THE UNIVERSITY OF CHICAGO

IMPLEMENTATION TECHNIQUES FOR NESTED-DATA-PARALLEL LANGUAGES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
ADAM MICHAEL SHAW

CHICAGO, ILLINOIS
AUGUST 2011

For Sindhu, my best friend, my partner in life and my biggest supporter.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

To everyone who helped me either with the content of my dissertation, or by offering encouragement as I stretched to complete my graduate work, I am sincerely grateful.

I would first of all like to thank my parents, Richard and Jane Shaw, who always put education first, and who believed in me. I would not be here without you, in any sense.

To my brother Daniel, who inspired me when I was a little kid and inspires me now.

My mother-in-law, Mrs. S. Rajalakshmi, has provided essential family support over the last phase of this program, enabling my wife and me to push forward with our careers in a busy time for us both. Thank you!

Thanks to my countless fine teachers, who exemplify what education should be about and left me permanently optimistic about the educational process, come what may. There are too many of you to name, but, when I step into the classroom, you are all there. In particular I would like to acknowledge Mrs. Smith, Mr. McGee and Mrs. DeLucia from The Country School, Andy Jaffe from Williams College, and Sharon Salveter from the University of Chicago.

To my fellow graduate students at the University of Chicago, who sustained me with friendship and help when I needed it. I would like to thank my research partners Mike Rainey and Lars Bergstrom, who have earned my sky-high respect, and my longstanding officemates Jacob Matthews and Borja Sotomayor, who were great companions to have on this sedentary journey. Lars has been especially helpful with technical details, wisdom and encouragement in the later stages of this process. I have also enjoyed and benefited from the company of Andy Terrel, Peter Brune, Casey Klein, Joshua Grochow, Paolo Codenotti, Ross Girshick, Matt Rocklin and many others.

To my committee, Anne Rogers, Matthew Fluet, and my advisor, John Reppy, for their investment in my progress along this path. Without your expertise, guidance, and instruction, this dissertation work would not have been possible. Thank you all. I also gratefully acknowledge the scholarly guidance of Dave MacQueen and Aaron Turon, who gave freely of their time to help me understand things better.

For her faith in me and her support in so many forms, I thank my wife Sindhu. Sindhu, your help has been indispensable. Any achievements of mine are at least half yours.

And a special acknowledgment to my daughter Sonali, my sunshine and my joy, and the best thing that ever happened to me, by a long shot.

Adam Shaw

Chicago, Illinois

August 2011

The research presented in this dissertation was supported by the National Science Foundation, under NSF Grants CCF-0811389, CCF-0937561, and CCF-1010568.

ABSTRACT

The implementation of nested-data-parallel programming languages is a challenging and important problem. Well-engineered parallel languages help programmers make better use of the parallel computing resources at their ever-increasing disposal. Nested data parallelism is an especially attractive model for parallel programming, since it enables high-level parallel programs to exploit parallel resources well, regardless of the regularity or irregularity of a particular programming problem. In nested-data-parallel programs, programmers can employ the idioms to which they are accustomed and achieve parallel speedups with only modest accommodations in their source code.

Nested-data-parallel languages have been in existence for over two decades now, but the ground has shifted underneath them. As a platform for parallel computing, multicore computers long ago superseded the wide-vector machines for which nested-data-parallel compilers were originally designed. Nevertheless, nested-data-parallel compilation, while adapted to ever more sophisticated languages, has remained fundamentally unaltered in its orientation toward vector instructions. This dissertation challenges longstanding techniques for compiling nested-data-parallel programs in a vector-machine style, on the basis that they are not appropriate for multicore targets.

We present hybrid flattening as a suitable alternative to traditional techniques for the compilation of nested data parallelism. Hybrid flattening transforms nested data structures in order to expose programs to various optimizations, while leaving control structures intact. We provide a semantics of hybrid flattening in the form of Flatland, a model language with a rewriting system, and prove properties about its formal integrity. We then define aggressive hybrid flattening, a practical application of Flatland's rewriting rules in an optimizing transformation. We demonstrate the effectiveness of aggressive hybrid flattening with our Parallel ML implementation, and we report encouraging experimental results across various benchmark applications.

CHAPTER 1
INTRODUCTION

Fast parallel computers have become common: they are relatively inexpensive and widely available. The ubiquity of parallel computing machines has given rise to a renewed demand for good parallel programming languages. Many such programming languages have been developed and are being developed now, but the research space is active and crowded with unanswered questions. What is the best paradigm for parallel programming? How can language designs help programmers manage the complexity of their applications? Should programming languages give programmers direct control over parallel resources, or should that control be automated? Answering these design questions leads to questions about implementation. How do we compile surface-level language constructs to target hardware? Which features can we support? Can our language support portability across diverse machines? How broad can we make our language's application domain?

Active projects in parallel programming language research, some with long histories, include NESL [8], Cilk [10], X10 [46], Chapel [14], Fortress [48], Data Parallel Haskell [15], and PML [21]. Each of these languages embodies its own vision for how parallel programming ought to be done, but they all include some form of *implicitly-threaded parallelism*. When a language supports implicitly-threaded parallelism, the programmer has mechanisms to provide hints to the compiler about parallel execution without directly specifying the low-level details of that execution, such as how it should map onto particular threads at runtime.

NESL is the seminal language supporting *nested data parallelism*. Nested data parallelism is a programming model allowing multiple levels of parallelism, both interprocedural (parallel operations) and intraprocedural (calling subroutines in parallel). Nested-data-parallel languages enable efficient parallelism over nested-data-parallel constructs, no matter how deeply nested and regardless of irregularity. Most recently, Data Parallel Haskell and PML have taken up the challenge of supporting nested data parallelism in the context of implicit threading. This dissertation uses PML as its experimental platform.

NESL took up the challenge of supporting nested data parallelism, and succeeded, long ago. Nevertheless, in integrating nested data parallelism into languages currently under development, there is work left to do. In the time since NESL's development, the hardware landscape has changed completely. NESL's compilation techniques are directed toward wide-vector parallel machines. In particular, NESL applies the *flattening transformation* to generate wide-vector executables. But multicore machines are different than wide-vector machines, with a completely different model of execution. We know flattening is not necessary for PML, whose implementation already enjoys considerable success with its flattening-free compiler technology. But is it desirable? We have previously conjectured that flattening would improve the performance of PML programs if it were incorporated into our system. To test this conjecture, we developed *hybrid flattening*, a novel program transformation designed with multicore machines as our intended targets. The present research describes and formalizes hybrid flattening as an abstract system, and furthermore demonstrates that a PML implementation of hybrid flattening does, in fact, improve the performance of PML programs.

1.1 Implicitly-Threaded Parallelism

Implicitly-threaded parallelism affords the programmer freedom to focus on the high-level concepts of solving his or her target problem, without having to manage parallelism directly. In a programming language that supports implicit threading, the programmer provides annotations to indicate which parts of the program might profitably be executed in parallel. The mapping of specific computations onto parallel threads is delegated to the compiler and runtime system; no threads exist at the language level. For any annotated computation, the implementation may choose to execute it in parallel or sequentially. The final decision rests with the system, since in a given context, the overhead of parallel execution could actually degrade rather than improve performance.

Implicit threading inhabits the middle ground between fully-implicit and explicitly-threaded parallelism. In fully-implicit parallelism, the programmer cedes all control of parallelism to the compiler. Programs do not indicate any parallel computation anywhere; the system identifies opportunities for parallelism and exploits them. The pH language [37] exemplifies this approach. In explicitly-threaded parallel languages, by contrast, the programmer controls all the parallelism directly, specifying exactly when to spawn threads, when to synchronize on pending results, when to release the resources of threads no longer needed back to the system, and so on. Parallel programming in C has this character.

Implicit threading is neither entirely declarative like fully-implicit parallelism, nor entirely direct like explicitly-threaded parallelism. Our language, PML, supports implicitly-threaded parallelism with a variety of constructs and idioms. (PML supports explicitly-threaded parallelism as well, which is outside the scope of the present work.) In the sequential parts of a PML program, the system makes no attempt to identify any parallelism. In the implicitly-threaded parts, parallel threads are created and managed automatically, out of the programmer's view. We see implicit threading as similar in spirit to garbage collection; the programmer relinquishes some control, and possibly some performance, in exchange for working with a body of code that is clearer, more concise, more modular, easier to maintain, and easier to build upon.

Implicitly-threaded language implementations must make good choices about how much parallelism to employ in executing annotated computations. Too much parallelism hurts performance because of its overhead costs; too little parallelism wastes opportunities to exploit all available resources. Finding the right amount of parallelism consistently and for a broad selection of programs is a hard problem. In PML, we employ both static and dynamic techniques to decide when parallel execution is warranted and ensure that it is well-distributed across processing elements at runtime.

1.2 Nested Data Parallelism

Modern languages with support for implicitly-threaded parallelism are, in many cases, augmented with support for nested data parallelism. Nested data parallelism is a declarative style for programming irregular parallel applications. In nested-data-parallel programming, nested arrays need not have regular structure — subarrays may have different lengths — yet parallel execution over those arrays remains efficient and balanced. Nested data parallelism makes it possible to write efficient parallel programs in familiar idioms. Common algorithms such as quicksort and Barnes-Hut n-body simulation [4] need not be reformulated to assume a regular shape; they can be expressed in the usual way, as irregular recursive divide-and-conquer problems, and the compiler and runtime system make sure the work is well-balanced at runtime. Nested data parallelism is attractive specifically because of its ability to allow programmers to write parallel programs in the style to which they are already accustomed.

Traditional flattening enables the efficient compilation of irregular nested data parallelism by transforming both data structures and control structures rather profoundly, to enable the use of efficient parallel vectorized and segmented operations on the transformed code. Hybrid flattening borrows certain elements of traditional flattening and dispenses with others — specifically, it imitates its data structure transformation, but leaves control structures intact — in order to take advantage of traditional flattening's performance-improving capabilities in the context of multicore computation.

1.3 Contributions

This dissertation contributes to the literature on compilation of nested-data-parallel languages in the following ways.

- We define *hybrid flattening* as a family of transformations over nested-data-parallel programs.
- We formalize hybrid flattening as a rewriting system over a model programming language, and prove important properties of the system.
- Within the framework of hybrid flattening, we define *aggressive hybrid flattening* as a particular strategy for compiling nested-data-parallel programs.

- We use the definition of aggressive hybrid flattening as the basis for implementing a compiler transformation in the PML compiler, `pmlc`.
- We measure flattened PML programs against unflattened ones, and demonstrate consistent performance gains of flattening in side-by-side comparisons across a range of benchmarks.

Both flattened and unflattened PML programs perform well with respect to fast sequential baselines, and scale well up to as many as 48 processors.

1.4 Outline of the Dissertation

The outline of this dissertation is as follows.

- Chapter 2 surveys the related work, providing a context for the departure of the present work from its predecessors.
- Chapter 3 describes PML and Manticore, our experimental platform, and argues a rationale for our novel hybrid flattening approach without providing full technical detail.
- Chapter 4 presents Flatland, a formal system describing the semantics of hybrid flattening transformations, and proves properties of its rewriting system such as type preservation. We use Flatland to define the specific rewriting strategy of aggressive hybrid flattening, a transformation designed with multicore compilation in mind.
- Chapter 5 explains the implementation of aggressive hybrid flattening in `pmlc` and shows how flattening exposes transformed programs to various optimizations not otherwise available.
- Chapter 6 reports performance results from a group of experiments testing the effectiveness of aggressive flattening in PML.
- Chapter 7 concludes and outlines directions for future work.

CHAPTER 2

RELATED WORK

Nested data parallelism has been an attractive parallel programming paradigm for more than two decades now. The early work on compilation of nested-data-parallel languages was directed at massively-parallel wide-vector machines. Such computers operate under a *single instruction, multiple data* (SIMD) regime. SIMD machines compute on wide vectors of scalar data in parallel, one instruction at a time. At any given moment, the instruction being performed by such a machine is the same at every location in the vector. SIMD hardware operates only in this way. SIMD-style computation is fundamentally different than computation on *symmetric multiprocessor* (SMP) machines, such as the multicore machines targeted by PML in the present work. In an SMP regime, a group of processors, each having access to a shared pool of memory, all compute at the same time independently of one another, and what synchronization is needed between the processors must be managed explicitly by software. Despite the fact that nested data parallelism has expanded beyond its SIMD origins to (increasingly common) symmetric multiprocessors, its compilation techniques have generally retained their original SIMD orientation, and have been subsequently adjusted to work in an SMP setting.

In this section, we consider related work save our own; the discussion of Manticore and PML research is folded into Chapter 3. All the related work is at some point concerned with immutable sequences of scalar data, but the terminology varies from one paper to the next. Blelloch's NESL work [8] uses "sequence"; Keller's dissertation [30] uses the term "vector"; Data Parallel Haskell papers generally use the term "array." Manticore and PML papers use "array." To unify terminology, we use "array" throughout this dissertation to refer to surface language constructs, in reference to others' work as well as our own, and we use "vector" to refer to hardware-level (compiled) data structures in SIMD and SIMD-style regimes.

2.1 NESL

Blelloch took up the challenge of compiling nested-data-parallel programs for wide-vector parallel machines in his 1988 dissertation (adapted and published as a book in 1990 [6]). In that work, and in a 1990 paper with Sabot [9], he presented the *flattening transformation*, which has since become the subject of several decades of further research. The original flattening translation research uses the programming language PARALATION LISP [45].

Blelloch presents NESL, an ML-like language with nested-data-parallel features, in 1993 [7], as an alternative surface language to PARALATION LISP that employs the same compilation techniques. NESL is a strict, parametrically-polymorphic language with ML-style type inference, with a built-in set of simple type classes such as `number`. NESL's design is minimal. It provides scalars of a few basic types and associated operators, sequences, simple datatypes and pattern matching, conditionals and let bindings, top-level function definitions, and a parallel *apply-to-each* construct that is the inspiration for, among other constructs, the *parallel comprehension* construct in PML.

Blelloch's flattening transformation entails substantial changes both to data structures and code. Its effect is to alter representations of nested data structures such that they consist only of flat vectors, which can then be operated on efficiently in parallel by correspondingly altered code. The flattening transformation is particularly effective in treating irregular parallelism: certain operations like the parallel *segmented sum* operation, where sums are computed over an array of arrays of numbers, complete in the same number of steps regardless of the irregularity of the shapes of the segments. This property (of segmented sums and other irregular operations) allows a wide variety of properly-encoded irregular parallel problems, many examples of which are detailed in Blelloch's dissertation, to execute in parallel on a SIMD machine without having to pay any performance penalty for their irregularity.

With respect to the representation of nested arrays, the flattening transformation operates as follows. Flattened arrays have two components: one or more flat data vectors (more than one in the case of unzipped tuples), containing the elements of the nested array in left-to-right order,

```
s  = [[1,2,3],[4,5],[6,7,8,9]]
sF = ([1,2,3,4,5,6,7,8,9], [3,2,4])
```

Figure 2.1: Flattened array with a lengths segment-descriptor.

and one or more *segment-descriptors*. Blelloch defines segment-descriptor as “any structure that defines the segmentation of a vector.” [1] In NESL, a segment-descriptor is always a flat vector of integers, and a flattened array carries with it one segment-descriptor for each level of nesting. There are various forms of segment-descriptors: *lengths*, *head flags*, and *head pointers*, which contain the lengths of subsequences, booleans marking segment beginnings, and indices of segment beginnings, respectively. Different segment-descriptor types have advantages and disadvantages: for example, head-flags is the same length as the flat data vector, which is desirable on a wide-vector machine, but cannot represent empty segments, while lengths can represent empty segments, but differs in length from the flat data vector. Figure 2.1 gives an example of a lengths-type segment-descriptor: `s` is the nested array of integers and `sF` is its flattened counterpart. In PML, we use our own *shape tree* expressions (discussed in the next chapter) to play the role of segment-descriptors, and among the segment-descriptor variants given by Blelloch, shape trees are closest to lengths.

The flattening transformation’s operations on data structures are complemented by more radical operations on control structures. To demonstrate its code transformation, we consider an example where factorial is applied in parallel to an array of integers, using NESL-like syntax in the following code excerpt.

```
val ns = [1,2,3,4];

function fact(n) =
  if (n < 2) then 1 else n * factorial(n-1);

val fs = { fact(n) : n in ns };
```

1. [6], p. 67.

```
function fact^(ns) = let
  if (length(ns) == 0) then ns else let
    val flags  = ns <^ dist(2, length(ns))
    val bases  = pack (ns, flags)
    val ones   = dist (1, length(bases))
    val nflags = not^ flags
    val rs0    = pack (ns, nflags)
    val rs1    = rs0 -^ dist(1, length(rs0))
    val rs2    = fact^ rs1
    val rs3    = ns *^ rs2
    in
      combine (ones, rs3, flags)
    end
```

Figure 2.2: Definition of `fact^`.

The value bound to the variable `fs` is a parallel *apply-to-each* expression. It states that `fact` is to be applied in parallel to all elements in `ns`. To execute this program is not a matter of simply using each processor to apply `fact` independently in parallel. Such an execution is impossible on NESL's SIMD targets: the program must proceed, a vector-wide instruction at a time, in lock-step toward its goal. To rewrite the apply-to-each expression bound to `fs`, NESL synthesizes a new function `fact^`. Whereas `fact` consumes an integer and produces an integer, `fact^`, its *vectorized* (suitable for vectors) counterpart, consumes an array of integers and produces an array of integers. That is, the type of `fact^` is the result of applying the array type constructor to the function's input and output types. Once `fact^` has been defined, the apply-to-each expression bound to `fs` is simply rewritten to `fact^ ns`. NESL follows a set of rules in transforming arithmetic operators, tests, and conditional expressions to arrive at the definition of `fact^` presented (once again, using a paraphrase of NESL's syntax) in Figure 2.2. The process by which `fact^` is synthesized from the definition of `fact` is implemented in the NESL compiler and described in prose in the NESL literature, but it was not presented as a formal rewriting system until later work (see below).

Some remarks are warranted to explain the code in Figure 2.2. Vectorized operators are marked with a caret (`^`). Vectorized operators represent pointwise application in parallel. The relationship between operators and their vectorized versions is as follows. Whereas the operator < is an infix operator for a pair of integer expressions, yielding a boolean, the operator `<^` is an infix operator on a pair of integer vector expressions of equal length, yielding a vector of booleans. The `dist` operator replicates its first argument across a vector of the length indicated by the second argument. Therefore the expression

```
ns <^ dist(2, length(ns))
```

evaluates to a vector of booleans, of the same length as `ns`, where for every element in `ns` less than 2 there appears a corresponding `true`, and for all the other elements `false`. The `pack` operator extracts all values from its first vector argument corresponding to `true` flags in its second vector argument and builds a fresh vector from them. The call to `combine` performs the inverse operation, choosing elements from one of two vectors, in order, according to the flags in its third argument, a vector of booleans.

The body of the original `fact` function is a conditional, branching according to a test of its (single scalar) argument. That conditional no longer appears in `fact^`: the flattening transformation has compiled it away. (The issue is confused by the fact that `fact^` has independently introduced a conditional, testing the length of the argument `ns` as its termination condition.) In place of `fact`'s original test, the boolean vector `flags` has been computed by the pointwise application of `< 2`, and `flags` has been used to extract a vector of base cases and a vector of recursive cases (`bases` and `rs0`, respectively). Those two vectors are then computed on, an instruction at a time, by subsequent operations. The reason underlying this conditional transformation is that in a SIMD regime, all elements are computed on simultaneously by a single vector instruction; no individual element can be singled out for special treatment. This restriction is at the heart of the present work, since in an SMP environment, it no longer exists, calling into question the aptness of this compilation strategy for multicore SMP machines.

Blelloch's thesis was written before commodity multicore machines existed. In characterizing the original motivation for his compilation techniques, Blelloch wrote, "to be useful, it must be possible to map [...] nested parallelism onto a flat parallel machine." [6] This is demonstrably no longer the case. Mapping nested parallelism onto a multicore machine is very much useful, and even with our prototype implementation, we can, by means of SMP parallelism, improve upon the performance of fast sequential programs by considerable margins (see Chapter 6).

NESL demonstrated the viability of the flattening transformation as a compilation technique, demonstrating particular success in the area of irregular parallel applications. Beyond its use as a research system, NESL has been used as a platform for teaching parallel programming at CMU for two decades. NESL's capabilities are sufficient to study and teach nested data parallel programming, but as a realistic programming language it is restricted. Basic features of modern functional languages are absent. To name two: there are user-defined datatypes and pattern matching in NESL, but there are no sum types or recursive types; furthermore, functions must be named and can only be defined at the top level. Sum and recursive types and first-class functions are absent from NESL specifically because its compilation technology — the flattening transformation — had no way to account for them. In a given application, NESL's missing features can be compensated for by proper encoding of the problem; in fact, a considerable portion of Blelloch's thesis is devoted to encoding algorithms on trees and graphs using nested-data-parallel arrays as a sort of universal representation. Yet NESL's incompleteness in providing the usual set of modern programming language mechanisms left open terrain in the language design space. Research remained to be done first to write down the flattening transformation as a formal system, then to extend it such that it could include a broader selection of programming language features. Subsequent projects aimed at exactly those targets.

2.2 Proteus

In 1993, Prins *et al.* [41, 38] presented Proteus, a functional nested-data-parallel language in the NESL mold. Proteus is a not a full-featured programming language in that it contains only as many constructs as necessary to demonstrate the authors' compilation techniques. In addition to scalar values, arithmetic operations and basic array operations, Proteus consists of function definition and application, let bindings, conditional expressions, and a declarative parallel iterator construct that is equivalent to NESL's apply-to-each form.

The Proteus work is the first to formalize flattening of nested-data-parallel programs as a rule-based translation. Their flattening transformation is, like NESL's, directed towards compilation for parallel vector machines, Its rules succinctly describe elements of the flattening transformation presented in this dissertation as well, including compilation of iterators to tabulations over integer ranges, and unzipping of arrays of tuples (both of which are part of PML's flattening implementation, discussed in Chapter 5).

The Proteus research describes two ways [41] of representing flattened nested arrays. Their *vector tree representation* gathers together the *value vector*, a flat vector of data containing the scalars from the original structure in left-to-right order, and another vector for each level of structure. This corresponds directly to NESL's flattened representation, with its flat data vectors and collections of segment-descriptors. Proteus's alternative representation is its *nesting tree representation*, which bears a resemblance to the representation used in the current implementation of PML.[2] A Proteus-style nesting tree is an n-ary tree. The children of internal nodes above the leaves represent segments; the children of the lowest level of nodes—that is, the leaves—contain, collectively, the data of the original nested array in left-to-right order. In PML, we also employ trees to capture nesting structure, but our shape trees are a separate structure from our flat data vectors (PML's representation scheme is given in full detail in the chapters that follow).

Proteus's flattening transformation is very much a predecessor of the current work. Certain of Proteus's transformation rules—those that describe the transformation of data structures—are echoed in our system. Proteus's essential point of divergence from the present work is, like NESL's, its orientation toward SIMD computing regimes, and everything that entails with respect to code

2. The Proteus papers do not directly discuss which of these representations they used in their software. A Proteus interpreter, but not a compiler, circa 1994, is still available at `ftp://ftp.cs.unc.edu/pub/projects/proteus/src`.

transformation.

2.3 Nepal and Data Parallel Haskell

NESL's limitations as a realistic programming language provided an outline for future research, as language designers naturally wished to integrate nested-data-parallel programming and standard modern programming language technologies (such as algebraic datatypes and higher-order functions). The step-by-step extension of NESL's foundation to a more feature-rich platform has ultimately taken the form of Data Parallel Haskell [15], having grown out of more than a decade of research in the context of various projects. The general contour of the research has been to start with NESL's flattening compilation as a foundation, then to extend incrementally the capabilities of the flattening transformation such that more and more modern mechanisms could be included in the surface language.

The first step in enriching NESL's flattening was Keller and Chakravarty's giving a type-theoretic foundation for flattening values of recursive types in 1998 [31]. In NESL, data structures such as trees and graphs need to be encoded as nested sequences. Keller and Chakravarty lifted this encoding burden by improving flattening to handle recursive-type values, so programmers could compute with recursive-type expressions inside arrays directly. As its model languages, Keller and Chakravarty's paper uses NESL as its source language and FKL (for "Flat Kernel Language") as the target of its flattening transformation. The same authors extended the flattening transformation shortly thereafter to sum types and separate compilation [16], using lambda-calculus-like model languages in their presentation. Note that these advances in what could be included in the surface of a nested-data-parallel language had no bearing on the target of the compilation; wide-vector machines continue to be the intended target of these systems.

Keller's 1999 thesis [30] includes a detailed formal treatment of the flattening transformation as adapted to accommodate nested arrays of recursive types, and furthermore extends compilation to account for parallel wide-vector distributed-memory machines. Compilation is modeled as a transformation from NKL, a nested kernel language, to FKL, a flat kernel language, and thereafter to DKL, a distributed kernel language. Flattening per NESL takes place in the compilation from NKL to FKL, and, at that stage in compilation, the system has in hand a program very much like one of NESL's post-flattening programs, with its `pack`- and `combine`-style operations (see Figure 2.2). From that point forward, compilation to DKL involves the introduction of the inverse `join` and `split` operators. The `split` operation, applied to a parallel array, has the effect of separating the array into chunks, each of which can be computed on separately in parallel, while the companion `join` reassembles the results of those pieces of computations into a parallel array. Each `join` is a synchronization point, so it is beneficial to remove them from transformed programs where possible. As Keller's dissertation states, the composition of `split` and `join` is equivalent to the identity function, and successive `split`/`join` applications can be eliminated through fusion; in fact this technique is still employed in Data Parallel Haskell (discussion of which follows).

With the introduction of *array closures*, it became possible to enable the flattening transformation to accommodate parallel arrays of first-class functions. Leshchinskiy *et al.* give an overview of *higher-order flattening* in a 2006 publication [34], and Leshchinskiy's 2005 dissertation [33] gives the topic of array closures a thorough theoretical treatment, as well as a novel formal presentation of the state-of-the-art enhanced flattening transformation including all the advancements achieved in their line of research.

The incremental enrichment of the flattening transformation led naturally to a design for a full-featured language, not lacking basic features or intended for proofs of concept, but meant to bring the spectrum of modern programming language features to bear on nested-data-parallel programming. Chakravarty *et al.* first present the language Nepal in 2001 [17], characterized as a version of Haskell including nested data parallelism. Nepal is succeeded by Chakravarty *et al.*'s Data Parallel Haskell [15], bringing together Nepal-style nested data parallelism with Haskell as implemented in the Glasgow Haskell Compiler (GHC) [26].

In 2008, Peyton Jones *et al.* give a thorough overview of the Data Parallel Haskell language and its compilation [39], including an updated account of how Data Parallel Haskell uses the flattening transformation in their implementation. In particular, and especially relevant to the present work, the 2008 paper needs to account for a major change in the landscape that has developed in the decades since the original appearance of the flattening transformation: that is, the widespread proliferation of multicore architectures. What happens after a system has compiled a Data Parallel Haskell program in the manner of NESL, and yet executes on an SMP rather than a SIMD machine? Their answer to this technical problem is to adapt the `split` and `join` mechanisms originally presented in Keller's dissertation to implement NESL-style operations (such as `pack` and `combine`) across the multiple processing elements on a multicore computer. Parallel computations are `split` across processing elements and subsequently `join`ed on completion. To eliminate unnecessary synchronization points, GHC uses rewrite rules [40] to erase successive applications of `split` and `join` (per the identity equivalence rule originally given by Keller). (In addition to this, various advanced fusion techniques are employed to streamline the resulting post-flattened program, an overview of which is given in their paper.) Though Data Parallel Haskell represents broad advances in NESL-style flattening in numerous ways, and although it has been adapted to run on multicore machines, its compilation strategy continues to reflect the SIMD orientation of its predecessors. This is the point from which the present work marks a distinct departure.

CHAPTER 3
THE RATIONALE FOR HYBRID FLATTENING

In this chapter, we present our novel approach to flattening for multicores in the context of the Manticore project. After a review of the related Manticore and PML literature in the first section, we give a rationale for our new formulation of *hybrid flattening* and intuitions about its utility.

3.1 Manticore and PML

The Manticore project is an infrastructure built to support languages with parallelism at multiple levels. Originally Manticore was the name of a programming language, although in more recent work, we have chosen to refer to the runtime system infrastructure as Manticore, and to the programming language built upon it (one of potentially many) as Parallel ML, or PML. We will consistently refer to the programming language as PML, even with respect to earlier work. We review the language and the infrastructure in turn.

3.1.1 The Language: PML

Fluet *et al.* presented the original PML design in 2007 [23]. The novel characteristic of the PML design was to combine both explicitly-threaded parallelism and implicitly-threaded parallelism in the same language. PML's explicitly-threaded parallel constructs are parallel versions of the core operations in CML [44], described most recently by Reppy *et al.* [43]. Its implicitly-threaded constructs, discussed below, are presented in detail in recent publications of Fluet *et al.* [21, 22].

PML provides implicitly-threaded parallel versions of a number of common sequential forms. PML's implicitly-threaded parallel constructs are parallel tuples, parallel arrays, parallel comprehensions, parallel bindings, and parallel cases. Each of these is a parallel form of a well-known sequential counterpart. This dissertation is concerned with the first three of these expression forms;

parallel bindings and parallel cases do not come into play. The detailed semantics of all forms is given by Fluet *et al.* [22].

Each implicitly-threaded construct hints to the compiler which computations are good candidates for parallel execution. The semantics of many of these constructs is sequential, and the system is always allowed to choose to execute them in a single thread. Sequential semantics provide the programmer with a clear deterministic programming model, and formalize the expected behavior of any PML compiler.

Parallel tuples are a lightweight notation for fork/join-style parallelism. Parallel tuples hint to the compiler that its arbitrarily (but finitely) many subexpressions are to be computed in parallel. Parallel tuples are written like SML's tuples, except they use parentheses with vertical bars as delimiters. A parallel tuple of n components is written as follows:

$$\texttt{(|}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\ \texttt{|)}$$

where e_1 to e_n are PML expressions. The subexpressions in a parallel tuple may differ in type; the expression as a whole has a product type of the usual form. In PML, any effects produced by the computations in a parallel tuple must take place in left-to-right order, in keeping with the sequential syntax of SML, although the expressions themselves may of course be evaluated at the same time.

A parallel array of n components is written as follows:

$$\texttt{[|}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\ \texttt{|]}$$

Once again, this construct recommends to the compiler that the n expressions inside the parallel array be evaluated in parallel, and once again the compiler may refuse. Each expression inside a parallel array must share the same type; if we call that type `t`, the whole expression has the type `t parray`.

PML provides *parallel comprehension* and *range* syntax to facilitate the specification of certain kinds of parallel arrays. It has a close analog in Haskell [28], the (sequential) list comprehension,

$$\frac{\Gamma \vdash e_t : \texttt{bool} \qquad \Gamma \vdash \mathrm{e}_1 : \tau_1 \quad \cdots \quad \Gamma \vdash \mathrm{e}_\mathrm{k} : \tau_\mathrm{k} \qquad \Gamma[\mathrm{p}_\mathrm{i} : \tau_\mathrm{i}] \vdash \mathrm{e}_0 : \tau}{\Gamma \vdash \texttt{[|}\ e_0 \mid p_1\ \textbf{in}\ e_1\texttt{,}\ \ldots\texttt{,}\ p_k\ \textbf{in}\ e_k\ \textbf{where}\ e_t\ \texttt{|]} : \tau\ \texttt{parray}}$$

Figure 3.1: Parallel comprehension typing.

and a closer analog in Data Parallel Haskell, which is also called a parallel comprehension and from which we borrow the name. The general form of a parallel comprehension is as follows:

$$\texttt{[|}\ e_0\ \texttt{|}\ p_1\ \textbf{in}\ e_1\texttt{,}\ \ldots\texttt{,}\ p_k\ \textbf{in}\ e_k\ \textbf{where}\ e_t\ \texttt{|]}$$

where e_0 is some expression (with free variables bound in the p_i) computing the elements of the array, the p_i are patterns binding the elements of the array-valued expressions e_i, and e_t is an optional boolean-valued expression over the p_i filtering the input. If the input arrays have different lengths, all are truncated to the length of the shortest input, and they are processed in parallel. The sequential semantics of parallel comprehension are similar to the semantics of mapping a function over a list using a standard implementation of `map` (such `List.map` from SML's Basis Library [25]). The main difference in the behavior of a comprehension and a typical `map` is that, when a pattern p_i does not match a given element of e_i, that element is skipped, leaving a gap in the result. For intuition about the typing of parallel comprehensions, a typing rule sketch (outside the context of a full typing judgment) is presented in Figure 3.1.

In PML, parallel comprehensions are the only native construct specifying a loop-like iteration. It is common to iterate over an integer interval, so PML provides a syntactic form called a range to facilitate it.

$$\texttt{[|}\ e_1\ \textbf{to}\ e_2\ \textbf{by}\ e_3\ \texttt{|]}$$

All the subexpressions of a range must by integers. Its start and end points represent the inclusive limits of an integer sequence. The limit on the left need not be less than the limit on the right. The expressions e_1 and e_2 correspond to the left and right limits of the sequence, and e_3 is the optional step size from one entry in the sequence to the next. If the clause **by** e_3 is omitted, the compiler assumes **by** `1` in its absence.

Retaining the names e_1, e_2 and e_3 for the start point, end point, and step size, respectively, of a range as sketched above, and for $e_3 \neq 0$, the sequence defined by that range is as follows:

$$\{n_k = e_1 + k(e_3) \mid k \in \mathbb{N}, (n_k = e_1) \vee (e_1 \leq n_k \leq e_2)\}$$

Under these semantics, a range is never fewer than one element in length, as the element e_1 is always part of the sequence, since $n_k = e_1$ is a sufficient condition for inclusion in the sequence and it is always the case that since $n_0 = e_1 + 0(e_3) = e_1$. Ranges where $e_3 = 0$ are undefined in the semantics and result in an exception's being raised at runtime in PML.

Ranges and parallel comprehensions are a facile combination, permitting the writing of concise code such as

```
[| n*n+1 | n in [| 1 to 6 |] |]
```

which evaluates to

```
[| 2, 5, 10, 17, 26, 37 |]
```

In our collection of PML benchmarks, many of our applications are expressed as parallel comprehensions over ranges. It is a genial idiom for declarative parallel programming and our compiler takes special care to handle it well. Regular nested parallel comprehensions, as they are often conveniently expressed as nested comprehensions over ranges, benefit from the tab flattening optimization made available by application of the flattening transformation, as we discuss below.

In its basis library, PML includes a collection of operators, such as `map`, `filter`, and `reduce` over parallel arrays, that make it possible to translate any nested-data-parallel NESL program into PML in a natural way. NESL programs and their PML translations are superficially similar. Like NESL and its relatives, PML is a nested-data-parallel language, enabling the same sort of declarative programming for regular and irregular parallel programs. Before the present work, however, the compilation of PML has not involved any form of the flattening transfor-

```
datatype 'a rope
  = Leaf of 'a seq
  | Cat of 'a rope * 'a rope
```

Figure 3.2: Polymorphic ropes in PML.

mation. Although PML has absorbed some of the surface features of the nested-data-parallel languages discussed in Chapter 2, its compilation has proceeded along entirely different lines.

Prior to the modifications outlined in the present document, PML has used *ropes* [12] to represent parallel arrays. Ropes, originally proposed as an alternative to strings, are persistent balanced binary trees with `seq`s, contiguous sequences of data, at their leaves. Figure 3.2 gives a simplified definition of our implementation of polymorphic ropes, suitable for the current presentation. (In our actual system, ropes' `Cat` nodes include length and depth information to speed balancing.) Read from left to right, the data elements at the leaves of a rope constitute the data of a parallel array it represents. Since ropes are physically dispersed in memory, they are well-suited to distributed construction, with different processors simultaneously working on different parts of the whole.

Concatenation of ropes is fast and inexpensive, requiring no data copying. Contrast rope concatenation to list or array concatenation, each of which requires linear copying. Furthermore, since a rope's data is distributed among its leaves, it occupies no single large chunk of memory. As a result it is a favorable representation for our garbage collector, which does not have to account for a large contiguous region of memory representing a single monolithic data structure. In addition, we use a rope's shape as a means of scheduling its parallel processing; its physical decomposition describes a parallel computational decomposition in a very direct way.

In the current non-flattening PML compiler, parallel arrays are mapped directly to ropes. The rope datatype and a broad set of rope operations, many of them parallel, are defined in the PML basis library, but none of the rope code is intended to be used directly in PML programs. There is a surface layer of syntax (*e.g.*, parallel comprehensions) and parallel array operations

(`PArray.map`, *etc.*) providing a layer of abstraction between parallel arrays at the surface, and ropes in compiled code. We operate underneath exactly this abstraction layer to change the internal representation of parallel arrays when flattening is enabled (see Chapter 5).

As a representation of parallel arrays, ropes, in comparison to flat sequences, have some weaknesses. First, random access to rope elements requires logarithmic rather than constant time. Second, ropes must remain balanced; balancing ropes can be costly. Third, mapping over multiple ropes is more complicated than mapping over multiple sequences, since the ropes can have different shapes; managing these complications can also be costly. Nevertheless, our performance studies [5, 42] have consistently demonstrated both good absolute performance and robust scaling for multiple processors across PML benchmarks using ropes as the foundational parallel data structure.

3.1.2 The Infrastructure: Manticore

PML runs under the Manticore runtime system. Auhagen *et al.* give a precise characterization of the current state of Manticore in a 2011 publication [3], which is recapitulated here in brief. Manticore's runtime system provides a hardware abstraction layer, a parallel garbage collector, a few scheduling primitives, and basic system services including networking and I/O. Its processor abstraction is the virtual processor, or *vproc*, each of which is hosted by its own pthread and pinned to a specific processor. Its garbage collector uses a split-heap architecture, following the parallel garbage collection design of Doligez, Leroy, and Gonthier [20, 19], and divides each processor's local memory into an old-data area and a nursery area, following Appel's semi-generational garbage collector design [2]. Garbage collection is either minor, major, or global; these are, respectively, entirely local per-processor collections, local collections with live data promoted into the global heap, and stop-the-world collections triggered by the global heap's size exceeding a limit.

Manticore is open to different task scheduling policies to determine the order in which to execute tasks and map tasks to processors. The scheduling policy used in the present work is *work stealing* [13, 27]. In work stealing, a group of workers, one per processor, collaborate on a given computation. Idle workers are made to bear most of the scheduling costs, while busy workers focus on finishing their work. We use the following well-known implementation of work stealing [11, 24]. Each worker maintains a deque (double-ended queue) of tasks, represented as thunks (suspended computations). When a worker reaches a point where it could execute two (mutually independent) computations in parallel, it pushes one of the two tasks onto the bottom of its deque, and it continues executing the other. Upon completion of its task, the worker attempts to pop a task off the bottom of its deque and run it. If the deque is not empty, then the task, which is necessarily the most recently pushed task, is simply run. If it is empty, on the other hand, all of the local tasks have been stolen by other workers, and the worker seeks to steal a task from the top of some other worker's deque. Potential victims are chosen at random from a uniform distribution.

One of the main advantages of traditional flattening-based compilation is its strength in handling irregular parallelism. In a SIMD regime, issues of processor utilization are eliminated by having marshaled the program into a series of whole-vector synchronized steps. NESL's achievement is to confer this benefit even to irregular programs. With PML, which does not run on SIMD machines and does not flatten during compilation (outside the present work), irregular parallel programs run the risk of using processors poorly; under-scheduled processors could be left idling while a few busy processors do all the work. Manticore mitigates against poor processor utilization both through work-stealing scheduling (see above) and *lazy tree splitting* [5].

Lazy tree splitting is Manticore's adaptation, for trees, of the *lazy binary splitting* technique of Tzannes *et al.* [49] for balancing parallel computation over flat arrays dynamically. In lazy binary splitting, decisions about delegating work among processors are based on dynamic estimations of load balance. We say a processor is *hungry* if it is idle and ready to take on new work, and *busy* otherwise. Splitting, whereby a processor offers half its remaining work to other hungry processors, is judged profitable when at least one remote processor is hungry. Communicating directly with other processors to check if they are hungry or not is costly, but, given a sufficiently inexpensive way to check for hungry remote processors, lazy binary splitting is an effective technique for achieving good load balance during parallel execution. Tzannes *et al.* have demonstrated the effectiveness of the following fast approximate hungry-processor check: if a processor's local work-stealing deque is empty, there is very likely a remote hungry processor, since all its work has been stolen; if its deque is not empty, the other processors are probably too busy to have stolen any work. This heuristic works surprisingly well considering its simplicity. The check is inexpensive, requiring only two local memory accesses and a compare instruction, and, in practice, it provides a sufficiently accurate estimate of whether or not splitting is profitable.

While lazy binary splitting is applicable to linear iterations over integer ranges or contiguous data structures, the technique must be modified to work on iterations over trees. The concept of "half of the rest of the work" is simple to determine in the case of a linear structure, but at the leaf of a tree (a rope, for example), it is less so. At a leaf, a typical tree-traversing program has no way to refer to any other part of the tree, since parts of the tree other than the leaf are generally not in view at that point. It is not immediately obvious where the notion of "half of the rest of tree" comes from. The innovation of lazy tree splitting is to use *zippers* [29] to provide exactly this capability. Once it is possible to divide the current remaining work over a tree in half, lazy binary splitting techniques apply to trees, and they have proven advantageous in practice.

By virtue of its work-stealing scheduling augmented with lazy tree splitting, PML is already an effective compiler of nested-data-parallel programs. We have previously conjectured [49] that flattening and lazy tree splitting would work well together; the concerns of lazy tree splitting, while related to those of flattening, are separate. We demonstrate below that, in conjunction with work stealing and lazy tree splitting, our version of flattening improves the performance of parallel execution, in the cases of both regular and irregular nested data parallelism.

3.2 Hybrid Flattening

We now give an overview of how we have adapted techniques based on the flattening transformation to improve the compilation of PML. In summary, we adopt NESL-style flattening of data structures, compiling nested parallel arrays to a customized flat representation, but, except insofar as we must transform code to operate on the flattened structures, we leave the code — and in particular, its control structures — essentially intact. Vectorized operations are not introduced. In full NESL flattening, it is necessary to represent control structures in data; it is not so in PML. Our work is fundamentally an experiment in what happens if one applies NESL-style data flattening in an existing SMP implementation. We name the technique *hybrid flattening*.

In full generality, hybrid flattening permits any amount of flattening (or, for that matter, unflattening) to occur at zero or more places in a program, while, in some sense, preserving the program's original meaning. Chapter 4 presents a theoretical basis for hybrid flattening, assigning the term a precise meaning and setting forth formalisms for rigorous treatment of hybrid flattening systems. Hybrid flattening need not be used to describe a transformation whereby every (or even any) parallel array is flattened: it does not itself express or embody a particular transformation policy. In this section, we describe our adaptation of hybrid flattening as the basis for an aggressive flattening transformation for PML. Leaving aside the formal details of hybrid flattening until the next chapter, the important point about hybrid flattening as applied to PML compilation is that control structures are left alone, not profoundly transformed in the manner NESL's treatment of `fact^` (see Figure 2.2 above).

Whereas PML without flattening compiles nested parallel arrays to nested ropes, PML with flattening compiles nested parallel arrays to flattened arrays. Flattened arrays, like nested arrays as compiled by NESL, consist of two pieces: a flat data vector, and a value representing the structure of the nested array called a *shape tree* (alternatively, a *shape*). By means of standard (in flattening) unzipping transformations, nested arrays of tuples are compiled to tuples of flattened arrays. In our implementation, flattened arrays are represented by the polymorphic `farray` datatype. To

```
datatype shape
  = Lf of int * int
  | Nd of shape list

datatype 'a farray
  = FArray of 'a rope * shape
```

Figure 3.3: Datatypes `shape` and `farray`.

represent the flat data vector part of flattened arrays, we use ropes, exploiting the considerable infrastructure we have already built to compute in parallel with them. Like ropes, flattened arrays are an internal representation, hidden from the programmer by parallel array syntax, types, and surface-level operations. Figure 3.3 gives the PML datatype definitions for both `shape` and `farray`. The datatype definition for `rope` is given in Figure 3.2 above.

To avoid confusion, in the examples that follow, we will refer to the `Leaf` and `Cat` constructors, from the datatype `rope`, and the `Lf` and `Nd` constructors, from `shape`, with qualified names such as `Rope.Leaf`, even though neither datatype is shown inside a module in Figures 3.2 or 3.3.

Shape trees are our adaptation of segment-descriptors in the NESL tradition. We discussed NESL's different segment-descriptor representations briefly in Section 2.1, including lengths, head flags and head pointers.[1] As mentioned, the latter two representations cannot account for empty subarrays, which is a crippling restriction. We immediately prefer the lengths representation for this reason, and it is the basis for shape trees. The lengths representation is not always preferred in SIMD contexts, since the length of lengths vectors generally differs from the lengths of flat data vectors, which can present difficulties on wide-vector machines and distributed-memory massively-parallel machines [30]. This length mismatch is an unimportant point in SMP execution, and so is not a concern in compiling PML. Furthermore, since we are not bound to use flat vector representations for everything, and to avoid contending with difficulties associated with keeping a bundle of separate but related lengths segment-descriptors consistent with one another, we package

1. In early implementations, flattened arrays carried a redundant bundle of collections of segment-descriptors, presumably so different types of segment-descriptors could be used when advantageous. Blelloch and Sabot mention this point in passing in 1990 [9], but do not elaborate on it.

our segment information in trees.

A `shape` is an n-ary tree whose leaves store integer pairs. Each leaf contains the starting index and the index of the elements following the segment of data in an `farray`. The shape `Shape.Lf (i, i+n)` describes a length-`n` segment starting at `i` and ending at the last position before `i+n`. (In practice, this choice of indices is a convenient convention that guards against common fencepost errors. For example, the length of a segment is computed simply by subtracting the low index from the high index.) A simple, flat parallel array of integers such as

```
[| 1, 2, 3 |]
```

has the following `farray` representation:[2]

```
FArray (Rope.Leaf [1,2,3], Shape.Lf (0,3))
```

The data in the original sequence appears here, at a `Rope.Leaf`, in the original order, and the accompanying `shape` — `Lf(0,3)` — means that the flattened array's only segment begins at position 0 and ends at position 2 (one before 3).

Nested parallel arrays are translated as follows. Consider the following nested array:

```
[| [| 1, 2 |], [| |], [| 3, 4, 5, 6 |] |]
```

Its flattened array representation is the following:

```
FArray (Rope.Leaf [1,2,3,4,5,6],
        Shape.Nd [Shape.Lf (0,2), Shape.Lf (2,2), Shape.Lf (2,6)])
```

The flat data appears in order in a `Rope.Leaf`. The shape is a `Nd` with three leaves: this means that the parallel array consists of three subsequences. The leaves tell us that the first sequence begins at position 0 and ends at 1, the second sequence is empty at position 2, and the third sequence begins at position 2 and ends at 5. This representation scales up to any nesting depth in a natural way.

2. For brevity, we present the sequence in the rope's `Leaf` node with list syntax. In our implementation, sequences at rope leaves are variously represented as vectors and arrays, but never as lists.

With shape trees, certain operations, otherwise complicated, are simple and efficient. Consider array subscript on a nested array of integers. This parallel array `x`

```
val x = [| [| 8, 7, 6 |], [| 5, 4 |] |]
```

is flattened to `xF` as follows:

```
val xF = FArray (Rope.Leaf [8,7,6,5,4],
                 Shape.Nd [Shape.Lf (0,3), Shape.Lf (3, 5)])
```

Assume we would like to select element 1 from `xF` (a value equivalent to the parallel array `[| 5, 4 |]`). We perform the operation simply by selecting the subtree at position 1 out of the shape tree of `xF`, which is, in this case, `Shape.Lf (3, 5)`. We can share the original flat data vector and need not copy any data from it. The result is the array `xF1` as follows:

```
val xF1 = FArray (Rope.Leaf [8,7,6,5,4],
                  Shape.Lf (3, 5))
```

The shape tree in `xF1` effectively tells us to ignore all data in the flat data vector before position 3. Under this implementation, flattened-array operators must be written to consult shape trees to determine which values are current in a given flattened-array value. In certain operations, such as flattened array concatenation, it does become necessary to clean flattened arrays — that is, to copy live data out of its flat data vector before further computation. But such copying need not take place at subscript time and may never be necessary; it can be delayed until demanded by a particular operation.

Examples containing short literal arrays, such as those presented in this section so far, are helpful for providing intuition about the transformation of data structures (and typical of the literature), but they may give the false impression that flattening nested arrays can always be performed at compile time. Literal values can be flattened at transformation time directly, the representation change incurring no runtime cost. In general, however, the compiler must cope with arbitrarily nested arrays whose dimensions are unknown until runtime. Consider this function `f`:

```
fun f n = [| [| i * j | j in [| 0 to i |] |]
                      | i in [| 0 to n |] |]
```

The parallel array `f` computes has no predetermined shape or size: it needs to be applied to an integer argument before such information exists. With respect to our flattening scheme, for such a parallel array of unknown dimensions, there is no way to compute the shape tree of its flattened representation in advance, let alone its flat data. Generally, although certain special cases can be analyzed at compile time, there is no universal way for the compiler to know the shape of a parallel array that is yet to be constructed. In many cases, the compiler needs to arrange for flattening to take place at runtime. We handle this issue, both in the formal system discussed in Chapter 4 and in our actual implementation, by inserting operators that generate flattened arrays once they have enough information to do so. We call these coercion operators, and discuss them in detail in the following chapters.

When nested arrays are transformed into flattened arrays, all operations applied to those array values must be correspondingly transformed. Our approach to this problem is to provide a core group of type-indexed families of array operators, each of which is implemented to perform its operation at every array type in its family. The group contains parallel array subscripting, and parallel maps, filters, and reductions over parallel arrays. Note that by including maps and filters, this core group subsumes what can be expressed in parallel comprehensions. All operations on parallel arrays are either members of this core group, or they are built from members of the group. As such, transformation of the type-indexed operators matching the transformations of data structures is sufficient to preserve the program's behavior.

As an example, consider the parallel array subscript operator. In PML, the syntax for parallel array subscripting is to write the infix operator `!` with a parallel array as its left argument and an integer as its right. Consider the following PML program, computing the first element of an array of pairs:

```
val ps = [| (1,2), (3,4) |]
val p0 = ps ! 0
```

PML infers the type `(int * int) parray` for `ps`. In the surface language, the subscript operator has the polymorphic type

```
'a parray * int -> 'a
```

so it is instantiated here to the type

```
(int * int) parray * int -> (int * int)
```

Consider the value of `ps` after flattening, which we will rename to `psF`:

```
val psF = (FArray (Rope.Leaf [1,3], Shape.Lf (0,2)),
           FArray (Rope.Leaf [2,4], Shape.Lf (0,2)))
```

Extracting the first element (element 0) of the transformed array (now a pair of arrays) is now an entirely different operation. The subscript operator must select the first element of both flat data vectors inside `psF`, and use them to construct a pair. In the course of flattening the program, the original polymorphic subscript operator must be replaced by the operator that performs this very different operation. PML uses type information to choose such replacement operators as transformation proceeds. The type of the transformed data structure is sufficient to specify the operator we need. Not all possible such operators can be written in advance, since the family of type-indexed array operators is, in theory, infinite, and in practice, very large. As such, when we have no predefined definition for a given operator (such as a particular flavor of array subscript), we synthesize the operator at compile time and introduce it into the program at the point of application. Such operators can all be determined mechanically from their type, and their compile-time synthesis (while tricky to implement) requires no special technical machinery.

Until this point, we have not discussed the possibility of flattening expressions to some intermediate extent, or selecting only certain expressions as candidates for flattening. It is possible that such mixed flattening transformations would be useful in practice. For example, while unzipping tuples of scalar values has some obvious potential advantages (see Section 3.3), it might not be the case that performance gains are realized by unzipping pairs whose components are pointers to heap-allocated values, especially if the unzipping operations must be performed at runtime. More generally, it might not be beneficial to expend computational resources flattening a nested array if the operations performed on its elements fall under some threshold of computational activity; one can imagine such a case where the overhead of flattening a structure overwhelms the possible benefits of having done so. The hybrid flattening framework in Chapter 4 provides a means of defining flattening transformations of this mixed character, although the transformation in itself offers no particular guidance about designing the appropriate heuristics and analyses. We have thus far only implemented an aggressive hybrid flattening in PML, although we remain interested in using our system as a platform for exploring different flattening strategies.

3.3 The Advantages of Hybrid Flattening

The purpose of flattening data structures and code for multicore nested data parallelism is not because it must be possible for our high-level declarative programs to run as a sequence of SIMD instructions. This is the task of flattening historically. We flatten nested data parallelism for multicores because it improves the performance of SMP execution, not only because flattened operations perform better than, or at least no worse than, their non-flattened counterparts, but because flattened programs are amenable to various powerful optimizations that cannot be applied to non-flattened programs. In this section, we consider these optimizations in turn. Note that not all of these optimizations are employed in the PML compiler as it stands: we give a current account of the status of PML in Chapter 5.

Monomorphization. Monomorphization is an optimization whereby a polymorphic data structure containing uniformly-represented (*i.e.*, boxed) elements is transformed to a representation containing raw (unboxed) elements in their place. Monomorphization is possible for various data structures, such as containers for boxed scalars. Consider a value of type `int rope` in PML. Without monomorphization, the system will build an `int rope` with sequences of pointers to heap-allocated integers at its leaves. Monomorphization can turn a PML `int rope` into an int_rope, a specialized datatype that represents only ropes of integers, and as such carries sequences of raw integer values at its leaves. Monomorphization of polymorphic values preserves original meaning in a faster and lighter-weight structure. Monomorphic int_ropes are less expensive to construct and traverse than polymorphic int ropes, as there are many fewer allocations in constructing them and less indirection in reading individual values from the leaves, which contain the data directly.

The flattening transformation, by virtue of unzipping arrays of tuples, exposes more opportunities for monomorphization than otherwise. In PML, arrays of double pairs, for example, become pairs of double arrays, which in turn become farrays, each containing a specialized rope of doubles as its flat data vector. Monomorphization is well known to be valuable even outside the context of nested-data-parallel compilation. MLton [35], an optimizing whole-program SML compiler, performs monomorphization to generate better-performing sequential code. PML stands to benefit from monomorphization even without flattening (PML currently does no monomorphization unless flattening is enabled), although it will never be the case that, without unzipping tuples, non-flattened PML will have as many opportunities to do it. Monomorphization is also a foundational component of the optimizations that follow, all of which benefit from the faster traversal and uniform layout of scalar vectors.

Map flattening. In nested data parallel code, nested map expressions over regular or irregular data structures can be flattened into efficient linear traversals. This optimization is applicable when the function applied to the nested map computes scalars from scalars; in the example here, we will use the function sqr that computes the square of its argument. The following code excerpt binds a nested parallel comprehension to the name a:

```
val nss = [| [| 1, 2, 3 |], [| 4 |], [| 5, 6, 7, 8 |] |]
val a   = [| [| sqr n | n in ns |] | ns in nss |]
```

The nested comprehension is equivalent to

```
PArray.map (PArray.map sqr) nss
```

and is rewritten as such by the compiler. In this example, the fact that `nss` has an irregular shape (that is, not all its inner arrays are of the same length) is irrelevant to the fact that we can use *map flattening* on it. The irregular array bound to `nss` is transformed to the following flattened array value:

```
val nssF = FArray (Rope.Leaf [1, 2, 3, 4, 5, 6, 7, 8],
                   Shape.Nd [Shape.Lf (0, 3),
                             Shape.Lf (3, 4),
                             Shape.Lf (4, 8)])
```

Note that, to compute the flattened value of `a`, the function `sqr` can be applied over the flat data vector in `nssF` without any involvement from the shape tree, and the shape tree in `nssF` can simply be shared with the flattened result. To make this concrete, if we name the flattened result `aF`, its computation is

```
val aF = FArray (Rope.mapP sqr (Rope.Leaf [1, 2, 3, 4, 5, 6, 7, 8]),
                 shapeOf nssF)
```

where `Rope.mapP` is an operation applying a function in parallel to all elements of a rope, and `shapeOf` is an operator returning the shape component of an `farray` value. This optimization can be applied to scalar-to-scalar nested maps regardless of the depth of nesting. Without map flattening, lazy tree splitting (see above) is employed by PML to compensate for the potential effect of irregular shapes in parallel array (*i.e.*, rope) data structures. While lazy tree splitting does improve the performance of irregular nested maps such as the computation of `a` above, it can certainly do no worse when applied to the simpler linear map-flattened computation of `aF` post-transformation.

Tab flattening. Nested parallel comprehensions over ranges have *regular* structure: at each dimension, the length of every array is fixed a constant. Two-dimensional regular arrays can be thought of as rectangles, three-dimensional regular arrays as cubes, and so on. The regularity of such structures can be exploited by the *tab flattening* optimization, which performs simple integer arithmetic operations to collapse multidimensional tabulations into linear ones.

Every one-dimensional parallel comprehension of scalars is trivially regular:

```
val xs = [| Double.fromInt i | i in [| 0 to 9 |] |]
```

The straightforward, and inefficient, implementation of this parallel comprehension is to translate it to a `map` over the parallel array containing the integers from 0 to 9.

```
PArray.map Double.fromInt [| 0 to 9 |]
```

This naïve translation entails building an ephemeral data structure that is immediately computed with and discarded. To save the cost associated with this intermediate structure, the compiler rewrites parallel comprehensions over ranges as tabulations:

```
PArray.tabulate (10, Double.fromInt)
```

Tabulating over integer intervals requires no intermediate data structures, and realizes a performance improvement over the build-and-map strategy outlined above. (Keep in mind that this is a parallel tabulation that distributes its rope construction among available processors.)

Nested parallel comprehensions naturally give rise to nested tabulations. The computation of `xss` in this excerpt

```
val xss = [| [| (i*10)+j | j in [| 0 to 9 |] |]
                         | i in [| 0 to 9 |] |]
```

can be naturally expressed by a tabulate within a tabulate as follows:

```
PArray.tabulate (10, fn i =>
  PArray.tabulate (10, fn j =>
    (i*10) + j))
```

This translation is already better than using maps with ephemeral structures, but the shape of our flattened array representations allows us to use tab flattening to improve on nested tabulations. Recall our evaluation of `xss` results in an `farray` containing a flat data vector and a shape tree. We name the result `xssF` and sketch it as follows:

```
  val xssF = FArray (Rope.Leaf [0, 1, 2, ..., 99],
                     Shape.Nd [Shape.Lf(0,10), ..., Shape.Lf(90,100)])
```

We can generate the flat data vector of `xssF` in one tabulation, over a single counter representing the total number of elements in the nested array, by performing the appropriate index arithmetic on the counter:

```
let fun k = let
      val (i, j) = (k div 10, k mod 10)
      in
        (i*10) + j
      end
  in
    PArray.tabulate (10*10, f)
  end
```

The shape tree in rectangular cases has a simple regular structure as well, and be computed from the dimensions of a regular array in a straightforward way.

Tab flattening operation scales to any number of dimensions for regular nested arrays. In PML, the implementation of tab flattening is complicated by the fact that ranges may step either up or down, they may have a stride greater than 1, and they need not begin at zero. A full treatment of tab flattening in our system, including the subtleties of its index arithmetic and details about how shape trees are computed for regular structures, is presented in Chapter 5.

Segmented instructions. One of NESL's important accomplishments was to formulate various *segmented instructions* so they could be computed in lockstep in parallel on vector hardware. As discussed in Chapter 2, NESL's fast segmented operations are an important element of NESL's ability to perform well on irregular nested-data-parallel programs, and an important one for PML to emulate.

NESL's segmented sum operation, for example, is able to compute the sums of a nested array of numbers in a fixed number of steps regardless of the irregularity of the array's structure. This operation is critical to the performance of sparse-matrix/vector multiplication, a common benchmark in the related work and one for which we present encouraging results in Chapter 6. Here is an example[3] of an irregular sum computation in PML:

```
let val nss = [| [| 1, 2 |], [| |], [| 3, 4, 5, 6 |] |]
  in
    [| sum ns | ns in nss |]
  end
```

If this parallel comprehension is rewritten such that the `sum` operation is simply mapped over the array-valued elements of `nss`, the irregularity of the structure of `nss`, if there is wide variation in the lengths of its elements, is bound to affect load balancing adversely. Dynamic techniques like lazy tree splitting help as much as they can, but it is better (as our results demonstrate) to implement a special segmented sum operation written to compute on a flattened rather than a nested array. For the present research, we implemented an efficient parallel segmented sum operation for PML for irregular nested arrays. PML's implementation of segmented sum is described in Chapter 5.

Vector-width instructions. Flattening was originally developed for the ultimate vector-width instructions: those of massively-parallel vector machines such as the CM-2 Connection Machine. Many multicore machines, including the Magny-Cours processors we use as our experimental platform, provide relatively narrow vector instructions for computing over groups of scalars at a time. (Our processors are capable of computing with 16-byte-wide vectors.) Through unzipping and monomorphization, our flattening transformation pushes nested arrays towards representation as flat vectors of raw scalars. Thus flattening puts us in a strong position to exploit vector instructions for in our executions. At the time of this writing, the PML compiler lacks support in its code generator for vector instructions, so for us this remains uncharted experimental territory. Nevertheless, flattening PML helps us move toward the goal of employing vector instructions in our generated code, and has been noted as an advantage of flattening in the related work from the beginning.

3. This example is of course orders of magnitude too small to cause any difficulties in practice.

3.4 Example: Sparse-Matrix Vector Multiplication

We close this section with an example demonstrating the application of aggressive hybrid flattening as implemented in PML. Further examples are discussed accompanying the experimental results in Chapter 6.

We sketch our transformation of a nested-data-parallel program to compute multiplication of a sparse matrix by a dense vector. Our code transformation follows closely along the lines of NESL-style flattening and superficially resembles it, but there are several points of divergence. One obvious difference is the absence of vector-width operations in our transformation: operations such as vector-width multiply and vector-width subscript are absent. We employ a segmented sum operation in our transformation, we have implemented segmented sum for multicore machines without using vector instructions (as NESL does) or emulating vector instructions on multicores with distributed operations (as Data Parallel Haskell does). PML's implementation of segmented sum for ropes is given in Chapter 5. We also differ from NESL-style flattening in our `farray` representation, pairing ropes with shape trees. All operations on `farray`s in PML are SMP-style parallel operations.

In the standard nested-data-parallel encoding of sparse-matrix vector multiplication [8, 15], sparse matrices are represented as arrays of parallel arrays of pairs. Each inner array represents a row, and within each row each pair represents a column index and a value. We say each such row is a sparse vector. A small sparse matrix is represented by the following irregular array:

```
[| [| (0, 0.1), (9, 0.2) |],
   [| (1, 0.3) |],
   [| |],
   [| (3, 0.4), (8, 0.5), (9, 0.6) |] |]
```

This particular array, after unzipping and flattening, consists of a pair of flattened arrays:

```
(IFArray (IntRope.Leaf [0, 9, 1, 3, 8, 9],
          Shape.Nd [Shape.Lf(0,2), Shape.Lf(2,3),
                    Shape.Lf(3,3), Shape.Lf(3,6)]),
 DFArray (DoubleRope.Leaf [0.1, 0.2, 0.3, 0.4, 0.5, 0.6],
          Shape.Nd [Shape.Lf(0,2), Shape.Lf(2,3),
                    Shape.Lf(3,3), Shape.Lf(3,6)]))
```

The first flattened array in the pair contains the indices in order, along with a shape tree containing nesting structure; the second array contains the values in order, and contain an identical shape tree. Note that both flat data vectors are monomorphic, stored in an `IntRope` and a `DoubleRope`, respectively: the compiler has exchanged specialized internal representations for the parallel arrays in the surface language. The flattened array constructors are also specialized to `IFarray` and `DFarray`, since per PML typing the polymorphic `FArray` constructor can only be used with polymorphic ropes. Thanks to the representation transformation performed by hybrid flattening in this case, we are able to operate on it fast in parallel.

The dense vector with which the sparse matrix is to be multiplied is also represented as a flattened array. Compiled dense vectors are simply a monomorphic flattened array containing a `DoubleRope` of data and a shape tree consisting of a single leaf.

The surface program to compute the product of a sparse matrix and a dense vector is as follows.

```
fun dotp (sv, v) = sum [| x * v!i | (i,x) in sv |]

fun smvm (sm, v) = [| dotp (sv,v) | sv in sm |]
```

Without performing any flattening steps, by rewriting the parallel comprehensions to maps over ropes, we already have a viable compilation strategy. But we shall forge ahead and flatten. Inlining the body of `dotp` into `smvm`, the body of `smvm` becomes

```
[| sum [| x * v!i | (i,x) in sv |] | sv in sm |]
```

When `sum` appears inside a parallel comprehension, it may be lifted out of that comprehension and replaced with a segmented sum operation, as follows:

```
segsum [| [| x * v!i | (i,x) in sv |] | sv in sm |]
```

Rewriting parallel comprehensions as maps gives

```
segsum (map (map (fn (i,x) => x * v!i)) sm)
```

Finally, a map-flattening rewrite, as discussed in the previous section, is applied to the nested maps in this expression, yielding

```
segsum (map (fn (i,x) => x * v!i) sm)
```

where this `map` is a function equipped to perform flattened mapping of a scalars-to-scalar function as described in the previous section. The results in Chapter 6 show `smvm`, transformed in this way, is able to outperform the untransformed program by a wide margin.

CHAPTER 4

A FORMAL SYSTEM FOR HYBRID FLATTENING

Our predecessors perform what we call *total flattening*. In total flattening, all arrays are transformed to collections of flat scalar vectors, and control structures are encoded in vectors of data. Total flattening was introduced in Blelloch and Sabot's original work [9], and it has been propagated through the literature, as a total transformation, all the way to the present, with one exception (Chakravarty *et al.*'s partial vectorization [18], discussed below). As we have argued, assumptions that made total flattening a profitable compilation strategy for massively-parallel vector machines do not hold on modern multicore hardware. In this chapter, we set forth a formal framework for designing and reasoning about *hybrid flattening*. In hybrid flattening, not every nested array in a program is necessarily flattened. A hybrid flattening transformation may flatten a program not at all, it may flatten it completely, or it may flatten some parts of a program to intermediate extents. Though hybrid flattening can transform data structures, it leaves control structures intact. Hybrid flattening describes flattening transformations designed, from the first, with multicore machines as intended targets.

Hybrid flattening transformations are able to perform both flattening and unflattening steps. The framework itself is agnostic with respect to how flattening or unflattening steps are applied, and to what end; it simply provides the operations needed to define transformations and a set of rules for introducing and eliminating flattening and unflattening steps. The framework could, in fact. be used to define an "unflattening" transformation, where the user writes down a completely flat program, and the compiler transforms it backwards into a nested data-parallel program. (In using terminology like "flatten" and "completely flat" here, we are appealing to intuition; precise characterizations of these concepts follow below.)

Flattening and unflattening operators are represented in our system by type-coercion operators (or simply "coercions"). For the coercion operator that transforms values of type τ_1 into values of type τ_2, we write $\tau_1 \triangleright \tau_2$. Coercions are not to be confused with type casts, which are often no-cost notifications inserted to placate a type checker (as in C, for example). Except for trivial identity coercions (which do come into play in our system), coercions are potentially expensive representation-changing operations, and, except for a critical few, we are eager to eliminate as many of them from the program as possible.

When we want to perform a flattening step, we insert a coercion. For example, a standard step in flattening transformations is to unzip arrays of pairs of scalars—that is, to reshape an array of pairs into a pair of arrays of equal length. The coercion that unzips an array of integer pairs is written $[(\mathit{int}, \mathit{int})] \triangleright ([\mathit{int}], [\mathit{int}])$. In an actual implementation (such as the one described in Chapter 5), this coercion is implemented with a type-specialized equivalent to SML's `ListPair.unzip` [25]. Its inverse coercion $([\mathit{int}], [\mathit{int}]) \triangleright [(\mathit{int}, \mathit{int})]$ (like `ListPair.zip`) is also part of the language of coercions. If it is ever the case that a pair of inverse coercions, like these two, are successively applied to a value (or, equivalently, composed with one another), they may be rewritten to an identity coercion, and subsequently removed from the program. The removal of such cancelling coercions is one of the key mechanisms by which performance optimizations are achieved in our system.

The system presented here, named *Flatland* (after Abbott's 1884 novella [1]), consists of the following parts: a monomorphic type system with a few common type constructors; an explicitly-typed nested data-parallel language with type coercion operators; a well-formedness judgment for typechecking; a set of rewriting rules for safely introducing type coercions into expressions; and a set of rules for moving coercions around within programs and eliminating them. Table 4.1 gives a summary of the syntactic forms we use in this chapter to define the judgments and relations of the system.

Well-formed programs are guaranteed to remain well-formed under any legal transformation in our rewriting system. The top-level type of the whole program remains fixed under transformation, but they types of the subexpressions within a program may change. The well-formedness guarantee is maintained by the following mechanism. For every coercion introduced into the pro-

Table 4.1: Flatland judgments and relations.

$\Gamma \vdash s : \nu$	shape tree typing
$\vdash \tau \triangleright \overline{\tau}\ ok$	well-formedness of coercions
$\Gamma \vdash e^{\tau}\ ok$	well-formedness of explicitly-typed terms
$\tau_1\ \mathbf{A}\ \tau_2$	τ_1 is an array of τ_2
$\tau_1\ \mathbf{L}\ \tau_2$	τ_1 and τ_2 are compatible representations
$e^{\tau} \mapsto \overline{e}^{\tau}$	hybrid flattening
$\mathbf{F}[\![\tau]\!] = \overline{\tau}$	aggressive type flattening
$\Delta \vdash e^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond \overline{e}^{\overline{\tau}}$	aggressive hybrid flattening for terms
$e^{\tau} \Downarrow \overline{e}^{\tau}$	aggressive hybrid flattening for whole programs

gram, inverse coercions are correspondingly introduced to maintain the stability of the types in the program. Every time a coercion is applied to the value to which a variable *x* is bound, for example, the inverse coercion is introduced at every use of coerced *x*. This ensures that no clients of *x* are put in a position to compute with a term of the wrong type, post-coercion. This kind of balanced introduction of coercions is performed by the system at every rewriting step. Type-indexed operators such as $\mathbf{map}$ and $\mathbf{filt}$ have the ability to absorb type coercions, since, for each one, there are many implementations from which to choose; this helps reduce the number of type coercions in the transformed program. We defer further discussion of this property of type-indexed operators until later.

Hybrid flattening presents us with choices: which terms should we flatten, and how much should we flatten them? Unzipping arrays of tuples, mentioned above, is a mandatory step in every previous flattening transformation [9, 38, 30, 33]. If the array to be unzipped is of pairs of scalars, unzipping stands to yield performance improvements, as the nonflat representation — an array of pointers to heap-allocated pairs — can be exchanged for a pair of monomorphic vectors of scalars. Not only are such vectors smaller and faster to compute with than vectors of boxed values, they can also be used in vector instructions, potentially yielding constant factor speedups. If, on the other hand, the array contains pairs of non-scalars (heap-allocated values), unzipping the array may not improve performance; the cost of unzipping, if performed at runtime, might dominate the flattened representation's benefits. More generally, if there is any overhead entailed either by flattening a data structure, or by computing with its flat representation, there will be cases where, depending on how the structure is used, flattening is not worth its cost.

Depending on the flattening policy put to use in a given system, our system can express our own adaptation of total flattening, or one of various other transformations of interest. The following transformations, among others, can be expressed in our framework:

- Flatten all arrays, always, including unzipping arrays that contain pairs.
- Flatten all nested arrays of scalars, but refrain from unzipping arrays of pairs.
- Flatten only array literals longer than some threshold value.

The language designer can use the framework to craft the transformation appropriate to the circumstances.

4.1 Flatland

We now present the language, static semantics, and rewriting rules of Flatland in turn.

4.1.1 The Language

Our model language is an explicitly-typed, monomorphic, strict, pure functional language with pairs, parallel arrays, and first-class functions. Flattened and non-flattened terms commingle in Flatland: there is no inherent distinction between source language and target language. *Parallel arrays*, or simply *arrays*, are sequences of expressions written inside square brackets, and *flattened parallel arrays*, or *flattened arrays*, are pairs of flat vectors of data and *shape trees* describing the nesting structure.

Figure 4.1 presents Flatland's types. Throughout this presentation, types are ranged over by the metavariable τ. We use subscript indices (τ_i) and overbars ($\overline{\tau}$) to distinguish types from one

$$
\begin{array}{rcll}
\tau & ::= & g & \textit{ground types} \\
& \mid & (\tau, \tau) & \textit{pairs} \\
& \mid & \tau \rightarrow \tau & \textit{functions} \\
& \mid & [\tau] & \textit{parallel arrays} \\
& \mid & \{\tau \,;\, \nu\} & \textit{flattened parallel arrays} \\
\\
\nu & ::= & \textit{lf} & \textit{structure of flat arrays} \\
& \mid & \textit{nd}(\nu) & \textit{structure of nested arrays} \\
\\
g & ::= & \textit{int} \mid \textit{bool} &
\end{array}
$$

Figure 4.1: Flatland: types.

another. The type language consists of ground types int and $bool$, pairs, functions, parallel arrays, and flattened parallel arrays. Parallel array types are written with square brackets, and flattened array types with curly braces. Flattened parallel types include shape types as subcomponents; this is the only place where shape types occur. Shape types are isomorphic to the natural numbers and they give the nesting depth of arrays they describe. Shape types are ranged over by the metavariable ν (for "nesting").

Figure 4.2 contains Flatland's term language. Every term t includes an explicit type as a superscript. For brevity, we elide type superscripts in the notation where the type is unnecessary. Where the type is not obvious, we include it. The metavariable b ranges over constants, and x ranges over variables. We assume there exists a basis of constants and operators, including both ground terms, such as integer and boolean constants, and a standard assortment of common primitive operators, such as integer addition and logical negation.

Parallel array terms in the source language are written with square brackets. Flattened arrays are written with curly braces. Every flattened array carries a shape tree. A shape tree is an n-ary tree whose leaves contain pairs of integers. Each leaf in a shape tree specifies the endpoints of a subsequence in its containing flattened array. We follow the convention that the first integer argument to a leaf is the first position of a segment, and the second integer argument is one more than the last position of a segment.

t	$::=$	e^{τ}	
e	$::=$	b	*ground terms*
	$\mid$	x	*variables*
	$\mid$	$\textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$	*conditionals*
	$\mid$	$\textbf{let}\ x = t\ \textbf{in}\ t$	*let expressions*
	$\mid$	$\textbf{fun}\, f\, x^{\tau} = t\ \textbf{in}\ t$	*function expressions*
	$\mid$	$t \circ t$	*function composition*
	$\mid$	$t\ t$	*application*
	$\mid$	(t, t)	*pairs*
	$\mid$	$\pi_i\ t$	*projection* $(i \in \{1, 2\})$
	$\mid$	$[t,\ \ldots,\ t]$	*arrays*
	$\mid$	$\{t,\ \ldots,\ t; s\}$	*flattened arrays*
	$\mid$	$t\ !_{\tau}\ t$	*array subscript*
	$\mid$	$\textbf{map}_{(\tau,\tau,\tau,\tau)}\ (t, t)$	*array map*
	$\mid$	$\textbf{filt}_{(\tau,\tau)}\ (t, t)$	*array filter*
	$\mid$	$\textbf{red}_{(\tau,\tau)}\ (t, t, t)$	*array reduction*
	$\mid$	$\tau \rhd \tau$	*type coercions*
s	$::=$	$\textbf{lf}(t, t)$	*leaves*
	$\mid$	$\textbf{nd}[s,\ \ldots,\ s]$	*nodes*
b	$::=$	$\mathrm{true} \mid \mathrm{false}$	
	$\mid$	$0 \mid 1 \mid \ldots$	
	$\mid$	$\mathrm{not} \mid + \mid \ldots$	

Figure 4.2: Flatland: terms.

Array subscript is written as the infix operator `!` and is typed as a function application to a pair. Map and filter have their commonly-accepted meanings (similar to the various `map` and `filter` functions in the SML basis). Map and filter are implemented as parallel operators in PML, but their parallel behavior does not come into play in the static system presented here. Reduce applies an associative operator to all elements in an array in no particular order, and uses a given zero argument as a left and right identity for that operator. Since the language is monomorphic, array operators do not have polymorphic implementations. Instead, we assume there is a type-indexed family of each such operator. We specify array operators by writing their type indices in a subscript. $!_{[\tau]}$ is the operator that selects elements of type τ from arrays of type $[\tau]$. $\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_3)}$ takes two arguments, a function of type $\tau_1 \to \tau_2$ and a term of array type τ_3 (Flatland's notion of array type is characterized precisely in the next section), and produces a term of array type τ_4. $\mathbf{filt}_{(\tau_1,\tau_2)}$ consumes a function of type $\tau_1 \to \mathit{bool}$ and a term of array type τ_2 and produces a (filtered) term of array type τ_2. $\mathbf{red}_{(\tau_1,\tau_2)}$ is applied to a triple: an operator of type $(\tau_1, \tau_1) \to \tau_1$, an identity of type τ_1, and a term of array type τ_2, yielding a term of type τ_1. The typing rules in the following section explain the constraints on the types indexing these operators.

As stated above, coercion operators are written $\tau \triangleright \overline{\tau}$, and we will discuss their properties in detail in the following section.

4.1.2 Static Semantics

Coercion operators are functions whose name indicates their type: a well-formed coercion operator $\tau \triangleright \overline{\tau}$ has the type $\tau \to \overline{\tau}$. Figure 4.3 gives the well-formedness judgment on coercions. The rules state the conditions under which a given type coercion is admitted into the language. Rule CO-FL says that coercions from $[\tau]$ to $\{\tau\ ;\ \mathit{lf}\}$ are legal; this coercion corresponds to a step of flattening a parallel array. The rule CO-FR performs the opposite coercion, rewriting flattened arrays of shape *lf* to parallel arrays. CO-ZL and CO-ZR correspond to unzipping arrays of pairs and zipping pairs of arrays, respectively. The rest of the rules are similar characterizations of traditional flattening

$$\frac{}{\vdash \tau \triangleright \tau \ \mathit{ok}} \ \text{CO-ID}$$

$$\frac{}{\vdash [\tau] \triangleright \{\tau \ ; \ \mathit{lf}\} \ \mathit{ok}} \ \text{CO-FL} \qquad \frac{}{\vdash \{\tau \ ; \ \mathit{lf}\} \triangleright [\tau] \ \mathit{ok}} \ \text{CO-FR}$$

$$\frac{}{\vdash \{\tau \ ; \ \mathit{nd}(\nu)\} \triangleright \{\{\tau \ ; \ \nu\} \ ; \ \mathit{lf}\} \ \mathit{ok}} \ \text{CO-LL}$$

$$\frac{}{\vdash \{\{\tau \ ; \ \nu\} \ ; \ \mathit{lf}\} \triangleright \{\tau \ ; \ \mathit{nd}(\nu)\} \ \mathit{ok}} \ \text{CO-LR}$$

$$\frac{}{\vdash [(\tau_1, \tau_2)] \triangleright ([\tau_1], [\tau_2]) \ \mathit{ok}} \ \text{CO-ZL} \qquad \frac{}{\vdash ([\tau_1], [\tau_2]) \triangleright [(\tau_1, \tau_2)] \ \mathit{ok}} \ \text{CO-ZR}$$

$$\frac{}{\vdash \{(\tau_1, \tau_2) \ ; \ \nu\} \triangleright (\{\tau_1 \ ; \ \nu\}, \{\tau_2 \ ; \ \nu\}) \ \mathit{ok}} \ \text{CO-ZFL}$$

$$\frac{}{\vdash (\{\tau_1 \ ; \ \nu\}, \{\tau_2 \ ; \ \nu\}) \triangleright \{(\tau_1, \tau_2) \ ; \ \nu\} \ \mathit{ok}} \ \text{CO-ZFR}$$

$$\frac{\vdash \tau \triangleright \overline{\tau} \ \mathit{ok}}{\vdash [\tau] \triangleright [\overline{\tau}] \ \mathit{ok}} \ \text{CO-A} \qquad \frac{\vdash \tau \triangleright \overline{\tau} \ \mathit{ok}}{\vdash \{\tau \ ; \ \nu\} \triangleright \{\overline{\tau} \ ; \ \nu\} \ \mathit{ok}} \ \text{CO-F}$$

$$\frac{\vdash \tau_1 \triangleright \overline{\tau}_1 \ \mathit{ok} \qquad \vdash \tau_2 \triangleright \overline{\tau}_2 \ \mathit{ok}}{\vdash (\tau_1 \rightarrow \tau_2) \triangleright (\overline{\tau}_1 \rightarrow \overline{\tau}_2) \ \mathit{ok}} \ \text{CO-FUN}$$

$$\frac{\vdash \tau_1 \triangleright \overline{\tau}_1 \ \mathit{ok} \qquad \vdash \tau_2 \triangleright \overline{\tau}_2 \ \mathit{ok}}{\vdash (\tau_1, \tau_2) \triangleright (\overline{\tau}_1, \overline{\tau}_2) \ \mathit{ok}} \ \text{CO-PAIR}$$

$$\frac{\vdash \tau_1 \triangleright \tau_2 \ \mathit{ok} \qquad \vdash \tau_2 \triangleright \tau_3 \ \mathit{ok}}{\vdash \tau_1 \triangleright \tau_3 \ \mathit{ok}} \ \text{CO-TRANS}$$

Figure 4.3: Well-formedness judgment on coercions.

$$\frac{\Gamma \vdash {e_1}^{int}\ ok \qquad \Gamma \vdash {e_2}^{int}\ ok}{\Gamma \vdash \mathbf{lf}({e_1}^{int}, {e_2}^{int}) : \mathit{lf}} \ \text{SHAPE-LF}$$

$$\frac{\Gamma \vdash s_1 : \nu \quad \cdots \quad \Gamma \vdash s_n : \nu}{\Gamma \vdash \mathbf{nd}[s_1, \ldots, s_n] : \mathit{nd}(\nu)} \ \text{SHAPE-ND}$$

Figure 4.4: Shape types.

and unflattening steps. It is necessary to mention a potential trouble spot in Figure 4.3. The unzip operators (CO-ZL, CO-ZFL) always succeed and produce two arrays of equal length. Their corresponding zip operators (CO-ZR, CO-ZFR) by contrast, have the potential to fail, because they might be applied to different-length inputs, in which case their behavior is undefined. We handle this issue by assuming that in every case the two zip coercions are applied to same-length arrays. (The point is moot with respect to aggressive hybrid flattening in Section 4.2, since in that system coercions are banned from source programs.) Nonsensical coercions like $int \triangleright (int, int)$ are not admitted by the system.

By the judgments in Figure 4.3, for every well-formed coercion, the inverse coercion exists and is well-formed.

Lemma 4.1.1 (well-formed coercions are invertible)**.** *If* $\vdash \tau \triangleright \overline{\tau}$ *ok, then* $\vdash \overline{\tau} \triangleright \tau$ *ok.*

Proof. The proof is by induction on the height of the deductions for the rules in Figure 4.3. The full proof is given in Appendix A. □

We present the static semantics of Flatland in Figures 4.4 and 4.5. Figure 4.4 defines the typing of shape trees, and Figure 4.5 contains a well-formedness judgment on terms. When a term e^{τ} is well-formed in environment Γ, we write $\Gamma \vdash e^{\tau}$ *ok*. The typing of shape trees is given in 4.4. Every term is explicitly typed, so deciding well-formedness is a matter of checking that type annotations are correct. For example, 1^{int} is well-formed, but 1^{bool} is not. In the static semantics, Γ is a finite map from variables to types. When we extend Γ to map x to τ, we write $\Gamma[x^{\tau}]$. We assume the existence of a basis environment BE mapping ground terms and operators to their types (see

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x^{\tau}\ ok}\ \text{OK-VAR} \qquad \frac{BE(b) = \tau \qquad (BE = \text{basis env})}{\Gamma \vdash b^{\tau}\ ok}\ \text{OK-B}$$

$$\frac{\Gamma \vdash {e_1}^{bool}\ ok \qquad \Gamma \vdash {e_2}^{\tau}\ ok \qquad \Gamma \vdash {e_3}^{\tau}\ ok}{\Gamma \vdash (\mathbf{if}\ {e_1}^{bool}\ \mathbf{then}\ {e_2}^{\tau}\ \mathbf{else}\ {e_3}^{\tau})^{\tau}\ ok}\ \text{OK-IF}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1}\ ok \qquad \Gamma[x^{\tau_1}] \vdash {e_2}^{\tau_2}\ ok}{\Gamma \vdash (\mathbf{let}\ x = {e_1}^{\tau_1}\ \mathbf{in}\ {e_2}^{\tau_2})^{\tau_2}\ ok}\ \text{OK-LET}$$

$$\frac{\Gamma' = \Gamma[f^{\tau_0 \to \tau_1}] \qquad \Gamma'[x^{\tau_0}] \vdash {e_1}^{\tau_1}\ ok \qquad \Gamma' \vdash {e_2}^{\tau_2}\ ok}{\Gamma \vdash (\mathbf{fun}\, f\, x^{\tau_0} = {e_1}^{\tau_1}\ \mathbf{in}\ {e_2}^{\tau_2})^{\tau_2}\ ok}\ \text{OK-FUN}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1}\ ok \qquad \Gamma \vdash {e_2}^{\tau_2}\ ok}{\Gamma \vdash ({e_1}^{\tau_1}, {e_2}^{\tau_2})^{(\tau_1,\tau_2)}\ ok}\ \text{OK-PAIR} \qquad \frac{\Gamma \vdash e^{(\tau_1,\tau_2)}\ ok \qquad i \in \{1,2\}}{\Gamma \vdash (\pi_i\ e^{(\tau_1,\tau_2)})^{\tau_i}\ ok}\ \text{OK-PROJ}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to \tau_2}\ ok \qquad \Gamma \vdash {e_2}^{\tau_1}\ ok}{\Gamma \vdash ({e_1}^{\tau_1 \to \tau_2}\ {e_2}^{\tau_1})^{\tau_2}\ ok}\ \text{OK-APP} \qquad \frac{\Gamma \vdash {e_1}^{\tau}\ ok \quad \cdots \quad \Gamma \vdash {e_n}^{\tau}\ ok}{\Gamma \vdash [{e_1}^{\tau}, \ldots, {e_n}^{\tau}]^{[\tau]}\ ok}\ \text{OK-ARR}$$

$$\frac{\Gamma \vdash {e_1}^{\tau}\ ok \quad \cdots \quad \Gamma \vdash {e_n}^{\tau}\ ok \qquad \Gamma \vdash s : \nu}{\Gamma \vdash \{{e_1}^{\tau}, \ldots, {e_n}^{\tau}; s\}^{\{\tau\,;\,\nu\}}\ ok}\ \text{OK-FARR}$$

$$\frac{\vdash \tau \rhd \overline{\tau}\ ok}{\Gamma \vdash (\tau \rhd \overline{\tau})^{\tau \to \overline{\tau}}\ ok}\ \text{OK-COERCE} \qquad \frac{\Gamma \vdash {e_1}^{\tau_2 \to \tau_3}\ ok \qquad \Gamma \vdash {e_2}^{\tau_1 \to \tau_2}\ ok}{\Gamma \vdash ({e_1}^{\tau_2 \to \tau_3} \circ {e_2}^{\tau_1 \to \tau_2})^{\tau_1 \to \tau_3}\ ok}\ \text{OK-COMP}$$

$$\frac{\Gamma \vdash {e_1}^{\tau}\ ok \qquad \Gamma \vdash {e_2}^{int}\ ok \qquad \tau' = (!\ \tau)}{\Gamma \vdash ({e_1}^{\tau}\ !_{\tau}\ {e_2}^{int})^{\tau'}\ ok}\ \text{OK-SUB}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to \tau_2}\ ok \qquad \Gamma \vdash {e_2}^{\tau_3}\ ok \qquad \tau_3\ \mathbf{A}\ \tau_1 \qquad \tau_4\ \mathbf{A}\ \tau_2}{\Gamma \vdash (\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_4)}\ ({e_1}^{\tau_1 \to \tau_2}, {e_2}^{\tau_3}))^{\tau_4}\ ok}\ \text{OK-MAP}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to bool}\ ok \qquad \Gamma \vdash {e_2}^{\tau_2}\ ok \qquad \tau_2\ \mathbf{A}\ \tau_1}{\Gamma \vdash (\mathbf{filt}_{(\tau_1,\tau_2)}\ ({e_1}^{\tau_1 \to bool}, {e_2}^{\tau_2}))^{\tau_2}\ ok}\ \text{OK-FILT}$$

$$\frac{\Gamma \vdash {e_1}^{(\tau_1,\tau_1) \to \tau_1}\ ok \qquad \Gamma \vdash {e_2}^{\tau_1}\ ok \qquad \Gamma \vdash {e_3}^{\tau_2}\ ok \qquad \tau_2\ \mathbf{A}\ \tau_1}{\Gamma \vdash (\mathbf{red}_{(\tau_1,\tau_2)}\ ({e_1}^{(\tau_1,\tau_1) \to \tau_1}, {e_2}^{\tau_1}, {e_3}^{\tau_2}))^{\tau_1}\ ok}\ \text{OK-RED}$$

Figure 4.5: Well-formedness judgment on terms.

$$\begin{array}{rcl} !\,[\tau] & = & \tau \\ !\,\{\tau\ ;\ \mathit{lf}\} & = & \tau \\ !\,\{\tau\ ;\ \mathit{nd}(\nu)\,\} & = & \{\tau\ ;\ \nu\,\} \\ !\,(\tau_1, \tau_2) & = & (!\,\tau_1, !\,\tau_2) \end{array}$$

Figure 4.6: Array-type subscripting.

rule OK-B).

Most of the well-formedness rules are familiar administrative rules that recursively propagate the judgment through subexpressions in natural ways. We draw attention to those rules that are distinct in our system. The rule on coercions (OK-COERCE) appeals to the earlier set of rules for well-formedness of coercions. The rule for array subscript (OK-SUB) uses the array-selection notation ($!\ \tau$), defined below, to compute the result type of the expression. The rules for the other three type-indexed array operators, OK-MAP, OK-FILT, and OK-RED, refer to the relation $\mathbf{A}$ in their premises. Broadly speaking, $\tau_1\ \mathbf{A}\ \tau_2$ should be read "τ_1 is an array of τ_2." The definition of $\mathbf{A}$ is discussed below.

For array type τ, we use the notation $(!\ \tau)$ to mean the type of the element selected by subscript out of a value of type τ. The return type of a particular subscript operator is calculated from its domain. Figure 4.6 gives the definition of $(!\ \tau)$. If $(!\ \tau)$ cannot be computed from the rules in Figure 4.6, then $(!\ \tau)$ is undefined.

For a type τ, we cannot simply write $[\tau]$ for array of τ, because $\{\tau\ ;\ \mathit{lf}\}$ is also an array of τ. Furthermore, if τ is a pair type (τ_1, τ_2), then $[(\tau_1, \tau_2)]$ and $([\tau_1], [\tau_2])$ (and more) are also arrays of τ, and so on. Thus we define the relation $\tau_1\ \mathbf{A}\ \tau_2$ for "τ_1 is an array of τ_2." For arrays of $(\mathit{int}, \mathit{int})$,

$$\frac{}{\tau \mathbf{L} \tau}\ \text{L-REFL} \qquad \frac{\tau_1 \mathbf{L} \tau_2}{\tau_2 \mathbf{L} \tau_1}\ \text{L-SYMM} \qquad \frac{\tau_1 \mathbf{L} \tau_2 \qquad \tau_2 \mathbf{L} \tau_3}{\tau_1 \mathbf{L} \tau_3}\ \text{L-TRANS}$$

$$\frac{}{[(\tau_1, \tau_2)] \mathbf{L} ([\tau_1], [\tau_2])}\ \text{L-ZPR}$$

$$\frac{}{\{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\} \mathbf{L} \{\tau\ ;\ \mathit{nd}(\nu)\}}\ \text{L-LF}$$

$$\frac{}{\{\{\tau\ ;\ \nu\}\ ;\ \mathit{nd}(\nu')\} \mathbf{L} \{\{\tau\ ;\ \mathit{nd}(\nu)\}\ ;\ \nu'\}}\ \text{L-ND}$$

$$\frac{\tau_1 \mathbf{L} \tau_2}{[\tau_1] \mathbf{L} [\tau_2]}\ \text{L-PARR} \qquad \frac{\tau_1 \mathbf{L} \tau_2}{[\tau_1] \mathbf{L} \{\tau_2\ ;\ \mathit{lf}\}}\ \text{L-PF} \qquad \frac{\tau_1 \mathbf{L} \tau_2}{\{\tau_1\ ;\ \nu\} \mathbf{L} \{\tau_2\ ;\ \nu\}}\ \text{L-FARR}$$

$$\frac{\tau_1 \mathbf{L} \tau_1' \qquad \tau_2 \mathbf{L} \tau_2'}{(\tau_1, \tau_2) \mathbf{L} (\tau_1', \tau_2')}\ \text{L-PAIR} \qquad \frac{\tau_1 \mathbf{L} \tau_1' \qquad \tau_2 \mathbf{L} \tau_2'}{\tau_1 \to \tau_2 \mathbf{L} \tau_1' \to \tau_2'}\ \text{L-FUN}$$

$$\frac{\tau_1 \mathbf{L} [\tau_2]}{\tau_1 \mathbf{A} \tau_2}\ \text{ARRAY-OF}$$

Figure 4.7: Definition of $\tau_1\ \mathbf{A}\ \tau_2$ and its auxiliary relation $\tau_1\ \mathbf{L}\ \tau_2$.

for example, all of the following are established by $\mathbf{A}$:

$$\begin{aligned}
[(int, int)] &\quad \mathbf{A} \quad (int, int) \\
([int], [int]) &\quad \mathbf{A} \quad (int, int) \\
\{(int, int)\ ;\ \mathit{lf}\} &\quad \mathbf{A} \quad (int, int) \\
(\{int\ ;\ \mathit{lf}\}, \{int\ ;\ \mathit{lf}\}) &\quad \mathbf{A} \quad (int, int)
\end{aligned}$$

These types correspond to arrays of integer pairs and all their equivalent representations under well-formed coercions. The relation $\mathbf{A}$ is defined in Figure 4.7.

The mechanism by which $\mathbf{A}$ is defined is as follows. We define an auxiliary relation $\mathbf{L}$ which establishes that two types are both representations of the same scalar or array type. We use the name $\mathbf{L}$ for the relation because it tells us that one type is "on the same level as" another. The first three rules in $\mathbf{L}$ define it as a reflexive, symmetric, and transitive relation. Rule L-PARR states that if τ_1 and τ_2 are related by $\mathbf{L}$, then wrapping both τ_1 and τ_2 in the array constructor preserves the relation. L-LF tells us that a nested flattened array of τ with inner shape ν and outer shape *lf* is related to a flattened array of the same type with shape $nd(\nu)$. L-ND is a similar judgment for the case when the outer shape is $nd(\nu)$ for some ν. The rest of $\mathbf{L}$'s rules are straightforward recursive judgments along similar lines. Now consider the rule ARRAY-OF. Note that $\mathbf{A}$ only ever uses $\mathbf{L}$ with an array type as its right argument. This is how $\mathbf{A}$ establishes that τ_1 is in fact an array of τ_2, because τ_1 must be a representation of $[\tau_2]$ under $\mathbf{L}$.

All arrays of type τ can be coerced between one another.

Lemma 4.1.2 (coercions within $\mathbf{A}$). *If* τ_1 $\mathbf{A}$ τ*, then* τ_2 $\mathbf{A}$ $\tau \Leftrightarrow \vdash \tau_1 \triangleright \tau_2$ *ok.*

Proof. The proof appears in Appendix A. □

To illustrate how $\mathbf{A}$ is used in the well-formedness rules, we consider the judgment for type-indexed filter:

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to bool}\ ok \qquad \Gamma \vdash {e_2}^{\tau_2}\ ok \qquad \tau_2\ \mathbf{A}\ \tau_1}{\Gamma \vdash (\mathbf{filt}_{(\tau_1,\tau_2)}\ ({e_1}^{\tau_1 \to bool}, {e_2}^{\tau_2}))^{\tau_2}\ ok}\ \text{OK-FILT}$$

The premises state the usual condition on the first argument to filter: that it is a predicate on τ_1. The premises also state that τ_2 $\mathbf{A}$ τ_1, meaning τ_2 is one of the representations of an array of τ_1. The rule states that if the expression e_2 is well-formed at τ_2, the application of filter is altogether well-typed at τ_2. The rules OK-MAP and OK-RED employ $\mathbf{A}$ similarly.

4.1.3 Rewriting Rules

The remaining piece of the Flatland system is its small-step rewriting system for introducing, eliminating, and propagating representation transformations through expressions. We present the system as a relation, using the syntactic form $e^\tau \mapsto \overline{e}^\tau$ to mean "e rewrites to $\overline{e}$." The rules are shown in Figures 4.8, 4.9, and 4.10, each of which we consider in turn.

$$\frac{}{e^\tau \mapsto (\tau \triangleright \tau)\ e^\tau}\ \text{CD-IDI} \qquad \frac{}{(\tau \triangleright \tau)\ e^\tau \mapsto e^\tau}\ \text{CD-IDE}$$

$$\frac{\vdash \tau_1 \triangleright \tau_2\ \mathit{ok} \qquad \vdash \tau_2 \triangleright \tau_3\ \mathit{ok}}{\tau_1 \triangleright \tau_3 \mapsto (\tau_2 \triangleright \tau_3) \circ (\tau_1 \triangleright \tau_2)}\ \text{CD-CI} \qquad \frac{}{(\tau_2 \triangleright \tau_3) \circ (\tau_1 \triangleright \tau_2) \mapsto \tau_1 \triangleright \tau_3}\ \text{CD-CE}$$

$$\frac{}{t_1\ (t_2\ t_3) \mapsto (t_1 \circ t_2)\ t_3}\ \text{CD-CU} \qquad \frac{}{(t_1 \circ t_2)\ t_3 \mapsto t_1\ (t_2\ t_3)}\ \text{CD-CF}$$

$$\frac{}{((\overline{\tau}_1 \triangleright \tau_1)\ t_1, (\overline{\tau}_2 \triangleright \tau_2)\ t_2)^{(\tau_1,\tau_2)} \mapsto (((\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2))\ (t_1, t_2))^{(\tau_1,\tau_2)}}\ \text{CD-PAIR}$$

$$\frac{}{(\pi_1\ ((\overline{\tau}_1 \triangleright \tau_1)\ t_1, t_2))^{\tau_1} \mapsto ((\overline{\tau}_1 \triangleright \tau_1)\ (\pi_1\ (t_1, t_2)))^{\tau_1}}\ \text{CD-FST}$$

$$\frac{}{(\pi_2\ (t_1, (\overline{\tau}_2 \triangleright \tau_2)\ t_2))^{\tau_2} \mapsto ((\overline{\tau}_2 \triangleright \tau_2)\ (\pi_2\ (t_1, t_2)))^{\tau_2}}\ \text{CD-SND}$$

$$\frac{}{(\textbf{if}\ t_1\ \textbf{then}\ (\overline{\tau} \triangleright \tau)\ t_2\ \textbf{else}\ (\overline{\tau} \triangleright \tau)\ t_3)^\tau \mapsto ((\overline{\tau} \triangleright \tau)\ (\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3))^\tau}\ \text{CD-IF}$$

$$\frac{}{(((\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2)\ t_1)\ ((\overline{\tau}_1 \triangleright \tau_1)\ t_2))^{\tau_2} \mapsto ((\overline{\tau}_2 \triangleright \tau_2)\ (t_1\ t_2))^{\tau_2}}\ \text{CD-APP}$$

$$\frac{}{\begin{array}{l}((\overline{\tau}_2 \to \overline{\tau}_3 \triangleright \tau_2 \to \tau_3)\ t_1 \circ (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2)\ t_2)^{\tau_1 \to \tau_3} \\ \quad \mapsto ((\overline{\tau}_1 \to \overline{\tau}_3 \triangleright \tau_1 \to \tau_3)\ (t_1 \circ t_2))^{\tau_1 \to \tau_3}\end{array}}\ \text{CD-COMP}$$

$$\frac{}{[(\overline{\tau} \triangleright \tau)\ t_1,\ \ldots,\ (\overline{\tau} \triangleright \tau)\ t_n]^{[\tau]} \mapsto (([\overline{\tau}] \triangleright [\tau])\ [t_1,\ \ldots,\ t_n]^{[\overline{\tau}]})^{[\tau]}}\ \text{CD-ARR}$$

$$\frac{}{\{(\overline{\tau} \triangleright \tau)\ t_1, \ldots, (\overline{\tau} \triangleright \tau)\ t_n; s\}^{\{\tau\ ;\ \nu\}} \mapsto ((\{\overline{\tau}\ ;\ \nu\} \triangleright \{\tau\ ;\ \nu\})\ \{t_1. \ldots, t_n; s\})^{\{\tau\ ;\ \nu\}}}\ \text{CD-FARR}$$

$$\frac{}{(\textbf{let}\ x = t_1\ \textbf{in}\ (\overline{\tau} \triangleright \tau)\ t_2)^\tau \mapsto ((\overline{\tau} \triangleright \tau)\ (\textbf{let}\ x = t_1\ \textbf{in}\ t_2))^\tau}\ \text{CD-LET}$$

$$\frac{}{(\textbf{fun}\ f\ x^\tau = t_1\ \textbf{in}\ (\overline{\tau} \triangleright \tau)\ t_2)^\tau \mapsto ((\overline{\tau} \triangleright \tau)\ (\textbf{fun}\ f\ x^\tau = t_1\ \textbf{in}\ t_2))^\tau}\ \text{CD-FUN}$$

$$\frac{\vdash \overline{\tau} \triangleright \tau\ \mathit{ok}}{x^\tau \mapsto ((\overline{\tau} \triangleright \tau)\ \overline{x}^{\overline{\tau}})^\tau}\ \text{CD-VAR} \qquad \frac{e^\tau \mapsto \overline{e}^\tau}{C[e^\tau] \mapsto C[\overline{e}^\tau]}\ \text{CD-CTXT}$$

Figure 4.8: Coercion distribution rules.

$$\frac{\vdash \tau \triangleright \overline{\tau}\ ok}{(t_1\ !_{\tau}\ t_2)^{(!\ \tau)} \mapsto ((!\ \overline{\tau} \triangleright !\ \tau)\ (((\tau \triangleright \overline{\tau})\ t_1)\ !_{\overline{\tau}}\ t_2)^{(!\ \overline{\tau})})^{(!\ \tau)}}\ \text{CD-SUB}$$

$$\frac{\overline{\tau}_2\ \mathbf{A}\ \tau_1}{(\mathbf{filt}_{(\tau_1,\tau_2)}\ (t_1, t_2))^{\tau_2} \mapsto ((\overline{\tau}_2 \triangleright \tau_2)\ (\mathbf{filt}_{(\tau_1,\overline{\tau_2})}\ (t_1, (\tau_2 \triangleright \overline{\tau}_2)\ t_2)))^{\tau_2}}\ \text{CD-FILT}$$

$$\frac{\overline{\tau}_3\ \mathbf{A}\ \tau_1 \qquad \overline{\tau}_4\ \mathbf{A}\ \tau_2}{(\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_4)}\ (t_1, t_2))^{\tau_4} \mapsto ((\overline{\tau}_4 \triangleright \tau_4)\ (\mathbf{map}_{(\tau_1,\tau_2,\overline{\tau}_3,\overline{\tau}_4)}\ (t_1, (\tau_3 \triangleright \overline{\tau}_3)\ t_2)))^{\tau_4}}\ \text{CD-MAP}$$

$$\frac{\overline{\tau}_2\ \mathbf{A}\ \tau_1}{(\mathbf{red}_{(\tau_1,\tau_2)}\ (t_1, t_2, t_3))^{\tau_1} \mapsto (\mathbf{red}_{(\tau_1,\overline{\tau}_2)}\ (t_1, t_2, (\tau_2 \triangleright \overline{\tau}_2)\ t_3))^{\tau_1}}\ \text{CD-RED}$$

Figure 4.9: Coercion introductions with type-indexed operators.

Figure 4.8 contains administrative rules whereby coercions can be introduced into programs, unfolded into compositions of coercions, and moved around within expressions. We call these the coercion distribution rules. We consider a few of them for illustration. The rule CD-IDI states that identity coercions can be applied to expressions anywhere; its dual CD-IDE states that identity coercions can be discarded wherever they appear. The rule CD-CI states that coercion $\tau_1 \triangleright \tau_3$ can be rewritten as a composition of two coercions, using τ_2 as an intermediate representation, as long as coercions $\tau_1 \triangleright \tau_2$ and $\tau_2 \triangleright \tau_3$ are well-formed. The rules from CD-PAIR to the bottom of the figure (save the last rule) have to do with hoisting coercions out of subexpressions. The rule CD-IF, for example, states that when the same coercion appears in both branches of a conditional, it may be lifted out of the branches and applied to the modified conditional as a whole. The final rule states that for any context computing on term e^{τ}, if e rewrites to $\overline{e}$, then $\overline{e}$ may be used in the same context.

Figure 4.9 shows rules for introducing coercions at type-indexed operators such that the type indices of the operator may be changed. These rules are critical: they allow us to exchange one type-indexed operator for another in the process of transforming a program. The coercions introduced in the process of those exchanges can be manipulated per the rules is Figure 4.8 and in many

$$\frac{\vdash \tau_1 \rhd \overline{\tau}_1\ \mathit{ok} \qquad \overline{x}\ \text{fresh} \qquad S = [x^{\tau_1}/((\overline{\tau}_1 \rhd \tau_1)\ \overline{x}^{\overline{\tau}_1})^{\tau_1}]}{(\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)^{\tau_0} \mapsto (\mathbf{let}\ \overline{x} = (\tau_1 \rhd \overline{\tau}_1)\ t_1\ \mathbf{in}\ S\ t_2)^{\tau_0}}\ \text{CD-LET-PROP}$$

$$\frac{\begin{array}{c}\vdash \tau_0 \rhd \overline{\tau_0}\ \mathit{ok} \qquad \overline{f}, \overline{x}\ \text{fresh} \\ S = [f^{\tau_0 \to \tau_1}/(\overline{f}^{\overline{\tau}_0 \to \tau_1} \circ (\tau_0 \rhd \overline{\tau}_0))^{\tau_0 \to \tau_1}] \\ S' = S[x^{\tau_0}/((\overline{\tau}_0 \rhd \tau_0)\ \overline{x}^{\overline{\tau}_0})^{\tau_0}]\end{array}}{(\mathbf{fun}\, f\, x^{\tau_0} = t_1\ \mathbf{in}\ t_2)^{\tau_2} \mapsto (\mathbf{fun}\, \overline{f}\, \overline{x}^{\overline{\tau}_0} = S'\ t_1\ \mathbf{in}\ S\ t_2)^{\tau_2}}\ \text{CD-FUN-PROP}$$

Figure 4.10: Coercion propagation rules.

cases eliminated. It is difficult to see their utility by inspecting the rules alone; we will give an illustrative example shortly.

Figure 4.10 gives a final pair of rewriting rules allowing propagation of coercions into the scopes of let bindings and functions. Consider the rule CD-LET-PROP. We determine in the premises that $\vdash \tau_1 \rhd \overline{\tau}_1$ *ok*. Therefore we know we can coerce t_1 to another representation. We apply the coercion to t_1 and bind the result to the fresh variable name $\overline{x}$. We then repair the body of the expression t_2, which expects x of type τ, by substituting the new variable $\overline{x}$ at every use of x, and coercing every occurrence of $\overline{x}$ back to its original type so as to preserve the well-typedness of t_2. The rule for functions is similar: it is essentially a modification of CD-LET-PROP such that similar treatment is given to the function name and the function argument where they are in scope.

Taken together, the rules in Figures 4.8, 4.9, and 4.10 constitute a type-preserving rewrite system over Flatland.

Theorem 4.1.3 ($\mapsto$ preserves types)**.** *If* $\Gamma \vdash e^{\tau}$ *ok and* $e^{\tau} \mapsto^{*} \overline{e}^{\tau}$*, then* $\Gamma \vdash \overline{e}^{\tau}$ *ok.*

Proof. The proof is by induction over the judgments of the $\mapsto$ relation.

The full proof of the theorem appears in Appendix A. □

Consider the contrived case where the whole program under transformation is the expression $[1, 2, 3]^{[int]}$. We cannot perform any useful flattening on this program, since the type of the whole program must be preserved. The effect of this constraint is that original program can only be transformed in an expression that is coerced back and forth from its original representation:

$$\begin{array}{rcl} & & ([1,2,3]^{[int]}) \\ (\text{CD-IDI}) & \mapsto & ([int] \triangleright [int])\ ([1,2,3]^{[int]}) \\ (\text{CD-CI}) & \mapsto & ((\{int\ ;\ lf\} \triangleright [int]) \circ ([int] \triangleright \{int\ ;\ lf\}))\ ([1,2,3]^{[int]}) \end{array}$$

We can simplify the last expression by collapsing its pair of inverse coercions into an identity coercion, via rule CD-CE, then removing the identity coercion, via CD-IDE. We have done nothing more than take a circular path back to the original expression.

Propagation rules enable us to make meaningful representations inside expressions. We transform the following program as an example.

$$\textbf{let}\ ns = [1,2,3]\ \textbf{in}\ (ns\ !_{[int]}\ 0)$$

The step-by-step transformation of this let-expression appears in Figure 4.11. Some abbreviations are needed to prevent the example from becoming impossibly verbose. Therefore, we let

$$\downarrow = [int] \triangleright \{int\ ;\ lf\}$$

and

$$\uparrow = \{int\ ;\ lf\} \triangleright [int]$$

Think of $\downarrow$ as "flatten" and $\uparrow$ as "unflatten." In Figure 4.11, the parallel array $[1,2,3]$ is transformed to its flattened array equivalent $((\downarrow\ [1,2,3])$, which evaluates to $\{1,2,3;\mathbf{lf}(0,3)\})$. Furthermore, by rewriting, we exchange one type-indexed subscript operator for another, thereby eliminating coercion operations. The coercions $\downarrow$ and $\uparrow$ are inverse coercions. Note we have

$$([int] \triangleright \{int\ ;\ lf\}) \circ (\{int\ ;\ lf\} \triangleright [int]) \mapsto \{int\ ;\ lf\} \triangleright \{int\ ;\ lf\}$$

$$\begin{array}{rcl}
 & & \mathbf{let}\ \mathit{ns} = [1,2,3]\ \mathbf{in}\ (\mathit{ns}\ !_{[\mathit{int}]}\ 0) \\
(\text{CD-LET-PROP}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ (\uparrow\ \overline{\mathit{ns}})\ !_{[\mathit{int}]}\ 0 \\
(\text{CD-SUB}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ (\mathit{int} \triangleright \mathit{int})\ ((\downarrow\ (\uparrow\ \overline{\mathit{ns}}))\ !_{\{\mathit{int}\ ;\ \mathit{lf}\}}\ 0) \\
(\text{CD-IDE}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ (\downarrow\ (\uparrow\ \overline{\mathit{ns}}))\ !_{\{\mathit{int}\ ;\ \mathit{lf}\}}\ 0 \\
(\text{CD-CU}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ ((\downarrow \circ \uparrow)\ \overline{\mathit{ns}})\ !_{\{\mathit{int}\ ;\ \mathit{lf}\}}\ 0 \\
(\text{CD-CE}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ ((\{\mathit{int}\ ;\ \mathit{lf}\} \triangleright \{\mathit{int}\ ;\ \mathit{lf}\})\ \overline{\mathit{ns}})\ !_{\{\mathit{int}\ ;\ \mathit{lf}\}}\ 0 \\
(\text{CD-IDE}) & \mapsto & \mathbf{let}\ \overline{\mathit{ns}} =\downarrow\ [1,2,3]\ \mathbf{in}\ (\overline{\mathit{ns}}\ !_{\{\mathit{int}\ ;\ \mathit{lf}\}}\ 0)
\end{array}$$

Figure 4.11: Transforming an expression by $\mapsto$.

which effectively makes the composition of $\downarrow$ and $\uparrow$ mutually annihilating. This property is always true of inverse coercions by specialization of rule CD-CE. Post-transformation, the representation of the array bound to *ns* is coerced exactly once, to the differently-typed fresh variable $\overline{\mathit{ns}}$. We can change the representation at transformation time; it need not be delayed to a later phase. In our PML implementation, we perform representation transformations like this one during compilation, and no runtime coercion takes place.

It is worth noting that in this example, we have selected rewriting rules entirely at our discretion: the system itself provided no guidance as to which steps to follow. The question remains as to how to formalize a strategy for applying Flatland's rewriting rules so as to move towards a desired goal. The following section shows us one way to use the mechanisms of Flatland to implement a particular representation-transformation policy.

4.2 Aggressive Hybrid Flattening

We define *aggressive hybrid flattening* informally as follows. We start with a source program of type τ containing parallel arrays but no flattened arrays. The source program is free of coercions. The transformation traverses the source program and replaces every parallel array value with its *flat* equivalent (for a definition of "flat," given below). All bindings and uses of array-valued expressions are correspondingly transformed. Coercions are folded into type-indexed array operators where they appear. The resulting program contains no parallel arrays, only flattened arrays, and it remains well-typed at its original type τ. The formalization of aggressive hybrid flattening given in this section is the blueprint of the PML implementation discussed in Chapter 5.

We now describe a source language and a target language for aggressive hybrid flattening, each of which is a well-defined subset of the terms in Figure 4.2. The source language models the surface language of PML, and the target language models compiled PML with flattened arrays. The distinction between source language and target language can be drawn according to types. A *flattened-array type* is a type of the form $\{\tau\ ;\ \nu\}$ for any τ and ν (see Figure 4.1). We then define *source type* as follows:

Definition 4.2.1. A *source type* is a type that is neither a flattened-array type, nor contains any flattened-array types. Equivalently, source types are those generated by the grammar

$$\tau ::= g \mid \tau \to \tau \mid (\tau, \tau) \mid [\tau]$$

We define *source programs* as a term e^{τ} for source type τ, all of whose subterms have source types, and containing no coercions anywhere. Thus, we have made it illegal to write down flattened arrays anywhere in a source program. The transformation itself will introduce all flattened-array values and all coercions into the target program.

The restrictions on values one can write down in a source program simplify all well-formedness judgments involving type-indexed array operators. Consider the rule OK-FILT from Figure 4.5.

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to \mathit{bool}}\ \mathit{ok} \qquad \Gamma \vdash {e_2}^{\tau_2}\ \mathit{ok} \qquad \tau_2\ \mathbf{A}\ \tau_1}{\Gamma \vdash (\mathbf{filt}_{(\tau_1,\tau_2)}\ ({e_1}^{\tau_1 \to \mathit{bool}}, {e_2}^{\tau_2}))^{\tau_2}\ \mathit{ok}}\ \textsc{ok-filt}$$

The general rule accommodates applying a predicate on type τ_1 to any array, or any coercion thereof, whose elements are of type τ_1; this is captured by our definition of $\mathbf{A}$. In the context of source programs, we no longer need the filter rule to apply to whole families of array types

$$\frac{\Gamma \vdash {e_1}^{[\tau]}\ \mathit{ok} \qquad \Gamma \vdash {e_2}^{\mathit{int}}\ \mathit{ok}}{\Gamma \vdash ({e_1}^{[\tau]} \,!_{[\tau]}\, {e_2}^{\mathit{int}})^{\tau}\ \mathit{ok}}\ \textsc{ok-sub-s}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to \tau_2}\ \mathit{ok} \qquad \Gamma \vdash {e_2}^{[\tau_1]}\ \mathit{ok}}{\Gamma \vdash (\mathbf{map}_{(\tau_1,\tau_2,[\tau_1],[\tau_2])}\ ({e_1}^{\tau_1 \to \tau_2}, {e_2}^{[\tau_1]}))^{[\tau_2]}\ \mathit{ok}}\ \textsc{ok-map-s}$$

$$\frac{\Gamma \vdash {e_1}^{\tau_1 \to \mathit{bool}}\ \mathit{ok} \qquad \Gamma \vdash {e_2}^{[\tau_1]}\ \mathit{ok}}{\Gamma \vdash (\mathbf{filt}_{(\tau_1,[\tau_1])}\ ({e_1}^{\tau_1 \to \mathit{bool}}, {e_2}^{[\tau_1]}))^{[\tau_1]}\ \mathit{ok}}\ \textsc{ok-filt-s}$$

$$\frac{\Gamma \vdash {e_1}^{(\tau_1,\tau_1) \to \tau_1}\ \mathit{ok} \qquad \Gamma \vdash {e_2}^{\tau_1}\ \mathit{ok} \qquad \Gamma \vdash {e_3}^{[\tau_1]}\ \mathit{ok}}{\Gamma \vdash (\mathbf{red}_{(\tau_1,[\tau_1])}\ ({e_1}^{(\tau_1,\tau_1) \to \tau_1}, {e_2}^{\tau_1}, {e_3}^{[\tau_1]}))^{\tau_1}\ \mathit{ok}}\ \textsc{ok-red-s}$$

Figure 4.12: Modified well-formedness judgments for source-language type-indexed operator forms.

at a time, because source programs only ever contain the array type $[\tau]$ for any given τ. We can simplify OK-FILT for source programs such that its second argument is restricted to be a plain-vanilla parallel array; we do so in the modified rule OK-FILT-S in Figure 4.12. In the same figure, the rules for array subscript, map, and reduce are modified along these lines. These typing rules also forbid array operators from treating pairs of arrays as arrays. This way, only transformation steps can introduce unzipped array representations.

The target language of aggressive hybrid flattening is defined in terms of *flat types*.

Definition 4.2.2. A type τ is *flat* if

- it is a ground type g,
- it is a function type $\tau_1 \to \tau_2$ and τ_1 and τ_2 are flat,
- it is a pair type (τ_1, τ_2) and τ_1 and τ_2 are flat, or
- it is an array type $\{\tau\ ;\ \nu\}$ and τ is a ground type or a flat function type.

$$
\begin{aligned}
\mathbf{F}[\![g]\!] &= g \\
\mathbf{F}[\![[g]]\!] &= \{g \;;\; \mathit{lf}\} \\
\mathbf{F}[\![\tau_1 \to \tau_2]\!] &= \mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!] \\
\mathbf{F}[\![(\tau_1, \tau_2)]\!] &= (\mathbf{F}[\![\tau_1]\!], \mathbf{F}[\![\tau_2]\!]) \\
\mathbf{F}[\![[\tau_1 \to \tau_2]]\!] &= \{\mathbf{F}[\![\tau_1 \to \tau_2]\!] \;;\; \mathit{lf}\} \\
\mathbf{F}[\![[(\tau_1, \tau_2)]]\!] &= (\mathbf{F}[\![[\tau_1]]\!], \mathbf{F}[\![[\tau_2]]\!]) \\
\mathbf{F}[\![[[\tau]]]\!] &= \mathbf{N}[\![(\mathbf{F}[\![[\tau]]\!])]\!] \\
\\
\mathbf{N}[\![(\tau_1, \tau_2)]\!] &= (\mathbf{N}[\![\tau_1]\!], \mathbf{N}[\![\tau_2]\!]) \\
\mathbf{N}[\![\{\tau \;;\; \nu\}]\!] &= \{\tau \;;\; \mathit{nd}(\nu)\}
\end{aligned}
$$

Figure 4.13: Aggressive type flattening.

If a type is not flat, we say it is *nonflat*. Note that source types and flat types do not partition the space of Flatland types: that is, source types and nonflat types are not the same. For example, $[(\mathit{int}, \mathit{bool})]$ is a source type, and the related type $(\{\mathit{int} \;;\; \mathit{lf}\}, \{\mathit{bool} \;;\; \mathit{lf}\})$ is a flat type. But the type $\{(\mathit{int}, \mathit{bool}) \;;\; \mathit{lf}\}$ is neither a a source type nor a flat type; it is disqualified as a source type since it is a flattened-array type and disqualified as a flat type since it includes a pair inside an array.

A *target program* is an expression whose outermost type is a source type, yet all of whose subexpressions have flat types. The restriction on its outermost type is a necessary consequence of the type-preservation property of rewriting in Flatland. The restriction is enforced by the application of one last "unflattening" coercion to the transformed program at the top level. Within the program, all subexpressions are flattened.

The function $\mathbf{F}[\![\tau]\!]$ in Figure 4.13 defines a mapping from source types to flat types.

Lemma 4.2.1 (**F** maps source types to flat types)**.** *For τ, a source type, $\mathbf{F}[\![\tau]\!]$ is flat.*

Proof. The proof is by induction over the structure of source types.

The full proof appears in Appendix A. □

$$\frac{\{\} \vdash e^{\tau} \searrow (\overline{\tau} \rhd \tau) \diamond \overline{e}^{\overline{\tau}}}{e^{\tau} \Downarrow ((\overline{\tau} \rhd \tau)\ \overline{e}^{\overline{\tau}})^{\tau}}\ \textsc{flatten}$$

Figure 4.14: Top-level aggressive hybrid flattening.

We use **F** to calculate target types during aggressive flattening: it guides our choices in inserting type coercions.

The following lemma states that type coercions suggested by **F** are all always well-formed.

Lemma 4.2.2 (**F**'s coercions are well-formed)**.** *If* $\mathbf{F}[\![\tau]\!] = \overline{\tau}$*, then* $\vdash \tau \rhd \overline{\tau}$ *ok.*

Proof. The proof appears in Appendix A. □

Flattening of whole programs is written as a type-preserving relation $\Downarrow$. Figure 4.14 gives the sole judgment for $\Downarrow$, which immediately delegates its work to an auxiliary relation $\searrow$. $\Downarrow$ and $\searrow$ are formulated as a big-step rewriting semantics. Whole-program flattening consists transforming a program e^{τ} (of source type τ) to another program $\overline{e}^{\overline{\tau}}$ (of flat type $\overline{\tau}$) and then coercing the transformed program back to τ at the top level. Note that in the cases where τ is, for example, a ground type, the outermost coercion is an identity coercion and has no effect. (This situation is suggested by the example in Figure 4.11.)

The auxiliary relation of the aggressive flattening transformation is given in Figures 4.15 and 4.16. The syntax of $\searrow$ is as follows:

$$\Delta \vdash e^{\tau} \searrow (\overline{\tau} \rhd \tau) \diamond \overline{e}^{\overline{\tau}}$$

Δ is a finite map from variable terms to variable terms; it is used to implement propagations through let-expressions and functions, as in CD-LET-PROP and CD-FUN-PROP above. On the right-hand side of the relation, a diamond ($\diamond$) is used to construct a pair out of a coercion and a transformed expression. The relation produces an unflattening coercion, along with the transformed expression, for use in one of the following ways. If the expression transformed is the whole program, the $\Downarrow$ relation applies the coercion to preserve the program's original type (as per FLATTEN in Figure 4.14). If the expression is not the whole program, the accompanying coercion is used as

$$\frac{}{\Delta \vdash b^{\tau} \searrow (\tau \triangleright \tau) \diamond b^{\tau}}\ \textsc{B-BASE}$$

$$\frac{\Delta(x^{\tau}) = \overline{x}^{\overline{\tau}}}{\Delta \vdash x^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond \overline{x}^{\overline{\tau}}}\ \textsc{B-VAR}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{bool} \searrow (bool \triangleright bool) \diamond {\overline{e}_1}^{bool} \\ \Delta \vdash {e_2}^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond {\overline{e}_2}^{\overline{\tau}} \\ \Delta \vdash {e_2}^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond {\overline{e}_2}^{\overline{\tau}}\end{array}}{\Delta \vdash (\mathbf{if}\ {e_1}^{bool}\ \mathbf{then}\ {e_2}^{\tau}\ \mathbf{else}\ {e_3}^{\tau})^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond (\mathbf{if}\ {\overline{e}_1}^{bool}\ \mathbf{then}\ {\overline{e}_2}^{\overline{\tau}}\ \mathbf{else}\ {\overline{e}_3}^{\overline{\tau}})^{\overline{\tau}}}\ \textsc{B-IF}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_1}^{\overline{\tau}_1} \\ \Delta' = \Delta[x^{\tau_1} \mapsto \overline{x}^{\overline{\tau}_1}],\ \overline{x}\ \text{fresh} \\ \Delta' \vdash {e_2}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_2}\end{array}}{\begin{array}{l}\Delta \vdash (\mathbf{let}\ x = {e_1}^{\tau_1}\ \mathbf{in}\ {e_2}^{\tau_2})^{\tau_2} \\ \quad \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond (\mathbf{let}\ \overline{x} = {\overline{e}_1}^{\overline{\tau}_1}\ \mathbf{in}\ {\overline{e}_2}^{\overline{\tau}_2})^{\overline{\tau}_2}\end{array}}\ \textsc{B-LET}$$

$$\frac{\begin{array}{c}\Delta' = \Delta[f^{\tau_0 \rightarrow \tau_1} \mapsto \bar{f}^{\overline{\tau}_0 \rightarrow \overline{\tau}_1}],\ \bar{f}\ \text{fresh} \\ \Delta'' = \Delta'[x^{\tau_0} \mapsto \overline{x}^{\overline{\tau}_0}],\ \overline{x}\ \text{fresh} \\ \Delta'' \vdash {e_1}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_1}^{\overline{\tau}_1} \\ \Delta' \vdash {e_2}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_2}\end{array}}{\Delta \vdash (\mathbf{fun}\ f\ x^{\tau_0} = {e_1}^{\tau_1}\ \mathbf{in}\ {e_2}^{\tau_2})^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond (\mathbf{fun}\,\bar{f}\,\overline{x}^{\overline{\tau}_0} = {\overline{e}_1}^{\overline{\tau}_1}\ \mathbf{in}\ {\overline{e}_2}^{\overline{\tau}_2})^{\overline{\tau}_2}}\ \textsc{B-FUN}$$

$$\frac{\Delta \vdash {e_1}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_1}^{\overline{\tau}_1} \qquad \Delta \vdash {e_2}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_2}}{\Delta \vdash ({e_1}^{\tau_1}, {e_2}^{\tau_2})^{(\tau_1,\tau_2)} \searrow ((\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2)) \diamond ({\overline{e}_1}^{\overline{\tau}_1}, {\overline{e}_2}^{\overline{\tau}_2})^{(\overline{\tau}_1,\overline{\tau}_2)}}\ \textsc{B-PAIR}$$

$$\frac{\Delta \vdash e^{(\tau_1,\tau_2)} \searrow ((\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2)) \diamond \overline{e}^{(\overline{\tau}_1,\overline{\tau}_2)}}{\Delta \vdash (\pi_1\ e^{(\tau_1,\tau_2)})^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond (\pi_1\ \overline{e}^{(\overline{\tau}_1,\overline{\tau}_2)})^{\overline{\tau}_1}}\ \textsc{B-FST} \qquad (\textsc{B-SND}\ \text{sim.})$$

Figure 4.15: Aggressive hybrid flattening, group 1.

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_1\to\tau_2} \searrow (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2) \diamond {\overline{e}_1}^{\overline{\tau}_1\to\overline{\tau}_2} \\ \Delta \vdash {e_2}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_2}^{\overline{\tau}_1}\end{array}}{\Delta \vdash ({e_1}^{\tau_1\to\tau_2}\ {e_2}^{\tau_1})^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond ({\overline{e}_1}^{\overline{\tau}_1\to\overline{\tau}_2}\ {\overline{e}_2}^{\overline{\tau}_1})^{\overline{\tau}_2}}\ \text{B-APP}$$

$$\frac{\mathbf{F}[\![[\tau]]\!] = \overline{\tau}}{\Delta \vdash e^{[\tau]} \searrow (\overline{\tau} \triangleright [\tau]) \diamond (([\tau] \triangleright \overline{\tau})\ e^{[\tau]})^{\overline{\tau}}}\ \text{B-ARR}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_2\to\tau_3} \searrow (\overline{\tau}_2 \to \overline{\tau}_3 \triangleright \tau_2 \to \tau_3) \diamond {\overline{e}_1}^{\overline{\tau}_2\to\overline{\tau}_3} \\ \Delta \vdash {e_2}^{\tau_1\to\tau_2} \searrow (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_1\to\overline{\tau}_2}\end{array}}{\Delta \vdash (e_1 \circ e_2)^{\tau_1\to\tau_3} \searrow (\overline{\tau}_1 \to \overline{\tau}_3 \triangleright \tau_1 \to \tau_3) \diamond (\overline{e}_1 \circ \overline{e}_2)^{\overline{\tau}_1\to\overline{\tau}_3}}\ \text{B-COMP}$$

$$\frac{\Delta \vdash {e_1}^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond {\overline{e}_1}^{\overline{\tau}} \qquad \Delta \vdash {e_2}^{\mathit{int}} \searrow (\mathit{int} \triangleright \mathit{int}) \diamond {\overline{e}_2}^{\mathit{int}}}{\Delta \vdash ({e_1}^{\tau}\ !_{\tau}\ {e_2}^{\mathit{int}})^{(!\ \tau)} \searrow ((!\ \overline{\tau}) \triangleright (!\ \tau)) \diamond ({\overline{e}_1}^{\overline{\tau}}\ !_{\overline{\tau}}\ {\overline{e}_2}^{\mathit{int}})^{(!\ \overline{\tau})}}\ \text{B-SUB}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_1\to\tau_2} \searrow (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2) \diamond {\overline{e}_1}^{\overline{\tau}_1\to\overline{\tau}_2} \\ \Delta \vdash {e_2}^{\tau_3} \searrow (\overline{\tau}_3 \triangleright \tau_3) \diamond {\overline{e}_2}^{\overline{\tau}_3} \\ \mathbf{F}[\![\tau_4]\!] = \overline{\tau}_4\end{array}}{\begin{array}{l}\Delta \vdash (\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_4)}\ ({e_1}^{\tau_1\to\tau_2}, {e_2}^{\tau_3}))^{\tau_4} \\ \quad \searrow (\overline{\tau}_4 \triangleright \tau_4) \diamond (\mathbf{map}_{(\overline{\tau}_1,\overline{\tau}_2,\overline{\tau}_3,\overline{\tau}_4)}\ ({\overline{e}_1}^{\overline{\tau}_1\to\overline{\tau}_2}, {\overline{e}_2}^{\overline{\tau}_3}))^{\overline{\tau}_4}\end{array}}\ \text{B-MAP}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_1\to \mathit{bool}} \searrow (\overline{\tau}_1 \to \mathit{bool} \triangleright \tau_1 \to \mathit{bool}) \diamond {\overline{e}_1}^{\overline{\tau}_1\to \mathit{bool}} \\ \Delta \vdash {e_2}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_2}\end{array}}{\begin{array}{l}\Delta \vdash (\mathbf{filt}_{(\tau_1,\tau_2)}\ ({e_1}^{\tau_1\to \mathit{bool}}, {e_2}^{\tau_2}))^{\tau_2} \\ \quad \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond (\mathbf{filt}_{(\overline{\tau}_1,\overline{\tau}_2)}\ ({\overline{e}_1}^{\overline{\tau}_1\to \mathit{bool}}, {\overline{e}_2}^{\overline{\tau}_2}))^{\overline{\tau}_2}\end{array}}\ \text{B-FILT}$$

$$\frac{\begin{array}{c}\Delta \vdash {e_1}^{(\tau_1,\tau_1)\to\tau_1} \searrow ((\overline{\tau}_1, \overline{\tau}_1) \to \overline{\tau}_1 \triangleright (\tau_1, \tau_1) \to \tau_1) \diamond {\overline{e}_1}^{(\overline{\tau}_1,\overline{\tau}_1)\to\overline{\tau}_1} \\ \Delta \vdash {e_2}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_2}^{\overline{\tau}_1} \\ \Delta \vdash {e_3}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond {\overline{e}_3}^{\overline{\tau}_2}\end{array}}{\begin{array}{l}\Delta \vdash (\mathbf{red}_{(\tau_1,\tau_2)}\ ({e_1}^{(\tau_1,\tau_1)\to\tau_1}, {e_2}^{\tau_1}, {e_3}^{\tau_2}))^{\tau_2} \\ \quad \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond (\mathbf{red}_{(\overline{\tau}_1,\overline{\tau}_2)}\ ({\overline{e}_1}^{(\overline{\tau}_1,\overline{\tau}_1)\to\overline{\tau}_1}, {\overline{e}_2}^{\overline{\tau}_1}, {\overline{e}_3}^{\overline{\tau}_2}))^{\overline{\tau}_2}\end{array}}\ \text{B-RED}$$

Figure 4.16: Aggressive hybrid flattening, group 2.

a building block for further coercions as program transformation proceeds outward. See B-IF for an example: the identical coercions that float out of recursive transformations of the two branches of the conditional are used as the coercion returned with the transformed conditional as a whole.

Most of the action in aggressive flattening takes place at array terms (see B-ARR). This is the rule where coercions — from parallel arrays to flattened arrays — are introduced. The rule B-VAR substitutes typed variables for their flattened replacements per the map carried by Δ. The array-operator rules (B-SUB, B-MAP, B-FILT, and B-RED) exchange operators indexed by source type to operators indexed by the corresponding flat types. The other rules are simply recursive administrative rules propagating transformations through expressions.

The big-step semantics is syntax-directed: there is exactly one rule to apply for every distinct syntactic form, yielding exactly one deterministic result. As such it describes not only a semantic specification but also an algorithm.

Finally we prove that the big-step rewriting relation $\Downarrow$ can be encoded in the small-step rewriting relation $\mapsto$.

Theorem 4.2.3 ($\mapsto$ encodes $\Downarrow$). *If* $e^{\tau} \Downarrow ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$*, then* $e^{\tau} \mapsto^{*} ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$.

Proof. The proof is by induction over the judgments in $\searrow$.

The full proof appears in Appendix A. □

The next two chapters discuss the use of the aggressive hybrid flattening system presented here to construct a compiler transformation and demonstrate its success in improving parallel performance for nested-data-parallel programs on multicore machines.

4.3 Related Work

Our system bears some resemblance to Leroy's system for boxing and unboxing of objects [32], which also includes mechanisms for pushing coercions through expressions while preserving types.

Leroy is not concerned with array flattening: his system is concerned with arity raising and avoiding boxing primitive values when they can be stored as raw data. The present work bears a closer relationship to the partial vectorization of Chakravarty *et al.* [18]. In the partial vectorization framework, flat terms and nonflat terms coexist, as in Flatland, but there are key distinctions between their system and ours. First, their surface program may contain no flattened terms; flattening is performed by the compiler, and there is no way to write down a flattened structure in a surface program in their system. Second, their strategy identifies which terms must be flattened and which terms must not be flattened, but the distinction is binary. There is no notion of flattening up to a point, or of a partially-flattened data structure. When flattening occurs, it is complete. It is possible for a particular value, in their system, to be used in two contexts (possibly more) where in one case the context demands a flat representation and in another a nonflat one. In such circumstances, the system introduces type coercions so that a flat value is coerced to a nonflat equivalent where necessary, and vice versa. The partial vectorization research grew out of the desire to integrate nested data parallelism into the GHC implementation of Haskell. Many of GHC's operators (its I/O routines, for example) are not implemented to cope with flattened structures. To integrate nested data-parallel programming into the infrastructure of GHC, a mechanism was needed to allow values to cross the boundary between the two spaces. Partial vectorization is their solution to this technical problem.

CHAPTER 5
IMPLEMENTATION

In this chapter, we describe the realization of aggressive hybrid flattening, as formalized in Chapter 4, in the PML compiler, `pmlc`. The flattening transformation is implemented as an optional AST-to-AST pass during compilation; it is activated by a command-line control when the compiler is invoked.

We begin the chapter with a comparison of Flatland and PML. We then describe some of the relevant details of the PML compiler. We then discuss optimizations we can apply as a result of having flattened a program.

5.1 Flatland vs. PML

First we consider PML in relation to the Flatland model. PML is a general-purpose programming language with a full assortment of modern functional programming features. It supports algebraic datatypes, ML-style type inference, pattern matching with exhaustiveness checking, and a variety of explicitly- and implicitly-threaded constructs for parallel computation. Flatland, by contrast, is minimal, containing only parallel arrays and four operators for computing with parallel arrays: subscript, map, filter and reduce. Nevertheless, the Flatland system has proven to be a useful tool for developing transformations in `pmlc` in practice. We use Flatland to reason about PML's parallel comprehensions by viewing them as syntactic abstractions for calls to `map` and `filter`, and design transformation for the simpler (Flatland-like) language. (This is similar to the approach taken by Data Parallel Haskell, which performs desugaring of comprehensions before any of its flattening steps [39].) The rules in Flatland's rewriting system (the $\mapsto$ relation in Chapter 4) have been directly applicable in implementing aggressive hybrid flattening in `pmlc`.

In the type systems, the main difference between Flatland and PML is Flatland's lack of support for parametric polymorphism. Compilation of parametrically-polymorphic functions is chal-

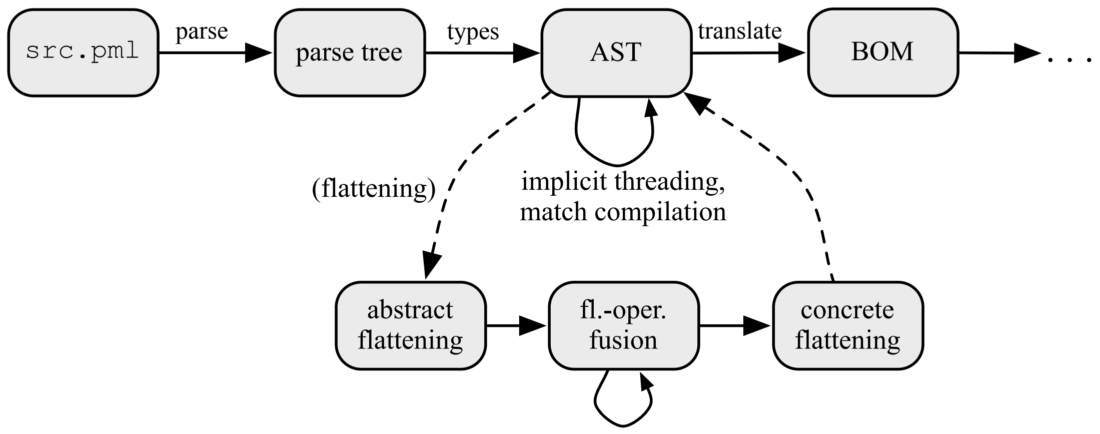


Figure 5.1: `pmlc` with optional flattening.

lenging when designing a representation transformation. Parametrically-polymorphic functions demand uniform representations, while flattening is directed toward building customized representations for faster processing. Therefore there is an inherent tension between the two representation styles. Moving forward, we see several ways to account for polymorphism: either by designing new formal mechanisms in the semantic model, or by monomorphizing the program before applying any flattening steps. In the latter approach, we could follow the example of MLton [35], which resolves all polymorphism at an intermediate compilation stage (between an abstract syntax tree and a normalized intermediate representation), and perform flattening as given in the present work on the monomorphic program.

5.2 `pmlc`: the PML Compiler

The PML compiler, `pmlc`, successively transforms programs into a standard sequence of intermediate representations. Figure 5.1 depicts the relevant phases of PML compilation. Without flattening, PML compilation proceeds as follows. The source file is parsed (if the input program is syntactically well-formed) into an untyped parse tree. The parse tree is then passed to a type-checker which either produces a typed abstract syntax tree (AST) or fails in the case of an ill-typed program. Before translation into the next representation, the typed AST is subject to a series of transformations. First, its high-level implicitly-threaded expressions—namely, parallel tuples, parallel bindings, parallel cases, parallel arrays, and parallel comprehensions—are rewritten into various expression forms closer to the core language. Parallel comprehensions, in particular, are rewritten as maps, filters, and tabulations. Next, pattern matches are checked for completeness and simplified, whereby complex pattern matches are compiled away. Then the AST is translated into a lower-level normalized typed language, BOM. From that point on, the compiler rewrites the program per continuation-passing style (CPS), from which a control-flow graph (CFG) is built, and so on to code generation.

When flattening is enabled, compilation proceeds, in the AST-to-AST transformation phase, in the following modified way. Parallel arrays last past the implicit-thread rewriting phase; they are not rewritten and persist as their own `PArray` form of AST node. Flattening occurs after pattern-match compilation. The actions of the transformation are organized into phases. The first phase is *abstract flattening*. In the abstract flattening phase, every parallel array type is transformed to to an *abstract flattened array type*, and every parallel array value to an *abstract flattened array value*. These are abstract in the sense that their implementations remain unspecified. Flattened-type choices are guided by **F**'s mapping from nonflat types to flat types given in Figure 4.13. We call this phase abstract since the compiler does not, at this point, commit to any particular concrete flattened-array representations: the flattened-array types and coercions in this phase are symbolic. The second phase is an optimizing *flatten-operator fusion* phase, in which identity coercions are discarded and pairs of cancelling coercions are eliminated until no further fusion is possible. The third and last phase is *concrete flattening*. During this phase, the compiler commits to a particular concrete representation of flattened arrays, and abstract coercions are replaced by concrete implementations corresponding to the choice of representation. The phases of flattening in `pmlc` are described in Section 5.3.

We now characterize two critical elements of the machinery of `pmlc`. First we discuss how `pmlc` computes with ropes in parallel, since efficient parallel rope processing underlies all the results reported in the following chapter. We then explain our rope-customized implementation of the segmented sum operation, which, by virtue of being fundamental to many common nested-data-parallel algorithms, is an important operation to support well for nested-data-parallel systems.

5.2.1 Parallel Processing of Ropes

Even without flattening, PML achieves excellent speedups on multicore machines, due to a successful strategy combining parallel computations over rope data structures with work-stealing scheduling and lazy tree splitting (see Section 3.1). Fast parallel processing of ropes is crucial to the performance results given in this dissertation. This section presents a description of the internal workings of parallel rope processing in PML.

PML provides fast parallel maps, filters, and reductions over rope data structures, and constructs ropes in parallel with tabulations. We focus this discussion on building ropes in parallel by tabulation. A detailed account of how PML implements the wider spectrum of parallel rope computations is given in other work [5].

The PML basis library contains an assortment of routines for parallel tabulation. One of these is `tabFromTo`. Its three arguments are a lower and upper bound, both integers, and a function of type `int -> t` (for some type `t`) mapping integer to values of type `t`. The function builds a rope whose elements are the results of having applied the function arguments to all integers in the interval between the (inclusive) bounds.

Figure 5.2 presents the code for `Rope.tabFromTo`. The rope datatype used here is the one presented in Figure 3.2. `maxLeafSize` is the upper bound on the length of sequences at a rope's leaves. Its value is global to the program and fixed at a default value of 512 (this default can be overridden by a command-line switch). In a given call to `tabFromTo`, either the whole interval can be packed into a single `Rope.Leaf`, or its construction must be recursively subdivided into a pair of ropes, in which case the pair is joined together under a `Rope.Cat` node. The code is

```
fun tabFromTo (lo, hi, f) = let
  fun t (lo, hi) =
    if (lo > hi) then Rope.Leaf (Seq.empty)
    else let
      val n = hi-lo+1
      in
        if (n <= maxLeafSize) then
          Rope.Leaf (Seq.tabulate (n, fn i => f (lo+i)))
        else let
          val m = (hi+lo) div 2
          in
            Rope.Cat (| t (lo, m), t (m+1, hi) |)
          end
      end
  in
    t (lo, hi)
  end
```

Figure 5.2: `Rope.tabFromTo`

nearly identical to the code one would write in SML. With respect to parallelism, all the action is in the argument to the constructor `Rope.Cat`, which is written as a parallel tuple. The parallel annotations divide the work into two roughly equal parallel parts, and this parallel subdivision occurs recursively all the way down to the leaves. At each leaf, sequences are tabulated sequentially.

The plot in Figure 5.3 shows the scaling behavior of `tabFromTo` in a particular case. Two versions of `tabFromTo` are tested here: one with a parallel tuple in the argument to `Rope.Cat`, and one with a sequential tuple in the same place. The numbers gathered for this plot are from tabulating from 0 to 999999, computing the Fibonacci number of the position mod 20 at each location. To compute Fibonacci, we use the naïve exponential functional implementation. The plot shows that the parallel annotations in `tabFromTo` enable it to achieve scaling performance that is close to ideal.

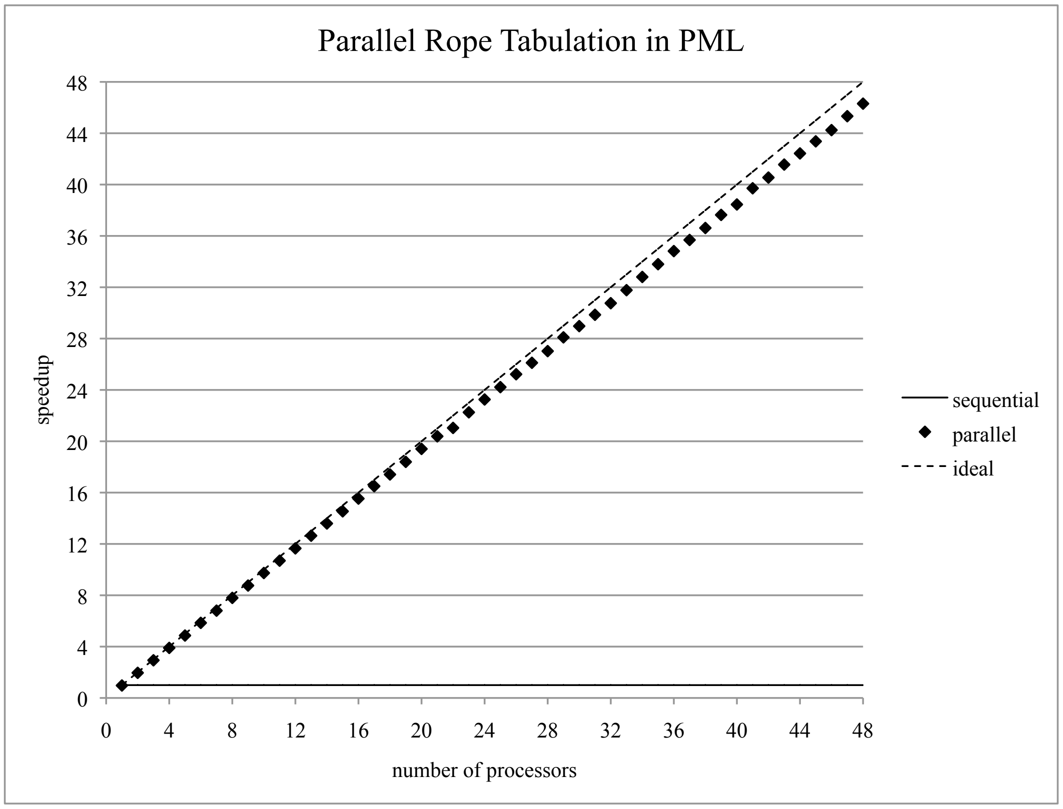


Figure 5.3: `Rope.tabFromTo` with and without parallelism.

5.2.2 Segmented Sum

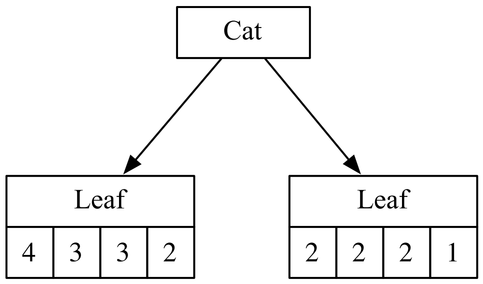
The success of NESL was due in part to its efficient implementations of segmented operations, including segmented sum. Segmented sum plays an important role in common benchmarks such as sparse-matrix vector multiplication (see Chapter 6). Segmented sum consumes an array of arrays of numbers (integers, in our example below), and produces an array of numbers consisting of the sums of all subarrays. NESL computes segmented sum with vector instructions; the implementation is based on a parallel +-scan over a vector of scalars and a head-flags segment descriptor [6]. In contrast to NESL's, PML's implementation of segmented sum consists of a linear transformation of a shape tree followed by a parallel rope reduction. We give an algorithm in this section that computes fast parallel segmented sums over flattened arrays that is insensitive to variation in the segment lengths of the argument.

The straightforward implementation of segmented sum is simply `PArray.map sum`, but, the more variation there is in the lengths of the subarrays, the more such an implementation will be subject to the usual challenges of irregularity. We now explain our alternative solution to this problem. Let `segs` be an array of arrays of integers as follows:

```
val segs = [| [|4|], [|3,3|], [|2,2,2,2|], [|1|] |]
```

The segmented sum of `segs` is `[|4,6,8,1|]`.

Per standard PML-style flattening, we transform `segs` to a pair containing a rope and a shape tree. We assume, for purposes of illustration, that ropes have leaf size 4; in practice they are never so small. In that case flattened `segs` is represented by the rope



and the shape tree

```
Shape.Nd [Shape.Lf(0,1), Shape.Lf(1,3), Shape.Lf(3,7), Shape.Lf(7,8)]
```

We proceed by dividing the information in the shape tree into leaf-size pieces, where each piece is a list of pairs indicating which segments have elements at each leaf, and how many elements belong to each segment. For example, the leaf at left contains 1, then 2, then 1 element of segments 0, 1, and 2 of `segs`, respectively. By a linear traversal of the shape tree, we can extract the following information about the segmentation for the four elements in the the left leaf: `[(0,1),(1,2),(2,1)]`. For the leaf on the right, we compute `[(2,3),(3,1)]`.

With these lists of pairs in hand, we compute segmented sums at each leaf independently. We gather partial totals at each leaf by segment, and pair each partial total with its segment index. For the leaf at left, we compute `[(0,4),(1,6),(2,2)]`, and at right we compute `[(2,6),(3,1)]`. The lists are then joined together such that like-indexed partial totals are combined, giving `[(0,4),(1,6),(2,8),(3,1)]`. Finally, after all partial sums have been joined, the segmented sums are given by discarding the segment indices from this result.

This algorithm requires a linear traversal of the shape tree at the outset, but once that has completed, the rest of the computation can be performed as an n-way fork-join over sequential computations at the leaves. Ropes are balanced, so the parallel reduction of this operation presents no particular scheduling challenges, but our lazy tree splitting scheduler makes adjustments in case of poor load balancing. Note that joining the partial totals at the leaves only ever entails one comparison, as it is only the last pair on the left and the first pair on the right that might share an index. Furthermore, partial totals may be joined in the same way at successive `Cat` nodes going up the tree, so the reduction acts just like a parallel sum. This segmented sum implementation is demonstrably better than simply mapping `sum` over an array of arrays (see Section 6.4), but it demands that the data structure be flattened before it can be employed.

5.3 Flattening in Phases

As mentioned above, flattening in `pmlc` is accomplished in three successive phases: abstract flattening, flattening-operator fusion, and concrete flattening. This section describes the operations of these phases in more detail.

5.3.1 Abstract Flattening

Chapter 4 presents coercions only symbolically, suggesting no implementation for any particular operator. Figure 5.4 gives abstract implementations for coercions. In the figure, we use the usual infix circle notation for standard function composition, and an alternate notation (infix $\otimes$) for

$$\begin{array}{rcl}
[\tau] \triangleright \{\tau\ ;\ \mathit{lf}\} & = & \mathbf{toFA}_{[\tau]} \\
\{\tau\ ;\ \mathit{lf}\} \triangleright [\tau] & = & \mathbf{fromFA}_{\{\tau\ ;\ \nu\}} \\
([\tau_1], [\tau_2]) \triangleright [(\tau_1, \tau_2)] & = & \mathbf{zip}_{([\tau_1],[\tau_2])} \\
[(\tau_1, \tau_2)] \triangleright ([\tau_1], [\tau_2]) & = & \mathbf{unzip}_{([\tau_1],[\tau_2])} \\
(\{\tau_1\ ;\ \nu\}, \{\tau_2\ ;\ \nu\}) \triangleright \{(\tau_1, \tau_2)\ ;\ \nu\} & = & \mathbf{zip}_{(\{\tau_1\ ;\ \nu\},\{\tau_2\ ;\ \nu\})} \\
\{(\tau_1, \tau_2)\ ;\ \nu\} \triangleright (\{\tau_1\ ;\ \nu\}, \{\tau_2\ ;\ \nu\}) & = & \mathbf{unzip}_{\{(\tau_1,\tau_2)\ ;\ \nu\}} \\
\{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\} \triangleright \{\tau\ ;\ \mathit{nd}(\nu)\} & = & \mathbf{cat}_{\{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\}} \\
\{\tau\ ;\ \mathit{nd}(\nu)\} \triangleright \{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\} & = & \mathbf{seg}_{\{\tau\ ;\ \nu\}} \\
\{\{\tau\ ;\ \mathit{nd}(\nu_1)\}\ ;\ \nu_2\} \triangleright \{\{\tau\ ;\ \nu_1\}\ ;\ \mathit{nd}(\nu_2)\} & = & \mathbf{cat}_{\{\{\tau\ ;\ \nu_1\}\ ;\ \mathit{nd}(\nu_2)\}} \\
\{\{\tau\ ;\ \nu_1\}\ ;\ \mathit{nd}(\nu_2)\} \triangleright \{\{\tau\ ;\ \mathit{nd}(\nu_1)\}\ ;\ \nu_2\} & = & \mathbf{seg}_{\{\{\tau\ ;\ \mathit{nd}(\nu_1)\}\ ;\ \nu_2\}} \\
[\tau] \triangleright [\overline{\tau}] & = & \mathbf{map}_{[\tau]}\ (\tau \triangleright \overline{\tau}) \\
\{\tau\ ;\ \nu\} \triangleright \{\overline{\tau}\ ;\ \nu\} & = & \mathbf{map}_{\{\tau\ ;\ \nu\}}\ (\tau \triangleright \overline{\tau}) \\
\tau_1 \rightarrow \tau_2 \triangleright \overline{\tau}_1 \rightarrow \tau_2 & = & \lambda f. f \circ (\overline{\tau}_1 \triangleright \tau_1) \\
\tau_1 \rightarrow \tau_2 \triangleright \tau_1 \rightarrow \overline{\tau_2} & = & \lambda f.\ (\tau_2 \triangleright \overline{\tau_2}) \circ f \\
(\tau_1, \tau_2) \triangleright (\overline{\tau_1}, \tau_2) & = & \tau_1 \triangleright \overline{\tau_1} \otimes \tau_2 \triangleright \tau_2 \\
(\tau_1, \tau_2) \triangleright (\tau_1, \overline{\tau_2}) & = & \tau_1 \triangleright \tau_1 \otimes \tau_2 \triangleright \overline{\tau_2}
\end{array}$$

Figure 5.4: Abstract implementations of coercions.

lateral composition of functions as follows:

$$f \otimes g \quad \stackrel{\text{def}}{=} \quad \lambda(a, b).(f\ a, g\ b)$$

In Figure 5.4, each Flatland coercion on the left corresponds to an abstract implementation on the right. The operators on the right, whose full definitions are not enumerated here, all have straightforward implementations in PML.

- **toFA** and **fromFA**. The operator **toFA** puts the elements in a parallel array into the flat data vector of a flattened array, and pairs that vector with a shape tree consisting of a single leaf from 0 to the number of elements in the array. **fromFA** is the inverse operation, which pulls the flat data vector out of a depth-1 flattened array and discards its shape information.
- **zip** and **unzip**. These operators are analogous to `ListPair.zip` and `ListPair.unzip` from the SML basis library, adapted to work either on parallel arrays or flattened arrays as needed.

- **cat** and **seg**. The **cat** operator is similar to `Vector.concat` in the SML basis, which concatenates a vector of vectors of elements into one vector by successively appending them. This **cat** is similar, although it also records the structure of the array of arrays it concatenates as it goes, preserving it in a shape tree. The operation **seg** is the inverse, repackaging an array into an array of arrays based on the segment information stored in a shape tree.

- **map** is analogous to `Vector.map` in the SML basis, adapted to iterate over parallel arrays and flattened arrays.

The other abstract implementations in Figure 5.4 are, by these rules, recursively synthesized out of their elemental components as needed. The compiler performs a similar synthesis of coercion operators, using Figure 5.4 as a guide.

The abstract flattening phase closely follows the big-step flattening semantics in Section 4.2. Abstract flattening has the following effects. Coming into abstract flattening, every parallel array literal appears as a `PArray` node in the AST, containing a list of expressions to be evaluated in parallel. Abstract flattening replaces every `PArray` node with an `FArray` node, carrying the same list of expressions as well as a `Shape.Leaf` indicating the length of the array. The type of the `FArray`'s elements is not necessarily scalar — that is, the `PArray` could be nested — so the compiler inserts a coercion operator to flatten nested arrays all the way down to their flat representations. Thus flattened arrays are given not only a new abstract syntactic form, but also a new type. Variables bound to parallel arrays in the source program are assigned fresh variables with flattened types, and substitutions are performed across the program to replace the old variables with the new.

During flattening, we treat a foundational group of parallel array operations specially in the compiler. This is the core group of blessed operators mentioned in Section 3.2. In `pmlc`, these are `PArray.map`, `PArray.filter`, `PArray.reduce`, `PArray.sub` (also written with infix `!` in PML), `PArray.tabulate` and its variants, `PArray.range`, and `PArray.app`. (Note that in our implementation, this group is larger than strictly necessary. In practice, it was easier to handle simple operations such as `length` directly than to encode them with primitives.) Each of the core operators is bound to a working polymorphic implementation in the PML basis library. If the flattening transformation is not enabled, the default polymorphic operations is used for each blessed operator. With flattening enabled, each of these operators is replaced during abstract flattening by a symbolic value (a member of an ML datatype) and a semantic type representing a type-indexed implementation of that operator. For example, `PArray.length` applied to an array of doubles is replaced by the value `PALength(Ty.Parr(Ty.Double))`, indicating which operator it is and at which type it is instantiated. The symbolic value of the operator is carried along until the concrete flattening phase, where it is replaced by an actual implementation.

Coercion operators are assigned symbolic values during abstract flattening, but the symbols are more specific than the plain types in Flatland's coercions ($\tau_1 \triangleright \tau_2$). Using Figure 5.4 as a guide, we resolve type coercions to the combinations of primitives (such as $\mathbf{unzip}$ and $\mathbf{cat}$) that appear on the right-hand side of that figure. Coercions that survive the second fusion phase are matched to implementations during concrete flattening as well.

The outcome of abstract flattening is that the type constructor `parray` does not appear in programs past this phase of transformation. Every parallel array is replaced with a flattened equivalent.

The following example demonstrates the operation of the abstract flattening phase. In this example, a function of type

```
int parray -> (bool * double) parray
```

is transformed to a function of type

```
(int, lf) farray -> ((bool, lf) farray) * ((double, lf) farray)
```

(For clarity, we use ML syntax here to represent compiler-internal semantic types.) The type constructor `parray` appears nowhere in the program subsequent to transformation.

5.3.2 Flatten-Operator Fusion

The compiler represents abstract coercions as members of the following algebraic datatype:

```
datatype fl_op
  = ID of ty
  | Unzip of ty
  | Concat of ty
  | Map of fl_op * shape_ty
  | Compose of fl_op * fl_op
  | CrossCompose of fl_op list
```

This datatype does not represent the "unflattening" operators (**fromFA**, **zip**, and **seg**) from the abstract coercion implementations in Figure 5.4. Such operators are necessary to complete the Flatland system, but they are not needed during `pmlc`'s aggressive flattening, which only ever coerces in the flat direction. Moreover, **toFA** is unneeded as a consequence of the previous phase's having replaced `PArray` nodes by `FArrays`.

The purpose of the flatten-operator fusion phase is to identify certain combinations of coercion operators in order to eliminate them. The compiler uses pattern matching to find its targets. Its actions are carried out by application of a set of legal rewrite rules, which are as follows:

- The application of any operator `ID` to any term `e` is rewritten to `e`.
- Any operator of the form `Map(ID, _)` is rewritten to `ID` of the appropriate type.
- The composition of `ID` with any operator `op` is rewritten to `op`.
- Any cross composition, all of whose elements are `ID` operators, is rewritten to `ID` of the appropriate (tuple) type.

The fusion process iterates until the compiler is unable to find any more fusion opportunities.

5.3.3 Concrete Flattening

In concrete flattening, the compiler commits to particular representations of data structures and implementations of the operations that work on them. Entering this phase, flattened arrays exist as their own kind of AST node, with their own types. During this phase, they are realized as rope/shape tree pairs, as discussed in Chapter 3. For ropes of scalars, the compiler selects monomorphic ropes to build monomorphic flattened array representations.

The shape type component of flattened array types is used during this phase to select the implementations of type-indexed operators, but after it has been so used, it is dropped. From this point forward, the depth of a flattened array no longer part of its type. This is not a problem; the shape type has already served its purpose. For example, the types `(int,lf) farray` and `(int, lf nd) farray` are both translated to `int_farray` during this phase. The inclusion of shape types past this phase would require PML to support dependent types [51], which it currently does not.

Coercion operators are realized in one of two ways. Sufficiently common coercions are already built into the basis library, in which case they are implemented with calls into the basis. If a coercion does not appear in the basis, it is synthesized by composition of other coercion operators, which may in turn need to be recursively synthesized.

The following example demonstrates the operation of the concrete flattening phase. Here, `PArray.map` is applied to an array of pairs of ground terms:

```
val f : int * bool -> int = ...
val a = PArray.map f [| (1, true), (2, false), ... |]
```

Since we are going to both unzip and monomorphize the array of pairs as part of the flattening transformation, we need a version of `map` that traverses not a single polymorphic data structure (as it would without flattening), but rather two monomorphic flattened arrays of two different types at the same time. Furthermore, it must produce a monomorphic array of integers. In the present case, we need an operator with the following signature:

```
val map_ibi : int_farray * bool_farray -> int_farray
```

The type of `map_ibi` gives sufficient information for its implementation. The code of this and other operators follows a standard pattern, so synthesis of such operators is a mechanical process.

PML flattening has been designed for modularity. There is a useful abstraction barrier between its second and third phases. Concrete flattening need not take place in only one way. In the current implementation, there is one particular strategy for representing flattened arrays (the various monomorphic versions of `farray`), but the three-phase design here makes it possible to experiment with different representations.

5.4 Optimizations

Flattening exposes optimizations to PML programs, which account for the performance improvements we report in Chapter 6. Two are discussed here: monomorphization and tab flattening. In addition to these currently-implemented optimizations, flattened PML is suitable for use with vector-width (*e.g.*, SSE) instructions, but our compiler technology does not yet support them.

5.4.1 Monomorphization

The compiler selects monomorphic representations for flattened arrays of scalars. This is manifest in the rope component of the flat arrays. The polymorphic rope datatype (Figure 3.2) points to sequences of boxed, heap-allocated values at its leaves, whereas monomorphic ropes such as `IntRope` and `DoubleRope` point to contiguous sequences of unboxed values at their leaves. Monomorphic ropes are smaller and faster to compute with than polymorphic ones, and we can observe performance benefits when using them even in benchmarks when no flattening is involved.

Monomorphization is complemented by unzipping of arrays of tuples. An array of pairs of integer and doubles, when unzipped, becomes an arrays of integer and an array of doubles; these in turn are represented by a flattened array with an `IntRope` and another with a `DoubleRope`. In the past, this rearrangement was necessary to accommodate NESL-era hardware. On multicores, the unzipping part of this transformation is not mandatory — the integer and double values could be packed into a flat vector of their own, with alternating integers and doubles in contiguous memory — but it is desirable because such vectors are easily amenable to vector-wide processing. When we build support for vector-width instructions in PML, we expect to observe substantial performance benefits on programs that make use of them over unzipped, monomorphized flattened array structures.

5.4.2 Tab Flattening

Flattening is directed at both regular and irregular parallel structures, but regular parallel computations are an important special case that occur frequently in practice. In particular, regular multidimensional tabulations are common in the computation of images and mathematical structures (see Chapter 6). Tab flattening is an optimization that can be applied to regular nested array structures. We gave an overview of tab flattening in Section 3.3; here we discuss the details of its implementation in `pmlc`.

Tab flattening is performed on regular parallel comprehensions. An array of arrays is rectangular at $m \times n$ if all its m inner arrays have the same length n. Generalized to higher dimensions, nested arrays with this property are *regular*. The compiler recognizes parallel comprehensions to be regular arrays if, at every level, they compute over a range and have no filtering **where** clause. (We inline range values into the right-hand-sides of parallel comprehensions in an earlier pass so these criteria are met more often.) The shape of the regular arrays can be computed in advance, and the necessary index arithmetic is performed according to a pattern as described below.

The one-dimensional case requires no flattening *per se*, but it is a useful starting point for explaining the general case. Consider some expression e computing over the range `f` to `t` by `s`.

```
[| e | i in [| f to t by s |] |]
```

The names `f`, `t`, and `s` inside the range are mnemonics for from, to, and step, respectively. In order to translate this comprehension into a tabulation, we first generate the function `g` over its free variable `i` (bound on the right-hand-side in the comprehension):

```
fun g i = e
```

We will use the following function `indexMap1D` to map a single counter value to the corresponding element in the range `[| f to t by s |]`. The following higher-order function generates one dimensional index maps based on the values in a range.

```
fun indexMap1D (f,t,s) = (fn k => f + (k * s))
```

If we let f, t, and s be the integer values corresponding to `f`, `t`, and `s` in the PML expression, applying the semantics of ranges given in Chapter 3 yields the following sequence:

$$\{n_k = f + k(s) \mid k \in \mathbb{N}, (n_k = f) \vee (f \leq n_k \leq t)\}$$

Let n be the number of elements in the sequence so defined. The value of n is given by

$$1 + \max(0, (t - f)/s)$$

We write this formula as a function in PML and name it `nElts`.

```
fun nElts (f, t, s) = 1 + Int.max (0, (t-f) div s)
```

We assemble these components together to arrive at the following translation of the original expression:

```
fun tab1D ((f,t,s), g) = let
  val n = nElts (f,t,s)
  val indexMap = indexMap1D (f,t,s)
  val data = Rope.tabulate (n, g o indexMap)
  val shape = Shape.Lf (0, n)
  in
    FArray (data, shape)
  end
```

We have omitted the details of computing a regular `shape` for a regular nested array, but the process is straightforward. We have not specified the type of e in this example, but in an actual compilation, both Rope and FArray would have been monomorphized to particular types in tab1D.

The two-dimensional case is a natural extension of the previous one. We begin with a regular two-dimensional nested comprehension:

```
[| [| e | j in [| f2 to t2 by s2 |] |]
        | i in [| f1 to t1 by s1 |] |]
```

We build a function g consisting of the body of the comprehension over its free variables i and j:

```
fun g (i, j) = e
```

We can build two-dimensional index maps with the following higher-order function

```
fun indexMap2D ((f1,t1,s1), (f2,t2,s2)) = let
  val d1 = nElts (f1,t1,s1)
  val d2 = nElts (f2,t2,s2)
  in
    fn k => (f1 + ((k div d2) mod d1) * s1,
             f2 + (k mod d2) * s2)
  end
```

and then compose the pieces as follows:

```
fun tab2D ((f1,t1,s1),(f2,t2,s2)) = let
  val n1 = nElts (f1,t1,s1)
  val n2 = nElts (f2,t2,s2)
  val n = n1*n2
  val indexMap = indexMap ((f1,t1,s1),(f2,t2,s2))
  val data = Rope.tabulate (n, g o indexMap)
  val shape =
    Shape.Nd (List.tabulate (n1, fn i => Shape.Lf (i*n2, i*n2+n2)))
  in
    FArray (data, shape)
  end
```

The benefit of having performed this optimization comes from being able to build a scalar rope directly, as opposed to a rope of ropes.

In pmlc, we generalize this technique to any number of dimensions. The process we follow is a straightforward extension of what we have already shown. Scaling the index map function up to m dimensions, we have

```
fun indexMap_m_D ((f1,t1,s1),(f2,t2,s2),...,(fm,tm,sm)) = let
  val d1 = nElts (f1,t1,s1)
  val d2 = nElts (f2,t2,s2)
  ...
  val dm = nElts (fm,tm,sm)
  in
    fn k => (f1 + ((k div (d2 * ... * dm)) mod d1) * s1,
             f2 + ((k div (d3 * ... * dm)) mod d2) * s2,
             ...,
             fm + ((k mod dm) * sm))
  end
```

The rest of the translation up to `m` dimensions follows from what we have already shown.

CHAPTER 6
EVALUATION

This section demonstrates that, across a variety of benchmarks, PML programs compiled with the flattening transformation outperform their non-flattened counterparts. Both flattened and non-flattened PML executables scale well on many processors (as many as 48 on our experimental platform), and both perform well in comparison to similar programs written in sequential languages.

We use SML execution times for our sequential baselines. SML programs were compiled with MLton, version r7549 [35], a whole-program optimizing compiler for Standard ML. For two benchmarks, we also wrote C programs as additional points of reference. We provide comparisons to C execution times in those cases.

The benchmark data presented in this section comes from experiments run on a Dell PowerEdge R815 server with 48 cores and 128 GB of DDR3, 1333 MHz RAM. The operating system is x86_-64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-27. The 48 cores are on four 12-core AMD Opteron 6172 Magny-Cours processors, each of which operates at 2.1 GHz and has 64 KB each of instruction and data L1 cache and 512 KB of L2 cache. There are eight 6 MB L3 caches, each of which is shared by six cores.

6.1 Mandelbrot

We compute the Mandelbrot set by means of a function `elt` which consumes a pair of integers and produces an integer. The argument to `elt` represents a location in the complex plane. Its return value is the number of iterations required, according to the standard iterating Mandelbrot set membership test, for a given point to diverge outside the set (by having a modulus greater than 2). A point is a member of the Mandelbrot set if it fails to diverge before reaching a fixed upper limit of iterations (we use 1000).

Table 6.1: Mandelbrot: execution times (seconds) with standard deviations, per number of processors, problem size 8196.

	PML, flat	PML, not flat	SML	C
1	272.20 (.000)	275.35 (.000)	140.13 (.000)	135.73 (.000)
2	136.36 (.000)	138.37 (.000)		
4	68.20 (.000)	70.99 (.011)		
8	34.11 (.000)	38.22 (.018)		
16	17.06 (.000)	17.60 (.002)		
32	8.55 (.000)	8.96 (.006)		
48	5.74 (.002)	6.09 (.007)		

Our PML benchmark program uses a SML-style implementation of `elt` (standard definition not shown) as a function with a recursive inner loop. The function is used with a nested parallel comprehension to compute Mandelbrot set membership over a rectangular array of pairs. For a given problem size n, specified by a command-line argument, the square of side length 4 centered around the origin is effectively overlaid with an $n \times n$ lattice of points in the complex plane. (The mapping of integer pairs to lattice points is handled in `elt`.) The Mandelbrot set is computed over the nested array as follows:

```
fun mandelbrot n = let
  val rng = [| 0 to (n-1) |]
  in
    [| [| elt (i, j) | j in rng |] | i in rng |]
  end
```

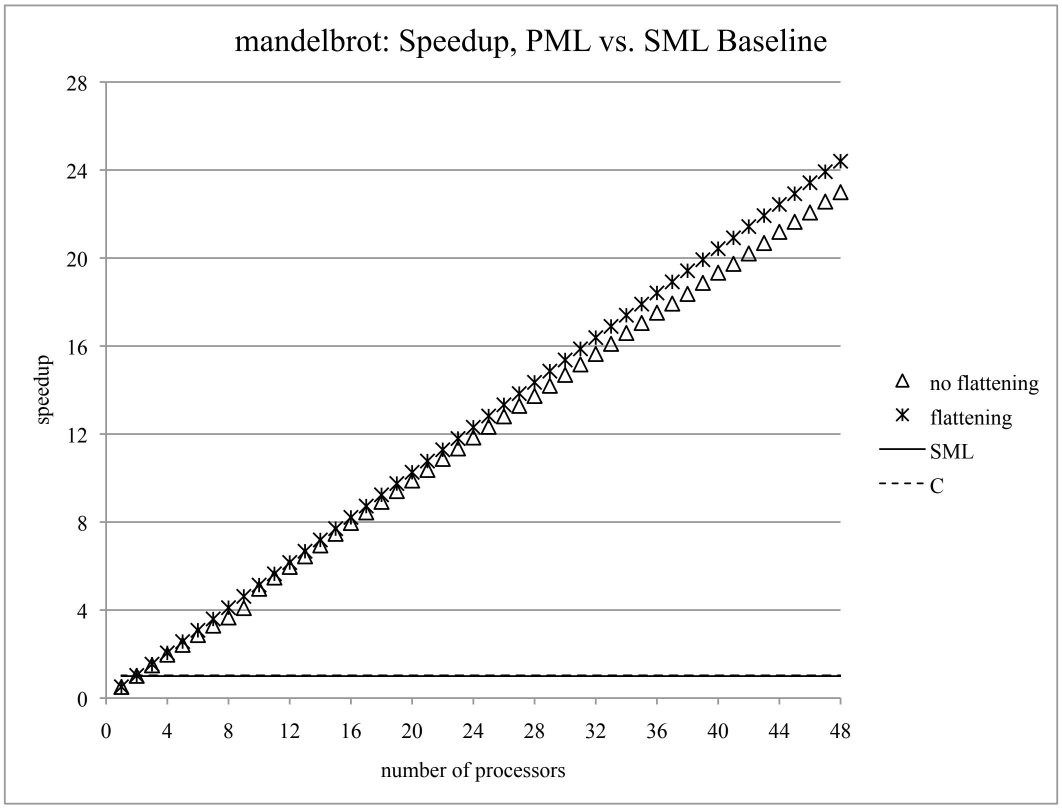
Note the computation is regular in the shape of the structures over which it is computed, but irregular in the sense that widely varying amounts of work (a reflection of the fractal shape of Mandelbrot set's boundary) are necessary per application of `elt`. The SML baseline program is the same as the PML program with necessary adjustments, most notably the substitution of nested calls to `Vector.tabulate` for PML's nested parallel comprehensions. The C baseline uses loops in place of recursive functions and builds an array of arrays of integers. The C executable was compiled with `gcc`, version 4.4.3, at optimization level 2.

For our experiments, we ran flattened and non-flattened versions of Mandelbrot in PML with problem size 8196 for all numbers of processors from 1 to 48. We ran sequential SML and C programs at the same problem size. C is faster than SML by a modest margin. Table 6.1 reports selected execution times from the experiments. Each execution time is given in seconds and is the arithmetic mean of 40 individual executions. The standard deviation of each group of 40 runs is given as a percentage of the mean and is written in parentheses after each time. Standard deviations were low in all cases, less (usually much less) than 2% in every case except for one aberration (3.2% for PML, not flattened, at 9 processors). In comparison to sequential execution times, absolute performance of Mandelbrot is excellent. At 2 processors, PML runs slightly faster than the MLton baseline, and by 4 processors far surpasses it.

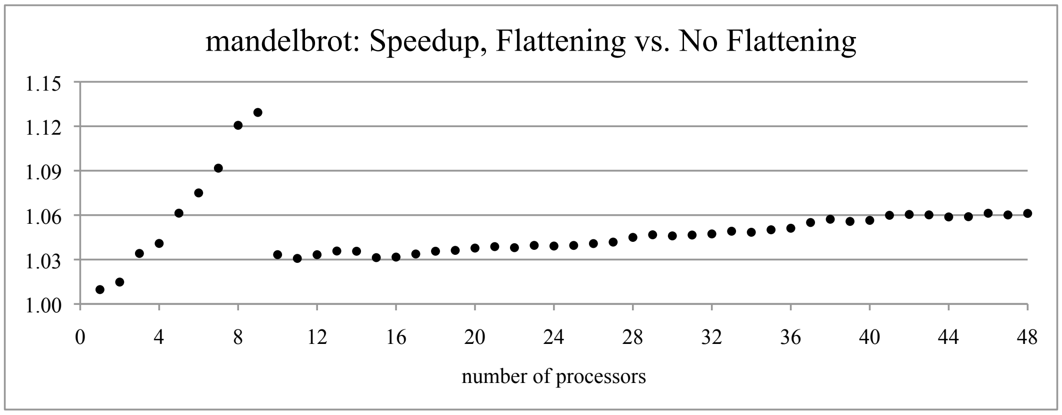
Figure 6.1(a) shows PML speedups, with and without flattening, against the SML baseline. We display C performance on the plot (barely distinguishable from SML) as an additional point of reference. Flattened PML outperforms non-flattened PML in every case. At 48 processors, flattened PML improves on baseline performance by a factor of just over 24.

Figure 6.1(b) gives the speedup of PML with flattening over PML without flattening as the ratio of the non-flattened time to the flattened time. The higher the data point in the plot, the faster the flattened executable. There is a dramatic widening in the performance gap up to nearly 13% at 9 processors, immediately after which flattening settles down to a 3% speedup and trends gradually upward to 6% for 48 processors.

We attribute the speedup achieved by flattening Mandelbrot to monorphization and tab flattening. Since the main inner loop of computing Mandelbrot is a rectangular nested parallel comprehension, the compiler automatically rewrites it to a linear tabulation per the scheme outlined in Section 5.4.2. Furthermore, because the value of that nested comprehension is monomorphized to an `int_farray` (containing a specialized `int_rope`), the tabulation is able to produce a lightweight rope structure with raw integers, as opposed to pointers to heap-allocated integers, at the leaves.



(a) Flattened and non-flattened PML vs. SML baseline.



(b) Speedup of flattened over non-flattened PML.

Figure 6.1: Mandelbrot speedups.

Table 6.2: Raytracer: execution times (seconds) with standard deviations, per number of processors, problem size 1024.

	PML, flat	PML, not flat	SML
1	61.04 (.005)	63.79 (.007)	45.70 (.001)
2	30.12 (.000)	30.98 (.001)	
4	15.07 (.001)	15.50 (.002)	
8	7.55 (.001)	7.76 (.003)	
16	3.79 (.001)	3.89 (.003)	
32	1.92 (.002)	1.97 (.004)	
48	1.32 (.006)	1.36 (.013)	

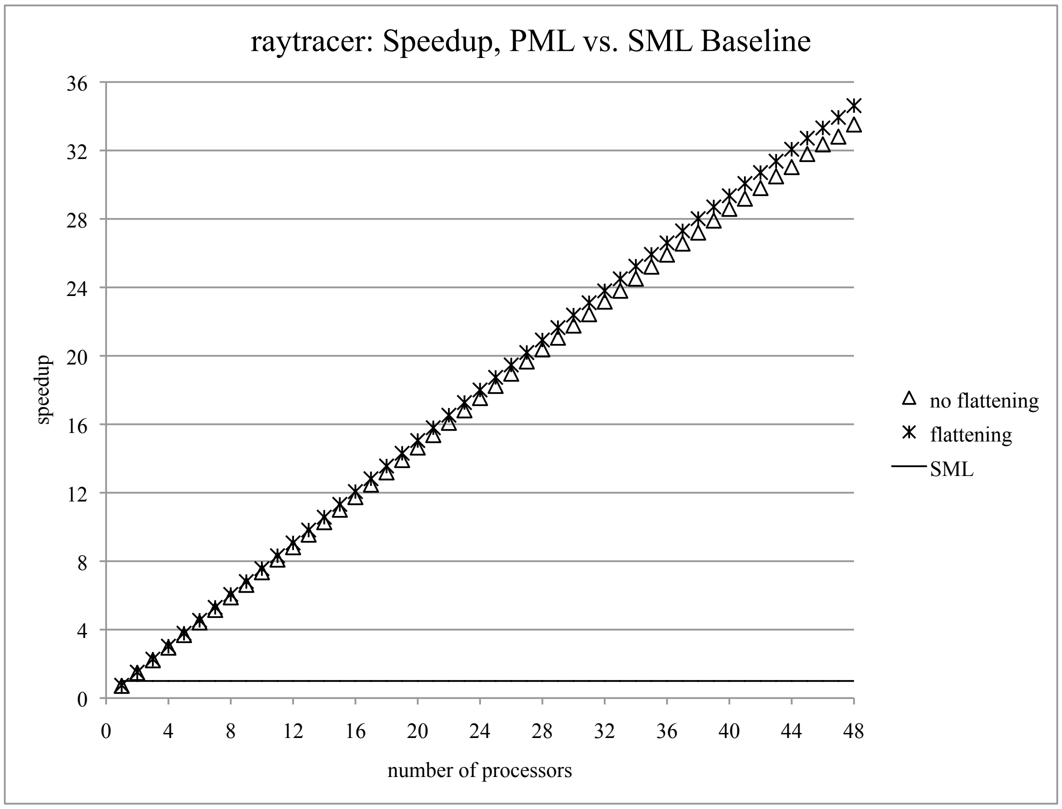
6.2 Raytracer

Our ray tracing benchmark computes the image of a scene graph consisting of a group of overlapping spheres with transparency and reflection. The code is translated from a parallel program in the implicitly-parallel language Id90 [36]. It is a brute-force implementation and does not use any acceleration data structures.

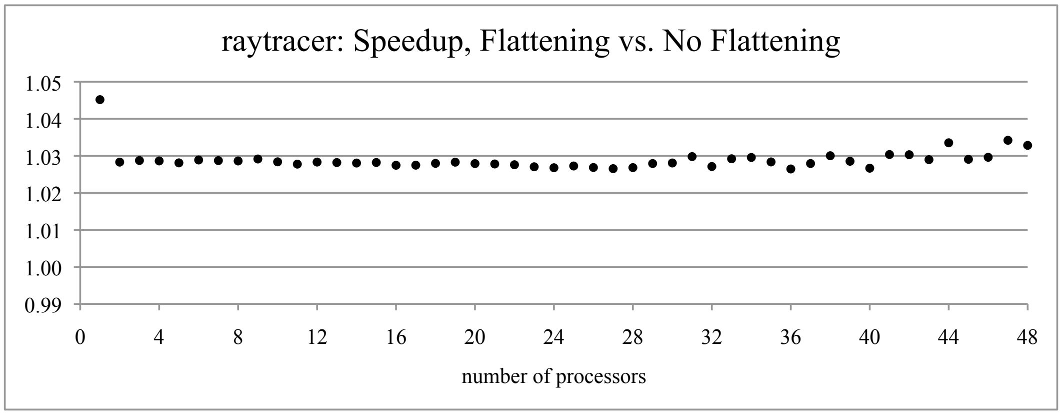
Ray tracing works in our benchmark by computing a rectangular parallel array of color values, each representing a pixel in the generated image. The function `trace` (definition not shown) consumes a pixel location and traces a ray through the scene graph, returning the pixel's color in the result. We write the body of the main function as a nested parallel comprehension:

```
fun raytracer n = let
  val side = [| 0 to (n-1) |]
  in
    [| [| trace (i, j) | j in side |] | i in side |]
  end
```

The program computes an size $n \times n$ array of colors for a command-line integer argument n. For the present work, our raytracer stops short of writing the color values it computes to an image file; writing to a file involves no parallel computation and therefore contains no interesting information for our measurements. (The program is able to report its colors at the console on demand, for the purposes of checking its output.)



(a) Flattened and non-flattened PML vs. SML baseline.



(b) Speedup of flattened over non-flattened PML.

Figure 6.2: Raytracer speedups.

Table 6.2 reports selected mean runtimes in seconds, with (low) standard deviations as a percentage of mean runtimes in parentheses. The SML baseline code was similar to the PML code, minus parallel constructs. We did not port the ray tracer implementation to C. For our experiments, we computed images of size 1024 $\times$ 1024. Figure 6.2(a) presents speedups of flattened and unflattened PML over the baseline. Except at a single processor, flattened PML is always fastest. PML's raytracer has excellent absolute performance with respect to a sequential SML baseline, achieving a better than thirtyfold speedup on 48 processors. Figure 6.2(b) gives the speedup of flattened PML over non-flattened PML; the improvement hovers consistently around 3%.

Like Mandelbrot, ray tracing takes an unpredictable amount of time on a per-element basis, though its structure as whole is regular. Also like Mandelbrot, this problem benefits from monomorphization and tab flattening, to which we attribute its faster execution times.

6.3 Mandelbulb

The Mandelbulb [50] is a fractal-like three dimensional solid, inspired by the Mandelbrot set. The computation of the Mandelbulb is over the cube of edge length 4 centered at the origin of a 3-dimensional coordinate space. Mandelbulb applies a test to each location in this centered cube. Like the Mandelbrot set test, it is an iterative computation which runs either until a value diverges (by having a modulus greater than 2) or the number of test iterations reaches an upper limit (we set the limit at 1000). The implementation of Mandelbulb is similar to the implementation of Mandelbrot, but the function that tests a location's membership in the set, which we again call `elt`, now consumes three integers, and the nested parallel comprehension at the heart of the program is three layers deep. Its results is a three-dimensional $n \times n \times n$ array of integers.

Table 6.3: Mandelbulb: execution times (seconds) with standard deviations, per number of processors, problem size 64.

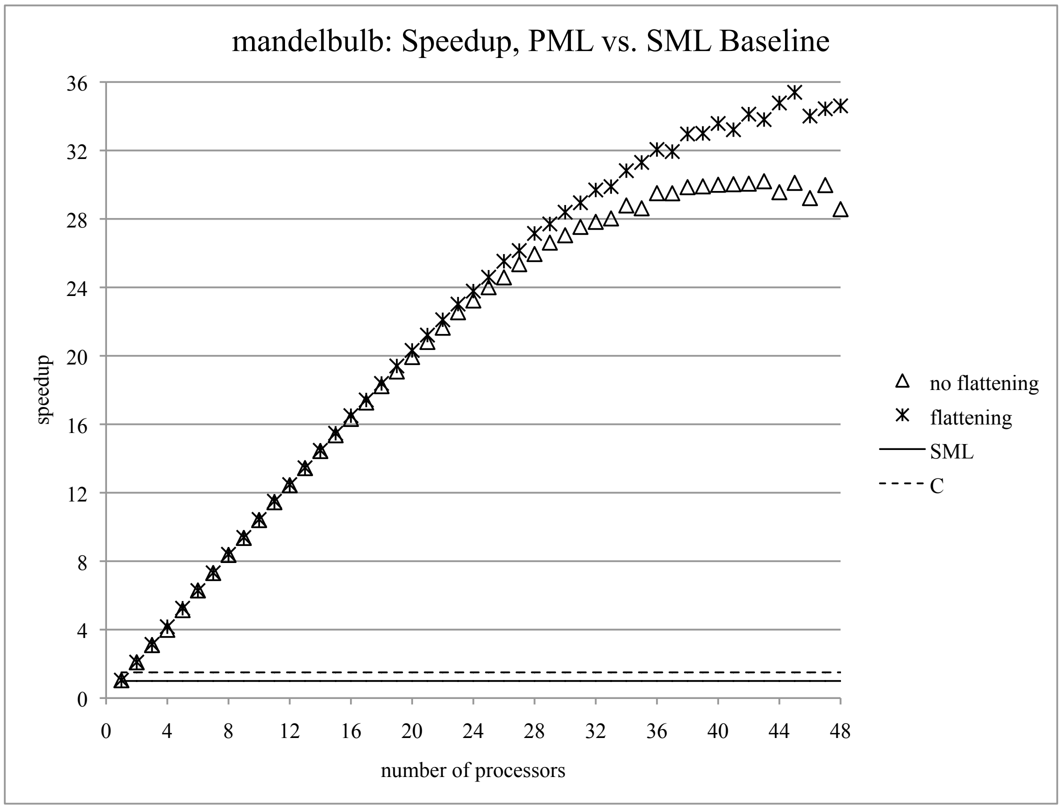
	PML, flat	PML, not flat	SML	C
1	6.36 (.014)	6.37 (.010)	6.66 (.014)	4.43 (.008)
2	3.17 (.010)	3.18 (.010)		
4	1.60 (.012)	1.67 (.089)		
8	0.79 (.010)	0.97 (.010)		
16	0.40 (.009)	0.41 (.010)		
32	0.22 (.035)	0.24 (.042)		
48	0.19 (.085)	0.23 (.127)		

```
fun mandelbulb n = let
  val range = [| 0 to n-1 |]
  in
    [| [| [| elt (i,j,k) | k in range |]
                         | j in range |]
                         | i in range |]
  end
```

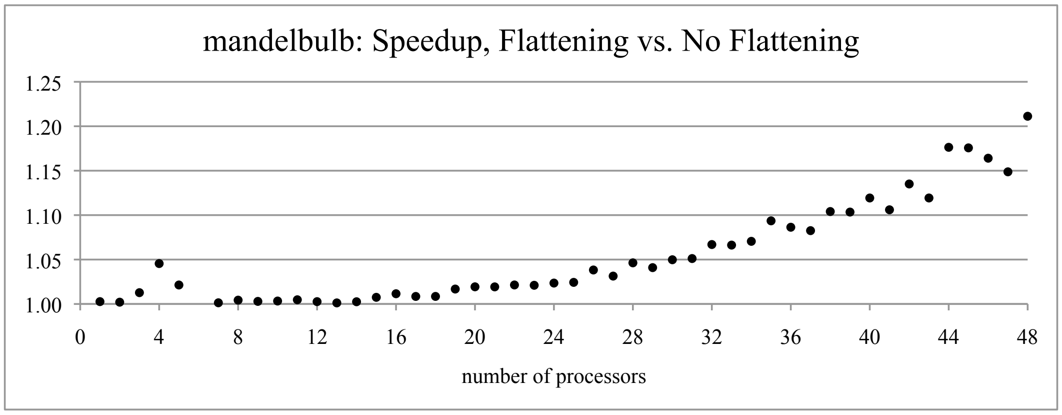
For our purposes, Mandelbulb provides a useful test for measuring whether the optimizations that yield performance improvements in regular two-dimensional examples such as Mandelbrot do as well in three dimensions.

Table 6.3 contains experiment results for executions computing a size 64 $\times$ 64 $\times$ 64 Mandelbulb volume. The SML baseline is a straightforward modification of the PML code, as in the previous two cases. We also wrote Mandelbulb in C. The C implementation is written as a main triply-nested for-loop, calling a while-loop-based implementation of the Mandelbulb membership test. The C code was compiled with `gcc`, version 4.4.3, at optimization level 2. In this case, C is faster than SML, representing a speedup of about 50%. We nevertheless continue to use SML as our baseline for consistency with our other experiments. Performance of PML Mandelbulb is good with respect to both sequential baselines.

Figure 6.3(a) shows speedups of flattened and non-flattened PML programs relative to SML. We include C on the plot for further comparison. Both flattened and non-flattened PML proceed



(a) Flattened and non-flattened PML vs. SML baseline.



(b) Speedup of flattened over non-flattened PML.

Figure 6.3: Mandelbulb speedups.

roughly linearly up to about 28 processors, at which flattened PML begins to realize an increasing gain over the non-flattened version. Figure 6.3(b) shows the ratio of execution times for non-flattened and flattened programs on 1 to 48 processors. Flattened PML realizes a 2.5% gain over non-flattened PML at 24 processors. From that point forward, flattened code pulls away from non-flattened code, and achieves a better than 20 percent speedup at 48 processors. Mandelbulb is subject to the same tab-flattening and monomorphization optimizations as the two previous experiments.

6.4 Sparse-Matrix Vector Multiplication

Among our experiments, sparse-matrix vector multiplication profits most from transformation. Sparse-matrix vector multiplication is expressed concisely as the following group of one-line PML functions:

```
fun plus (x:double, y:double) = x+y

fun sum (xs : double parray) = PArray.reduce plus (0.0) xs

fun dotp (sv, v) = sum [| x * (v!i) | (i,x) in sv |]

fun smvm (sm, v) = [| dotp (sv, v) | sv in sm |]
```

The function `sum` is implemented as a call to `PArray.reduce`, which performs a tree-shaped collapsing reduction over a rope of double-precision floating-point numbers. Sum as implemented on top of `PArray.reduce` processes the rope of doubles in parallel, where the sums of the values at each leaf are computed sequentially, and the sums of the values under each internal node are added to one another once they have both been computed. This happens over the whole tree in parallel, and the sum of the whole is the sum computed at the rope's topmost node. The function `dotp` is the application of `sum` to multiplication operations, with a parallel array of double representing a dense vector, over a parallel array of pairs. The function `smvm` is, in turn, the application of `dotp` over the members of an array of such arrays of pairs.

Table 6.4: SMVM: execution times (seconds) with standard deviations, per number of processors, problem size 20000.

	PML, flat	PML, not flat	SML
1	39.22 (.006)	163.34 (.004)	14.01 (.012)
2	24.25 (.014)	106.81 (.015)	
4	17.58 (.041)	69.04 (.136)	
8	12.44 (.062)	48.26 (.154)	
16	5.99 (.091)	23.40 (.123)	
32	3.50 (.136)	12.80 (.123)	
48	2.40 (.119)	9.63 (.135)	

The flattened version of `smvm` uses a monomorphic segmented sum operation as discussed in Chapter 3. Our flattened `smvm` is written explicitly as a separate program, since the compiler is not yet implemented to rewrite parallel comprehensions to segmented operations automatically. [1]

```
fun smvm (sm, v) = let
  val prods = products (sm, v)
  val sums = segsum prods
  in
    sums
  end
```

The function `products` (definition not shown), consumes a sparse vector, represented by a pair of flattened arrays, and a dense vector, represented by a single flattened array. The sparse vector's pair contains a flattened array of indices containing an `int_rope` of raw integers as its flat data vector, and a flattened array of values containing a `double_rope` of raw doubles.

The computation of `products` runs in one (parallel) elementwise pass over the two flattened arrays in the sparse vector (see the discussion of map flattening in Section 3.3). The function `segsum` executes a fast segmented sum over the result of `products`; its implementation is as given in Chapter 5. In our experiments, an irregular sparse matrix is constructed according to a deterministic algorithm so the runs between implementations are comparable. For the program

1. This rewriting presents no special difficulty. If the compiler can identify `sum`, it can rewrite expressions of the form `[| sum xs | xs in xss |]` to `segsum xss`. Analogous rewrites are available for as many sum-like operators (reductions) the compiler is able to recognize.

runs reported in our plots, we constructed sparse matrices of 20000 rows of varying lengths, and dense vectors of width 1000.

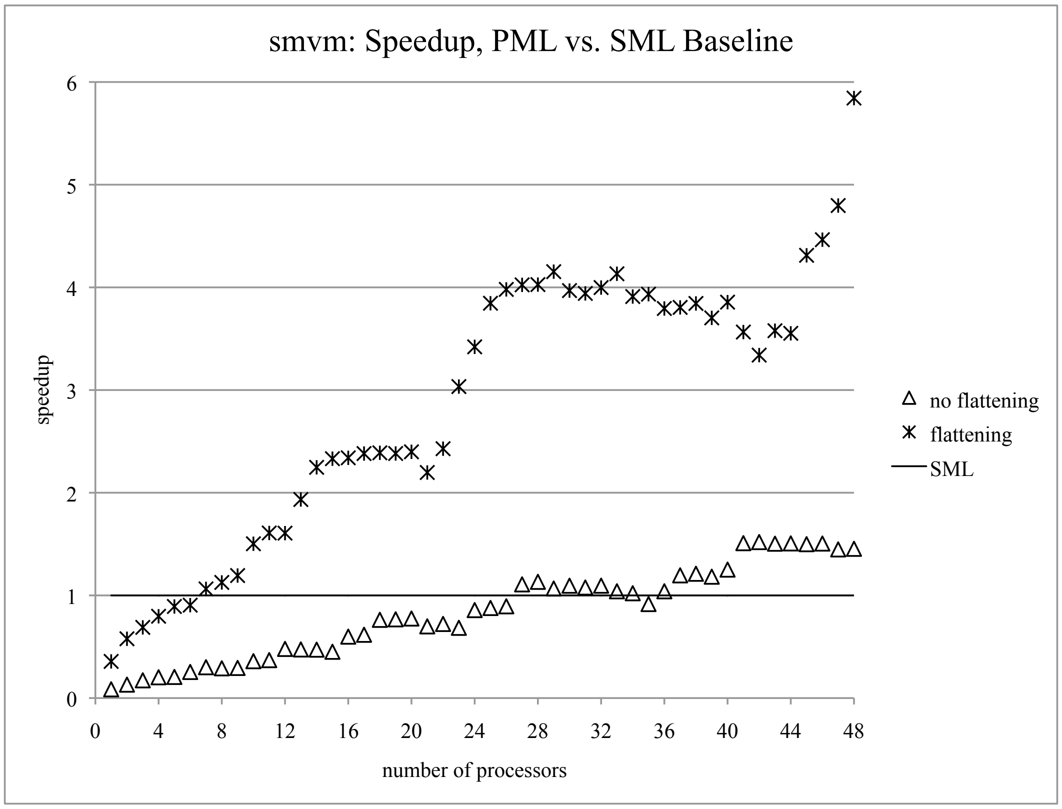
In our experiments, the impact of flattening on performance is dramatic in the case of `smvm`. The flattened version of the program is between a factor of 3 and 5 faster than the non-flattened version. Figure 6.4(b) gives the ratio of running times of the non-flattened to the flattened code. The higher the value of a given data point in the plot, the faster flat `smvm` is than non-flat `smvm`.

We show the performance of flattened and non-flattened `smvm` relative to a sequential SML baseline in Figure 6.4(a). The SML implementation uses vectors (in the sense of `Vector.vector`) of doubles to represent dense vectors, vectors of pairs of `int`s and `real`s to represent sparse vectors, and vectors of sparse vectors to represent sparse matrices. It is a straightforward high-level SML implementation of this problem.

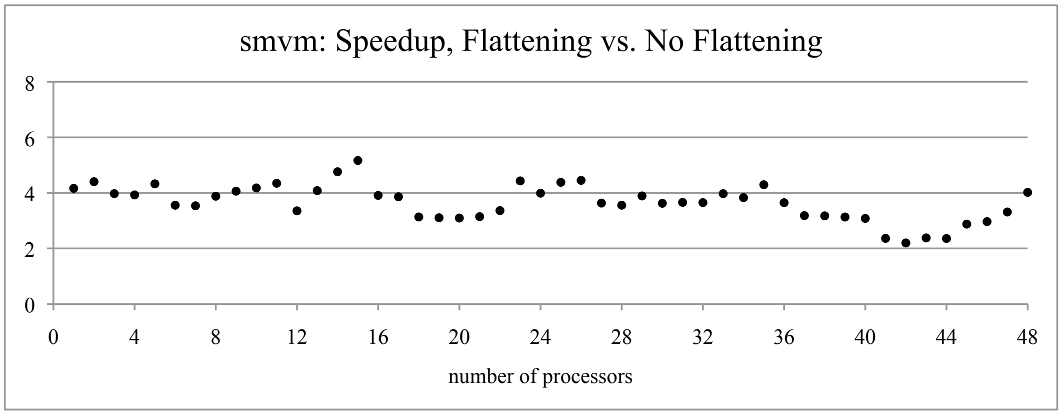
```
fun dotp v sv = let
  val sum = Vector.foldl op+ 0.0
  fun mul (i, x) = x * Vector.sub (v, i)
  in
    sum (Vector.map mul sv)
  end

fun smvm (sm, v) = Vector.map (dotp v) sm
```

Figure 6.4(a) gives the speedups of PML over its SML baseline, both with and without flattening. Flattened `smvm` is substantially faster than non-flattened `smvm` for all numbers of processors up to 48, and furthermore has much better performance with respect to the fast sequential baseline used here. Flattened `smvm` outpaces the baseline at 7 processors, whereas non-flattened `smvm` needs 27 processors to accomplish the same. PML with no flattening must compensate for the irregularity of the problem with dynamic scheduling adjustments. The nested program does scale roughly linearly with increasing numbers of processors, but the rate of improvement is low and absolute performance is comparable to sequential performance, even with 48 processors. The flattened program, by contrast, is mainly a call to segmented sum on a flattened irregular structure. Whatever the irregularity of that structure in the source program, at runtime it is a flat rope struc-



(a) Flattened and non-flattened PML vs. SML baseline.



(b) Speedup of flattened over non-flattened PML.

Figure 6.4: Sparse-matrix vector multiplication speedups.

ture whose segmented sum can be computed by an embarrassingly-parallel rope reduction (see Section 5.2.2). We attribute the dramatic improvement of flattening to this property of segmented sum.

CHAPTER 7

CONCLUSION

The implementation of nested-data-parallel programming languages is a challenging and important problem. Well-engineered parallel languages help programmers make better use of the parallel computing resources at their ever-increasing disposal. Nested data parallelism is an especially attractive model for parallel programming, since it enables high-level parallel programs to exploit parallel resources well, regardless of the regularity or irregularity of a particular programming problem. In nested-data-parallel programs, programmers can employ the idioms to which they are accustomed and achieve parallel speedups with only modest accommodations in their source code.

Nested-data-parallel languages have been in existence for over two decades now, but the ground has shifted underneath them. As a platform for parallel computing, multicore computers long ago superseded the wide-vector machines for which nested-data-parallel compilers were originally designed. Nevertheless, nested-data-parallel compilation, while adapted to ever more sophisticated languages, has remained fundamentally unaltered in its orientation toward vector instructions. This dissertation challenges long-standing techniques for compiling nested-data-parallel programs in a vector-machine style, on the basis that they are not appropriate for multicore targets.

We have presented hybrid flattening as a suitable alternative to traditional techniques. Hybrid flattening transforms nested data structures to expose programs to various optimizations, while leaving control structures intact. Furthermore, hybrid flattening supports a choice of flattening policies, ranging from no data flattening to full flattening of all nested data. We have provided a semantics of hybrid flattening in the form of Flatland, a model language with a rewriting system, and proven properties about its formal integrity. Flatland is mechanism without policy, providing no particular guidance on how to transform programs. In order to be useful in designing an optimizing transformation, Flatland must be coupled with a strategy for application of its rewriting rules. We have embedded such a strategy in our formal semantics of aggressive hybrid flattening, which we have proven to be expressible under Flatland. Aggressive hybrid flattening is a variation of traditional total flattening, flattening nested arrays in essentially the same way.

We used the formalization of aggressive hybrid flattening as the basis for a prototype implementation of a compiler transformation in the compiler for PML, `pmlc`. This enabled us to test the effects of flattening PML programs across various benchmarks. According to our tests, flattening improves the performance of regular nested-data-parallel programs significantly, and it improves that of irregular programs dramatically. As we had conjectured in earlier work [5], flattening and lazy tree splitting are a felicitous combination. Although our implementation of hybrid flattening is in PML, the semantic specification given in this document is sufficiently abstract to be applicable to nested-data-parallel languages broadly.

7.1 Future Work

Using the present material as a starting point, many lines of research appear worth pursuing. This section outlines a few possible directions.

On the semantic front, Flatland's core language could be extended to model a wider variety of language constructs. Algebraic datatypes with sum types would be a natural addition. There already exist well-known schemes for flattening sum types [16], which are unzipping-style transformations across variants. Flatland should appropriate these with suitable modifications. Because of the way sum types are flattened, future work along these lines might entail allowing hybrid flattening to perform some control changes.

Flatland's type system could be enriched to account for polymorphism, although it is not immediately clear how to write parametrically-polymorphic functions over arrays whose transformed representations are drawn from a family of possibilities. Data Parallel Haskell provides some guidance in this area. During compilation, all functions in a Data Parallel Haskell program are vectorized, and the non-vectorized and vectorized functions are paired together [33]. Both the original function and its vectorized version are then available at all uses; the function needed is simply selected from the pair. But with the large groups of equivalent representations in Flatland, a pair would not suffice, and generating all possible vectorizations of Flatland functions might be impractical, or even impossible. We might be able to determine which vectorized versions of which functions a program needs per a static analysis — control-flow analysis [47] comes to mind — and thereby avert a combinatorial explosion in compiled code. This appears to be a sizeable research problem in and of itself.

It is tempting to augment Flatland with a cost semantics, so transformations could be proven to achieve performance improvements. Any such cost semantics should account for parallel execution, since that is our target domain. It would be useful to be able to use Flatland to model transformation-driven optimizations in languages such as PML.

Flatland is designed to accommodate different transformation strategies, so we naturally look forward to using it to define an assortment of transformations. We already have two points of comparison for any such new transformation: compilation with no flattening, and compilation with aggressive flattening per Chapter 5. One can imagine defining a variety of transformations according to various heuristics. Aggressive hybrid flattening in Chapter 4 uses **F** (Figure 4.13) to make coercion choices. **F** is not a heuristic so much as a blanket policy: flatten everything. We have suggested in our discussion that more nuanced strategies may be useful. We might not always unzip array of pairs, depending on those pairs' types, for example. Furthermore, we can imagine using a static analysis of running time to guide flattening choices. Perhaps when the per-element computation in a parallel array is estimated to be uniformly light, no flattening should be undertaken on it. These and other strategies remain to be explored, both formally and empirically.

On the hardware front, we would like to make use of vector instructions on multicores. Our own 48-core machine supports 16-byte-wide vector operations, but our compiler's back-end code generator (ML-RISC) does not support them. We are considering a new back-end implementation in order to address this shortcoming. If we were able to make use of vector instructions in `pmlc`, we expect flattening would prove substantially more useful than it already has, especially in the case of arithmetically-intense parallel computations.

Finally, the abstraction layer between abstract and concrete flattening (Chapter 5) allows us to experiment with the representation of flattened arrays in the compiled code. We are not permanently tied to ropes. We are interested in experimenting with other flattened array representations whose flat data vectors are represented in some other way, such as contiguous large arrays, which might enable novel (for PML) optimizations such as in-place update of flat data vectors. We expect employing an assortment of vector representations other than ropes in our representation transformations will give us ways to improve performance still further.

APPENDIX A

PROOFS

A.1 Auxiliary Lemmas

Lemma A.1.1 (context replacement)**.** *Given* $\Gamma \vdash C[e^\tau]$ *ok and* $\overline{e}^\tau$ *such that* $\forall \Gamma' \,.\, \Gamma' \vdash e^\tau$ *ok* $\implies \Gamma' \vdash \overline{e}^\tau$ *ok,* $\Gamma \vdash C[\overline{e}^\tau]$ *ok.*

Proof. We have $\Gamma \vdash C[e^\tau]$ *ok*. Within the proof of the well-formedness of $C[e^\tau]$, there must exist some Γ' such that $\Gamma' \vdash e^\tau$ *ok*. Then, by the inductive hypothesis, $\Gamma' \vdash \overline{e}^\tau$ *ok*. Then $\Gamma \vdash C[\overline{e}^\tau]$ *ok* by induction on the rules of the well-formedness judgment. □

Lemma A.1.2 (substitution)**.** *For* $x \notin \mathrm{dom}(\Gamma)$*, if* $\Gamma[x^{\overline{\tau}}] \vdash e^\tau$ *ok and* $\Gamma \vdash \overline{e}^{\overline{\tau}}$ *ok, then* $\Gamma \vdash ([\overline{e}/x]\ e^\tau)$ *ok.*

Proof. The proof is by induction on the rules of the well-formedness judgment in the standard way. □

Lemma A.1.3. *If* $\Gamma \vdash e^\tau$ *ok and* $x \notin (\mathrm{dom}(\Gamma) \cup \mathrm{FV}(\mathrm{e}^\tau))$*, then* $\Gamma[x^{\overline{\tau}}] \vdash e^\tau$ *ok.*

Proof. The proof is by induction on $\Gamma \vdash e^\tau$ *ok*. □

Lemma A.1.4 (weakening)**.** *If* $\Gamma[x^{\overline{\tau}}] \vdash e^\tau$ *ok and* $x \notin \mathrm{FV}(\mathrm{e}^\tau)$*, then* $\Gamma \vdash e^\tau$ *ok.*

Proof. The proof is by induction on $\Gamma \vdash e^\tau$ *ok*. □

Lemma A.1.5. *If* $x \notin \mathrm{FV}(\mathrm{t})$*, then* $x \notin \mathrm{FV}([x/\mathrm{t}]\mathrm{e})$*)*.

Proof. The proof is by induction on the definition of substitution. □

Lemma A.1.6. *If* $\vdash \tau \rhd \overline{\tau}$ *ok, then* $\vdash\ !\,\tau \rhd\ !\,\overline{\tau}$ *ok.*

Proof. For convenience, we repeat the definition of $!\,\tau$.

$$\begin{array}{rcl} !\,[\tau] & = & \tau \\ !\,\{\tau\ ;\ \mathit{lf}\} & = & \tau \\ !\,\{\tau\ ;\ \mathit{nd}(\nu)\,\} & = & \{\tau\ ;\ \nu\,\} \\ !\,(\tau_1,\tau_2) & = & (!\,\tau_1, !\,\tau_2) \end{array}$$

Let *select types* be the name for types τ for which $!\,\tau$ is defined.

By inspection of the definition of !, select types are exactly either parallel array types, flattened array types, of pairs of select types, although the inner component τ is unrestricted.

The proof is by induction over the height of all well-formedness judgments on select types. We do not consider rule CO-FUN (! is undefined on function types).

(base case) CO-ID Immediate.

(base case) CO-FL For $\vdash\ [\tau] \triangleright \{\tau\ ;\ \mathit{lf}\}\ \mathit{ok}$, $!\ [\tau] = \tau$ and $!\ \{\tau\ ;\ \mathit{lf}\} = \tau$, so the conclusion is immediate from CO-ID.

(base case) CO-FR Similar to the CO-FL case.

(base case) CO-LL For $\vdash\ \{\tau\ ;\ \mathit{nd}(\nu)\,\} \triangleright \{\{\tau\ ;\ \nu\,\}\ ;\ \mathit{lf}\}\ \mathit{ok}$, $!\ \{\tau\ ;\ \mathit{nd}(\nu)\,\} = \{\tau\ ;\ \nu\,\}$ and $!\ \{\{\tau\ ;\ \nu\,\}\ ;\ \mathit{lf}\} = \{\tau\ ;\ \nu\,\}$.

(base case) CO-LR Similar to the CO-LL case.

(base cases) CO-ZFL and CO-ZFR The same as CO-LL and CO-LR, *mutatis mutandis*.

(base case) CO-ZL For $\vdash\ [(\tau_1,\tau_2)] \triangleright ([\tau_1],[\tau_2])\ \mathit{ok}$, $!\ [(\tau_1,\tau_2)] = (\tau_1,\tau_2)$ and $!\ ([\tau_1],[\tau_2]) = (!\ [\tau_1], !\ [\tau_2]) = (\tau_1,\tau_2)$. The conclusion is immediate from CO-ID.

(base case) CO-ZR Similar to the CO-ZR case.

CO-A By definition of ! for $[\tau]$ and $[\overline{\tau}]$ and induction on the premise of CO-A.

CO-F Similar to the CO-A case.

CO-PAIR By induction on the premises.

CO-TRANS By induction on the premises.

$\square$

Lemma A.1.7. *For τ flat where $\tau = \{\tau' \,;\, \nu\}$ or $\tau = (\tau_1, \tau_2)$, $\mathbf{N}[\![\tau]\!]$ flat.*

Proof. The proof is by induction on argument to $\mathbf{N}$.

(base case) $\{\tau \,;\, \nu\}$ We have $\{\tau \,;\, \nu\}$ flat.

$\mathbf{N}[\![\{\tau \,;\, \nu\}]\!] = \{\tau \,;\, \mathit{nd}(\nu)\}$ is flat by definition.

(τ_1, τ_2) We have (τ_1, τ_2) flat.

By induction, $\mathbf{N}[\![\tau_1]\!]$ and $\mathbf{N}[\![\tau_2]\!]$ are flat, so $(\mathbf{N}[\![\tau_1]\!], \mathbf{N}[\![\tau_2]\!])$ is flat by definition.

$\square$

Lemma A.1.8. *For type $\tau = \{\tau' \,;\, \nu\}$ or $\tau = (\tau_1, \tau_2)$, $\vdash [\tau] \triangleright \mathbf{N}[\![\tau]\!]$ ok.*

Proof. The proof is by induction on the definition of $\mathbf{N}$.

(base case) $\{\tau \,;\, \nu\}$ By definition, $\mathbf{N}[\![\{\tau \,;\, \nu\}]\!] = \{\tau \,;\, \mathit{nd}(\nu)\}$.

By CO-FL, CO-LR, and CO-TRANS, $\vdash [\{\tau \,;\, \nu\}] \triangleright \{\tau \,;\, \mathit{nd}(\nu)\}$ *ok*.

(τ_1, τ_2) By definition, $\mathbf{N}[\![(\tau_1, \tau_2)]\!] = (\mathbf{N}[\![\tau_1]\!], \mathbf{N}[\![\tau_2]\!])$.

By the inductive hypothesis, $\vdash [\tau_1] \triangleright \mathbf{N}[\![\tau_1]\!]$ *ok* and $\vdash [\tau_2] \triangleright \mathbf{N}[\![\tau_2]\!]$ *ok*.

Then $\vdash ([\tau_1], [\tau_2]) \triangleright (\mathbf{N}[\![\tau_1]\!], \mathbf{N}[\![\tau_2]\!])$ *ok* by CO-PAIR.

By CO-ZL, $\vdash [(\tau_1, \tau_2)] \triangleright ([\tau_1], [\tau_2])$ *ok*.

So, by CO-TRANS, $\vdash [(\tau_1, \tau_2)] \triangleright (\mathbf{N}[\![\tau_1]\!], \mathbf{N}[\![\tau_2]\!])$ *ok*.

$\square$

A.2 Lemmas and Theorems from the Main Document

Lemma 4.1.1 (well-formed coercions are invertible, p. 47)**.**

If $\vdash \tau \triangleright \overline{\tau}$ *ok, then* $\vdash \overline{\tau} \triangleright \tau$ *ok.*

Proof. The proof is by induction over the heights of the judgments in Figure 4.3.

base cases CO-ID is immediate. The following pairs of rules proves the claim for one another:

CO-FL and CO-FR; CO-LL and CO-LR; CO-ZL and CO-ZR; CO-ZFL and CO-ZFR.

CO-A . We have $\vdash [\tau] \triangleright [\overline{\tau}]$ *ok.*

From the premise, $\vdash \tau \triangleright \overline{\tau}$ *ok.*

By inductive hypothesis on the premise, $\vdash \overline{\tau} \triangleright \tau$ *ok.*

Then by CO-A, $\vdash [\overline{\tau}] \triangleright [\tau]$ *ok.*

CO-F . Identical to case CO-A, *mutatis mutandis*.

CO-FUN . We have $\vdash \tau_1 \to \tau_2 \triangleright \overline{\tau}_1 \to \overline{\tau}_2$ *ok*

From the premises, $\vdash \tau_1 \triangleright \overline{\tau}_1$ *ok* and $\vdash \tau_2 \triangleright \overline{\tau}_2$ *ok.*

By inductive hypotheses on the premises, $\vdash \overline{\tau}_1 \triangleright \tau_1$ *ok* and $\vdash \overline{\tau}_2 \triangleright \tau_2$ *ok.*

Then by CO-FUN, $\vdash \overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2$ *ok*

CO-PAIR We have $\vdash (\tau_1, \tau_2) \triangleright (\overline{\tau}_1, \overline{\tau}_2)$ *ok*

From the premises, $\vdash \tau_1 \triangleright \overline{\tau}_1$ *ok* and $\vdash \tau_2 \triangleright \overline{\tau}_2$ *ok.*

By inductive hypotheses on the premises, $\vdash \overline{\tau}_1 \triangleright \tau_1$ *ok* and $\vdash \overline{\tau}_2 \triangleright \tau_2$ *ok.*

Then by CO-PAIR, $\vdash (\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2)$ *ok.*

CO-TRANS We have $\vdash \tau_1 \triangleright \tau_3$ *ok.* From the premises, $\vdash \tau_1 \triangleright \tau_2$ *ok* and $\vdash \tau_2 \triangleright \tau_3$ *ok.* By induction, $\vdash \tau_3 \triangleright \tau_2$ *ok* and $\vdash \tau_2 \triangleright \tau_1$ *ok.* Then by CO-TRANS, $\vdash \tau_3 \triangleright \tau_1$ *ok.*

□

Lemma 4.1.2 (coercions within **A**, p. 51)**.**

If τ_1 **A** τ*, then* τ_2 **A** $\tau \Leftrightarrow \vdash \tau_1 \triangleright \tau_2$ *ok.*

Proof. The proof follows immediately from Lemmas A.2.1 and A.2.1 below. □

Lemma A.2.1. *If* τ_1 **A** τ *and* τ_2 **A** τ*, then* $\vdash \tau_1 \triangleright \tau_2$ *ok.*

Proof. We have τ_1 **A** τ and τ_2 **A** τ.

By the premise of ARRAY-OF, we have τ_1 **L** $[\tau]$ and τ_2 **L** $[\tau]$.

By L-SYMM, τ_2 **L** $[\tau] \implies [\tau]$ **L** τ_2.

So, by L-TRANS, τ_1 **L** τ_2.

Therefore, to prove the lemma, it is sufficient to show τ_1 **L** $\tau_2 \implies \vdash \tau_1 \triangleright \tau_2$ *ok.*

The proof proceeds by induction on the judgments in **L**.

(base case) L-REFL In the case that $\tau_1 = \tau_2 = \tau'$, $\vdash \tau' \triangleright \tau'$ *ok* is immediate by CO-ID.

L-SYMM We know that τ_1 **L** τ_2. By the premise of L-SYMM, τ_2 **L** τ_1. By induction, $\vdash \tau_2 \triangleright \tau_1$ *ok.*

By Lemma 4.1.1, $\vdash \tau_1 \triangleright \tau_2$ *ok.*

L-TRANS By induction on the premises and CO-TRANS.

(base case) L-ZPR By CO-ZL.

(base case) L-LF By CO-LR.

(base case) L-ND By CO-NR.

L-PARR By induction on the premise and CO-A.

L-PF By induction on the premise and CO-FL.

L-FARR By induction on the premise and CO-F.

L-PAIR By induction on the premise and CO-PAIR.

L-FUN By induction on the premise and CO-FUN.

□

Lemma A.2.2. *If* τ_1 $\mathbf{A}$ τ *and* $\vdash \tau_1 \triangleright \tau_2$ *ok, then* τ_2 $\mathbf{A}$ τ*.*

Proof. The proof is by induction over the well-formedness judgments for coercions.

In the proof, we use $\overline{\tau}$ as the name of the type τ_1 and τ_2 are both related to.

As in Lemma A.2.1, we appeal to the only rule of $\mathbf{A}$ to reduce the problem to the following: if τ_1 $\mathbf{L}$ $\overline{\tau}$ and $\vdash \tau_1 \triangleright \tau_2$ *ok*, then τ_2 $\mathbf{L}$ $\overline{\tau}$.

CO-ID Immediate for $\vdash \tau_1 \triangleright \tau_1$ *ok*.

CO-FL We have $\vdash [\tau] \triangleright \{\tau\ ;\ \mathit{lf}\}$ *ok* and $[\tau]$ $\mathbf{L}$ $\overline{\tau}$.

By L-PF and L-TRANS, $\{\tau\ ;\ \mathit{lf}\}$ $\mathbf{L}$ $\overline{\tau}$.

CO-FR Symmetric to CO-FL.

CO-LL We have $\vdash \{\tau\ ;\ \mathit{nd}(\nu)\} \triangleright \{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\}$ *ok* and $\{\tau\ ;\ \mathit{nd}(\nu)\}$ $\mathbf{L}$ $\overline{\tau}$.

By L-TRANS and L-LF, $\{\{\tau\ ;\ \nu\}\ ;\ \mathit{lf}\}$ $\mathbf{L}$ $\overline{\tau}$.

CO-LR Symmetric to CO-LL.

CO-ZL and CO-ZR Like the previous pairs of cases, but making use of L-ZPR.

CO-ZFL and CO-ZFR Like the previous pair of cases, but making use of L-PF, L-PAIR, and L-ZPR.

CO-A We have $\vdash [\tau_1] \triangleright [\tau_2]$ *ok* and $[\tau_1]$ $\mathbf{L}$ $\overline{\tau}$.

By CO-A, $\vdash \tau_1 \triangleright \tau_2$ *ok.*

By L-PARR and L-TRANS, $[\tau_2]$ $\mathbf{L}$ $\overline{\tau}$.

CO-F Like the previous case, but making use of L-FARR.

CO-FUN We have $\vdash \tau_1 \to \tau_2 \triangleright \overline{\tau}_1 \to \overline{\tau}_2$ *ok* and $\tau_1 \to \tau_2 \ \mathbf{L}\ \overline{\tau}$.

By CO-FUN, $\vdash \tau_1 \triangleright \overline{\tau}_1$ *ok* and $\vdash \tau_2 \triangleright \overline{\tau}_2$ *ok*.

Then by L-FUN, $\overline{\tau}_1 \to \overline{\tau}_2 \ \mathbf{L}\ \overline{\tau}$.

CO-PAIR Similar to CO-FUN, using L-PAIR in place of L-FUN.

CO-TRANS We have $\vdash \tau_1 \triangleright \tau_3$ *ok* and $\tau_1 \ \mathbf{L}\ \overline{\tau}$.

By inductive hypotheses on the premises of CO-TRANS, $\tau_1 \ \mathbf{L}\ \tau_2$ and $\tau_2 \ \mathbf{L}\ \tau_3$ for some τ_2.

Then by L-TRANS, $\tau_3 \ \mathbf{L}\ \overline{\tau}$.

□

Theorem 4.1.1 ($\mapsto$ preserves types, p. 54)**.**

If $\Gamma \vdash e^\tau$ *ok and* $e^\tau \mapsto^* \overline{e}^\tau$*, then* $\Gamma \vdash \overline{e}^\tau$ *ok.*

Proof. The proof proceeds by examining each rule in the flattening relation. Assume that for some Γ we have $\Gamma \vdash {e_0}^{\tau_0}$ *ok* and ${e_0}^{\tau_0} \mapsto {\overline{e}_0}^{\tau_0}$. We need to show that $\Gamma \vdash {\overline{e}_0}^{\tau_0}$ *ok*.

CD-IDI We assume $\Gamma \vdash {e_0}^{\tau_0}$ *ok*.

By definition, $\vdash (\tau_0 \triangleright \tau_0)$ *ok*.

Then by OK-COERCE and OK-APP, $\Gamma \vdash ((\tau_0 \triangleright \tau_0)\ e_0)^{\tau_0}$ *ok*.

CD-IDE We assume $\Gamma \vdash ((\tau_0 \triangleright \tau_0)\ e_0)^{\tau_0}$ *ok*.

Then by OK-APP, $\Gamma \vdash {e_0}^{\tau_0}$ *ok*.

CD-CI Let $\tau_0 = \tau_1 \to \tau_3$ and $e_0 = \tau_1 \triangleright \tau_3$. We assume $\vdash \tau_1 \triangleright \tau_3$ *ok*, and $\vdash \tau_1 \triangleright \tau_2$ *ok*.

By OK-COERCE, $\Gamma \vdash (\tau_1 \triangleright \tau_2)^{\tau_1 \to \tau_2}$ *ok* and $\Gamma \vdash (\tau_2 \triangleright \tau_3)^{\tau_2 \to \tau_3}$ *ok*.

Then by OK-COMP, $\Gamma \vdash ((\tau_2 \triangleright \tau_3) \circ (\tau_1 \triangleright \tau_2))^{\tau_1 \to \tau_3}$ *ok*.

CD-CE We assume $\vdash (\tau_2 \rhd \tau_3)$ *ok* and $\vdash (\tau_1 \rhd \tau_2)$ *ok*.

By OK-COERCE, $\Gamma \vdash (\tau_2 \rhd \tau_3)^{\tau_2 \to \tau_3}$ *ok* and $\Gamma \vdash (\tau_1 \rhd \tau_2)^{\tau_1 \to \tau_2}$ *ok*.

Then by OK-COMP, $\Gamma \vdash (\tau_1 \rhd \tau_3)^{\tau_1 \to \tau_3}$ *ok*.

CD-CU Let $t_1 = {e_1}^{\tau_1 \to \tau_0}$, $t_2 = {e_2}^{\tau_2 \to \tau_1}$, and $t_3 = {e_3}^{\tau_2}$.

We assume $\Gamma \vdash t_1$ *ok*, $\Gamma \vdash t_2$ *ok*, $\Gamma \vdash t_3$ *ok*, and $\Gamma \vdash (t_1\ (t_2\ t_3))^{\tau_0}$ *ok*.

By OK-COMP, $\Gamma \vdash (t_1 \circ t_2)^{\tau_2 \to \tau_0}$ *ok*.

Then by OK-APP, $\Gamma \vdash ((t_1 \circ t_2)\ t_3)^{\tau_0}$ *ok*.

CD-CF We assume $\Gamma \vdash ((t_1 \circ t_2)\ t_3)^{\tau_0}$ *ok*.

By OK-APP, $\Gamma \vdash t_3$ *ok* and $\Gamma \vdash (t_1 \circ t_2)^{\tau_2 \to \tau_0}$ *ok*, where $t_3 = {e_3}^{\tau_2}$.

By OK-COMP, $\Gamma \vdash t_1$ *ok* and $\Gamma \vdash t_2$ *ok*, where $t_1 = {e_1}^{\tau_1 \to \tau_0}$ and $t_2 = {e_2}^{\tau_2 \to \tau_1}$.

By OK-APP, $\Gamma \vdash (t_2\ t_3)^{\tau_1}$ *ok*.

Then by OK-APP, $\Gamma \vdash (t_1\ (t_2\ t_3))^{\tau_0}$ *ok*.

CD-PAIR Let $t_1 = {e_1}^{\overline{\tau}_1}$ and $t_2 = {e_2}^{\overline{\tau}_2}$.

We assume $\Gamma \vdash (((\overline{\tau}_1 \rhd \tau_1)\ t_1)^{\tau_1}, ((\overline{\tau}_2 \rhd \tau_2)\ t_2)^{\tau_2})^{(\tau_1, \tau_2)}$ *ok*.

By OK-PAIR, $\Gamma \vdash ((\overline{\tau}_1 \rhd \tau_1)\ t_1)^{\tau_1}$ *ok*, and $\Gamma \vdash ((\overline{\tau}_2 \rhd \tau_2)\ t_2)^{\tau_2}$ *ok*.

By OK-APP, $\Gamma \vdash (\overline{\tau}_1 \rhd \tau_1)^{\overline{\tau}_1 \to \tau_1}$ *ok*, $\Gamma \vdash (\overline{\tau}_2 \rhd \tau_2)^{\overline{\tau}_2 \to \tau_2}$ *ok*, $\Gamma \vdash {e_1}^{\overline{\tau}_1}$ *ok*, and $\Gamma \vdash {e_2}^{\overline{\tau}_2}$ *ok*.

By OK-PAIR, $\Gamma \vdash (t_1, t_2)^{(\overline{\tau}_1, \overline{\tau}_2)}$ *ok*.

By CO-PAIR, $\vdash (\overline{\tau}_1, \overline{\tau}_2) \rhd (\tau_1, \tau_2)$ *ok*.

Then, by OK-APP, $\Gamma \vdash (((\overline{\tau}_1, \overline{\tau}_2) \rhd (\tau_1, \tau_2))\ (t_1, t_2))^{(\tau_1, \tau_2)}$ *ok*.

CD-FST Let $t_1 = {e_1}^{\overline{\tau}_1}$ and $t_2 = {e_2}^{\tau_2}$.

We assume $\Gamma \vdash (((\overline{\tau}_1 \rhd \tau_1)\ t_1)^{\tau_1}, t_2)^{(\tau_1, \tau_2)}$ *ok*.

By OK-PAIR, $\Gamma \vdash ((\overline{\tau}_1 \triangleright \tau_1)\ t_1)^{\tau_1}$ *ok*, and $\Gamma \vdash {t_2}^{\tau_2}$ *ok*.

By OK-APP, $\Gamma \vdash (\overline{\tau}_1 \triangleright \tau_1)^{\overline{\tau}_1 \to \tau_1}$ *ok* and $\Gamma \vdash {e_1}^{\overline{\tau}_1}$ *ok*.

By OK-PAIR, $\Gamma \vdash (t_1, t_2)^{(\overline{\tau}_1, \tau_2)}$ *ok*.

Then by OK-PROJ for $i = 1$, $\Gamma \vdash \pi_1\ (t_1, t_2)^{\overline{\tau}_1}$ *ok*.

Then by OK-APP, $\Gamma \vdash ((\overline{\tau}_1 \triangleright \tau_1)\ \pi_1\ (t_1, t_2))^{\tau_1}$ *ok*.

CD-SND The same as the CD-FST case, with appropriate modifications.

CD-IF Let $t_1 = {e_1}^{bool}$, $t_2 = {e_2}^{\overline{\tau}_0}$, and $t_3 = {e_3}^{\overline{\tau}_0}$.

We assume $\Gamma \vdash (\textbf{if}\ t_1\ \textbf{then}\ ((\overline{\tau}_0 \triangleright \tau_0)\ t_2)^{\tau_0}\ \textbf{else}\ ((\overline{\tau}_0 \triangleright \tau_0)\ t_3)^{\tau_0})^{\tau_0}$ *ok*.

By OK-IF, $\Gamma \vdash {e_1}^{bool}$ *ok*, $\Gamma \vdash ((\overline{\tau}_0 \triangleright \tau_0)\ t_2)^{\tau_0}$ *ok*, and $\Gamma \vdash ((\overline{\tau}_0 \triangleright \tau_0)\ t_3)^{\tau_0}$ *ok*.

By OK-APP, $\Gamma \vdash (\overline{\tau}_0 \triangleright \tau_0)^{\overline{\tau}_0 \to \tau_0}$ *ok*, $\Gamma \vdash {e_2}^{\overline{\tau}_0}$ *ok*, and $\Gamma \vdash {e_3}^{\overline{\tau}_0}$ *ok*.

Then by OK-IF, $\Gamma \vdash (\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3)^{\overline{\tau}_0}$ *ok*.

Then by OK-APP, $\Gamma \vdash ((\overline{\tau}_0 \triangleright \tau_0)\ ((\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3)^{\overline{\tau}_0}))^{\tau_0}$ *ok*.

CD-APP Let $t_1 = {e_1}^{\overline{\tau}_1 \to \overline{\tau}_2}$ and $t_2 = {e_2}^{\overline{\tau}_1}$.

We have $\Gamma \vdash (((\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2)\ t_1)\ ((\overline{\tau}_1 \triangleright \tau_1)\ t_2))^{\tau_2}$ *ok*.

By OK-APP and OK-COERCE,

- $\vdash \overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2$ *ok*,
- $\vdash \overline{\tau}_1 \triangleright \tau_1$ *ok*,
- $\Gamma \vdash {e_1}^{\overline{\tau}_1 \to \overline{\tau}_2}$ *ok*, and
- $\Gamma \vdash {e_2}^{\overline{\tau}_1}$ *ok*.

By CO-FUN, $\vdash \overline{\tau}_2 \triangleright \tau_2$ *ok*.

By OK-APP, $\Gamma \vdash (t_1\ t_2)^{\overline{\tau}_2}$ *ok*.

By OK-COERCE and OK-APP, $\Gamma \vdash ((\overline{\tau}_2 \triangleright \tau_2)\ (t_1\ t_2))^{\tau_2}$ *ok*.

CD-COMP The distinct part of proving this case is establishing $\vdash \overline{\tau}_1 \to \overline{\tau}_3 \triangleright \tau_1 \to \tau_3$ *ok*.

We have $((\overline{\tau}_2 \to \overline{\tau}_3 \triangleright \tau_2 \to \tau_3)\ t_1 \circ (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2)\ t_2)^{\tau_1 \to \tau_3}$.

By OK-COMP, OK-APP, and OK-COERCE, $\vdash \overline{\tau}_2 \to \overline{\tau}_3 \triangleright \tau_2 \to \tau_3$ *ok*.

Then by CO-FUN, $\vdash \overline{\tau}_3 \triangleright \tau_3$ *ok*.

Similarly, by OK-COMP, OK-APP, and OK-COERCE, $\vdash \overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2$ *ok*.

Then by CO-FUN, $\vdash \overline{\tau}_1 \triangleright \tau_1$ *ok*.

The by CO-FUN, $\vdash \overline{\tau}_1 \to \overline{\tau}_3 \triangleright \tau_1 \to \tau_3$ *ok*.

The rest follows.

CD-ARR We have $\Gamma \vdash [(\overline{\tau} \triangleright \tau)\ t_1, \ldots, (\overline{\tau} \triangleright \tau)\ t_n]^{[\tau]}$ *ok*.

By OK-ARR, OK-APP, and OK-COERCE, we know that $\Gamma \vdash t_i^{\overline{\tau}}$ *ok* for $1 \leq i \leq n$, and $\vdash \overline{\tau} \triangleright \tau$ *ok*.

Then, by OK-PARR, $\Gamma \vdash [t_1, \ldots, t_n]^{[\overline{\tau}]}$ *ok*.

By CO-A, $\vdash [\overline{\tau}] \triangleright [\tau]$ *ok*.

So, by OK-COERCE and OK-APP, $\Gamma \vdash (([\overline{\tau}] \triangleright [\tau])\ [t_1, \ldots, t_n])^{[\tau]}$ *ok*.

CD-FARR Similar to CD-ARR, *mutatis mutandis*.

CD-LET Similar to the OK-IF case.

CD-FUN Similar to the OK-IF case.

CD-VAR We have $\Gamma \vdash x^{\tau}$ *ok*.

From the premise, $\vdash \overline{\tau} \triangleright \tau$ *ok*.

By OK-APP, $\Gamma \vdash ((\overline{\tau} \triangleright \tau)\ x)^{\tau}$ *ok*.

CD-CTXT By Lemma A.1.1.

CD-SUB Let $t_1 = {e_1}^{\tau}$.

We have $\Gamma \vdash (t_1 \;!_{\tau}\; t_2)^{!\,\tau}$ *ok*.

From the premise, $\vdash \tau \rhd \overline{\tau}$ *ok*.

By OK-SUB, $\Gamma \vdash {e_1}^{\tau}$ *ok*.

By OK-APP and OK-SUB, $\Gamma \vdash ((\tau \rhd \overline{\tau})\; t_1 \;!_{\overline{\tau}}\; t_2)^{!\,\overline{\tau}}$ *ok*.

By Lemmas 4.1.1 and A.1.6, $\vdash !\; \overline{\tau} \rhd !\; \tau$ *ok*.

By OK-APP, $\Gamma \vdash ((!\; \overline{\tau} \rhd !\; \tau)\; ((\tau \rhd \overline{\tau})\; t_1 \;!_{\overline{\tau}}\; t_2))^{!\,\tau}$ *ok*.

CD-FILT Let $t_1 = {e_1}^{\tau_1 \to bool}$ and $t_2 = {e_2}^{\tau_2}$.

We have $\Gamma \vdash (\mathbf{filt}_{(\tau_1, \tau_2)}\; (t_1, t_2))^{\tau_2}$ *ok*.

From the premise, $\overline{\tau}_2 \;\mathbf{A}\; \tau_1$.

By OK-FILT, $\Gamma \vdash {e_1}^{\tau_1 \to bool}$ *ok*, $\Gamma \vdash {e_2}^{\tau_2}$ *ok*, and $\tau_2 \;\mathbf{A}\; \tau_1$.

By $\tau_2 \;\mathbf{A}\; \tau_1$ and $\overline{\tau}_2 \;\mathbf{A}\; \tau_1$ and Lemma 4.1.2, $\vdash \overline{\tau}_2 \rhd \tau_2$ *ok* and $\vdash \tau_2 \rhd \overline{\tau}_2$ *ok*.

By OK-APP and OK-FILT, $\Gamma \vdash (\mathbf{filt}_{(\tau_1, \overline{\tau}_2)}\; (t_1, (\tau_2 \rhd \overline{\tau}_2)\; t_2))^{\overline{\tau}_2}$ *ok*

By OK-COERCE and OK-APP, $\Gamma \vdash ((\overline{\tau}_2 \rhd \tau_2)\; (\mathbf{filt}_{(\tau_1, \overline{\tau}_2)}\; (t_1, (\tau_2 \rhd \overline{\tau}_2)\; t_2)))^{\tau_2}$ *ok*.

CD-MAP Let $t_1 = {e_1}^{\tau_1 \to \tau_2}$ and $t_2 = {e_2}^{\tau_3}$.

We have $\Gamma \vdash (\mathbf{map}_{(\tau_1, \tau_2, \tau_3, \tau_4)}\; (t_1, t_2))^{\tau_4}$ *ok*.

By OK-MAP, we have $\tau_3 \;\mathbf{A}\; \tau_1$ and $\tau_4 \;\mathbf{A}\; \tau_2$.

From the premises of CD-MAP, we have $\overline{\tau}_3 \;\mathbf{A}\; \tau_1$ and $\overline{\tau}_4 \;\mathbf{A}\; \tau_2$.

By Lemmas 4.1.2 and 4.1.1, we have $\vdash \tau_3 \rhd \overline{\tau}_3$ *ok* and $\vdash \overline{\tau}_4 \rhd \tau_4$ *ok*.

By the premise of OK-MAP, $\Gamma \vdash {e_2}^{\tau_3}$ *ok*.

Then by OK-COERCE and OK-APP, we have $\Gamma \vdash (\tau_3 \rhd \overline{\tau}_3)\; t_2$ *ok*.

By OK-COERCE, OK-APP, and OK-MAP, $\Gamma \vdash (\mathbf{map}_{(\tau_1, \tau_2, \overline{\tau}_3, \overline{\tau}_4)}\; (t_1, (\tau_3 \rhd \overline{\tau}_3)\; t_2))^{\overline{\tau}_4}$ *ok*.

Then by OK-COERCE and OK-APP, $\Gamma \vdash ((\overline{\tau}_4 \triangleright \tau_4)\ (\mathbf{map}_{(\tau_1,\tau_2,\overline{\tau}_3,\overline{\tau}_4)}\ (t_1, (\tau_3 \triangleright \overline{\tau}_3)\ t_2)))^{\tau_4}$ *ok*.

CD-RED Similar to CD-FILT.

We have $\Gamma \vdash (\mathbf{red}_{(\tau_1,\tau_2)}\ (t_1, t_2, t_3))^{\tau_1}$ *ok*.

By the OK-RED and CD-RED, $\tau_2\ \mathbf{A}\ \tau_1$ and $\overline{\tau}_2\ \mathbf{A}\ \tau_1$.

Then by Lemma 4.1.2, $\vdash (\tau_2 \triangleright \overline{\tau}_2)$ *ok*.

By OK-COERCE and OK-APP, $\Gamma \vdash ((\tau_2 \triangleright \overline{\tau}_2)\ t_2)^{\overline{\tau}_2}$ *ok*, and the rest follows.

CD-LET-PROP Let $t_1 = {e_1}^{\tau_1}$, $t_2 = {e_2}^{\tau_0}$, and ${e_0}^{\tau_0} = (\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)^{\tau_0}$.

We assume $\vdash \tau_1 \triangleright \overline{\tau}_1$ *ok*, $\Gamma \vdash {e_0}^{\tau_0}$ *ok*. By CD-LET-PROP, we have ${e_0}^{\tau}\!\mapsto\!{\overline{e}_0}^{\tau_0}$ where

$${\overline{e}_0}^{\tau_0} = (\mathbf{let}\ \overline{x} = ((\tau_1 \triangleright \overline{\tau}_1)\ ({e_1}^{\tau_1}))^{\overline{\tau}_1}\ \mathbf{in}\ [x/(\overline{\tau}_1 \triangleright \tau_1)\ \overline{x}]({e_2}^{\tau_0}))^{\tau_0}$$

for $\overline{x}$ fresh.

The subexpressions of $\overline{e}_0$ are well-formed under Γ:

(a) By OK-LET, $\Gamma \vdash {e_1}^{\tau_1}$ *ok* and $\Gamma[x^{\tau_1}] \vdash {e_0}^{\tau_0}$ *ok*.

Then by OK-COERCE and OK-APP, $\Gamma \vdash ((\tau_1 \triangleright \overline{\tau}_1)\ ({e_1}^{\tau_1}))^{\overline{\tau}_1}$ *ok*.

(b) By OK-VAR, OK-COERCE, and OK-APP, $\Gamma[x^{\tau_1}][\overline{x}^{\overline{\tau}_1}] \vdash ((\overline{\tau}_1 \triangleright \tau_1)\ \overline{x}^{\overline{\tau}_1})^{\tau_1}$ *ok*.

Then by Lemma A.1.3, $\Gamma[x^{\tau_1}][\overline{x}^{\overline{\tau}_1}] \vdash {e_2}^{\tau_0}$ *ok*.

Then by Lemma A.1.2, $\Gamma[x^{\tau_1}][\overline{x}^{\overline{\tau}_1}] \vdash ([x/(\overline{\tau}_1 \triangleright \tau_1)\ \overline{x}]({e_2}^{\tau_0}))^{\tau_0}$ *ok*.

Then by Lemmas A.1.4 and A.1.5, $\Gamma[\overline{x}^{\overline{\tau}_1}] \vdash ([x/(\overline{\tau}_1 \triangleright \tau_1)\ \overline{x}]({e_2}^{\tau_0}))^{\tau_0}$ *ok*.

Then by (a) and (b) and OK-LET, $\Gamma \vdash {\overline{e}_0}^{\tau_0}$ *ok*.

CD-FUN-PROP Let $t_1 = e^{\tau_1}$ and $t_2 = e^{\tau_2}$.

We have $\Gamma \vdash (\mathbf{fun}\, f\, x^{\tau_0} = t_1\ \mathbf{in}\ t_2)^{\tau_2}$ *ok*.

From the premise, we have $\vdash \tau_0 \triangleright \overline{\tau}_0$ *ok*.

From OK-FUN, we have $\Gamma[x^{\tau_0}][f^{\tau_0 \to \tau_1}] \vdash {e_1}^{\tau_1}$ *ok* and $\Gamma[f^{\tau_0 \to \tau_1}] \vdash {e_2}^{\tau_2}$ *ok*.

Let $\overline{e}^{\tau_2} = (\mathbf{fun}\, \bar{f}\, \bar{x}^{\overline{\tau}_0} = [x/(\overline{\tau}_0 \triangleright \tau_0)\, \bar{x}][f/\bar{f} \circ (\tau_1 \triangleright \overline{\tau}_1)]\ t_1 \ \mathbf{in}\ [f/\bar{f} \circ (\tau_1 \triangleright \overline{\tau}_1)]\ t_2)^{\tau_2}$.

Let $\Gamma' = \Gamma[x^{\tau_1}][f^{\tau_0 \to \tau_1}][\bar{x}^{\overline{\tau}_1}][\bar{f}^{\overline{\tau}_1 \to \tau_2}]$.

We also know $\Gamma' \vdash ((\overline{\tau}_1 \triangleright \tau_1)\, \bar{x})^{\tau_1}$ *ok* and $\Gamma' \vdash ((\bar{f} \circ (\tau_1 \triangleright \overline{\tau}_1)))^{\tau_1 \to \tau_2}$ *ok*.

By Lemmas A.1.2, A.1.3, A.1.4, and A.1.5,

$$\begin{aligned}
&\Gamma[\bar{x}^{\overline{\tau}_1}][\bar{f}^{\overline{\tau}_0 \to \tau_1}] \vdash [x/(\overline{\tau}_0 \triangleright \tau_0)\, \bar{x}][f/\bar{f} \circ (\tau_1 \triangleright \overline{\tau}_1)]\ t_1\ \textit{ok} \\
&\Gamma[\bar{f}^{\overline{\tau}_0 \to \tau_1}] \vdash [f/\bar{f} \circ (\tau_1 \triangleright \overline{\tau}_1)]\ t_2\ \textit{ok}
\end{aligned}$$

Thus $\Gamma \vdash \overline{e}^{\tau_2}$ *ok* by OK-LET.

□

Lemma 4.2.1 (**F** maps source types to flat types, p. 59)**.**

For τ, a source type, $\mathbf{F}[\![\tau]\!]$ is flat.

Recall by definition a source type is one of those generated by the grammar

$$\tau ::= g \mid \tau \to \tau \mid (\tau, \tau) \mid [\tau]$$

By definition a flat type is either a ground type, a function type built of flat types, a pair type built of flat types, or a flattened array type $\{\tau\ ;\ \nu\}$ for τ not an array and not a pair.

Proof. The proof is by induction on the argument to **F**. The array case is expanded into four cases, for arrays containing ground types, function, pairs, and arrays.

(base case) ground types $\mathbf{F}[\![g]\!] = g$, g is flat by definition.

(base case) arrays of ground types $\mathbf{F}[\![[g]]\!] = \{g\ ;\ \mathit{lf}\}$, $\{g\ ;\ \mathit{lf}\}$ is flat by definition.

functions $\mathbf{F}[\![\tau_1 \to \tau_2]\!] = \mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]$. By induction, $\mathbf{F}[\![\tau_1]\!]$ and $\mathbf{F}[\![\tau_2]\!]$ are flat. Then the result $\mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]$ is flat by definition.

pairs Like the previous case, *mutatis mutandis*.

arrays of functions $\mathbf{F}[\![[\tau_1 \to \tau_2]]\!] = \{\mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]\ ;\ \mathit{lf}\}$ By induction, $\mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]$ is flat, so by definition $\{\mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]\ ;\ \mathit{lf}\}$ is flat.

arrays of pairs $\mathbf{F}[\![[(\tau_1, \tau_2)]]\!] = (\mathbf{F}[\![[\tau_1]]\!], \mathbf{F}[\![[\tau_2]]\!])$. By inductive hypotheses, both components of the result are flat, so the result is flat.

arrays of arrays $\mathbf{F}[\![[[\tau]]]\!] = \mathbf{N}[\![\mathbf{F}[\![[\tau]]\!]]\!]$.

By definition of $\mathbf{F}$ over arrays, $\mathbf{F}[\![[\tau]]\!]$ is either a flattened array or a pair.

By inductive hypothesis, $\mathbf{F}[\![[\tau]]\!]$ is flat.

The result is flat by Lemma A.1.7.

□

Lemma 4.2.2 (**F**'s coercions are well-formed, p. 60)**.**

If $\mathbf{F}[\![\tau]\!] = \overline{\tau}$, *then* $\vdash \tau \rhd \overline{\tau}$ *ok.*

Proof. The proof is by induction over the structure of source types.

(base case) g We have $\mathbf{F}[\![g]\!] = g$. $\vdash g \rhd g$ *ok* by CO-ID.

(base case) $[g]$ We have $\mathbf{F}[\![[g]]\!] = \{g\ ;\ \mathit{lf}\}$. $\vdash [g] \rhd \{g\ ;\ \mathit{lf}\}$ *ok* by CO-FL.

$\tau_1 \to \tau_2$ We have $\mathbf{F}[\![\tau_1 \to \tau_2]\!] = \mathbf{F}[\![\tau_1]\!] \to \mathbf{F}[\![\tau_2]\!]$.

Let $\overline{\tau}_1 = \mathbf{F}[\![\tau_1]\!]$ and $\overline{\tau}_2 = \mathbf{F}[\![\tau_2]\!]$.

By the inductive hypothesis on τ_1 and τ_2, $\vdash \tau_1 \rhd \overline{\tau}_1$ *ok* and $\vdash \tau_2 \rhd \overline{\tau}_2$ *ok.*

Then by CO-FUN, $\vdash \tau_1 \to \tau_2 \rhd \overline{\tau}_1 \to \overline{\tau}_2$ *ok.*

(τ_1, τ_2) We have $\mathbf{F}[\![(\tau_1, \tau_2)]\!] = (\mathbf{F}[\![\tau_1]\!], \mathbf{F}[\![\tau_2]\!])$.

Let $\overline{\tau}_1 = \mathbf{F}[\![\tau_1]\!]$ and $\overline{\tau}_2 = \mathbf{F}[\![\tau_2]\!]$.

By the inductive hypothesis on τ_1 and τ_2, $\vdash \tau_1 \triangleright \overline{\tau}_1$ *ok* and $\vdash \tau_2 \triangleright \overline{\tau}_2$ *ok*.

Then by CO-PAIR, $\vdash (\tau_1, \tau_2) \triangleright (\overline{\tau}_1, \overline{\tau}_2)$ *ok*.

$[\tau_1 \rightarrow \tau_2]$ We have $\mathbf{F}[\![[\tau_1 \rightarrow \tau_2]]\!] = \{\mathbf{F}[\![\tau_1 \rightarrow \tau_2]\!]\ ;\ \mathit{lf}\}$.

Let $\overline{\tau}_1 \rightarrow \overline{\tau}_2 = \mathbf{F}[\![\tau_1 \rightarrow \tau_2]\!]$.

By the inductive hypothesis, $\vdash \tau_1 \rightarrow \tau_2 \triangleright \overline{\tau}_1 \rightarrow \overline{\tau}_2$ *ok*.

Then by CO-FL, $\vdash [\tau_1 \rightarrow \tau_2] \triangleright \{\overline{\tau}_1 \rightarrow \overline{\tau}_2\ ;\ \mathit{lf}\}$ *ok*.

$[(\tau_1, \tau_2)]$ We have $\mathbf{F}[\![[(\tau_1, \tau_2)]]\!] = (\mathbf{F}[\![[\tau_1]]\!], \mathbf{F}[\![[\tau_2]]\!])$.

Let $\overline{\tau}_1 = \mathbf{F}[\![[\tau_1]]\!]$ and $\overline{\tau}_2 = \mathbf{F}[\![[\tau_2]]\!]$.

By the inductive hypthesis, $\vdash [\tau_1] \triangleright \overline{\tau}_1$ *ok* and $\vdash [\tau_2] \triangleright \overline{\tau}_2$ *ok*.

By CO-PAIR, $\vdash ([\tau_1], [\tau_2]) \triangleright (\overline{\tau}_1, \overline{\tau}_2)$ *ok*.

By CO-ZL, $\vdash [(\tau_1, \tau_2)] \triangleright ([\tau_1], [\tau_2])$ *ok*.

Then by CO-TRANS, $\vdash [(\tau_1, \tau_2)] \triangleright (\overline{\tau}_1, \overline{\tau}_2)$ *ok*.

$[[\tau]]$ We have $\mathbf{F}[\![[[\tau]]]\!] = \mathbf{N}[\![\mathbf{F}[\![[\tau]]\!]]\!]$

We need to show $\vdash [[\tau]] \triangleright \mathbf{N}[\![\mathbf{F}[\![[\tau]]\!]]\!]$ *ok*.

By inspection of the definition of **F** and Lemma 4.2.1, if the argument to **F** is an array type $[\tau]$, then **F** produces either a flat array type or a pair of flat types.

We need consider only these two cases.

$\{\overline{\tau}\ ;\ \nu\}$ Let $\mathbf{F}[\![[\tau]]\!] = \{\overline{\tau}\ ;\ \nu\}$.

We have $\mathbf{N}[\![\{\overline{\tau}\ ;\ \nu\}]\!] = \{\overline{\tau}\ ;\ \mathit{nd}(\nu)\}$.

To show: $\vdash [[\tau]] \triangleright \{\overline{\tau}\ ;\ \mathit{nd}(\nu)\}$ *ok*.

By the inductive hypothesis on **F**, $\vdash [\tau] \triangleright \{\overline{\tau}\ ;\ \nu\}$ *ok*.

By CO-FL, $\vdash [[\tau]] \triangleright \{\{\overline{\tau}\ ;\ \nu\}\ ;\ \mathit{lf}\}$ *ok*.

By CO-LR, $\vdash \{\{\overline{\tau}\ ;\ \nu\}\ ;\ \mathit{lf}\} \triangleright \{\overline{\tau}\ ;\ \mathit{nd}(\nu)\}\ \mathit{ok}$.

By CO-TRANS, $\vdash [[\tau]] \triangleright \{\overline{\tau}\ ;\ \mathit{nd}(\nu)\}\ \mathit{ok}$, which was to be shown.

$(\overline{\tau}_1, \overline{\tau}_2)$ Let $(\overline{\tau}_1, \overline{\tau}_2) = \mathbf{F}[\![[\tau]]\!]$.

We need $\vdash [[\tau]] \triangleright \mathbf{N}[\![(\overline{\tau}_1, \overline{\tau}_2)]\!]\ \mathit{ok}$.

By inductive hypothesis, we have $\vdash [\tau] \triangleright (\overline{\tau}_1, \overline{\tau}_2)\ \mathit{ok}$.

Then by CO-A, $\vdash [[\tau]] \triangleright [(\overline{\tau}_1, \overline{\tau}_2)]\ \mathit{ok}$.

By Lemma A.1.8 (see below), $\vdash [(\overline{\tau}_1, \overline{\tau}_2)] \triangleright \mathbf{N}[\![(\overline{\tau}_1, \overline{\tau}_2)]\!]\ \mathit{ok}$.

Then by CO-TRANS, we have $\vdash [[\tau]] \triangleright \mathbf{N}[\![(\overline{\tau}_1, \overline{\tau}_2)]\!]\ \mathit{ok}$.

□

Theorem 4.2.3 ($\mapsto$ encodes $\Downarrow$, p. 63)**.**

If $e^{\tau} \Downarrow ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$, *then* $e^{\tau} \mapsto^* ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$.

Proof. Given: $e^{\tau} \Downarrow ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$.

Then by FLATTEN, $\{\} \vdash e^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond \overline{e}^{\overline{\tau}}$.

Then, to show: $\{\} \vdash e^{\tau} \searrow (\overline{\tau} \triangleright \tau) \diamond \overline{e}^{\overline{\tau}} \implies e^{\tau} \mapsto^* ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^{\tau}$.

B-BASE To show: $b^{\tau} \mapsto^* ((\tau \triangleright \tau)\ b^{\tau})^{\tau}$

The conclusion is immediately true by CD-IDE.

B-VAR $x^{\tau} \mapsto ((\overline{\tau} \triangleright \tau)\ \overline{x}^{\overline{\tau}})^{\tau}$ immediate from CD-VAR.

B-IF We have $(\mathbf{if}\ {e_1}^{bool}\ \mathbf{then}\ {e_2}^{\tau}\ \mathbf{else}\ {e_3}^{\tau})^{\tau}$.

By the premises of the judgment, we have $\Delta \vdash {e_1}^{bool} \searrow (\mathit{bool} \triangleright \mathit{bool}) \diamond {\overline{e}_1}^{bool}$.

By induction and CD-IDE, we have ${e_1}^{bool} \mapsto^* (\mathit{bool} \triangleright \mathit{bool})\ {\overline{e}_1}^{bool} \mapsto {\overline{e}_1}^{bool}$.

By induction, we have ${e_2}^{\tau} \mapsto^* (\overline{\tau} \triangleright \tau)\ {\overline{e}_2}^{\overline{\tau}}$ and ${e_3}^{\tau} \mapsto^* (\overline{\tau} \triangleright \tau)\ {\overline{e}_3}^{\overline{\tau}}$.

By three applications of CD-CTXT we have

$$(\textbf{if } {e_1}^{bool} \textbf{ then } {e_2}^{\tau} \textbf{ else } {e_3}^{\tau})^{\tau} \mapsto^{*} (\textbf{if } {\overline{e}_1}^{bool} \textbf{ then } (\overline{\tau} \triangleright \tau)\, {\overline{e}_2}^{\overline{\tau}} \textbf{ else } (\overline{\tau} \triangleright \tau)\, {\overline{e}_3}^{\overline{\tau}})^{\tau}$$

Then by CD-IF we have

$$\begin{aligned}&(\textbf{if } {\overline{e}_1}^{bool} \textbf{ then } (\overline{\tau} \triangleright \tau)\, {\overline{e}_2}^{\overline{\tau}} \textbf{ else } (\overline{\tau} \triangleright \tau)\, {\overline{e}_3}^{\overline{\tau}})^{\tau} \\ &\quad \mapsto ((\overline{\tau} \triangleright \tau)\, (\textbf{if } {\overline{e}_1}^{bool} \textbf{ then } {\overline{e}_2}^{\overline{\tau}} \textbf{ else } {\overline{e}_3}^{\overline{\tau}}))^{\tau}\end{aligned}$$

which proves the case.

B-LET By induction on the premises and CD-LET to hoist coercion $\overline{\tau}_2 \triangleright \tau_2$ out of the scope of the let-expression.

B-FUN Similar to the previous case, but using CD-FUN.

B-PAIR By induction on the premises and CD-PAIR to factor the coercions out of the expessions in the pair.

B-FST By induction on the premise and CD-FST to hoist the coercion $\overline{\tau}_1 \triangleright \tau_1$ out of the first expression in the pair.

B-SND Similar to B-FST.

B-APP By induction on the premises and CD-APP.

B-ARR We have $\mathbf{F}[\![[\tau]]\!] = \overline{\tau}$.

By CO-ID, $\vdash \overline{\tau} \triangleright \overline{\tau}$ *ok*.

By Lemma 4.2.2, $\mathbf{F}[\![[\tau]]\!] = \overline{\tau} \implies \vdash [\tau] \triangleright \overline{\tau}$ *ok*.

By Lemma 4.1.1, $\vdash \overline{\tau} \triangleright [\tau]$ *ok*.

Therefore by CD-CI, $\overline{\tau} \triangleright \overline{\tau} \mapsto (\overline{\tau} \triangleright [\tau]) \circ ([\tau] \triangleright \overline{\tau})$.

By CD-IDI, $e^{\overline{\tau}} \mapsto (\overline{\tau} \triangleright \overline{\tau})\ e^{\overline{\tau}}$.

Since $\overline{\tau} \triangleright \overline{\tau} \mapsto (\overline{\tau} \triangleright [\tau]) \circ ([\tau] \triangleright \overline{\tau})$, by CD-CTXT, $(\overline{\tau} \triangleright \overline{\tau})\ e^{\overline{\tau}} \mapsto (((\overline{\tau} \triangleright [\tau]) \circ ([\tau] \triangleright \overline{\tau}))\ e^{\overline{\tau}})^{\overline{\tau}}$.

By CD-CF, $(((\overline{\tau} \triangleright [\tau]) \circ ([\tau] \triangleright \overline{\tau}))\ e^{\overline{\tau}})^{\overline{\tau}} \mapsto ((\overline{\tau} \triangleright [\tau])\ (([\tau] \triangleright \overline{\tau})\ e^{\overline{\tau}}))^{\overline{\tau}}$, which was to be shown.

B-COMP By induction on the premises and CD-COMP.

B-SUB By induction on the premises, we have ${e_1}^{\tau} \mapsto^* ((\overline{\tau} \triangleright \tau)\ {\overline{e}_1}^{\overline{\tau}})^{\tau}$.

We also have ${e_2}^{int} \mapsto^* ((int \triangleright int)\ {\overline{e}_2}^{int})^{int}$ which $\mapsto {\overline{e}_2}^{int}$ by CD-IDE.

By two applications of CD-CTXT, we step from the original term to

$$(((\overline{\tau} \triangleright \tau)\ {\overline{e}_1}^{\overline{\tau}})^{\tau}\ !_{\tau}\ {\overline{e}_2}^{int})^{!\,\tau}$$

By Lemma A.1.6, $\vdash\ !\ \overline{\tau} \triangleright !\ \tau\ ok$.

Then, by CD-SUB, we can step to

$$((!\ \overline{\tau} \triangleright !\ \tau)\ ((\tau \triangleright \overline{\tau})\ (\overline{\tau} \triangleright \tau)\ {\overline{e}_1}^{\overline{\tau}})^{\overline{\tau}}\ !_{\overline{\tau}}\ {\overline{e}_2}^{int})^{!\,\tau}$$

We cancel $\tau \triangleright \overline{\tau}$ and $\overline{\tau} \triangleright \tau$ by CD-CF, CD-CE, and CD-IDE to arrive at

$$((!\ \overline{\tau} \triangleright !\ \tau)\ ({\overline{e}_1}^{\overline{\tau}}\ !_{\overline{\tau}}\ {\overline{e}_2}^{int})^{!\,\overline{\tau}})^{!\,\tau}$$

B-MAP Similar to CD-SUB, using CD-MAP and Lemmas 4.2.2 and 4.1.2.

B-FILT Similar to CD-SUB, using CD-FILT and Lemma 4.1.2.

B-RED Similar to CD-SUB, using CD-RED and Lemma 4.1.2.

□

REFERENCES

[1] E. A. Abbott. *Flatland: A Romance of Many Dimensions*. 1884.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *SP&E*, 19(2):171–183, 1989.

[3] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011*, New York, NY, June 2011. ACM.

[4] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, Dec. 1986.

[5] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. In *ICFP '10*, pages 93–104, New York, NY, September 2010. ACM.

[6] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[7] G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of C.S., CMU, Pittsburgh, PA, Apr. 1993.

[8] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, 21(1):4–14, 1994.

[9] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *JPDC*, 8(2):119–134, 1990.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95*, pages 207–216, New York, NY, July 1995. ACM.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[12] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *SP&E*, 25(12):1315–1330, Dec. 1995.

[13] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, New York, NY, Oct. 1981. ACM.

[14] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *HIPS '04*, pages 52–60, Los Alamitos, CA, Apr. 2004. IEEE Computer Society Press.

[15] M. M. T. Chakravarty, R. L. eshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, pages 10–18, New York, NY, Jan. 2007. ACM.

[16] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In *ICFP '00*, pages 94–105, New York, NY, Sept. 2000. ACM.

[17] M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Euro-Par '01*, volume 2150 of *LNCS*, pages 524–534, New York, NY, Aug. 2001. Springer-Verlag.

[18] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial Vectorisation of Haskell Programs. In *DAMP '08*, New York, NY, Jan. 2008. ACM.

[19] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, pages 70–83, New York, NY, Jan. 1994. ACM.

[20] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL '93*, pages 113–123, New York, NY, Jan. 1993. ACM.

[21] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, pages 119–130, New York, NY, Sept. 2008. ACM.

[22] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, 2010. Accepted.

[23] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*, pages 37–44, New York, NY, Jan. 2007. ACM.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, New York, NY, June 1998.

[25] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2004.

[26] GHC. The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc`.

[27] R. H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*, pages 9–17, New York, NY, Aug. 1984. ACM.

[28] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[29] G. Huet. The zipper. *JFP*, 7(5):549–554, 1997.

[30] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 1999.

[31] G. Keller and M. M. T. Chakravarty. Flattening trees. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 709–719, London, UK, 1998. Springer-Verlag.

[32] X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92*, pages 177–188, New York, NY, 1992. ACM.

[33] R. Leshchinskiy. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 2005.

[34] R. Leshchinskiy, M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *ICCS '06*, number 3992 in LNCS, pages 920–928, New York, NY, May 2006. Springer-Verlag.

[35] MLton. The MLton Standard ML compiler. Available at `http://mlton.org`.

[36] R. S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[37] R. S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[38] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *FoMPP5*, pages 186–193, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[39] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *APLAS '08*, pages 138–138, New York, NY, Dec. 2008. Springer-Verlag.

[40] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233, Sept. 2001.

[41] J. F. Prins and D. W. Palmer. Transforming high-level data-parallel programs into vector operations. In *PPoPP '93*, pages 119–128, New York, NY, May 1993. ACM.

[42] M. Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, Aug. 2010. Available from `http://manticore.cs.uchicago.edu`.

[43] J. Reppy, C. Russo, and Y. Xiao. Parallel Concurrent ML. In *ICFP '09*, pages 257–268, New York, NY, August–September 2009. ACM.

[44] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[45] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, MA, 1989.

[46] V. Saraswat and N. Nystrom. Report on the experimental language X10. Technical report, IBM Research, 2005.

[47] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, School of C.S., CMU, Pittsburgh, PA, May 1991.

[48] G. L. Steele Jr. Parallel programming and code selection in Fortress. In *PPoPP '06*, page 1, New York, NY, Mar. 2006. ACM. Keynote talk; slides available from `http://research.sun.com/projects/plrg/CGOPPoPP2006public.pdf`.

[49] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*, pages 179–190, New York, NY, Jan. 2010. ACM.

[50] D. White. Mandelbulb: The Unravelling of the Real 3D Mandelbrot Fractal. `http://www.skytopia.com/project/fractal/mandelbulb.html`, viewed 24 June 2011.

[51] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227, New York, NY, USA, 1999. ACM.