to \text{Step } \alpha$

data Next $(s, \alpha : \text{box}) : \text{box}$ **where**

Yield : $s \times \alpha \to \text{Next } s\ \alpha$

Done : Next $s\ \alpha$

The list $[0, 1, 2]$ would be encoded using a stepper function g that increments a counter and yields the counter's current value:

$$\begin{aligned}&\textbf{let } g\ i = \textbf{if } i == 3 \textbf{ then } \text{Done } \textbf{else } \text{Yield } (i+1)\ i\\&\textbf{in } \text{Step } 0\ g\end{aligned}$$

Mapping a function f over this value would produce a new stepper function h that applies f to the value yielded by g:

$$\begin{aligned}&\textbf{let } g\ i = \textbf{if } i == 3 \textbf{ then } \text{Done } \textbf{else } \text{Yield } (i+1)\ i\\&\quad h\ i = \textbf{case } g\ i\\&\qquad\quad \textbf{of } \{\text{Done} \to \text{Done}; \text{Yield } s\ x \to \text{Yield } s\ (f\ x)\}\\&\textbf{in } \text{Step } 0\ h\end{aligned}$$

The Done and Yield values constructed in g are immediately used in h. Inlining followed by case-of-case transformation [57] reorganizes code to eliminate these temporary data structures, producing a new stepper function that is like g, except that it calls f on the value it yields:

$$h\ i = \textbf{if } i == 3 \textbf{ then } \text{Done } \textbf{else } \text{Yield } (i+1)\ (f\ i)$$

Triolet's virtual data structure encoding also takes advantage of compile-time optimizations that eliminate temporary data structures.

Steppers are a fairly versatile sequential encoding. Although nested traversals are fusible, optimizing the fused code to a nested loop is difficult [17] and has only recently been demonstrated using a custom optimization engine [18]. In experiments using Eden, steppers were

slower by roughly a factor of two to five than imperative loop nests. A nested traversal produces a stepper (the outer loop) whose suspended state contains another stepper (one instance of the inner loop). Each outer loop iteration constructs a new inner stepper. In typical loops, all inner stepper functions are instantiations of the same static function, but the compiler cannot take advantage of this to inline the code. Instead, `Next` objects containing boxed values are created dynamically. Triolet's compiler does no better in this situation.

Folds A data structure can be encoded as a function that folds over its elements in some predetermined order. The function calls a given worker function on each data structure element to update an accumulator. The fusion scheme for lists that uses this as the canonical form of loops is called fold/build fusion [16]. The list $[0, 1, 2]$ has the following fold encoding.

$$\begin{aligned}
&\lambda w\ z.\ \textbf{let}\ \mathsf{loop}\ i\ x = \textbf{if}\ i == 3 \\
&\qquad\qquad\qquad\qquad\ \textbf{then}\ x \\
&\qquad\qquad\qquad\qquad\ \textbf{else}\ \mathsf{loop}\ (i+1)\ (w\ i\ x) \\
&\qquad\ \textbf{in}\ \mathsf{loop}\ 0\ z
\end{aligned}$$

Its meaning is clearer after unrolling the loop to get $\lambda w\ z.\ w\ 2\ (w\ 1\ (w\ 0\ z))$, which calls w to update an accumulator with the values 0, 1, and 2 in turn. Nested traversals do not pose the same optimization trouble for folds that they do for steppers. In a nested traversal, the worker function passed into w calls another fold function that contains its own loop. Inlining moves the value of w to its callsite in the body of `loop`, bringing the inner fold function along to produce a nested loop.

Unlike indexers and steppers, folds offer no flexibility in execution order. A fold processes each data structure element in sequence without interruption. This inflexibility rules out fusion of `zip`, which pairs up elements at corresponding indices in multiple data structures. A fused `zip` would read from each of several data structures in an interleaved fashion. It is a common pattern to store data in a structure-of-arrays format, then zip the arrays together in preparation for a loop that uses all the fields. Folds do not support this pattern.

Collectors A collector is an imperative variant of a fold. Instead of updating an accumulator, the worker function uses side effecting operations to update its output value. Collectors are used by Scala's collection library [56] and SkeTo [32]. Chapel implements `zip` by treating the first iterator as a parallel collector and the other as indexers [81]. Triolet uses collectors in sequential code for histogramming and for packing the results of variable-length output traversals into an array.

Conversions The rows of Table 5.1 are in decreasing order of how much the user of a virtual data structure can control its execution order. Indexers offer the greatest control, steppers offer less, and folds and collectors offer no control. A higher-control encoding can be converted to a lower-control one. Although no encoding supports zips and mutation, for instance, one could fuse `histogram(n, map(f, zip(a, b)))` by zipping and mapping over indexers, converting the result to a collector, and computing a histogram of the result. A collector that is created from indexer $\langle n, f \rangle$ loops over all indices up to n, calls f on each index, and passes the generated values to the worker function:

$$
\begin{aligned}
&\mathsf{idxToColl}\ \langle n, f \rangle = \\
&\quad \lambda w\ s.\ \textbf{let}\ \mathsf{loop}\ i\ s_2 = \textbf{if}\ i == n \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ s_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{loop}\ (i+1)\ (w\ (f\ i)\ s_2) \\
&\qquad\qquad \textbf{in}\ \mathsf{loop}\ 0\ s
\end{aligned}
$$

However, this conversion removes the potential for parallelization, since a collector's use of side effects is not compatible with parallel execution.

Triolet's iterator library is layered on top of a library of fusible operations for manipulating each of these virtual data structures. We use conventional names for these library functions along with a subscript to indicate what encoding they are implemented for, e.g., $\mathsf{map}_{\mathsf{Idx}}$, $\mathsf{map}_{\mathsf{Step}}$, $\mathsf{map}_{\mathsf{Fold}}$, and $\mathsf{map}_{\mathsf{Coll}}$ are map functions over indexers, steppers, folds, and collectors. We use conversion functions named by their input and output encoding, such as

idxToColl.

5.2 Hybrid Iterators

There is at least one fusible encoding supporting every desirable feature in Table 5.1, and this suggests that a hybrid encoding could overcome the limitations in the previous section. Triolet's encoding is hybrid in two ways. First, the library builds nested loops, possibly with a different encoding at each nesting level. Second, the library converts between encodings when needed. To illustrate, consider the computation of `sum(filter(lambda x: x > 0), xs)`, which selects the positive numbers in array `xs` and sums them. Suppose `xs` has the value $[1, -2, -4, 1, 3, 4]$. The call to `filter` returns $[1, 1, 3, 4]$. For the implementation of `sum`, indexers are the only parallelizable, fusible form at our disposal so far. Using indexers, each thread is assigned a specific number of elements to process. For instance, one thread may sum the first two values while the other sums the last two. Unfortunately, computing which index each output of `filter` resides at requires a complete pass through the data, making a fusible indexer encoding impossible.

A better fusion strategy is to partition the input array `xs` across threads and have each thread sequentially filter and sum one partition. The key to fusion is that our implementation of `filter` does not reassign indices, but rather produces either zero or one output at each index so that it is compatible with indexer-based parallelization and fusion. Conceptually, the call to `filter` transforms $[1, -2, -4, 1, 3, 4]$ into the nested list $[[1], [], [], [1], [3], [4]]$, then the call to `sum` partitions this nested list into $[[1], [], []]$ and $[[1], [3], [4]]$ and sums the two parts in parallel. By encoding the nested list as an indexer of steppers, we ensure that the filter computation is fused with the summation.

In general, a loop may have arbitrarily nested filter operations and/or traversals. Each level of nesting may produce a predetermined number of values using an indexer, or a variable number of values using a stepper. Thus an iterator can consist of an indexer containing values, a stepper containing values, an indexer containing iterators, or a stepper containing

iterators. We name these cases IdxFlat, StepFlat, IdxNest, and StepNest:

```
data Iter (α : box) : box where
    IdxFlat : Idx α → Iter α
    StepFlat : Step α → Iter α
    IdxNest : Idx (Iter α) → Iter α
    StepNest : Step (Iter α) → Iter α
```

Nested iterators can be understood as loop nests where all loops work together to produce a sequence of values.

Triolet's iterators are flexible enough to fuse all the difficult patterns in Table 5.1, while also keeping indexer-based loops available so that they can be distributed across parallel tasks. Figure 5.1 shows Iter-based implementations of some common skeleton functions. The functions zip, filter, concatMap, and collect implement four of the five features from Table 5.1. Section 5.5 addresses the remaining feature, parallelism. Reductions are illustrated by sum. Each function inspects the form of its arguments and computes a result accordingly.

In most cases, these functions just call a lower-level fusible function at each level. For instance, the sum over an iterator is computed by computing a partial sum at each nesting level, using either $\mathsf{sum}_{\mathsf{Idx}}$ or $\mathsf{sum}_{\mathsf{Step}}$. In a nested iterator, sum calls itself to sum inner loops.

The interesting cases are where a function converts between encodings or adds a level of nesting. In the sum-of-filter example, filter receives an an IdxFlat term. The body of filter (as shown in the first equation) calls $\mathsf{map}_{\mathsf{Idx}}$ to transform each element of the indexer into a stepper. The outer level of the resulting iterator remains indexer-based and parallelizable, while the inner level decides whether to process a given value. The functions filter and sum examine their argument values and build return values in such a way that, after inlining, each Iter term is a short-lived value that compiler optimizations can eliminate. The summation simplifies by inlining filter, then sum, then the recursive call to sum:

$$
\begin{aligned}
&\mathsf{zip}\ (\mathsf{IdxFlat}\ xs)\ (\mathsf{IdxFlat}\ ys) = \mathsf{IdxFlat}\ (\mathsf{zip}_{\mathsf{Idx}}\ xs\ ys)\\
&\mathsf{zip}\ \mathsf{xs}\ \mathsf{ys} = \mathsf{StepFlat}\ (\mathsf{zip}_{\mathsf{Step}}\ (\mathsf{toStep}\ \mathsf{xs})\ (\mathsf{toStep}\ \mathsf{ys}))\\
&\quad \text{where}\ \ \mathsf{toStep}\ (\mathsf{IdxFlat}\ xs) = \mathsf{idxToStep}\ xs\\
&\qquad\qquad \mathsf{toStep}\ (\mathsf{StepFlat}\ xs) = xs\\
&\qquad\qquad \mathsf{toStep}\ (\mathsf{IdxNest}\ xss) = \mathsf{concatMap}_{\mathsf{Step}}\ \mathsf{toStep}\ (\mathsf{idxToStep}\ xss)\\
&\qquad\qquad \mathsf{toStep}\ (\mathsf{StepNest}\ xss) = \mathsf{concatMap}_{\mathsf{Step}}\ \mathsf{toStep}\ xss
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{filter}\ f\ (\mathsf{IdxFlat}\ xs) = \mathsf{IdxNest}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{StepFlat} \circ \mathsf{filter}_{\mathsf{Step}}\ f \circ \mathsf{unit}_{\mathsf{Step}})\ xs)\\
&\mathsf{filter}\ f\ (\mathsf{StepFlat}\ xs) = \mathsf{StepFlat}\ (\mathsf{filter}_{\mathsf{Step}}\ f\ xs)\\
&\mathsf{filter}\ f\ (\mathsf{IdxNest}\ xss) = \mathsf{IdxNest}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{filter}\ f)\ xss)\\
&\mathsf{filter}\ f\ (\mathsf{StepNest}\ xss) = \mathsf{StepNest}\ (\mathsf{map}_{\mathsf{Step}}\ (\mathsf{filter}\ f)\ xss)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{concatMap}\ f\ (\mathsf{IdxFlat}\ xs) = \mathsf{IdxNest}\ (\mathsf{map}_{\mathsf{Idx}}\ f\ xs)\\
&\mathsf{concatMap}\ f\ (\mathsf{StepFlat}\ xs) = \mathsf{StepNest}\ (\mathsf{map}_{\mathsf{Step}}\ f\ xs)\\
&\mathsf{concatMap}\ f\ (\mathsf{IdxNest}\ xss) = \mathsf{IdxNest}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{concatMap}\ f)\ xss)\\
&\mathsf{concatMap}\ f\ (\mathsf{StepNest}\ xss) = \mathsf{StepNest}\ (\mathsf{map}_{\mathsf{Step}}\ (\mathsf{concatMap}\ f)\ xss)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{collect}\ (\mathsf{IdxFlat}\ xs) = \mathsf{idxToColl}\ xs\\
&\mathsf{collect}\ (\mathsf{StepFlat}\ xs) = \mathsf{stepToColl}\ xs\\
&\mathsf{collect}\ (\mathsf{IdxNest}\ xss) = \lambda w\ s_1.\ \mathsf{idxToColl}\ xss\ (\lambda xs\ s_2.\ \mathsf{collect}\ xs\ w\ s_2)\ s_1\\
&\mathsf{collect}\ (\mathsf{StepNest}\ xss) = \lambda w\ s_1.\ \mathsf{stepToColl}\ xss\ (\lambda xs\ s_2.\ \mathsf{collect}\ xs\ w\ s_2)\ s_1
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{sum}\ (\mathsf{IdxFlat}\ xs) = \mathsf{sum}_{\mathsf{Idx}}\ xs\\
&\mathsf{sum}\ (\mathsf{StepFlat}\ xs) = \mathsf{sum}_{\mathsf{Step}}\ xs\\
&\mathsf{sum}\ (\mathsf{IdxNest}\ xss) = \mathsf{sum}_{\mathsf{Idx}}\ (\mathsf{map}_{\mathsf{Idx}}\ \mathsf{sum}\ xss)\\
&\mathsf{sum}\ (\mathsf{StepNest}\ xss) = \mathsf{sum}_{\mathsf{Step}}\ (\mathsf{map}_{\mathsf{Step}}\ \mathsf{sum}\ xss)
\end{aligned}
$$

Figure 5.1: Triolet iterator functions.

$$
\begin{aligned}
&\mathsf{sum}\ (\mathsf{filter}\ f\ (\mathsf{IdxFlat}\ ys)) \\
&\quad = \mathsf{sum}\ (\mathsf{IdxNest}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{StepFlat} \circ \mathsf{filter}_{\mathsf{Step}}\ f \circ \mathsf{unit}_{\mathsf{Step}})\ ys)) \\
&\quad = \mathsf{sum}_{\mathsf{Idx}}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{sum} \circ \mathsf{StepFlat} \circ \mathsf{filter}_{\mathsf{Step}}\ f \circ \mathsf{unit}_{\mathsf{Step}})\ ys) \\
&\quad = \mathsf{sum}_{\mathsf{Idx}}\ (\mathsf{map}_{\mathsf{Idx}}\ (\mathsf{sum}_{\mathsf{Step}} \circ \mathsf{filter}_{\mathsf{Step}}\ f \circ \mathsf{unit}_{\mathsf{Step}})\ ys)
\end{aligned}
$$

Iterators are completely eliminated, leaving behind indexer and stepper code that is further optimized into a simple loop nest.

Zipping together two flat indexers uses an indexer-based zip function to bring together the elements at each index. If one or both iterators have variable-length components, it is necessary to identify corresponding elements of the input iterators by traversing them sequentially or by computing the positions of their output elements. Triolet's implementation traverses them sequentially by converting both arguments to steppers and traversing them together.

The variable-output functions filter and concatMap work similarly to each other. They add a level of loop nesting in order to preserve the potential parallelism of indexers and avoid the overhead of stepper-based nested traversals. Functions that consume iterators, like collect and sum, transform each level of nesting into a loop.

The functions in Figure 5.1 need to be inlined to enable subsequent optimizations. Compilers are normally reluctant to inline recursive functions, as doing so can blow up code size and/or execution time. We manually annotate these functions to indicate that they should be inlined only when the compiler knows their Iter argument's constructor, which ensures that inlining only occurs when it would expose further optimization opportunities. Inlining eventually terminates because each level of recursion consumes one level of statically known loop nesting.

5.3 Imperative Algorithms

Some library functions can be implemented more efficiently using mutable arrays. Mutable values are used locally within some leaf tasks, such as the mutable array in a histogramming computation. However, unrestricted use of mutable data can lead to race conditions and sharing of mutable data between nodes, which has undesirable consequences for correctness and performance. It is helpful to keep track of where mutable objects are used. Imperative library functions confine side effects to the scope of a single thread using a pattern modeled on state transformers [82]. Store fragments, representing the state of a thread-local mutable object, are passed around locally within a task to track dependences. Once imperative updates are complete, the mutable object reference is converted to an immutable reference, consuming the mutable store. The immutable reference is returned.

5.4 Multidimensional Iterators

So far, the Iter data type is good for variable-length and nested looping patterns, but is awkward for looping over multidimensional arrays. On the other hand, indexer-based libraries like Repa and loop-based functional languages like Single Assignment C are well suited to loops over multidimensional arrays, but they do not support fusion of nested, variable-length traversals. This section generalizes Iter for multidimensional loops and arrays.

Matrix transposition is an example of an algorithm that is awkward to write using 1D arrays. The transpose of a matrix `A` can be written by giving the transposed matrix's elements as a function of their indices. For a given matrix `A` whose dimensions are `width` and `height`, transposition would be written in Triolet as `[A[x,y] for (y, x) in range((height, width))]`. The functions `map` (implicitly called by the comprehension) and `range` are overloaded for multidimensional iteration spaces.

Simulating a multidimensional loop using 1D iterators would introduce overhead. Using a one-dimensional comprehension `[ ... for i in range(height * width)]` would require reconstructing the 2D indices `x` and `y` from `i` using expensive division and

modulus operations. Alternatively, using nested comprehensions `[[ ... for x in width] for y in height]` results in an array of arrays, which adds an additional pointer indirection to subsequent lookups.

We introduce a type class called Domain to characterize index spaces. Each index space is a type that is a member of Domain. One-dimensional organizations of data, as we have been discussing up until now, have type Seq. A value of type Seq holds an array length. Two-dimensional arrays have a width and a height, so a 2D domain Dim2 holds a pair of integers.

data Seq : box **where** seq : Int → Seq

data Dim2 : box **where** dim2 : Int × Int → Dim2

Each domain type d has an associated type Index d whose values identify individual indices within a domain. An Index Seq is an Int and an Index Dim2 is an ⟨Int, Int⟩.

A number of functions are defined differently for each domain type. The definition of class Domain, below, lists overloaded types and functions that are defined differently for each domain d and are used here.

class Domain (d : box) **where**

  type Index d : box

  idxToFold : $(\alpha \to \beta \to \beta) \to \beta \to$ Idx $d\ \alpha \to \beta$

  idxToColl : $(\alpha \to$ State $\to$ State$) \to$ Idx $d\ \alpha \to$ State $\to$ State

  zipWith : $(\alpha \to \beta \to \gamma) \to$ Iter $d\ \alpha \to$ Iter $d\ \beta \to$ Iter $d\ \gamma$

Each of these functions is related to looping over all indices in a domain. The functions idxToFold and idxToColl convert an indexer to a fold or collector that loops over all points in the domain. The function zipWith visits all points in the intersection of two domains.

We also generalize Idx to arbitrary domains d:

type Idx $d\ \alpha = \langle d, \mathsf{Index}\ d \to \alpha \rangle$

We then generalize Iter over arbitrary domains. Every Idx α is converted to a Idx $d\ \alpha$, producing the following generalized algebraic data type.

data Iter $(d, \alpha : \mathsf{box}) : \mathsf{box}$ **where**

IdxFlat : Idx $d\ \alpha \to$ Iter $d\ \alpha$

StepFlat : Step $\alpha \to$ Iter Seq α

IdxNest : Domain $d \times$ Idx d (Iter Seq α) $\to$ Iter Seq α

StepNest : Step (Iter Seq α) $\to$ Iter Seq α

Only the IdxFlat constructor can create iterators of arbitrary domain types. It simply allows an Idx $d\ \alpha$ to be used through the Iter interface. The other three constructors contain variable-length traversals, which do not preserve array dimensionality—removing arbitrary elements of a 2D array does not in general yield a 2D array, for instance—so it does not make sense for them to build multidimensional iterators.

IdxNest puts the contents of a multidimensional indexer into a 1D order. Essentially, this iterator "forgets" the multidimensional structure of the indexer. To do a variable-length traversal over a multidimensional array, users would start by flattening it to a sequence using the following function.

flatten (IdxFlat xs) = IdxNest ($\mathsf{map}_{\mathsf{Idx}}\ \mathsf{unit}_{\mathsf{Seq}}\ xs$)

5.5 Parallelism

Triolet's runtime uses Threading Building Blocks for thread parallelism and MPI for distributed parallelism. We implement wrapper functions that expose these interfaces as generic

parallel skeletons. On top of these wrappers, we layer high-level skeletons that allow users to select what degree of parallelism to use.

We add a field to `Iter` holding a flag to indicate what degree of parallelism to use. Users designate an iterator as parallel by calling `localpar` (for thread parallelism) or `par` (for distributed and thread parallelism) on it, thereby setting the flag. Parallel skeletons inspect the flag and invoke the appropriate distributed, threaded, and sequential functions. For instance, a distributed-parallel histogram performs a distributed reduction, which performs one threaded reduction per node, which sequentially builds one histogram per thread.

Triolet includes runtime facilities for serializing and deserializing objects to byte arrays. The compiler automatically generates serialization code from the definitions of algebraic data types; we override the compiler for a few types (Section 5.6). Functions are represented by heap-allocated closures and are also serialized. Serializing an object transitively serializes all objects that it references. Pointers to global data are serialized as a segment identifier and offset. Since the majority of serialized data typically resides in pointer-free arrays, such arrays are serialized using a block copy to minimize serialization time.

5.6 Array Partitioning

Distributed traversal of an array should partition the array across distributed tasks, but the compiler-generated serialization code would send the entire array to every task. To support array partitioning with minimal implementation complexity, iterators keep track of array partitions so that the runtime can send a partition to each distributed task. In principle, Triolet's compiler could take the compiler-driven approach of analysis to discover what data a parallel loop accesses, followed by transformation to copy only the relevant data to remote processors. Triolet employs a much simpler library-driven solution that takes advantage of the divide-and-conquer nature of skeletons.

Consider again the loop `sum(filter(lambda x: x > 0), xs)`. If this data is split between two tasks, each task would ideally receive half of the array `xs`. If the domain of `xs` is

d and the block of memory holding array elements is a, the expression would return a flat indexer that reads an element of a and pairs it with its index, IdxFlat $\langle d, \lambda i.\ \langle a[i], i\rangle\rangle$. It is this reference to a that, in the simple serialization strategy, drags the entire array along when it is serialized.

We enhance the functionality of indexers to address this problem. We call the new data structure a slice. Slices of arrays keep track of what subarray the library intends to use. Library functions partition data by modifying the subarray information, and the runtime uses the subarray information to serialize only the data that a remote task actually needs.

The Slice data type is defined as follows.

data Slice $d\ \alpha$ **where**

Slice : $\beta \to$	*source data*
$d \to$	*domain*
$(\beta \to$ Index $d \to \alpha) \to$	*lookup*
$(\beta \to d \to$ Index $d \to$ Slice $d\ \alpha) \to$	*partition*
$(\beta \to$ Serializer (Slice $d\ \alpha)) \to$	*serializer*
$(\beta \to$ Deserializer (Slice $d\ \alpha)) \to$	*deserializer*
$(\beta \to \beta) \to$	*preserve*
Slice $d\ \alpha$	

This is a well-known object-oriented pattern: the first field is a piece of data, of some hidden type β, and the remaining fields are methods for accessing that data. Traversing an array creates a Slice whose source data holds the array and the location of the subarray that will be accessed. The next two fields provide functionality equivalent to an indexer's domain and lookup function. The *partition* field extracts a subset of the slice. It takes a domain and offset, and constructs a new slice. Partitioning an array is a constant-time operation: it merely creates a new slice whose source data contains the original array and the location of the subarray. The next two fields are serializer and deserializer functions, whose types we have abbreviated as Serializer (Slice $d\ \alpha$) and Deserializer (Slice $d\ \alpha$). The compiler-

generated serialization functions for this data type delegate their work to these functions. When an array slice is partitioned and then serialized to a buffer, the serializer function copies the subarray to the buffer (along with the Slice's other fields). The deserializer reads the subarray out into a new array.

There is one catch to this interplay between the library and the runtime. Compile-time optimizations are often smart enough to statically eliminate slice objects, preventing slice-based array partitioning from taking place at run time. The *preserve* field of Slice is used for selectively inhibiting these optimizations. Distributed low-level skeletons, like the parallel sum in `sum(par(myLargeArray))`, invoke the field to ensure that slices are preserved, while shared-memory and sequential skeletons allow slices to be optimized away. This function calls an externally defined identity function on the parts of the source data that must be accessed through the slice, preventing the compiler from analyzing the flow of values and bypassing the slice object.

The final iterator type, after adding the changes from Section 5.5 and this section, is shown below. Each constructor holds a parallelism hint telling the library whether to execute in parallel. Indexer-based iterators contain an input data structure in the form of a slice object. When an iterator over an array is sent from one node to another, the slice object directs the runtime system to copy only the iterator's input subarray.

data Iter d α **where**

IdxFlat : ParHint → Slice d α → Iter d α

StepFlat : ParHint → Step α → Iter Seq α

IdxNest : ParHint → Domain d ⇒ Slice d (Iter Seq α) → Iter Seq α

StepNest : ParHint → Step (Iter Seq α) → Iter Seq α

CHAPTER 6

Evaluation

To show how a solid skeleton framework can deliver high performance without burdening programmers, four benchmarks from the Parboil benchmark suite [59] have been converted into Triolet, Eden, and C+MPI+OpenMP. This chapter discusses these benchmarks' performance and the degree to which different languages required manual optimizations. The benchmarks chosen were those found, by inspection, to contain loops with high compute intensity, meaning that they perform many operations per byte of input. When parallelizing on a cluster, high compute intensity allows to amortize the overhead of message passing.

Because the goal is to demonstrate the performance of high-level code, benchmarks were initially written in a straightforward style, using higher-order functions in Triolet and Eden and simple loop nests in C. Some manual optimizations were performed on the Eden code to resolve issues that severely degraded performance: Loops were manually blocked, using lists at the outer level and arrays at the inner levels, and histograms were rewritten to use imperative loops nests. Distributed parallel loops use the same problem decomposition strategy in all languages. Because the same parallelization strategies are used, performance comparisons reveal overhead incurred in each language.

For each benchmark, performance is normalized as speedup against sequential C to provide a measure of absolute performance. As a highly efficient implementation layer, C+MPI+OpenMP serves as a useful reference point against which to evaluate the scalability and parallel overhead of the high-level languages.

C code is compiled with GCC 4.7.3 `-O3`. Eden code is compiled with GHC-Eden 7.6.1 `-O2`

Parts of this chapter appeared in the ACM Symposium on Principles and Practice of Parallel Programming [79]. The material is used with permission.

with LLVM 3.2 as the backend. All three versions use OpenMPI as a distributed communication layer. Tests are run on a group of eight Amazon EC2 cluster compute nodes with two 8-core Xeon E5-2670 processors per node (a total of 16 cores per node). Hyperthreading was disabled. Parboil includes a range of input problem sizes for each benchmark. Data sets were selected to have a sequential C running time between 20 and 200 seconds, which is large enough for the C+MPI+OpenMP code to scale up to the full test system. Reported parallel times are the average of five runs.

6.1 Eden Overview

Eden is a distributed-parallel extension of GHC [49]. Although Eden is not designed to compete with low-level programming models, it is able to achieve significant performance and scalability at small core counts when used in concert with high-performance array libraries. Eden code is valid Haskell code that executes sequentially if compiled with an unmodified GHC. Eden's runtime system uses MPI to create and communicate among parallel execution contexts. Eden, unlike GHC, does not support shared-memory multithreading. Though Haskell uses lazy evaluation, data are fully evaluated before being sent from one execution context to another. Eden provides a library of parallel list-processing skeletons as a higher-level abstraction over processes.

Skeletons are implemented to use a two-level work distribution similar to that used in Triolet and C+MPI+OpenMP. By taking advantage of the cluster's topology, these skeletons outperform the flat communication strategy of the skeletons included in Eden. The main process distributes work to one process in each node, which further distributes work to other processes in the same node. This avoids a communication bottleneck at the main process that occurs in a flat communication strategy. Results are aggregated in the same two-level way. As in Eden's skeleton library, one processor on each node is reserved for communication so that communication is not delayed by the execution of other work.

Table 6.1: Lines of source code in parallel kernels in each benchmark.

	Triolet			Eden			C+MPI+OpenMP
Benchmark	Compute	Marshal	Total	Compute	Marshal	Total	Total
`mri-q`	21	20	41	27	40	67	151
`tpacf`	49	47	96	66	52	118	151
`sgemm`	31	24	55	145	33	178	273
`cutcp`	59	23	82	70	52	132	194

6.2 Source Code Size

Source code size serves as an approximate measure of the code complexity involved in different programming styles. When one language allows a kernel to be written in less code, it is either because the language provides more concise building blocks, such as one-line list comprehensions taking the place of multi-line `for` loops, or because more of the work is done by a runtime system or a library. Code size is a crude measure because it is influenced by superficial differences in programming style. Nevertheless, it reveals some significant trends across benchmarks.

Table 6.1 tabulates the number of lines of code used for each parallel kernel. In Triolet and Eden, total lines of code are classified into "compute" for computing results and "marshal" for marshaling data between languages. Marshaling is needed because Triolet and Eden manage data structures on a separate heap. Code for input, output, timing, and using external code (i.e., `#include` directives in C and `import` statements in Haskell) is not counted.

The Triolet and Eden code of `mri-q`, `tpacf`, and `cutcp` are similar in size and smaller than the C+MPI+OpenMP code. Much of the C+MPI+OpenMP code consists of MPI-level loop blocking and communication. Some of the C+MPI+OpenMP code is partially replicated and specialized so that the MPI master and worker ranks execute different code. For instance, the master and worker nodes execute the same computation in parallel, but with different communication code interspersed with the computation code. While this code replication is avoidable, it arguably reflects a realistic development practice.

The Triolet code of `sgemm` is much shorter than the other two languages because the Triolet code relies on Triolet's library for 2D block decomposition. Neither Eden nor MPI have

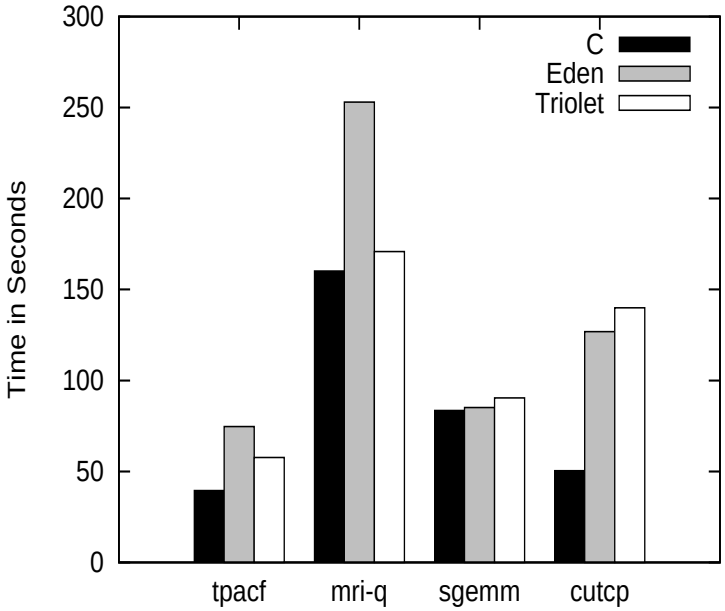


Figure 6.1: Sequential execution time of benchmarks.

such library features. Block decomposition took over 120 lines of code in Eden and C with MPI. The same code was reused in Eden for inter-node and intra-node work decomposition. In C+MPI+OpenMP an OpenMP `collapse` directive was used for intra-node work decomposition.

Overall, the high-level languages reduce code size by providing more functionality in library code.

6.3 Sequential Execution Time

Comparing the sequential execution time of the benchmarks across languages reveals how well the Triolet and Haskell compilers eliminate abstraction overhead. Kernel code of the benchmarks consists of simple loop nests that perform arithmetic and read and write arrays. In C these operations are written more or less directly, while in Triolet and Haskell they are written with generic higher-order functions that must be simplified away.

Sequential execution times are shown in Figure 6.1. The execution time of high-level code ranges from on par with C code to almost 3× slower. Flat loops compile to efficient code:

performance parity is achieved on the flat loop benchmarks, `mri-q` and `sgemm`, except for `mri-q` in Eden. Eden's runtime is 50% longer because the backend misses a floating-point optimization that computes `sinf` and `cosf` together when they are called with the same argument. Nested loop benchmarks `tpacf` and `cutcp` lose more performance relative to C, with the deeply nested loop from `cutcp` losing the most. Due to the size of the generated code, it is difficult to identify the source of the overhead. In parallel execution, these execution time differences are less significant than differences due to parallelization overhead.

6.4 Parallel Speedup and Scalability

6.4.1 MRI-Q

The main loop of `mri-q` computes a non-uniform 3D inverse Fourier transform to create a 3D image. It is heavily computation-bound and has a simple memory access pattern, making it straightforward to parallelize. In Triolet, it can be distilled down to a line of code:

```
[sum(ftcoeff(k, r) for k in ks) for r in par(zip3(x, y, z))]
```

The code consists of a parallel map over image pixels, summing contributions from all frequency-domain samples. The body of `ftcoeff`, not shown, performs several floating-point math operations. This code yields parallel performance on par with manually written MPI and OpenMP (Figure 6.2).

Eden does not provide a data structure supporting an efficient, parallel map operation. Nevertheless, one can be built from lists (supporting a parallel, high-overhead map) and vectors (supporting a sequential, efficient map), borrowing a technique from another Haskell library [37]. Arrays are *chunked* into lists of 1k-element vectors. The parallel loop is tiled to match the structure of the data: `mapFarmB` traverses the list in parallel, while `V.map` traverses a vector. (The chunk-building code is not shown.)

```
mapFarmB (V.map (\r -> V.sum $ V.map (ftcoeff r) ks)) points
```

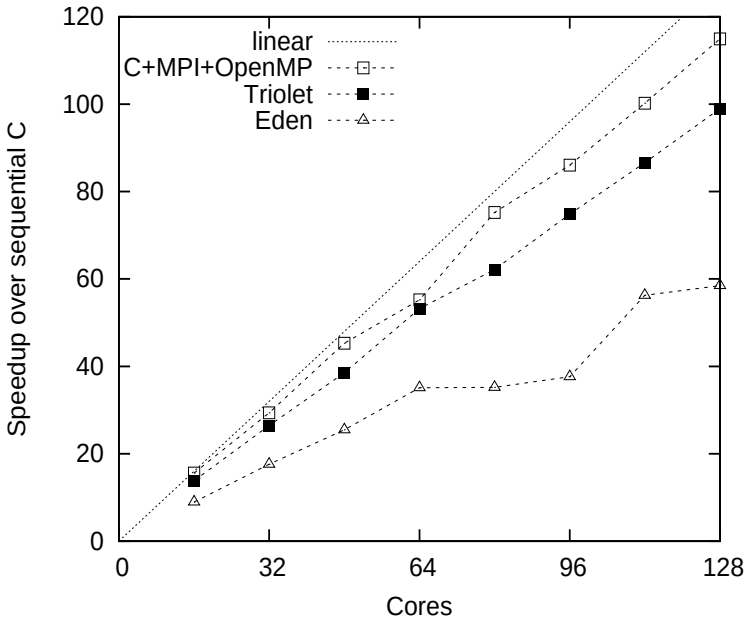


Figure 6.2: Scalability and performance of `mri-q` implemented in different languages.

While the Eden code scales well, it loses performance due to a missed floating-point optimization. The performance loss is also visible in the sequential Eden execution time (Section 6.3).

C+MPI+OpenMP is the most verbose, dedicating more code to partitioning data across MPI ranks than to the actual numerical computation. While `mri-q`'s communication pattern fits MPI's scatter, gather, and broadcast primitives, these were not as efficient as the Triolet code; the fastest version used nonblocking, point-to-point messaging.

6.4.2 SGEMM

The scaled product αAB of two 4k×4k element matrices is computed in `sgemm`. The multiplication is parallelized after transposing matrices so that the innermost loop accesses contiguous matrix elements. The array layout reflects this benchmark's original design for wide vector parallelism on GPUs; to multiply matrices efficiently without using vector hardware, both input arrays and the output array are transposed in separate loops.

The product of A and B^T is written in Triolet as a parallel evaluation of dot products:

```
def dot(u, v):
  return sum(x*y for (x, y) in zip(u, v))

zipped_AB = outerproduct(rows(A), rows(BT))
AB = [alpha*dot(u, v) for (u, v) in par(zipped_AB)]
```

The first two lines define dot product as multiplying two arrays elementwise and summing the products. The next line calls `rows` to reinterpret the matrices as 1D virtual arrays of arrays, where each inner array is one row, then `outerproduct` to zip A and B together into a 2D virtual array containing row i of B and row j of A at position i, j. The last line performs a parallel map to compute dot products and construct the array. The parallel map performs a block decomposition of C, distributing chunks of A and B across cluster nodes and cores. Slices created by the calls to `rows` carry the necessary information to enable this data distribution. Similar 2D decompositions are written as part of the parallel C+MPI+OpenMP and Eden code.

Transposition is a sequential bottleneck in Eden since it does too little work to parallelize profitably on distributed memory. We parallelize it over shared memory on a single node in Triolet and C+MPI+OpenMP. At 128 cores, transposition takes 35% of Eden's execution time.

All versions of the code exhibit limited scalability due to transposition time and communication time (Figure 6.3). C+MPI+OpenMP and Triolet spend similar amounts of time in communication and in parallel computation, resulting in similar performance. Triolet's performance stops rising toward eight nodes as it spends more time constructing messages. At eight nodes, 40% of Triolet's overhead relative to C+MPI+OpenMP is attributable to the garbage collector [83], which is slow when allocating objects comprising tens of megabytes. The garbage collection overhead was determined by comparing to the run time when libc `malloc` was substituted for garbage-collected memory allocation. The Eden code fails at two nodes because the array data for a chunk of work is too large for Eden's message-passing runtime to buffer.

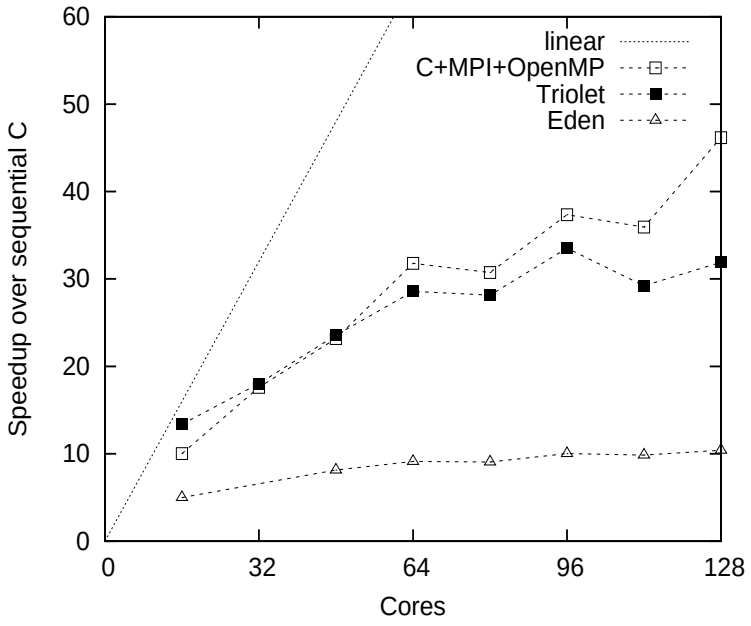


Figure 6.3: Scalability and performance of `sgemm` implemented in different languages.

6.4.3 TPACF

The `tpacf` application analyzes the angular distribution of observed astronomical objects. It uses histogramming and nested traversals, presenting a challenge for conventional fusion frameworks. Three histograms are computed using different inputs. One loop compares an observed data set with itself; one compares it with several random data sets; and one compares each random data set with itself. The code is parallelized across data sets and across elements of a data set.

In Triolet, the common code of the three loops is factored out and written once. The function on lines 1–4 of Figure 6.4 contains the common part of all three histogram computations, dealing with correlating pairs of values taken from a pair of data sets. This code maps `score` over all given pairs of objects to compute a similarity between members of each pair and collects the results into a new histogram. On lines 6–10, `randomSetsCorrelation` computes a parallel histogram over a collection of random data sets. Parameter `corr1` computes a histogram from one random data set, and `rands` is an array of random data sets. The function body consists of a parallel reduction that computes histograms of individual data

```
1  def correlation(size, pairs):
2    values = (score(size, u, v)
3              for (u, v) in pairs))
4    return histogram(size, values)
5
6  def randomSetsCorrelation(size, corr1, rands):
7    empty = [0 for i in range(size)]
8    def add(h1, h2):
9      return [x + y for (x, y) in zip(h1, h2)]
10   return reduce(add, empty, par(corr1(r) for r in rands))
11
12 def selfCorrelations(size, obs, rands):
13   def corr1(rand):
14     indexed_rand = zip(indices(domain(rand)), rand)
15     pairs = localpar((u, v)
16                      for (i, u) in indexed_rand
17                      for v in rand[i+1:])
18     return correlation(size, pairs)
19   return randomSetsCorrelation(size, corr1, rands)
```

Figure 6.4: Triolet code of `tpacf`'s self-correlation loop.

sets and adds them together.

The `selfCorrelations` of random data sets are computed in lines 12–19. The function `corr1` computes the self-correlation of one data set `rand` (lines 13–17). Self-correlation examines all unique pairs of values `(rand[i], rand[j])` where `j > i`. Line 15 pairs each element of `rand` with its index. For each index-element pair `(i, u)` (line 16), a slice consisting of elements at higher indices is extracted (`rand[i+1:]` on line 17), and `u` is paired with each element `v` of that slice (line 15). Line 18 computes a correlation histogram from these pairs. Line 19 runs `corr1` in parallel over the random data sets and sums the generated histograms. The other two parallel histogramming loops are defined similarly to `selfCorrelation`.

Triolet is expressive enough to efficiently partition the loop nest across nodes and cores without manual partitioning. The Eden and C+MPI+OpenMP implementations of `tpacf` partition data according to the number of available cores. In Eden, only the outer loop over `rands` is parallelized. In the original program, `rands` is an array containing 100 arrays, and

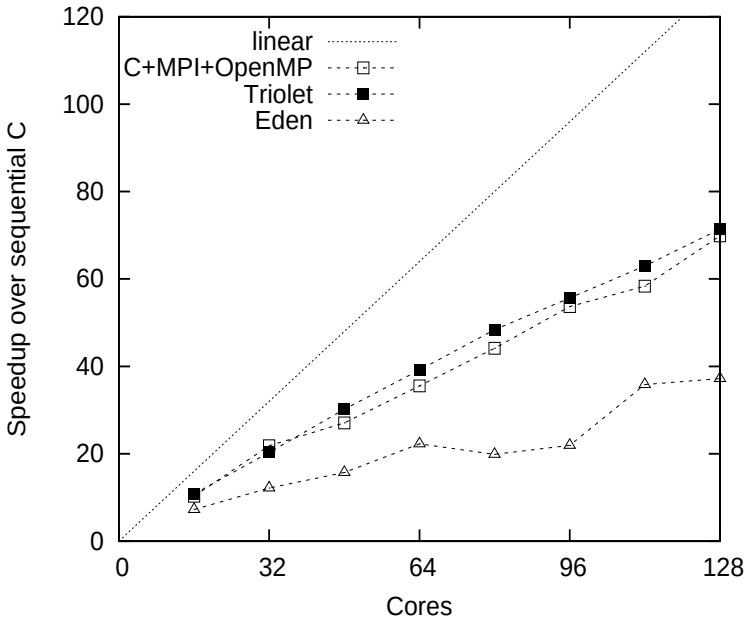


Figure 6.5: Scalability and performance of `tpacf` implemented in different languages.

consequently the loop cannot be parallelized on more than 100 cores. To use more cores, the Eden code subdivides these arrays, doubling the size of `rands`. In C+MPI+OpenMP, each core within a node operates on its own histogram array. Since the OpenMP runtime does not transparently manage per-core data, the C+MPI+OpenMP code explicitly creates the right number of arrays and determines which array to use within the loop body. For a programmer, manual partitioning typically entails one or more iterations of performance optimization. Triolet's library support for parallel histograms over nested iterators and runtime support for thread parallelism uses the parallel architecture efficiently.

Triolet and C+MPI+OpenMP scale similarly (Figure 6.5). Triolet is slightly faster due to a more even distribution of computation time across nodes. Eden has somewhat worse sequential performance and a higher communication overhead.

6.4.4 CUTCP

The `cutcp` benchmark is taken from a molecular modeling application. It computes the electrostatic potential induced by a collection of charged atoms at all points on a grid. An

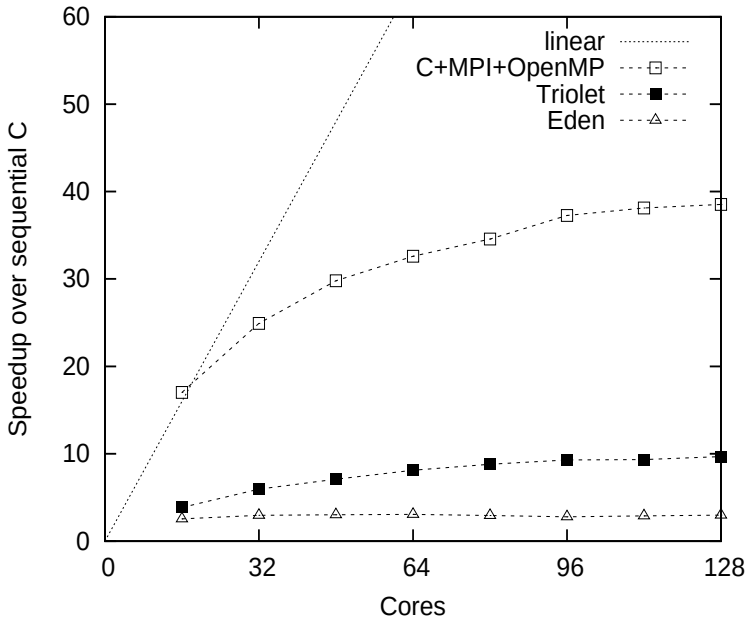
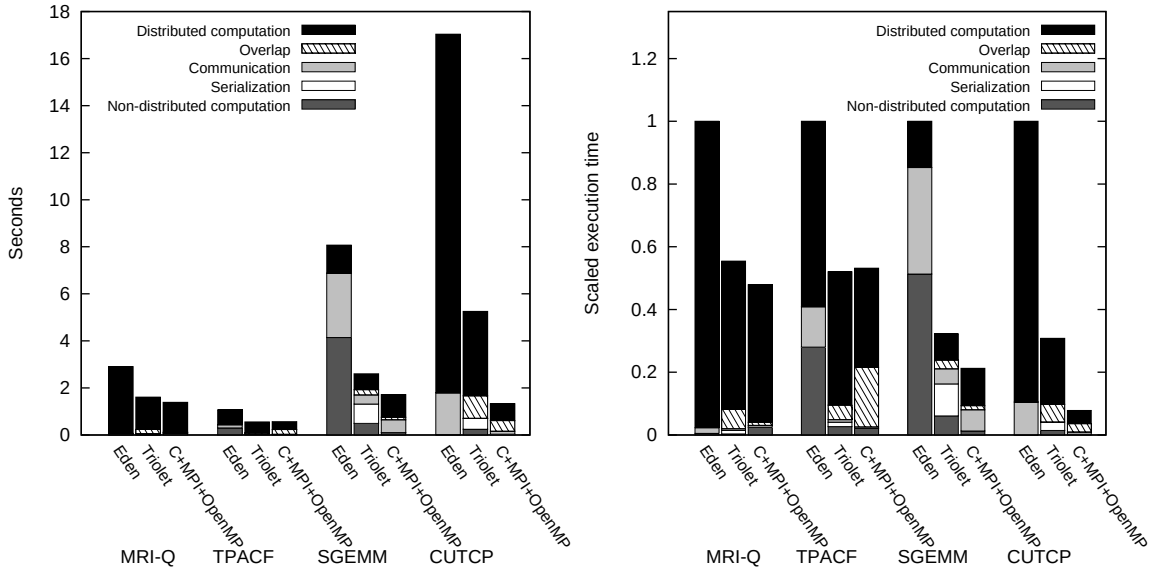


Figure 6.6: Scalability and performance of `cutcp` implemented in different languages.

atom's a field affects those grid points within a distance c. The body of the computation is essentially a floating-point histogram: it loops over atoms, loops over nearby grid points, skips points that are not within distance c, and updates the grid at the remaining points. This computation is done by nested loops and conditionals in the C code or nested traversals in Triolet. Subsets of atoms are processed in parallel.

As in `tpacf`, the Eden code's sequential performance suffers when using nested traversals. Using an imperative loop nest improves sequential performance by 5×, and the faster version is used for performance measurements.

Performance of Triolet and C+MPI+OpenMP saturates quickly (Figure 6.6), as the overhead of summing histograms dominates execution time. As in `sgemm`, Triolet has significant garbage collection overhead. Approximately 60% of Triolet's execution time at eight nodes arises from allocator overhead.



(a) Absolute time

(b) Normalized against slowest time for each benchmark

Figure 6.7: Execution time of benchmarks implemented in C+MPI+OpenMP, Triolet, and Eden on 128 cores. Bars are colored to show time spent in different stages of parallel computation. Part (b) shows the data of part (a) rescaled for comparison between implementations of a benchmark.

6.5 Breakdown of Parallel Execution Time

No single factor limits parallel scalability in these benchmarks. The breakdown of execution time in Figure 6.7 narrows down where overheads affect each benchmark. The figure shows execution time for all benchmarks in all languages on 128 cores. Time is classified based on the life cycle of distributed parallel loops into the following categories.

Distributed computation is when all processors are computing in parallel. This is the only category that utilizes all cores. Other categories indicate inefficient use of cores.

Overlap between distributed computation and communication occurs when some cores are doing distributed computation while others have not begun or have already finished. Overlap is likely to result from load imbalance or communication delays.

Communication is time spent sending data to cores at the beginning of a loop or receiving data from cores at the end.

Serialization is time spent packing or unpacking messages in the main thread.

Non-distributed computation is time spent in sequential or thread-parallel computation.

Measurements come from sampling the clock at the beginning and end of communication events in Triolet and C+MPI+OpenMP code. Additionally, in Triolet, the time spent serializing and deserializing messages in the main thread was measured. In C+MPI+OpenMP data is not serialized or deserialized.

Data was collected in Eden by measuring the reduction in execution time when distributed loop bodies or entire distributed loops were replaced by trivial computation (such as creating a zero-filled array). Evaluation of unused inputs was forced to ensure that they would be computed in Haskell's lazy execution model. Event times were not used because Eden's use of laziness allows different stages of execution to overlap, making it unclear how to categorize time. The Eden execution trace in Figure 6.8 shows an extreme example where four processors begin executing a parallel loop body after most others have finished. While asynchronous communication is also used in some Triolet and C+MPI+OpenMP code, they showed smaller delays. The measurement method used in Eden cannot identify overlap between stages and cannot distinguish serialization from communication; thus, only distributed computation, communication, and non-distributed computation are shown.

Distributed computation dominates the execution time of the compute-intensive benchmarks `mri-q` and `tpacf`. Eden's sequential execution in `tpacf` is spent in a self-correlation loop. This is a parallelizable triangular loop representing 0.3% of `tpacf`'s work. Triolet and C+MPI+OpenMP use TBB and OpenMP to dynamically balance work across threads. To parallelize it in Eden, work would have to be partitioned in a way that balances load.

Load imbalance in the C+MPI+OpenMP implementation of `tpacf` is evident in the large overlap time. The cause is unclear, since Triolet has less load imbalance with the same work distribution.

The communication-intensive benchmark `sgemm` spends the greatest fraction of time in communication and serialization in all implementations. Eden's large communication time is

Figure 6.8: An execution trace of `sgemm` on four cluster nodes as displayed by the Eden trace viewer. Each horizontal band is a core's execution timeline. Wide black bars indicate computation, wide grey bars indicated blocked or delayed execution, and narrow light bars indicate idling. The four bars displaced to the right show that four processors started late.

a combination of the additional cost of sending messages in two hops (inter-node followed by intra-node) and the work of partitioning and merging arrays between hops. Serialization and deserialization take 30% of Triolet's execution time. Even C+MPI+OpenMP's relatively short communication phase takes 30% of its total execution time.

Of Eden's sequential execution time in `sgemm`, 35% is spent transposing matrices. Transposition contains virtually no computation and so cannot amortize the cost of message passing in a distributed memory model. The remainder is spent splitting and merging arrays.

In `cutcp`, the time spent in distributed computation varies dramatically between implementations. The difference is not simply due to cross-language differences in efficiency, since it is larger than the difference in sequential execution time in Figure 6.1. As discussed in Section 6.4.4, `cutcp`'s use of per-processor private output arrays introduces additional work for creating and summing arrays. Some of the overhead in Triolet is attributable to the garbage collector. Eden's overhead may have the same origin.

CHAPTER 7

Related Work

This dissertation draws from a broad range of prior work on container libraries, functional languages, parallel languages, and algorithmic skeletons. This chapter compares in detail the work lying at the intersection of these categories—data-parallel, functional, distributed skeletons—and presents some representatives of more distantly related container traversal interfaces.

It is not always straightforward to identify differences and similarities between languages. Similar features may be presented in different ways. For instance, although Sisal's loops closely resemble Fortran loops, their behavior is closer to a parallel map due to the absence of side effects in the loop body. Different parallel execution models may be interconvertible in some situations. For instance, loop fusion can partially transform NESL's vector-parallel execution model into a thread-parallel execution model. To facilitate comparisons across languages, this chapter defines some language features and optimizations so as to apply to both loop-based and traversal-based programming interfaces.

Table 7.1 lists features of programming languages and libraries that use container traversals for expressing parallel loops. A programming abstraction is counted as a container traversal if it manages communication and parallelism of user-defined code over a container. That is, it processes a collection of inputs (such as a range of numbers, or the elements of a data structure) using a user-defined function to compute results from subsets of the data, and executions of user-defined functions do not interact with one another in normal usage.

The first five columns of the table show what kind of loops can be written in each framework.

Table 7.1: Feature comparison of parallel container traversal implementations.

	Kinds of loops					Loop optimizations					Runtime system features	
	Array traversal	Multidimensional array traversal	Nested traversal	Imperative loops	Sort by key	Fusion	Filter fusion	Nested traversal fusion	Imperative loop fusion	Array distribution	Communication	Automatic marshaling
Parallel loop												
Sisal	✓		✓	✓		✓			✓	✓	Shared or at sync	✓
SAC	✓	✓		✓		✓					Shared	
Delite	✓			✓	✓	✓	✓		✓		Shared	✓
Repa	✓	✓				✓					Shared	
Vector												
ZPL	✓	✓		✓		✓				✓	Explicit	
NESL	✓		✓	✓		✓			✓	✓	Shared	
DPH	✓					✓				✓	Shared	
Manticore	✓					✓				✓	Shared	
Copperhead	✓					✓				✓	Shared	✓
Collection												
Scala	✓		✓	✓	✓	✓	✓	✓	✓		Shared	
Query												
Spark	✓		✓	✓	✓						At sync	
DryadLINQ	✓		✓		✓	✓					At sync	✓
Functional data-parallel skeleton												
HDC	✓		✓							✓	At sync	✓
PMLS											At sync	✓
Eden											At sync	✓
HdpH											Explicit	
Triolet	✓	✓	✓	✓		✓	✓	✓	✓	✓	Shared & at sync	✓
Other data-parallel skeleton												
Muesli	✓	✓		✓						✓	Explicit	
SkeTo lists	✓			✓		✓	✓		✓	✓	Explicit	✓

Array traversal A framework supports array traversal if it has loops that partition array elements across processors. Frameworks that do not support array traversal use a different container type such as lists or lazy sequences.

Multidimensional array traversal If a framework's array traversal functionality can be applied to multidimensional arrays, it supports multidimensional array traversal.

Nested traversal A framework supports nested traversal if it has a way to expand a sequence by transforming each of its elements into an arbitrary number of new elements. The canonical nested traversal function is `concatMap`.

Imperative loops A framework is shown as supporting imperative loops if it allows mutable data structures to be updated within a loop. Updates are generally restricted to "output-only" operations that cannot be used for communication across loop iterations.

Sort by key Some frameworks support an operation that sorts and groups elements by a user-defined ordering or hash function. Variants of this include sorting by key, relational joins, and the sort phase of map-reduce frameworks. If a framework can only implement sorting via scan operations, it is not counted as having a sort by key operation. Scan-based sorting cannot "send" values directly to the destination indicated by their key, but must perform extra phases of communication.

The next five columns describe which optimizations are performed transparently by the framework. Optimizations facilitate high-level programming by obviating manual loop transformations.

Fusion A framework is shown as performing fusion if it can co-schedule the iterations of multiple skeletons such that one skeleton's output is directly consumed by another without saving it into a data structure or sending it between threads. Fusion can be achieved statically through loop transformations or dynamically through lazy evaluation.

Filter fusion This form of fusion involves fusing a skeleton that discards or skips values, such as `filter`, with a subsequent skeleton.

Nested traversal fusion This form of fusion involves fusing a nested traversal with a subsequent skeleton.

Imperative loop fusion This form of fusion involves loops that perform imperative updates.

Array distribution A framework distributes arrays if it partitions array data into chunks for parallel processing. A framework may distribute an array for the lifetime of the array, or only while traversing it. Distribution is not necessary in a shared memory system but may improve performance by reducing false sharing. In frameworks that distribute lists but not arrays, programmers can block workaround is to tile loops.

The last two columns describe runtime system features.

Communication Different runtime mechanisms may be used to share data between threads. Implementations relying on shared memory hardware use *shared memory* communication.

Distributed-memory language implementations may communicate only at synchronization points (*at sync*). An alternative used by some languages is to emulate shared memory at run time by dynamically passing messages when remote data is read or written; no language in the table does this.

Some skeleton libraries, lacking language and runtime support, rely on the programmer to insert *explicit* communication.

Sisal has a shared memory and a distributed memory implementation but the two models can't be used together, shown as "Shared or at sync." Triolet uses shared memory and distributed parallelism together, shown as "Shared & at sync."

Automatic marshaling Some distributed frameworks automatically determine how to marshal data structures, or turn them into messages for transmission over a network. In

distributed frameworks without automatic marshaling, programmers must write marshaling code, typically in the form of serialization functions for each data type. Automatic marshaling is inapplicable to shared memory frameworks.

7.1 Parallel Loop Languages

One line of language development extends the paradigm of mapping loop iterations to parallel processors. Because dependences are the primary obstacle to effective loop parallelization in imperative languages such as Fortran and C, these languages facilitate a programming style without side effects [84]. These languages tend to share the characteristics of indexers, supporting multidimensional arrays and loop fusion but lacking optimizations for variable-length loops.

Sisal is a functional language developed as an alternative to Fortran for parallel programming. Parallelism is expressed through parallel `for` loops. Values produced in a loop body can be collected into a new array and/or reduced to a result value. Table 7.1 describes Sisal 90 because the language implementation for this version of Sisal is described in detail [85]. It has an implementation for shared memory and another for distributed memory systems. The programming language only supports 1D arrays; nevertheless, the compiler supports multidimensional arrays and attempts to recognize programming idioms corresponding to loops over multidimensional arrays. Sisal compilers fuse loops on an iteration-by-iteration basis, so traversals with variable-length outputs cannot be fused [86].

Single Assignment C (SAC) is a functional language that expresses parallelism through array comprehensions [13]. SAC introduces support for multidimensional arrays and generic programming, whereas Sisal has been monomorphic until recently. Like Sisal, SAC's compiler performs loop fusion on an iteration-by-iteration basis and thus does not fuse traversals with variable-length outputs.

Delite [54, 55] is a framework for implementing embedded domain-specific languages. The framework is designed to be extended with new language features, and its extensibility

has been used to make several programming languages. Thus, Delite represents a family of programming languages that share a common infrastructure. The common infrastructure includes an intermediate program representation and optimizer that includes container traversal operations used for writing parallel loops. Delite has an extensible set of primitive container traversal operations, each of which is implemented by handwritten code for a given target. Loop fusion is based on rewriting combinations of container traversals.

Repa is a Haskell library for parallel loop programming [14]. Repa uses indexers to achieve loop fusion.

7.2 Vector Languages

Vector languages are loosely modeled on vector processors, where instructions operate in parallel on the elements of an array. Effectively, each vector operation is a 1D parallel loop over one or more arrays. While vector languages do not adhere rigorously to the model, they inherit a tendency to flatten all data structures into arrays of scalars and to flatten programs into sequential programs containing small, parallel inner loops. Loop fusion is an important transformation for merging loops, and it is employed by all vector languages.

ZPL was developed to provide a simple and accurate performance model for parallel programming. Programs execute in SIMD fashion and communication is explicit. It is a vector language in the sense that users are encouraged to view a program as if each statement executes in parallel across all processors or across all array elements.

The seminal vector language, NESL, has been deployed on a variety of computers, including some distributed memory systems [6]. The entry in Table 7.1 characterizes NESL's multicore implementation [10]. Although data structures are semantically immutable, NESL's runtime system opportunistically mutates arrays when it detects that the old array contents are no longer used, allowing imperative algorithms to run with efficient asymptotic performance [6]. The multicore backend can only fuse loops having the same number of iterations, so filter operations and nested traversals cannot be fully fused [10].

DPH (Data Parallel Haskell) generalizes the flattening transformation to work in the context of Haskell programs [8]. It employs rewrite-based loop fusion, which again can only fuse loops having the same number of iterations. Although DPH is designed to support nested traversal, the implementation in the latest release (version 0.7.0.1) is incomplete.

Manticore [9] and Copperhead [7] each depart from the vector model to improve program performance. Both languages compile nested loops as-is instead of flattening them, which avoids the overhead of reorganizing data. Neither language supports nested traversals.

7.3 Collection Libraries

Many programming languages have a standard library of collections. Collection libraries tend to be feature-rich (these rows of Table 7.1 are filled more densely than most other rows), but are not necessarily designed with parallelism in mind. Use of lazy sequences achieves a fused execution order for a broad range of looping patterns.

Scala collections [21] provides both parallel and sequential higher-order functions. Performance comparisons between Scala collections and Delite-hosted languages suggest that the former incurs significant overhead [55].

7.4 Query Languages

Query languages have feature sets based on traditional database query languages, but have recently begun to converge with list-processing and array-processing interfaces. Query languages tend to be designed for very large data sets where disk access latency and network throughput limitations are more significant than processing speed. Consequently, their designs do not emphasize efficient execution of inner loops, but rather aim to reduce disk and network I/O through efficient scheduling. Frameworks such as MapReduce [87] and Hadoop [88] simplify low-level issues of fault tolerance and distributed storage, but are difficult to program directly. Higher-level query languages provide container traversal interfaces

to these low-level frameworks.

DryadLINQ [28] is a version of LINQ for cluster computing. It uses services in the .NET framework to marshal functions and data across the network. Tasks are dynamically scheduled to balance load and reduce communication. Dependent tasks scheduled to the same processor communicate through iterators, yielding a fused execution behavior.

Spark [29, 89] provides an interface resembling Scala iterators for operating on distributed objects. Whereas most query languages store temporary data on disk, Spark can cache stored data in memory to reduce the effective access time.

7.5 Functional Data-Parallel Skeletons

Prior work on functional data-parallel skeletons has emphasized simple, high-level control of parallelism rather than generality or high absolute performance. Languages and libraries in this category minimize the differences between the framework and a preexisting library or language. For instance, rather than introducing new programming language features or new collection types, these frameworks adopt existing languages and data structures. Several of these frameworks (PMLS [90, 50], Eden [49], and HdpH [91]) adopt their host language's convention of using linked lists rather than arrays, which negatively impacts performance. Loidl et al. [92] compare the performance of functional skeleton frameworks.

HDC (Higher-order Divide and Conquer) is a skeleton-based parallel compiler for a subset of Haskell [48, 93]. HDC's core parallel abstraction is a divide-and-conquer parallel skeleton. Haskell lists are represented as arrays for efficiency. The compiler statically analyzes and transforms uses of skeletons to select a parallel schedule.

HdpH [91] provides a skeleton interface on top of a lower-level parallel library. HdpH's runtime system, unlike other functional skeleton frameworks, supports both shared and distributed parallelism. However, HdpH's skeletons either use node-level thread parallelism or global work stealing; to take advantage of locality, one must write a nested loop using two different skeletons.

7.6 Other Data-Parallel Skeletons

Some skeletons perform compile-time code transformation by using C++'s template metaprogramming facilities. Muesli [45] takes advantage of both shared and distributed memory parallelism. The SkeTo list library [32] uses a collector design to fuse loops over lists.

CHAPTER 8

Conclusion

This dissertation has presented Triolet, a system for high-level, high-performance parallel programming of clusters. Triolet uses container traversals as a familiar high-level programming abstraction. A parallel loop may be a composition of multiple traversals, each performing a simple transformation on a collection of values. The high-level semantics of container traversals convey useful parallelism and access pattern information to the implementation. Container traversal code is transformed at compile time into efficient parallel and sequential loops. For parallel loops that are naturally expressed using Triolet's library of container traversals, Triolet offers similar complexity to a sequential programming language and performance roughly comparable to manually parallelized C code.

Triolet's performance comes from several new techniques for mapping container traversal interfaces onto distributed hardware. Data is shared within a cluster node, reducing communication overhead and memory use relative to a purely message-passing implementation. Some container traversal functions construct problem-specific data distribution strategies. Some container traversal functions introduce nested loops to process irregular workloads without introducing costly communication phases. Like prior loop fusion frameworks for sequential and shared-memory parallel programs, Triolet embeds these techniques into a common internal representation of iterators and containers. Mapping a large collection of traversal functions into a common internal representation affords a degree of extensibility and composability, simplifying the task of designing a library that merges combinations of traversals into a single loop.

Triolet's results convey a few lessons for implementors of distributed container traver-

sal frameworks. Triolet demonstrates the usefulness of a flexible internal representation of communication and work distribution. Prior implementations of container traversal typically inject user code into pre-built parallel loops that distribute work and perform communication. In such a framework, developers reorganize loops and data structures to fit the pre-built computational structure, which often introduces overhead. Triolet's iterators allow more flexible construction of data distribution schemes and parallel loop nests, reducing the need to reorganize loops and storage.

Triolet also demonstrates the usefulness of combining shared memory and message passing in a container traversal framework. Although the combination's performance benefits are widely recognized, nearly all container traversal implementations use only one or the other. In distributed parallel loops on clusters, sharing within a cluster node reduces communication overhead and cache pressure. Moreover, some algorithms do too little computation to offset their message-passing overhead, and thus are better suited to shared-memory parallelization. These differences translate to large performance benefits for some parallel loops.

For programming language implementors, Triolet serves as a counterpoint to the notion that high-level, high-performance parallel programming requires language and compiler support for parallelism. Triolet's parallel features are implemented as library code. Libraries are easier than compilers to develop, extend, and integrate with other software, making libraries an attractive approach to implementing parallel features. Triolet extends prior work in library-driven loop optimization with more flexible library abstractions for building loops. The library design motivated a refinement to the design of compiler optimizations: inlining was enabled for some cases of structurally recursive functions. These concepts can guide the design of other parallel programming systems.

It remains to be seen how Triolet's library design can be adapted to other programming languages. The library relies on some uncommon type system features. Triolet's type system plays a role in explaining and validating the programming interface, implementation, and compiler optimizations, in resolving overloading, and in detecting erroneous programs. Higher-kinded types express aspects of the container interface. Existential polymorphism

hides the types of temporary data structures used by iterators while still making type information available to optimization. Polymorphic unboxed types allow generic code to operate on efficiently packed data. Triolet's library design can probably be used in languages lacking these features, though it may mean giving up some usability, clarity, or performance.

8.1 Future Directions

Triolet demonstrates loop parallelization on a computational cluster for several algorithms using a variety of common container traversals. There are a number of directions this work could be extended.

It is desirable to compile the same high-level code to a variety of computing platforms. Triolet currently uses multiple cores and cluster nodes. However, Triolet cannot currently exploit other sources of parallelism found in high-performance computing platforms today. Triolet does not utilize the computing capabilities of GPUs or the vector units of CPUs. Extending Triolet to support these targets would broaden its applicability.

Triolet's execution model is that of a sequential program containing parallel loops. All data resides in the main processor's memory except when a parallel loop is running. Many applications are a poor fit for this execution model. Alternatives are to persistently distribute data across the memories of different processors, store data on a distributed filesystem, or stream data through a system without permanently storing it. Others have argued that all these use cases can be targeted by a common container traversal interface [29]. These execution models have different performance implications, yet they all fit the container traversal programming style, suggesting that a common infrastructure may suit them.

Triolet has only scratched the surface of the algorithms that can be expressed as container traversals. It is an open question whether a broad set of traversals and traversal-related optimizations can fit into a single framework. Some classes of algorithms have interesting and nontrivial sharing patterns, and have been expressed as container traversals. Sorting, grouping, and relational joins have been demonstrated in database query languages

and database-inspired map-reduce frameworks [19, 28, 29]. Graph traversals, graph algorithms, and unstructured mesh algorithms have also been demonstrated [94, 95]. While loop parallelization of stencil algorithms has received much attention, container traversal expression of such algorithms has not been thoroughly explored [96]. Each of these classes of algorithms has its own characteristic data reuse patterns suited to specialized communication strategies. Attempting to develop a common interface for all these patterns could foster a better conceptual and theoretical framework for the efficient parallel implementation of container traversals.

REFERENCES

[1] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[2] K. Keutzer, B. Massingill, T. Mattson, and B. Sanders, "A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects," in *Proc. 2010 Workshop on Parallel Programming Patterns*, 2010, pp. 9:1–9:8.

[3] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 105–118.

[4] J. Sipelstein and G. Blelloch, "Collection-oriented languages," in *Proc. IEEE*, vol. 9, no. 4, 1991, pp. 504–523.

[5] D. Hillis and G. S. Jr., "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.

[6] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a portable nested data-parallel language," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 4–14, 1994.

[7] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an embedded data parallel language," in *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 47–56.

[8] M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data Parallel Haskell: A status report," in *Proc. Workshop on Declarative Aspects of Multicore Programming*, 2007, pp. 10–18.

[9] M. Fluet, M. Rainey, J. Reppy, and A. Shaw, "Implicitly threaded parallelism in Manticore," *Journal of Functional Programming*, vol. 20, no. 5–6, pp. 537–576, 2010.

[10] S. Chatterjee, "Compiling nested data-parallel programs for shared-memory multiprocessors," *ACM Transactions on Programming Langugages and Systems*, vol. 15, no. 3, pp. 400–462, 1993.

[11] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw, "Data-only flattening for nested data parallelism," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 81–92.

[12] J. Feo and D. Cann, “A report on the Sisal language project,” *Journal of Parallel and Distributed Computing*, vol. 10, pp. 349–366, 1990.

[13] S.-B. Scholz, “Single Assignment C—Efficient support for high-level array operations in a functional setting,” *Journal of Functional Programming*, vol. 3, no. 6, pp. 1005–1059, 2003.

[14] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, “Regular, shape-polymorphic, parallel arrays in Haskell,” in *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2010, pp. 261–272.

[15] P. Wadler, “Deforestation: Transforming programs to eliminate trees,” *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, Jan. 1988.

[16] A. Gill, J. Launchbury, and S. Peyton Jones, “A short cut to deforestation,” in *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1993, pp. 223–232.

[17] D. Coutts, R. Leshchinskiy, and D. Stewart, “Stream fusion: From lists to streams to nothing at all,” in *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2007, pp. 315–326.

[18] A. Farmer, C. Hoener zu Siederdissen, and A. Gill, “The HERMIT in the stream: Fusing stream fusion’s concatMap,” in *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2014, pp. 97–108.

[19] E. Meijer, “Confessions of a used programming language salesman,” in *Proc. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2007, pp. 677–694.

[20] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, “Abstraction mechanisms in CLU,” *Communications of the ACM*, vol. 20, no. 8, pp. 564–576, Aug. 1977.

[21] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky, “A generic parallel collection framework,” in *Euro-Par 2011: Proc. International Conference on Parallel Processing*, 2011, pp. 136–147.

[22] P. Hudak, “Building domain-specific embedded languages,” *ACM Computing Surveys*, vol. 28, no. 4, December 1996.

[23] H. Chafi, A. Sujeeth, K. Brown, H. Lee, A. Atreya, and K. Olukotun, “A domain-specific approach to heterogeneous parallelism,” in *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 35–46.

[24] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, “Shader algebra,” *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 787–795, Aug. 2004.

[25] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: Using data parallelism to program GPUs for general-purpose uses,” in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 325–335.

[26] M. Chakravarty, G. Keller, S. Lee, T. McDonell, and V. Grover, "Accelerating Haskell array codes with multicore GPUs," in *Proc. 6th Workshop on Declarative Aspects of Multicore Programming*, 2011, pp. 3–14.

[27] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Du Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *Proc. 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011, pp. 224–235.

[28] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Úlfar Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 1–14.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conference on Hot Topics in Cloud Computing*, 2010, pp. 10–10.

[30] H. J. Lee, K. Brown, A. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky, "Implementing domain-specific languages for heterogeneous parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 42–53, 2011.

[31] T. Veldhuizen, "Arrays in Blitz++," in *Proc. 2nd International Symposium on Computing in Object-Oriented Parallel Environments*, 1998, pp. 223–230.

[32] H. Tanno and H. Iwasaki, "Parallel skeletons for variable-length lists in SkeTo skeleton library," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, 2009, vol. 5704, pp. 666–677.

[33] N. Bell and J. Hoberock, *Thrust: A Productivity-Oriented Library for CUDA*. Morgan Kaufman, 2011, ch. 26.

[34] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[35] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[36] G. Mainland, R. Leshchinskiy, and S. Peyton Jones, "Exploiting vector instructions with generalized stream fusion," in *Proc. ACM International Conference on Functional Programming*, 2013, pp. 37–48.

[37] D. Coutts, D. Stewart, and R. Leshchinskiy, "Rewriting Haskell strings," in *Proc. International Conference on Practical Aspects of Declarative Languages*, 2007, pp. 50–64.

[38] S. Peyton Jones and S. Marlow, "Secrets of the Glasgow Haskell Compiler inliner," *Journal of Functional Programming*, vol. 12, no. 5, pp. 393–434, Jul. 2002.

[39] UPC Consortium, "UPC language specifications, v1.2," 2005. [Online]. Available: http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf

[40] R. Numrich and J. Reid, “Co-array Fortran for parallel programming,” *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[41] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, “Titanium: A high-performance Java dialect,” *Concurrency: Practice and Experience*, vol. 11, no. 11–13, pp. 825–836, 1998.

[42] 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. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005, pp. 519–538.

[43] B. Chamberlain, D. Callahan, and H. Zima, “Parallel programmability and the Chapel language,” *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312.

[44] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, “Productivity and performance using partitioned global address space languages,” in *Proc. International Workshop on Parallel Symbolic Computation*, 2007, pp. 24–32.

[45] P. Ciechanowicz and H. Kuchen, “Enhancing Muesli’s data parallel skeletons for multi-core computer architectures,” in *Proc. IEEE International Conference on High Performance Computing and Communications*, 2010, pp. 108–113.

[46] J. Protic, M. Tomasevic, and V. Milutinovic, “Distributed shared memory: Concepts and systems,” *IEEE Parallel & Distributed Technology: Systems & Technology*, vol. 4, no. 2, pp. 63–71, 1996.

[47] J. Carter, D. Khandekar, and L. Kamb, “Distributed shared memory: Where we are and where we should be headed,” *Hot Topics in Operating Systems*, p. 119, 1995.

[48] C. Herrmann and C. Lengauer, “HDC: A higher-order language for divide-and-conquer,” *Parallel Processing Letters*, vol. 10, no. 2–3, pp. 239–250, 2000.

[49] R. Loogen, Y. Ortega-mallén, and R. Peña-marí, “Parallel functional programming in Eden,” *Journal of Functional Programming*, vol. 15, no. 3, pp. 431–475, 2005.

[50] N. Scaife, S. Horiguchi, and G. M. P. Bristow, “A parallel SML compiler based on algorithmic skeletons,” *Journal of Functional Programming*, no. 4, pp. 615–650, 2005.

[51] V. Loup, “Computer “experiments” on classical fluids. I. thermodynamical properties of Lennard-Jones molecules,” *Physical Review*, vol. 159, pp. 98–103, 1967.

[52] P. Wadler, “List comprehensions,” in *The Implementation of Functional Programming Languages*, S. Peyton Jones, Ed. Prentice Hall, 1987, pp. 127–138.

[53] S. Peyton Jones, A. Tolmach, and T. Hoare, “Playing by the rules: Rewriting as a practical optimisation technique in GHC,” in *Proc. ACM SIGPLAN Haskell workshop*, 2001, pp. 203–233.

[54] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 89–100.

[55] A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *Proc. 27th European conference on Object-Oriented Programming*, 2013, pp. 52–78.

[56] T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky, "Optimizing data structures in high-level programs: New directions for extensible compilers based on staging," in *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, 2013, pp. 497–510.

[57] S. Peyton Jones, "Compiling Haskell by program transformation: A report from the trenches," in *Proc. European Symposium on Programming*, 1996, pp. 18–44.

[58] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee, "TIL: A type-directed optimizing compiler for ML," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.

[59] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-M. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," IMPACT Research Group, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[60] A. Appel, *Compiling with Continuations*. Cambridge University Press, 2007.

[61] A. de Medeiros Santos, "Compilation by transformation in non-strict functional languages," Ph.D. dissertation, University of Glasgow, 1995.

[62] C. Ellison and G. Roşu, "Defining the undefinedness of C," Computer Science Department, University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[63] K. Läufer and M. Odersky, "Polymorphic type inference and abstract data types," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1411–1430, 1994.

[64] S. Peyton Jones and J. Launchbury, "Unboxed values as first class citizens in a non-strict functional language," in *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, 1991, pp. 636–666.

[65] X. Leroy, "Unboxed objects and polymorphic typing," in *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, 1992, pp. 177–188.

[66] A. Ohori, "A polymorphic record calculus and its compilation," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 6, pp. 844–895, 1995.

[67] Z. Shao, "Flexible representation analysis," in *Proc. ACM SIGPLAN International Conference on Functional Programming*, 1997, pp. 85–98.

[68] A. Kennedy and D. Syme, “Design and implementation of generics for the .NET common language runtime,” in *Proc. ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2001, pp. 1–12.

[69] J. Schwartz, R. Dewar, E. Schonberg, and E. Dubinsky, *Programming with sets: An introduction to SETL*. New York, NY: Springer-Verlag, 1986.

[70] C. Lin and L. Snyder, “ZPL: An array sublanguage,” in *Proc. 6th International Workshop on Languages and Compilers for Parallel Computing*, 1994, pp. 96–114.

[71] L. Snyder, “The design and development of ZPL,” in *Proc. ACM SIGPLAN conference on History of programming languages*, 2007, pp. 8–1–8–37.

[72] S. Marlow, “Haskell 2010 language report,” 2010. [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/

[73] B. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[74] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, “The essence of compiling with continuations,” in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pp. 237–247.

[75] J. Reppy, “Optimizing nested loops using local CPS conversion,” *Higher Order and Symbolic Computation*, vol. 15, no. 2-3, pp. 161–180, Sep. 2002.

[76] J. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[77] A. Charguéraud and F. Pottier, “Functional translation of a calculus of capabilities,” in *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2008, pp. 213–224.

[78] J. Ferrante, K. Ottenstein, and J. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[79] C. Rodrigues, A. Dakkak, T. Jablin, and W.-M. Hwu, “Triolet: A programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing,” in *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 247–258.

[80] K. Kennedy and K. McKinley, “Maximizing loop parallelism and improving data locality via loop fusion and distribution,” in *Proc. International Workshop on Languages and Compilers for Parallel Computing*, 1994, pp. 301–320.

[81] B. Chamberlain, S.-E. Choi, S. Deitz, and A. Navarro, “User-defined parallel zippered iterators in Chapel,” in *Proc. Conference on Partitioned Global Address Space Programming Models*, 2011.

[82] J. Launchbury and S. Peyton Jones, “State in Haskell,” *Lisp in Symbolic Computation*, vol. 8, no. 4, pp. 293–341, 1995.

[83] H. Boehm and M. Weiser, “Garbage collection in an uncooperative environment,” *Software: Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[84] D. Cann, “Retire Fortran?: a debate rekindled,” *Communications of the ACM*, vol. 35, no. 8, pp. 81–89, Aug 1992.

[85] J.-L. Gaudiot, W. Böhm, W. Najjar, T. DeBoni, J. Feo, and P. Miller, “The Sisal model of functional programming and its implementation,” in *Proc. Second Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, 1997, pp. 112–123.

[86] R. Bernecky, S. Herhut, and S.-B. Scholz, “Symbiotic expressions,” in *Proc. International Conference on Implementation and Application of Functional Languages*, 2010, pp. 107–124.

[87] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, 2004, pp. 10–22.

[88] T. White, *Hadoop: The Definitive Guide*. O’Reilly Media, 2012.

[89] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-82, 2011.

[90] N. Scaife, G. Michaelson, and S. Horiguchi, “Comparative cross-platform performance results from a parallelizing SML compiler,” in *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, ser. IFL ’02, 2002, pp. 138–154.

[91] P. Maier and P. Trinder, “Implementing a high-level distributed-memory parallel Haskell in Haskell,” in *Implementation and Application of Functional Languages*, ser. Lecture Notes in Computer Science, 2012, vol. 7257, pp. 35–50.

[92] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Peña, S. Priebe, Á. Rebón, and P. Trinder, “Comparing parallel functional languages: Programming and performance,” *Higher-Order and Symbolic Computation*, vol. 16, no. 3, pp. 203–251, 2003.

[93] C. Herrmann, “The skeleton-based parallelization of divide-and-conquer recursions,” Ph.D. dissertation, University of Passau, 2000.

[94] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, “Liszt: A domain specific language for building portable mesh-based PDE solvers,” in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[95] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, “Simplifying scalable graph processing with a domain-specific language,” in *Proc. International Symposium on Code Generation and Optimization*, 2014, pp. 208–218.

[96] B. Lippmeier and G. Keller, “Efficient parallel stencil convolution in Haskell,” in *Proc. 4th ACM Symposium on Haskell*, 2011, pp. 59–70.