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. You know as well as I do, this is for you.
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... vi
LIST OF TABLES ........................................................... vii
ACKNOWLEDGEMENTS ...................................................... 1
ABSTRACT ................................................................. 2

CHAPTER

1 INTRODUCTION .......................................................... 3
  1.1 Implicit Threading .................................................. 3
  1.2 Nested Data Parallelism ........................................... 5
  1.3 Contributions ...................................................... 5
  1.4 Outline of the Dissertation ....................................... 6

2 RELATED WORK .......................................................... 7
  2.1 NESL .............................................................. 7
  2.2 Proteus ............................................................ 9
  2.3 Nepal and Data Parallel Haskell .................................. 10
  2.4 Manticore and PML ................................................ 11

3 DISCUSSION ............................................................. 13
  3.1 Implicitly-Threaded Parallelism in PML ......................... 13
  3.2 The Manticore Infrastructure ..................................... 16
  3.3 Ropes and Rope Tabulation ........................................ 17
  3.4 PML and the Flattening Transformation ........................ 21
  3.5 Optimizations Enabled by Flattening ......................... 25
  3.6 Examples ........................................................ 30

4 FORMALIZATION .......................................................... 31
  4.1 The Model Language ............................................... 32
  4.2 A Fully-Flattening Transformation .............................. 37

5 IMPLEMENTATION ......................................................... 47
  5.1 Implementation Overview ......................................... 47
  5.2 Abstract Flattening ................................................. 49
  5.3 Flatten Operator Fusion ........................................... 50
  5.4 Concrete Flattening ............................................... 51
  5.5 Optimizations ...................................................... 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 EVALUATION</td>
<td>58</td>
</tr>
<tr>
<td>6.1 Flat Arithmetic</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Mandelbrot</td>
<td>60</td>
</tr>
<tr>
<td>6.3 Ray Tracing</td>
<td>61</td>
</tr>
<tr>
<td>6.4 Mandelbulb</td>
<td>62</td>
</tr>
<tr>
<td>6.5 Sparse Matrix Vector Multiplication</td>
<td>63</td>
</tr>
<tr>
<td>7 CONCLUSION</td>
<td>66</td>
</tr>
<tr>
<td>7.1 Future Work</td>
<td>66</td>
</tr>
</tbody>
</table>

APPENDIX

A PROOFS                      | 68   |
REFERENCES                    | 69   |
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Flattening a nested vector using a “lengths” segment descriptor.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Rope.tabFromTo</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Speedup of parallel tabFromTo over sequential PML.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Speedup of parallel tabFromTo over sequential C.</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Rope.tabFromToStep</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Flatland: terms</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Flatland: types</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Well-formedness judgment on terms (1 of 2). Superscript types have been elided from most subexpressions in the consequents of these rules.</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Well-formedness judgment on terms (2 of 2).</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Flattening types</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Flattening transformation (1 of 2).</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Flattening transformation (2 of 2).</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>Type-indexed family of data-flattening operators ( \text{fl}_{\rho} )</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Type-indexed family of nesting operators (inverse of data-flattening operators) ( \text{nest}_{\rho} )</td>
<td>45</td>
</tr>
<tr>
<td>4.10</td>
<td>Coercion operators</td>
<td>46</td>
</tr>
<tr>
<td>4.11</td>
<td>Cancelling coercion operators</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>pmlc</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>pmlc with flattening</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>PML code for flat-arith</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Speedup of flat-arith, with flattening transformation applied, vs. itself at one processor.</td>
<td>59</td>
</tr>
<tr>
<td>6.3</td>
<td>Speedup of flat-arith vs. sequential C baseline.</td>
<td>60</td>
</tr>
<tr>
<td>6.4</td>
<td>PML code for mandelbrot</td>
<td>60</td>
</tr>
<tr>
<td>6.5</td>
<td>Speedup of mandelbrot, with flattening transformation applied, vs. itself at one processor.</td>
<td>61</td>
</tr>
<tr>
<td>6.6</td>
<td>Speedup of mandelbrot vs. sequential C baseline.</td>
<td>62</td>
</tr>
<tr>
<td>6.7</td>
<td>PML code for raytracer</td>
<td>62</td>
</tr>
<tr>
<td>6.8</td>
<td>Speedup of raytracer, with flattening transformation applied, vs. itself at one processor.</td>
<td>63</td>
</tr>
<tr>
<td>6.9</td>
<td>Speedup of raytracer vs. sequential SML baseline.</td>
<td>64</td>
</tr>
<tr>
<td>6.10</td>
<td>PML code for mandelbulb</td>
<td>64</td>
</tr>
<tr>
<td>6.11</td>
<td>Speedup of mandelbulb, with flattening transformation applied, vs. itself at one processor.</td>
<td>65</td>
</tr>
<tr>
<td>6.12</td>
<td>Speedup of mandelbulb vs. sequential C baseline.</td>
<td>65</td>
</tr>
</tbody>
</table>
LIST OF TABLES

4.1 Relations defined in this chapter. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
ACKNOWLEDGEMENTS

Acknowledgements go here.
ABSTRACT

Parallel programming languages offer programmers different ways to manage parallelism in their applications. In an implicitly parallel language, programs say nothing about what is to be computed in parallel; the system makes decisions entirely on its own. In an explicitly-threaded parallel language, by contrast, the program states exactly how parallel computation will be conducted, by spawning and managing threads of computation with specific explicit operations. Implicit and explicitly-threaded programming language implementations place control of parallelism entirely in the hands of compiler writers, in the first case, and end-user programmers, in the second.

Implicitly-threaded languages inhabit the middle ground between these two extremes. Implicitly-threaded languages give the programmer mechanisms for hinting to the compiler which computations should run in parallel. The exact details of how many threads there are, which computation maps onto which thread, how computations are scheduled, and even the final decision about whether a computation will be run in parallel at all, are all determined by the system. We believe this paradigm empowers programmers to guide and tune the parallel execution of a program while freeing them from the considerable extra burden of managing possibly dozens, or more, parallel computational threads. Implicitly-threaded parallel programs are natural, concise, and portable.

In this dissertation, we discuss techniques for the implementation of languages with implicitly-threaded parallel features. In particular, we define a novel transformation on a model parallel language that flattens and monomorphizes nested data structures such that computing with them in parallel becomes substantially faster. Using these transformation rules, we have implemented the transformation as a pass in the compiler for Parallel ML (PML), a heterogeneous parallel language with implicitly-threaded features. In practice, transformed programs perform better than untransformed ones, and scale much better, and compete favorably against efficient sequential programs in C and SML. With our system, running PML programs on a 48-core machine yields as much as a thirtyfold improvement over their sequential counterparts.
Parallel machines are now inexpensive and widely available, but writing parallel programs is still hard. Programmers continue to use languages to write parallel programs that fail to give them adequate support. Better languages will increase programmer productivity and enable applications to make better use of the fast multicore machines that have become common.

The demand for good parallel languages has given rise to a diverse variety of programming language research: recent and active projects include NESL [6], Cilk [8], X10 [38], Chapel [10], Fortress [40], Data Parallel Haskell [14], and PML [17], to name a few. 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 a way 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. This dissertation describes novel compilation techniques that improve the performance of programs written with implicitly-threaded parallel constructs, and demonstrates their success with experimental results.

1.1 Implicit Threading

Implicit threading is a promising paradigm for high-level parallel programming. In a programming language that supports implicit threading, the programmer provides high-level annotations to indicate parts of the program that 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 given annotated computation, the implementation may choose to execute it in a completely parallel way, in a completely sequential way, or somewhere in between. Note that the compiler and the runtime system are free to ignore any suggestion of parallelism in
any program, as in any particular statically-unknown context, the overhead of parallel execution might actually make a program slower.

We believe implicitly-threaded parallelism is the most promising programming model we have. It affords the programmer freedom to focus on the high-level concepts of solving his or her target problem, without getting caught up in the complexities of managing parallelism directly. The present work consists of the development, implementation and evaluation of techniques for maximizing the parallelism available in implicitly-threaded programs while minimizing the concomitant overhead.

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 the overhead induced in creating and managing it; too little parallelism misses opportunities to exploit the resources available in the hardware. Finding the right amount of parallelism consistently and for a broad selection of programs is a hard problem. Good results in this area will have widespread impact.

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 contain no direct mention of parallel computation, so the system tries to identify opportunities for parallelism and exploit them. The pH language [30] exemplifies this approach. By contrast, in explicitly-threaded parallel languages, the programmer controls all the parallelism directly, specifying exactly when to spawn threads, when to synchronize on pending results, and so forth. Writing parallel programs in C or Java, for example, has this character. Note that programmers who make use of pthreads for parallel programming are using a tool whose original intent was concurrency, rather than parallelism, and there are a variety of resulting difficulties. Surely we can do better!

Implicit threading has neither the hands-off character of fully implicit parallelism, nor the direct, low-level quality of fully explicit parallelism. In the explicitly sequential parts of a program,
the system makes no attempt to identify any parallelism, and in the implicitly parallel parts, unless the system decides against parallelism, 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 easy to build upon.

1.2 Nested Data Parallelism

Nested data-parallelism (NDP) [6] is a declarative style for programming irregular parallel applications. NDP languages provide language features favoring the NDP style, efficient compilation of NDP programs, and various common NDP operations like parallel maps, filters, and sum-like reductions. Irregular parallelism is achieved by the fact that nested arrays need not have regular, or rectangular, structure; i.e., subarrays may have different lengths. NDP programming is supported by a number of different parallel programming languages [14, 20], including our own Parallel ML (PML) [17].

[[ NDP makes it possible to write efficient parallel programs in familiar idioms. ex: mandelbrot, smvm ]]
• a novel representation for flattened arrays, using ropes and shape trees,

• transformation rules for flattening on a model language, making use of the aforementioned representation,

• techniques for implementing the model transformation in a full-featured parallel programming language,

• discussion of optimizations made possible by having performed the transformation, and

• experimental results demonstrating the performance benefits achieved by this transformation and subsequent optimizations.

The present work is conducted as part of the Manticore project [19], which consists of a runtime system and a language design and implementation for Parallel ML [17] (PML). PML supports both implicitly- and explicitly-threaded parallelism, mixing the features of core Standard ML [28] with parallel channel-based concurrency from CML [36] and nested data parallelism in the style of NESL [6].

1.4 Outline of the Dissertation

We begin with a survey of related work in Chapter 2. Chapter 3 presents a high-level discussion of the system presented in this dissertation without full technical detail. Chapter 4 presents a formal model of the mechanisms developed for this dissertation. Chapter 5 discusses the implementation of the present system in the PML compiler, and Chapter 6 gives performance results across a variety of benchmarks. Chapter 7 summarizes and outlines future work.
CHAPTER 2
RELATED WORK

In this section, we consider the related work that precedes and informs the present research. We are especially interested in identifying the differences between approaches as they relate to this dissertation.

2.1 NESL

The programming language NESL [6] is the original source for using the flattening transformation to compile of nested data parallel languages. Guy Blelloch wrote about the flattening transformation in his doctoral thesis [4] and sketched a compilation strategy, including flattening, for an existing parallel language, PARALATION LISP [37]. NESL the language does not appear in the literature until later.

In Blelloch’s thesis, flattening of vectors\(^1\) is presented as transforming nested arrays into a representation combining an array of elements (the elements of the nested sequence in a flattened array, in left-to-right order) with some number of segment descriptors, which he defines as “any structure that defines the segmentation of a vector.” (In his work “any structure” is always itself a vector, but, taken literally, the definition is broad enough to include our shape tree structure, discussed below, as well.) In his regime, a flattened array carries one segment descriptor for each level of nesting. Thus, more deeply nested arrays in the surface language are represented by larger collections of arrays in the representation. (Later, in her NESL-inspired dissertation [23], Keller would also use segment descriptors in flattened representations.)

Blelloch’s thesis presents various alternatives for the contents of segment descriptors: lengths, head flags, and head pointers, which contain the lengths of the subsequences, booleans marking

---

1. Blelloch refers to sequences as vectors; we use the term “arrays” in the present work to match the terminology of more recent research.
segment beginnings, and indices of segment beginnings, respectively. An example of Blelloch’s
lengths-type segment descriptor is given in Figure 2.1; \( S \) is the nested vector and \( S' \) is its flattened
counterpart. This scheme is conceptually close to our shape tree representation.

[[ NDP and irregular parallelism ]]

Blelloch’s thesis was written well before multicore machines became so widespread and inex-
expensive. In characterizing the original motivation for his compilation techniques, Blelloch wrote
that
to be useful, it must be possible to map [...] nested parallelism onto a flat parallel
machine. [4]

This dissertation demonstrates that this is no longer the case. In fact, mapping nested parallelism
onto a multicore machine is very much useful, and even with our prototype implementation we are
able to better sequential C programs by considerable margins (see Chapter 6).

NESL surfaces in the literature in the early 1990s [5] as an ML-inspired working platform
for exercising the ideas set forth in Blelloch’s thesis. Like ML, NESL is a strict, parametrically-
polymorphic language whose types are inferred, 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 op-
erators, sequences, simple datatypes and pattern matching, conditionals and let bindings, top-level
function definitions, and the parallel apply-to-each construct which is the progenitor of, among
other constructs, the parallel comprehension construct in PML.

NESL demonstrated the viability of the flattening transformation as a compilation technique.
Its compiler became the first public formalization of the flattening transformation, albeit in code;
no formal semantics of flattening as implemented for NESL has ever been presented. Beyond its use as a research system, NESL has been used as a platform for teaching parallel programming at CMU since the early 1990s. NESL’s capabilities are sufficient to study nested data parallelism, 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, but no sum types or recursive types; furthermore, functions must be named and can only be defined at the top level. These features are absent specifically because, at the time of NESL’s implementation, the flattening transformation had not yet been modified to account for them. Although NESL’s missing features can be accommodated for in the context of any particular application, its incompleteness as a modern functional language left open terrain in the language design space. (In fact, a considerable portion of Blelloch’s thesis is given over to illustrating how to encode algorithms on trees and graphs using vectors as a sort of universal representation.) Research remained to be done to first formalize the flattening transformation, then to extend it such that it could contain a broader selection of programming language features. Subsequent projects aimed at exactly those targets.

### 2.2 Proteus

In the early 1990’s, Prins et al. [33, 31] presented the Proteus programming language, a simple functional nested data parallel language in the NESL mold. Proteus is a not a realistic programming language in that it contains only enough features to demonstrate the authors’ compilation techniques. Specifically, 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 “iterator” construct that is isomorphic to PML’s parallel comprehension.

The Proteus work is the first to present 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, which is its most significant divergence from the present work. Nevertheless, the 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.

The Proteus research describes two ways [33] of representing flattened nested arrays. What they call the “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 is isomorphic to NESL’s flattened representation, based on flat data vectors accompanied by collections of “segment descriptor” vectors. The other Proteus representation is the “Nesting Tree Representation,” which bears a resemblance to the representation used in the current implementation of PML. 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 we pair those “shape trees” with 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, which also presents an account of the flattening transformation that is clearly specified on a minimal language. Certain of our transformation rules are directly prefigured in the Proteus work. Nevertheless, the results of their flattening transformation and ours are completely different. Their target code is in true SIMD style, where one instruction at a time is applied in parallel to arbitrarily long vectors of data, while our target code performs arbitrarily many sequential or parallel tasks in parallel.

### 2.3 Nepal and Data Parallel Haskell

NESL’s limitations as a realistic programming language provided something of an outline for future research, as language designers naturally wished to integrate nested data parallel programming with modern programming language technologies. The step-by-step extension of NESL’s founda-

---

tion to a more feature-rich platform ultimately took shape as Data Parallel Haskell [14], although it was two PhD theses and a decade of research in the making.

The first step in enriching NESL was the establishment of a type-theoretic foundation for recursive types in a nested data parallel setting [24]. As its model languages, this paper uses NESL as its source language and FKL (for “Flat Kernel Language”) as the target of its flattening transformation. The transformation is only sketched in this paper and the code is transformed “by hand”; actual compiler technology would follow.

Flattening broadened to include support for recursive types [24] (this paper uses NESL and FKL).

Extended further to sum types and separate compilation [11] (uses lambda-calculus-like model languages).

Keller’s thesis treats these topics on (strict) model languages in her thesis [23].

Haskell extended with these techniques. Language design presented under the name Nepal [12]. At this point, things are lazy.

Flattening extended to arrays of higher-order functions [27]. Haskell is the presentation language in this work. Same topics given heavy theoretical treatment in Leshchinskiy’s PhD thesis [26].

The language, now made to account for laziness by virtue of its admixture with Haskell, eventually starts to be presented under the name Data Parallel Haskell [14].

2.4 Manticore and PML

The Manticore project is an infrastructure built to support languages with parallelism at multiple levels: explicitly-threaded parallelism by means of CML-style concurrency, and implicitly-threaded parallelism via NESL-style nested data parallelism. Originally Manticore was the name of the programming language itself [19], although in more recent research [18] we have chosen to refer to the runtime system infrastructure as Manticore and the programming language built upon
it (one of potentially many) as Parallel ML, or PML.

[[ brief survey of Manticore/pml research ]]}

At the surface, PML supports the same constructs as NESL. Like PML, NESL is a strict nested data parallel language with implicitly-threaded parallel constructs, and both languages use ML-style type inference in typechecking. NESL’s flattening techniques, being targeted towards prefix scans and vector operations, are fundamentally different than PML’s. Furthermore, NESL, although it is an ML derivative, is not intended to be a full programming language; a large number of common programming tools are simply not available in NESL. PML, by contrast, is a full-featured ML-like programming language extended with explicitly- and implicitly-threaded parallel constructs.

DPH vs. PML:

- DPH – much more complicated type system
- strict vs. lazy
- Data Parallel Haskell’s flattening is more aggressive, unzipping datatypes and flattening arrays of functions, for example.
CHAPTER 3
DISCUSSION

In this section, we present a high-level introduction to our own particular formulation of the flattening transformation. In order to do so, we must first sketch some of the features of our language and implementation. We then discuss some of the optimizations that the flattening transformation makes available and work through several examples of applying those optimizations.

3.1 Implicitly-Threaded Parallelism in PML

The PML language provides a variety of parallel constructs. These can be grouped into explicitly-threaded parallel features and implicitly-threaded ones. PML’s explicitly-threaded constructs are essentially parallel formulations of channel-based concurrency as pioneered in CML [35]. Parallel CML has been presented in other publications [34] and is outside the scope of this dissertation. Implicitly-threaded constructs are those where the programmer provides annotations, or hints to the compiler, as to which parts of the program are profitable for parallel evaluation, while the mapping of computation onto parallel threads is left to the compiler and runtime system. [18] The full set of PML’s implicitly-threaded constructs consists of parallel arrays, parallel tuples, parallel bindings, and parallel cases. Syntax for common parallel array computations is also provided via parallel comprehensions and ranges. For the purposes of discussing the flattening transformation, we will focus on the parallel array construct and its various syntactic forms, considering parallel tuples as well when needed. Discussion and specifications of the meanings of parallel bindings and parallel cases have been given in prior publications [18].

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:

\[
(\mid e_1, \ldots, e_n \mid)
\]
This expression is a hint to the compiler that it is beneficial to evaluate $e_1, \ldots, e_n$ in parallel, although the compiler need not heed the suggestion. 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:

$$\langle | e_1, \ldots, e_n | \rangle$$

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.

Parallel tuples and parallel arrays differ as they do in sequential languages. The components of a parallel tuple can be of different types, but any parallel tuple inhabits a type the number and position of whose components is rigid. By contrast, the components of a parallel array must all share one type, but parallel arrays of any given type may have any natural length.

It is useful for parallel designations to be hints rather than mandatory directives, since it may be the case at runtime — and this may not be known until runtime — that certain components of a tuple or array do not merit the overhead of being run in parallel. A particular component might, for example, by virtue of inlining or some compile-time beta reduction, turn out to be a constant, in which case it need not be dynamically evaluated at all, let alone given the full parallel treatment.

PML provides parallel comprehension and range syntax to facilitate the specification of certain kinds of parallel arrays comprehensions. PML provides parallel comprehension syntax as a concise, high-level way to specify parallel arrays. It has a close analog in Haskell [21], the list comprehension. In its general form, a parallel comprehension is as follows:

$$\langle | e_0 | p_1 \text{ in } e_1, \ldots, p_k \text{ in } e_k \text{ where } \text{test} | \rangle$$

Each metavariable $e$ ranges over PML expressions, as does the metavariable $\text{test}$, and the metavariables $p$ range over patterns. The boolean expression $\text{test}$ acts a filter over those items on the right hand side of the bar. The patterns may do so as well, if they fail to match the items in their
accompanying expressions. The typing of parallel comprehensions is expressed in the following rule:

\[
\Gamma \vdash e_t : \text{bool} \quad \Gamma \vdash e_i : \tau_i, i \in [1, k] \quad \Gamma + \{ p_i : \tau_i \} \vdash e_0 : \tau
\]

\[
\Gamma \vdash [e_0 | p_1 \text{ in } e_1, \ldots, p_k \text{ in } e_k \text{ where } e_t] : \tau \text{ pararray}
\]

The semantics of this expression are similar to the semantics of mapping a function over a list using a standard implementation of \texttt{map} (such as the SML basis library’s \texttt{List.map}). There are two differences. First, when a pattern \( p_i \) does not match a given element of \( e_i \), that element is skipped, leaving a “gap” in the result. The second difference is that it may be evaluated in parallel.

Like \texttt{List.map}, parallel comprehensions are very much like loops. In PML, they 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 which does just that.

\[
[| e_1 \text{ to } e_2 \text{ by } e_3 |]
\]

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 step size from one entry in the sequence to the next. The clause \texttt{by } \( e_3 \) may be omitted, in which case the compiler fills in \texttt{by 1} in its absence.

If we name the start point, end point and step size \( f \), \( t \) and \( s \) (“from,” “to,” and “step,” respectively), the sequence is defined as follows:

\[
\{ n_k = f + ks \mid k \in \mathbb{N}, f \leq n_k \leq t \}
\]

Under these semantics, a range is never fewer than one element in length, as the first element \( e \) is always included in the sequence. The fact that this sequence always has length at least one follows from the fact that \( n_0 = f \) and therefore the condition \( f \leq n_0 \leq t \) is true for all values of \( f \), \( s \) and \( t \).

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

$$\left[ \begin{array}{c} 2, 5, 10, 17, 26, 37 \end{array} \right]$$

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. In particular, nested parallel comprehensions benefit from the optimization made available by the flattening transformation, as we discuss in detail below.

### 3.2 The Manticore Infrastructure

Auhagen et al. give a precise characterization of the Manticore infrastructure in a recent publication [2], which is recapitulated here in brief. Manticore’s runtime system provides a hardware abstraction layer, a parallel garbage collector, 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. The garbage collector uses a split-heap architecture, following the parallel garbage collection design of Doligez, Leroy, and Gonthier [16, 15], and divides each processor’s local memory into an old-data area and a nursery area, following Appel’s semi-generational garbage collector [1]. 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 uses the lazy tree splitting [3] (LTS) technique for managing the parallel decomposition of work on tree data structures. The foundational idea undergirding LTS is lazy binary splitting [41]. In lazy binary splitting, in processing parallel do-all loops, a processor checks for other “hungry” processors (awaiting work) as it iterates, and, if it finds one, hands over half of its remaining work to its hungry counterpart. LTS generalizes this technique to processors that, rather
than looping, are iterating over trees. This is accomplished by application of the zipper [22] to provide trees with cursors that can be used to navigate around trees in ways that are not possible (up, backwards, etc.) in simple standard recursive tree traversals.

The combination of LTS and using ropes to represent data sequences (see Section 3.3) is effective, and, prior to the present work, we had conjectured that LTS and flattening would work well together. LTS gives us good load balancing for parallel tree traversals, while flattening transforms data structures into simple, flat trees.

### 3.3 Ropes and Rope Tabulation

Ropes [9] are a balanced-tree representation of sequences first proposed in the literature in 1995. The PML compiler uses ropes extensively in its sequence representations. We can define the rope datatype in PML as follows:

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

In our compiler, ropes differ slightly from this definition: each Cat node, in addition to pointing to a pair of ropes, also contains an integer length and an integer depth. As such, querying the length or depth of a Cat node, once built, requires constant time. For the present purposes, we omit this concession to performance in the interest of clarity. We also do not specify here what the representation of seq is; one only needs to know that it is an ordered sequence of data, one that can be instantiated to any particular type. In the PML implementation used in this dissertation, seq is instantiated to array.

The length of each leaf’s data sequence is bounded above by a value maxLeafSize which is set at the moment the program begins executing; for this thesis work, its default value was 512. As long as the maxLeafSize is kept relatively small, each leaf is of modest size, requiring a little more than 512 (pointer-sized) words of memory.
fun tabFromTo (lo, hi, f) = let
  fun t (lo, hi) =
    if (lo > hi) then Leaf (Seq.empty)
    else let
      val n = hi-lo+1
      in
      if (n <= maxLeafSize) then
        Leaf (Seq.tabulate (n, fn i => f (lo+i)))
      else let
        val m = (hi+lo) div 2
        in
        Cat (| t (lo, m), t (m+1, hi) |)
      end
    end
    in
    t (lo, hi)
  end

Figure 3.1: Rope.tabFromTo

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 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, as its physical decomposition can be used to describe a computational decomposition in a very direct way.

[[ from John: “distributed construction.” ]] 

Parallel rope tabulation is at the heart of many of our results. In our implementation, parallel tabulation of polymorphic ropes is based on a routine called tabFromTo, which consumes lower and upper inclusive integer bounds and a function, and builds a rope whose elements are the results of having applied that function to all integers between the bounds. The code for Rope.tabFromTo is given in Figure 3.1. The code is nearly identical to the code one would
write in SML, except the arguments to the \texttt{Cat} constructor are written as a parallel tuple. As such, the two recursive calls to the locally-defined function \( t \) are made in parallel; if those calls make recursive calls themselves, those too are made in parallel, and so on down to the leaf nodes, where simple sequential tabulations take place. In practice this parallel decomposition of tabulation performs very well.

By testing two versions of \texttt{tabFromTo} alongside one another, we can demonstrate that our parallel tuple implementation is effective and scales very well. The plot in Figure 3.2 compares two PML implementations of \texttt{tabFromTo} as given in Figure 3.1. In the demonstration, each implementation is tabulating a million calls to a naïve implementation of \texttt{fib} (Fibonacci) for values in the interval \([0,19]\). The baseline implementation is \texttt{tabFromTo} with no parallel tuple at the \texttt{Cat} node, while the parallel program includes the parallel tuple as shown. While the runtime of the parallel version is ever so slightly slower than the sequential one on a single processor, the result of employing a parallel tuple in the implementation is a near-perfect linear speedup to 48 cores. with the performance at all 48 cores representing a better than 46-fold speedup over the sequential baseline. PML fares somewhat less well against C, a comparison to which is plotted.
in Figure 3.3 to give a sense of absolute performance. The C program baseline allocates a million int array and populates it with calls to fib on the same values as the PML programs. In this case, 7 cores are required for the PML program to run faster than the C program; at all 48 cores PML achieves about a seven-fold speedup over C.

The implementation of Rope.tabulate is simply a call to tabFromTo with the appropriate arguments:

```plaintext
fun tabulate (n, f) = tabFromTo (0, n-1, f)
```

Recall the optional by clauses of ranges, which specifies the stride a range takes as it traverses its interval. Tabulating with a step, not necessarily 1, is implemented by calling tabFromTo after
having done the necessary arithmetic. We call the PML implementation of this kind of tabulation \texttt{tabFromToStep}, and the code is given in Figure 3.4.

In the early implementations of the PML compiler, parallel arrays were mapped directly to ropes. We had many choices about how to compile parallel arrays, but for the advantages enumerated above, we chose ropes as our target. The rope datatype and a broad set of rope operations, many of them parallel, are defined in the PML basis library. This code is not intended to be used directly by the programmer, but rather is the object of parallel array compilation. Having developed advanced techniques for parallel computation over ropes [3], we have already demonstrated success in compiling PML without any flattening whatsoever. Nevertheless, we will show that the present strategy is both compatible with and improved upon by flattening. Without flattening, if a program ever builds a nested parallel array, the compiler simply build nested calls to tabulate in order to construct a rope of ropes. But the flattening transformation is specifically directed at eliminating nested parallel arrays (and, as a consequence, ropes of ropes), so it must adopt a different strategy. The next sections describe some of the techniques we use to flatten nested structures and demonstrate their improvements upon the methods described here.

### 3.4 PML and the Flattening Transformation

\textsc{Java} compilers would not do well to penalize programs that allocated lots of objects, nor would \textsc{ML} compilers do well to discourage programmers from programming in a heavily higher-order style. By the same token, PML compilers should support declarative parallel programs, in particular those that use parallel comprehensions, which PML’s syntax privileges highly.

Parallel comprehension syntax is an enticing feature of PML. Its declarative, high-level character encourages a conceptual, mathematical approach to parallel problem solving, intended to liberate the programmer from having to give too much consideration to a program’s mechanical and operational characteristics. This, in fact, gets to the core of our language design. Since our syntax effectively promotes a style of problem solving, it behooves us to support that style well
with our compiler, lest PML programmers be compelled to write awkward or unidiomatic code in order to achieve good performance.

The flattening transformation in the PML compiler is directed specifically toward that kind of support. Values in the surface language that exist as parallel arrays are translated into flat arrays, which in turn are built of ropes and a simple data structure we introduce here heretofore called “shape trees” or simply “shapes.” PML’s implements flat arrays with the \texttt{farray} datatype. Please note \texttt{farrays} are not part of the surface languages; they are an internal data structure used by the compiler in its transformations. They are implemented with PML code in the PML basis library.

PML’s \texttt{farray} representation pairs flat ropes and shapes to represent nested structures. The name \texttt{farray} means “flattened array,” that is, an array whose data has been flattened into a single linear vector of data but whose original structure has been preserved separately and recorded in an accompanying value.

\begin{verbatim}
datatype 'a farray
   = FArray of 'a rope * shape
\end{verbatim}

In this data structure, a \texttt{rope} is just an ordered sequence of data, and it may be thought of as such in abstract terms. Its choice is not mandatory. Nonetheless, since we already have so much infrastructure in the implementation of ropes, we decided to exploit it in the flattened representation as well.

### 3.4.1 The Rationale for Shape Trees

Segment descriptors contain nesting information in the FT literature. Various segment descriptor designs have been presented, including lengths, head flags and head pointers. The head flags representation suffers from the crippling restriction that it cannot represent empty subarrays. Neither the lengths representation nor the head pointer flags representation suffers from this restriction; lengths is the simpler of the two, and it is closest to ours.
In the earliest flattening work [7] each flattened array carried with it a redundant bundle of segment descriptors, each of which is expressing the structure in its own way. Presumably segment descriptors of different representations were selected and used in contexts where they were offered best performance, but Blelloch and Sabot do not discuss this part of their implementation in any detail.

Flattening predecessors objected to the lengths encoding since it yields an array whose length may not match the length of the data vector it describes. This was a concern when segment descriptors were distributed across different processors, which they used to be. Keller’s unnamed representation (perhaps we could call it “owners”) does not suffer from this problem, but it is designed for segment descriptors that are distributed “across processors.”

[[ What does DPH do? ]]

The considerations about the representations of segment descriptors have shifted since our predecessors’ work. Blelloch and Sabot were specifically interested in “existing parallel hardware,” which was, in 1990, entirely different from what we have today. In our multicore machines, we are no longer bound to represent everything as a vector, and we have no expectation of distributing our sequences across machines. Freed from needing to use vectors to represent nesting structure, we designed shape trees as a compact tree representation to fill the role shape descriptors have traditionally played.

A shape is an n-ary tree whose leaves store integer pairs.

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

Each leaf contains the starting index and the ubdex of the elements following the segment of data in an farray. The shape Lf(i, i+n) described a length-n segment starting at i. A simple, flat parallel array of integers such as

```
[| 1, 2, 3 |
```
has the following farray representation:

\[
\text{FArray (Leaf \{1,2,3\}, Lf(0,3))}
\]

(Note we present the sequence in the rope’s Leaf node using list syntax, although the reader will keep in mind that this sequence is not implemented as a list.) The data in the original sequence appears here, at a Leaf, in the original order, and the accompanying shape — Lf(0,3) — states that the flattened array’s only segment begins at position 0 and ends at position 2.

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:

\[
\text{FArray (Leaf \{1,2,3,4,5,6\}, Nd[Lf(0,2),Lf(2,2),Lf(2,6)])}
\]

Once again, the data appears in order in a rope’s Leaf node. The shape is a Nd of two leaves: this alone tells us that the parallel array, now flattened, consists of two subsequences. The leaves, in turn, tell us that the first sequence begins at position 0 and runs up to 3 exclusive, and the second sequence begins at position 3 and runs till the end.

For arrays of arrays of integers, this representation scales up to any nesting depth in a natural way. The transformation is more complicated when it involves tuples and other non-scalar values. We defer our discussion of flattening of non-scalars until a later section.

It is worth noting that the flattening transformation cannot, in general, be applied at compile time. Miniature literal examples like those immediately preceding are helpful for illustrating the goals of the transformation, but the compiler cannot in general assume that those are the kinds of parallel arrays it will encounter. In actual applications, the compiler must deal with parallel arrays that occur in arbitrary contexts and do arbitrary things:

\[
\text{fun } f \text{ n } = \text{ [ | [ | i mod j + j \text{ div } i | j \text{ in } [ | 0 \text{ to } i |] |]}
\]

\[
\text{ | i \text{ in } [ | 0 \text{ to } n |] |]}
\]

24
There is no way for the compiler to know, in general, the shape of a parallel array (such as this one) that is yet to be constructed. In many realistic cases, the best the compiler can do is to arrange for flattening to take place at the future moment when there is a value to flatten.

### 3.5 Optimizations Enabled by Flattening

The optimizations enabled by the flattening transformation have a particular character. They are worthwhile and provide dramatic performance gains, but, as they are systemic and wide-ranging, they place a heavy burden on the compiler writer. In other words, very powerful optimizations are available, but, as they are bound up with wholesale, program-wide representation changes, they come at a high price.

[[John put a question mark next to that paragraph. not sure why.]]

There are certain classic optimizations that, given the right infrastructure, are easy to perform and easy for the compiler to digest. Consider the following classic fusion optimization, known as “map/map” in the GHC literature [32].

\[
map \ f \ (map \ g \ xs) \Rightarrow \ map \ (f \circ g) \ xs
\]

(Note this optimization is only valid when the functions \(f\) and \(g\) are pure.) It is simple to implement this optimization and it saves linear time and space by skipping building an ephemeral intermediate list. Furthermore, its changes are entirely local: no client of the optimized code needs to know that the optimization has ever happened.

The optimizations made possible by flattening are reminiscent of the sort of manual optimizations C programmers have employed for decades. For example, a nested loop over two indices in C, like this one:

```c
for (int i=0; i<sz; i++)
    for (int j=0; j<sz; j++)
```

25
process(i, j);

can be transformed to one loop over a single index variable as follows:

    for (int k=0; k<sz*sz; k++)
        process(k/sz, k%sz);

In the latter case, the tests and jumps of the inner loop have been avoided, yet its behavior is identical to that of its more expensive counterpart.

One can similarly optimize data structures in C. Consider a two-dimensional square array of scalars of some dimension n. If the programmer chooses to represent this as an array of arrays a — in other words, as a structure of type int** — she will be able to refer to its elements in a natural way, as a[i][j]. This is good programming style, in the sense that the code reflects the programmer’s intention in a predictable, broadly understandable way. By contrast, the array could be represented as a linear array of scalars b of type int*. Referring to element i, j now entails a bit of arithmetic — b[i*w+j]. In this latter case, the clarity of the code has been compromised, slightly: the fact that b represents an array of arrays is no longer entirely clear from the code itself. But the programmer might very well be willing to tolerate a slight obfuscation in the code in exchange for better performance. Nevertheless, this hand optimization of the code is not local but pervasive. All clients of b must know that, although b is manifest as a linear array of scalar values, it represents a two-dimensional shape.

Note that as one scales up to more and more dimensions, the multi-dimensional array notation remains clear, whereas the flattened array notations becomes less and less so. Compare three dimensions in multi-dimensional style

    a[i][j][k]

to three simulated dimensions in flat style

    a[(i*d*w)+(j*w)+k]
In our setting, the flattening transformation enables our language design to inhabit a sweet spot between these two alternatives. We combine the clarity of multi-dimensional arrays in the surface syntax with the performance of flattened arrays. Our surface program contains nested representations akin to \( a[i][j] \), in that it is manifestly clear when structures represent two (or more) dimensions. The compiled program, on the other hand, transforms the data structure into a flattened (essentially linear) structure, and transforms all clients of that structure accordingly.

Furthermore, array flattening and parallelism make a great combination. It is much easier for the runtime system to schedule parallel work on a flat array of computations than on a nested one [39]. So, while array flattening is beneficial even in a traditional sequential setting such as C programming, it is beneficial in parallel programs to an even greater extent.

### 3.5.1 Unzipping and Monomorphization

unzipping.

monomorphization.

### 3.5.2 Tab Flattening

As discussed above, every nested parallel array is transformed to a flattened array by the flattening transformation. Since most data structures do not exist until run time, most representation transformations must take place at run time. For this purpose we employ a family of type-indexed flattening operators, inserted into the program by the compiler as needed.

Consider the following nested parallel comprehension.

```plaintext
val pcomp1 = let
  val rng = [| 0 to 2 |]
  in
    [| [| 10*i + j | j in rng |] | i in rng |]
  end
```

Absent the flattening transformation, \( pcomp1 \) would evaluate to the following value:

\[
[| [|| 0, 1, 2 ||], [|| 10, 11, 12 ||], [|| 20, 21, 22 ||] ||]
\]
Let us refer to this as \( pcomp_1 \)'s nested representation. The value bound to \( pcomp_1 \) has surface type \( \text{int parray parray} \). Per the flattening transformation, this value will be transformed into one of type \( \text{int farray} \). Recall \( \text{farray} \) is represented as a pair consisting of a flat rope of data and a shape tree (wrapped in an \( \text{FArray} \) constructor). As such, the parallel comprehension at hand must in fact to evaluate to the following:

\[
\text{FArray (Leaf [0, 1, 2, 10, 11, 12, 20, 21, 22], Nd [Lf (0, 3), Lf (3, 6), Lf (6, 9)])}
\]

We will refer to this value as \( pcomp_1 \)'s flat representation.

Our compiler recognizes parallel comprehensions that span ranges (in this case of \( pcomp_1 \), \([| 1 \text{ to 3 |}]\)) and rewrites them as calls to \( \text{PArray.tabulate} \). If the flattening transformation is not enabled, it rewrites \( pcomp_1 \) into nested calls to \( \text{tabulate} \), as follows:

\[
\text{val } pcomp_1 = \text{PArray.tabulate (3, fn } i => \\
\quad \text{PArray.tabulate (3, fn } j => 10 \ast i + j))
\]

It is straightforward for the compiler to apply the appropriate flattening operator to this nested \( \text{tabulate} \) in order to yield the desired flat array. Assuming that operator is named \( \text{fl_intParrParr} \), the flattening-transformed code is simply

\[
\text{val } pcomp_1 = \text{let } \\
\quad \text{val } n = \text{PArray.tabulate (3, fn } i => \\
\quad \quad \text{PArray.tabulate (3, fn } j => 10 \ast i + j)) \\
\quad \text{in } \\
\quad \text{fl_intParrParr } n \\
\text{end}
\]

This implementation has the benefit of being simple and local. But, where in the nested case we were building a single nested data structure, in this case we build a nested data structure only to feed it to the operator \( \text{fl_intParrParr} \) that builds an additional data structure of roughly the same size, and leaves the original nested structure to be garbage collected. In other words, by flattening this structure, we have basically doubled our allocation requirements and added linear time to our computation. It is possible that in such cases whatever benefits were to be had by flattening have been preempted by the extra work flattening has entailed.
Note that this simple strategy, while undesirable for the reasons stated here, is the necessary strategy when dealing with parallel array literals, that is, those that are spelled out with individual computation for elements and not created systematically by computational rules as is `pcompl` above. In the case of literals, there is no logic we can use to create the array, so we can do no better than to take it as is and flatten it with a flattening operator. It is certainly possible to do this at compile time for some common cases—for example, a parray whose elements are constants—but we have not yet built any infrastructure for doing so.

[[ John: “should be trivial. ?” ]]

There is a favorable alternative to the naïve nest-then-flatten strategy given above, which is to produce the desired flat structure directly. In doing so, the intermediate nested array construction is dispensed with altogether. This strategy is very successful in practice, as it enables the system to build a flat structure quickly and subsequently reap the benefits of processing that flat structure, which is the motivation for performing the transformation in the first place.

In the present example, this alternate compilation strategy will allow us to replace two tabulates—a tabulate of a tabulate—with a single tabulate that produces the flat vector of data directly. It is tempting to refer to this process as “tab fusion” (recall the “map/map” fusion transformation above, which replaces two maps with one), but I prefer to call it “tab flattening” as it is more wide-ranging that a typical fusion optimization, whose effects are local and invisible to clients of the optimized code. Its effects are not local and it is only possible because the system as a whole is designed to deal with fundamental representation changes in certain data structures.

The tab-flattening compilation strategy assumes the existence of an operator to create regular shapes. In this case we refer to the operator `regular2D` that, given two lengths, produces the shape tree corresponding to the lengths given. The `regular2D` operation is a simple, inexpensive computation. We also refer to `Rope.tabulate`, a parallel operator that produces a rope of data according to the given function for computing the individual elements.
val pcompl = let
  fun elt k = 10 * (k div 3) + (k mod 3)
  val data = Rope.tabulate (3∗3, elt)
  val shape = regular2D (3∗3)
  in
    FArray (data, shape)
  end

No intermediate structures are created here. Rope.tabulate is a fast, parallel operation, and this compilation strategy is effective, as our results show.

[[ John asks: “is this translation formalized somewhere?” it will be, in a later section. ]]

3.6 Examples

• mandelbrot/mandelbulb

• smvm

• flat arithmetic
CHAPTER 4
FORMALIZATION

As discussed in Chapter 2, previous flattening transformations were designed to transform declarative functional programs to vector code that processes long vectors of scalar data in parallel. In such cases, the SIMD operations in the target programs are entirely different than the operations in the surface code. By contrast, hybrid flattening transformations, while flattening data structures using similar techniques to prior work, produce a task-parallel program whose operators, while altered selectively, are in the same class as those in the surface language.

We are interested in a different class of transformations than those in earlier work, which perform wholesale flattening and radical transformation of the programs to which they are applied. We are interested instead in a family of flattening transformations which we call hybrid flattening, where terms can be flattened either entirely, not at all, or to some intermediate extent. In this section, we present a formal system that allows precise characterizations of hybrid flattening transformations. We first present a simple model language with functions, pairs, and arrays. Flattened and non-flattened terms can coexist in the model language. The model allows us to discuss the essential elements of flattening while sidestepping implementational details that contribute only non-essential complexity. After introducing the model, we present a formal transformation that flattens all array terms to the full extent. The set of rules presented in the transformation comprises the backbone of the actual PML implementation discussed in Chapter 5 and used to gather the results presented in Chapter 6.

Following Data Parallel Haskell’s terminology, we refer to parallel sequences as parallel arrays or simply arrays. Full flattening transforms all arrays in the source program into flattened arrays of scalars. Our flattened array representation pairs flat vectors of data with shape trees (defined and discussed below) describing the original nesting structure. Flattening arrays requires repeated application of two operations: concatenation, where the elements of a nested array are joined such

---

1. We draw inspiration from Leroy’s model systems containing both unboxed and boxed representations [25].
that they are a level less deeply nested, and unzipping, where arrays of pairs are rearranged into pairs of arrays. In concert with data structure flattening, the code surrounding nested arrays — the clients of those arrays, in other words — must be correspondingly transformed to process flattened arrays rather than nested ones.

4.1 The Model Language

First we present our model language, Flatland. It is an explicitly-typed, monomorphic, strict, pure functional language with pairs, parallel arrays, and first-class functions. We presuppose a basis of constants and operators, including both ground terms like integers and booleans, and a standard assortment of common primitive operators. The language also includes array operators subscript (\(!\)), map (\(\text{map}\)), filter (\(\text{filt}\)) and (associative) reduce (\(\text{reduce}\)). Since the language is monomorphic, there is not a single uniform implementation of array operators: we assume there is a type-indexed family of each such operator. When we want to refer explicitly to a particular specialized implementation of one of the array operators, we include its type in a subscript, as in \(!_\rho\) for type \(\rho\).

The grammar of terms in Flatland appears in Figure 4.1. Every term carries an explicit type as a superscript. This grammar is actually an abbreviated presentation of two grammars, one for the source language (i.e., the surface language), and one for the target language. If a syntactic form is present is only one of those languages, it is specified as such in a comment. Flattening transformations are applied to programs in the source language to produce programs in the target language. The metavariable conventions are as follows: \(b\) ranges over constants and \(x\) over variables.

The source language differs from the target language in the following ways. All terms in both languages are explicitly typed, but the target language uses a different, and disjoint, set of types (discussed below). Arrays in the source language are delimited by square brackets. Flattened arrays, which only appear in the target language, are delimited by curly braces. Furthermore,

---

2. It is named after Abbott’s novella of the same name (1884).
Table 4.1: Relations defined in this chapter.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e^\tau ) ok</td>
<td>well-formedness of explicitly-typed terms</td>
</tr>
<tr>
<td>( \tau \downarrow \tau )</td>
<td>type flattening (full)</td>
</tr>
<tr>
<td>( e^\tau \downarrow e^{(\tau/\rho)} )</td>
<td>term flattening (full)</td>
</tr>
</tbody>
</table>

non-empty flattened arrays carry an extra expression with them, which is its *shape tree*. A shape (alternatively, a *shape*) 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. Shape trees play the role that collections of *segment descriptors* play in predecessor languages such as NESL.

The types of Flatland are given in Figure 4.2. The variable \( \tau \) ranges over *surface types* (which we also refer to as *interface types*), and \( \rho \) ranges over *representation types*. Types in the source language are surface types alone. Types in the target language are of the form \( \langle \tau/\rho \rangle \), where a surface type is paired with a representation type. Representation types are determined by the rules of the flattening transformation: they cannot be chosen or specified in the surface language. The types ranged over by the variable \( \nu \) are *shape types*. (\( \nu \) is a mnemonic for “nesting.”) Shapes’ types record the nesting depth of a given flattened array and are isomorphic to the natural numbers. Shape types only occur as subcomponents of flattened array types.

The interface type of a term is the type it has in the surface program. It is manifest in the surface program as written. The representation type is an internal type which may or may not differ from the interface type. The representation type characterizes the concrete representation the system has chosen for a given structure in the surface program. Ground types are the same at the interface level and the representation level, but array types differ between interface and representation.

[[ need to discuss specialization of mapP, filtP, reduceP ]]
4.1.1 Static Semantics of Flatland

Since Flatland is an explicitly-typed language, we present its static semantics as a well-formedness judgment on terms. $1^{\text{int}}$ is well-formed; $1^{\text{bool}}$ is not. When a term $t$ is well-formed we write $t\ ok$. Typing is simple in Flatland and the rules contain no surprises. The rules for well-formedness are given in Figures 4.3 and 4.4.

4.1.2 Flatland vs. PML

PML is a general-purpose programming language with a full assortment of modern functional programming features. It supports algebraic datatypes, type inference, pattern matching with exhaustiveness checking, and a variety of explicitly- and implicitly-threaded constructs for parallel computation. Flatland contains only parallel arrays, and provides four operators for computing with parallel arrays: subscript, map, filter and reduce. As PML’s parallel comprehensions are syntactic sugar for appropriate applications of map and filter, transformations on Flatland’s parallel array computations represent transformations on PML’s parallel comprehensions after desugaring. In practice, Flatland has proven to be a useful model for developing and reasoning about the flattening transformation in PML. The rules here have been directly applicable in implementing full flattening for the PML compiler.

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 challenging when designing a representation transformation. Parametrically polymorphic functions demand uniform, pointer-heavy representations, while flattening is directed toward creating customized, monomorphic representations for faster processing. In this sense, the two compilation styles are at odds with one another. At the time of this writing, parametric polymorphism for parallel arrays is unimplemented in PML. This is a problem which may, in fact, never need to be resolved. Following the example of MLton [29], a whole-program compiler for SML, the PML compiler is moving toward full monomorphization during compilation. In monomorphization,
polymorphic code is instantiated at every concrete type where it is needed, and the program is purged of polymorphism in an intermediate stage of compilation (in MLton, it lies between an abstract syntax tree and a normalized representation). Once this new regime is in place, it will be possible to apply the flattening transformation in PML to a monomorphic program, and the treatment of flattening given here will not need to be adjusted to take polymorphism into account.

4.1.3 Decisions about Flattening

The formal system presented in this chapter is a framework for many flattening transformations. Values can be either flattened or not, in this framework, but it is also the case that they may be flattened to varying degrees. Decisions about flattening are made with the assistance of a heuristic function. Coercions between representation, when necessary, are inserted as part of the flattening process.

The original presentation of the flattening transformation by Blelloch and Sabot [7] was directed at the specific problem of compiling declarative languages to massively parallel vector machines. Because of the architectural requirements of the object code in their context, their flattening was of necessity wholesale flattening, which is to say, all nested arrays needed to be flattened in order for compilation to succeed. All the flattening transformation work up to the present is indebted to the original ideas in that paper, and while the original techniques have been supplemented with new theory and extended significantly, the total nature of the transformation has traditionally been assumed. The sole exception to total flattening is the recent work of Chakravarty et al. [13] on partial vectorization. In their work, they present a system that includes both flattened and non-flattened terms, and introduces coercions in the appropriate places where a value needs to cross a representation boundary between flatness and non-flatness.

The critical difference between the partial vectorization system and ours is that, when a value is flattened in their system, it is always flattened all the way; there is no way to perform flattening up to a point but no further. Our system, like theirs, coerces terms across representation boundaries
as needed, but it also allows terms themselves to be flattened partially. We employ heuristics to
decide whether and how much to flatten a particular value or not, and propagate our decisions
throughout our program by tracking them in an environment. When we arrive at a conflict between
one flattening decision and another, we insert type coercions to resolve it.

In Section 4.2, we present a flattening transformation which is aggressive, as most flattening
transformation are. Our system flattens every data structures and modifies client code accordingly.
The partial vectorization work points out cases where application of the flattening transfor-
mation might result in semantic difficulties, but their system does not identify when flattening might
worsen the performance of the program. Here we consider some situations where wholesale flat-
tening may not actually yield programs’ best performance. Given the appropriate heuristics, a
particular system should be able to moderate the amount of flattening it does, according to its best
estimate of the performance of the compiled code.

There are trival cases where flattening does not improve performance, such as when a nested
structure is computed with in such a way that its representation has little performance impact.
Perhaps a nested array is to be used only by being fed to a function that selects its first element.
In that scenario, any computational effort expended in changing the representation of the original
structure is likely to have been wasted. More generally, when a data structure is not subject to
traversal by any of the operations performed upon it, its representation, nested or otherwise, may
not have a significant effect on performance. There must exist a boundary on one side of which
flattening is profitable, and the other side of which it is not.

Arrays of tuples are typically unzipped as part of flattening [7, 33, 23, 26]. Our own flatten-
ing system as implemented does so, and demonstrates good results in our tests (see Chapter 6).
Nevertheless, there are scenarios where unzipping tuples would generate unnecessarily expensive
code. Assume we have a parallel array of pairs, where the components of the pairs are pointers to
heap-allocated data structures. If the pairs of pointers are unzipped, traversal of the transformed
structure now entails traversal of two sequences.
4.2 A Fully-Flattening Transformation

We present our flattening transformation as a system of rules in this section, but we first present some examples to provide intuition about the system. We use the following notation for two kinds of function composition employed in the transformation, which one might think of as vertical and horizontal composition, respectively:

\[
(f \circ g) x \overset{\text{def}}{=} f(g x)
\]

\[
(f \otimes g)(x, y) \overset{\text{def}}{=} (f x, g y)
\]

The flattening transformation turns every array in the surface language into a flattened array. This is the case even when the original array contains no nesting. For example, consider the array of integers

\[
[1, 2, 3] : \text{[int]}
\]

The data in this array requires no concatenation; it is flat already. The shape tree is one leaf, indicating that the flattened array consists of a single sequence, starting at position 0 and continuing up to position 3, exclusive.

\[
\{1, 2, 3 \mid \text{lf}(0, 3)\} : \langle \text{[int]} / \text{[int} ; \text{lf} \rangle
\]

Brace-delimited arrays are flattened arrays, and they have their own (brace-delimited) types. Note this term now carries a pair of types: its original array type from the surface language, and its flattened array type, stating that its elements are integers and, via its shape type, that it represents only one level of structure.
Arrays of pairs are unzipped into pairs of arrays, as well as flattened as above.

\[
[(1, 2)] : [(\text{int}, \text{int})]
\]

By means of unzipping and flattening, this term becomes

\[
(\{1; \text{lf}(0, 1)\}, \{2; \text{lf}(0, 1)\}) : \langle ([\text{int}], [\text{int}])/\langle [\text{int}], [\text{lf}] \rangle, \langle [\text{int}], [\text{lf}] \rangle \rangle
\]

The surface type is significant here: it records the original form of the term. It distinguishes this target term from the transformation of \([(1), (2)]\), which is transformed into the identical pair of singleton flattened arrays.

Our presentation of the flattening transformation is syntax-directed. It is designed so that the rules both specify the system and sketch a simple implementation of it.

We now present rules for the flattening of types and flattening of terms. While these two rule sets are distinct, we overload the symbol \(\downarrow\) to stand for both of them. Flattening of types is given in Figure 4.5. Flattening of terms is given in Figures 4.6 and 4.7.

### 4.2.1 Claims

We begin by defining the property of flatness for types, and prove related claims about the flattening relation on types.

**Definition 4.2.1.** A type \(\rho\) is flat if

- it is a ground type \(g\),

- it is a function type \(\rho_1 \rightarrow \rho_2\) and \(\rho_1\) and \(\rho_2\) are flat,

- it is a pair type \((\rho_1, \rho_2)\) and \(\rho_1\) and \(\rho_2\) are flat, or

- it is an array type \(\{\rho; \nu\}\) and \(\rho\) is flat and neither a pair type nor an array type.
The type flattening relation in Figure 4.5 always yields flat types. We will prove this claim below.

The $N$ operator defined in Figure 4.5 is used as an auxiliary in the flattening transformation on types. The $N$ operator performs “flat nesting” – that is, it takes a type and makes its array one level more deeply nested, by wrapping its nesting-tree types in another $nd()$.

We would like to prove that the flattening transformation on types yields a flat type, no matter its input. In order to show that, we first make a claim about the $N$ operator.

$N$ is defined only on flattened-array types, defined as follows.

**Definition 4.2.2.** A flattened-array type is either

- an array type $\{\rho; \text{lf}\}$ where $\rho$ is flat and $\rho$ is neither a pair type nor an array type, or
- a pair of flattened-array types.

Note that all flattened-array types are flat types, but not all flat types are flattened-array type (for example, any ground type $g$).

**Claim 4.2.1.** Let $\phi$ be a flattened-array type. Then $(N \phi)$ is a flat type.

**Proof.** The proof is by induction on the structure of the type $\phi$.

- If $\phi = \{\rho; \nu\}$ is flat, then $N \phi = \{\rho; nd(\nu)\}$ is also flat by definition.

- Let $\phi = (\rho_1, \rho_2)$ for flat types $\rho_1$ and $\rho_2$. There are two inductive hypotheses, one for each type $\rho_i$, that $N \rho_i$ is flat. Then by definition $N \phi = (N \rho_1, N \rho_2)$ is flat.

It is also the case that $(N \phi)$ is a flattened-array type, but we need not prove that stronger claim.

**Claim 4.2.2.** For all types $\tau$ and $\rho$ such that $\tau \downarrow \rho$, $\rho$ is a flat type.

**Proof.** By induction on the rules in Figure 4.5.
• The two axiomatic rules are base cases, both true by definition.

• For each non-axiomatic rule other than the nested array rule (the last one), the claim is true based on inductive hypotheses that the claims are true in the premises.

• For the nested array rule, our inductive hypothesis is that the type $\rho$ in the premise is flat. Furthermore $\rho$ is either a flat pair type or a flat array type, since, by inspection of the four rules transforming array types, array types are never transformed to ground types or function types. Then, by Claim 4.2.1, $N \rho$ is also flat.

\[\square\]

Theorem 4.2.3. Given: $e^\tau$ ok and $\tau \triangleright \downarrow \rho$ and $e^\tau \triangleright e^{(\tau/\rho)}$.

Then $e^{(\tau/\rho)}$ ok.

The proof of this theorem is in Appendix A.
\[
t ::= e^{(\tau/\tau)}
\]
\[
e ::= b
\]
\[
x
\]
\[
\text{if } t \text{ then } t \text{ else } t
\]
\[
\text{let } x^{\tau} = t \text{ in } t
\]
\[
\text{fun } f(x^{\tau}) = t
\]
\[
(t, t)
\]
\[
\text{fst } t
\]
\[
\text{snd } t
\]
\[
t t
\]
\[
[]
\]
\[
[t, \ldots, t]
\]
\[
\{\}
\]
\[
\{[t, \ldots, t] ; s\}
\]
\[
\tau \rightsquigarrow \tau
\]
\[
t !_{\tau} t
\]
\[
\text{map}_{\tau,\tau} t t t
\]
\[
\text{filt}_{\tau} t t t
\]
\[
\text{reduce}_{\tau} t t t
\]
\[
\overline{\text{map}}_{\tau,\tau,\nu} t t t
\]
\[
\overline{\text{filt}}_{\tau,\nu} t t t
\]
\[
\overline{\text{reduce}}_{\tau,\nu} t t t
\]
\[
\]
\[
b ::= \{\text{true, false, 0, 1, \ldots, not, +, \ldots}\}
\]
\[
s ::= \text{lf}(t, t)
\]
\[
\text{nd}[s, \ldots, s]
\]

Figure 4.1: Flatland: terms
T ::= \langle \tau / \tau \rangle \quad \text{interface, representation types}

\tau ::= g \quad \text{ground types} \\
| (\tau, \tau) \quad \text{pairs} \\
| [\tau] \quad \text{parallel arrays} \\
| \{\tau ; \nu\} \quad \text{flattened parallel arrays} \\
| \tau \rightarrow \tau \quad \text{functions}

\nu ::= \text{If} \quad \text{structure of flat arrays} \\
| \text{nd}(\nu) \quad \text{structure of nested arrays}

\quad g = \{\text{int, bool, \ldots}\}

Figure 4.2: Flatland: types.

\[
\begin{align*}
\Gamma(x) = \tau & \quad \text{BE}(b) = \tau \quad (\text{BE = basis env}) \\
\Gamma \vdash x^\tau \ ok & \quad \vdash b^\tau \ ok \\
\Gamma \vdash e_1\text{bool} \ ok & \quad \Gamma \vdash e_2^\tau \ ok \quad \Gamma \vdash e_3^\tau \ ok \\
\Gamma \vdash (\text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3)^\tau \ ok \\
\Gamma \vdash e_1^{\tau_1} \ ok & \quad \Gamma[x \mapsto \tau_1] \vdash e_2^{\tau_2} \ ok \\
\Gamma \vdash (\text{let} \ x^{\tau_1} = e_1 \ \text{in} \ e_2)^{\tau_2} \ ok \\
\Gamma \vdash e_1^{\tau_1} \ ok & \quad \Gamma \vdash e_2^{\tau_2} \ ok \\
\Gamma \vdash (e_1, e_2)^{\tau_1, \tau_2} \ ok \\
\Gamma \vdash e^{\tau_1, \tau_2} \ ok & \quad \Gamma \vdash (\text{fst} \ e)^{\tau_1} \ ok \\
\Gamma \vdash (\text{snd} \ e)^{\tau_2} \ ok
\end{align*}
\]

Figure 4.3: Well-formedness judgment on terms (1 of 2). Superscript types have been elided from most subexpressions in the consequents of these rules.
\[
\frac{\Gamma \vdash []^{\tau} \text{ok}}{\Gamma \vdash [e_1, \ldots, e_n]^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash \{\tau; \nu\} \text{ok}}{\Gamma \vdash \{\tau, \nu\} \text{ok}}
\]

\[
\frac{\Gamma \vdash e_1^{\tau}, \ldots, e_n^{\tau} \text{ok} \quad s : \nu}{\Gamma \vdash \{e_1, \ldots, e_n; s\}^{\tau; \nu} \text{ok}}
\]

\[
\frac{\Gamma \vdash e_1^{\tau} \text{ok} \quad \Gamma \vdash e_2^{\text{int}} \text{ok}}{\Gamma \vdash (e_1^{\tau} e_2)^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash e_1^{\{\tau; \nu\}} \text{ok} \quad \Gamma \vdash e_2^{\text{int}} \text{ok}}{\Gamma \vdash (e_1^{\{\tau; \nu\}} e_2)^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash e_1^{\tau} \text{ok}}{\Gamma \vdash (e_1^{\tau} e_2)^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash e_1^{\{\tau; \nu\}} \text{ok} \quad \Gamma \vdash e_2^{\text{int}} \text{ok}}{\Gamma \vdash (e_1^{\{\tau; \nu\}} e_2)^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash (\text{fun} f(x^\tau) = e_2^{\tau})^{\tau \rightarrow \tau'} \text{ok}}{\Gamma \vdash e_1^{\tau \rightarrow \tau'} \text{ok} \quad \Gamma \vdash e_2^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash (\text{map}_{\tau, \tau'} e_1 e_2)^{\tau'} \text{ok}}{\Gamma \vdash e_1^{\tau \rightarrow \tau'} \text{ok} \quad \Gamma \vdash e_2^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash (\text{filt}_{\tau} e_1 e_2)^{\tau} \text{ok}}{\Gamma \vdash e_1^{\tau \rightarrow \text{bool}} \text{ok} \quad \Gamma \vdash e_2^{\tau} \text{ok}}
\]

\[
\frac{\Gamma \vdash \text{reduce}_{\tau} e_1 e_2 e_3^{\tau} \text{ok}}{\Gamma \vdash e_1^{(\tau, \tau) \rightarrow \tau} \text{ok} \quad \Gamma \vdash e_2^{\tau} \text{ok} \quad \Gamma \vdash e_3^{[\tau]} \text{ok}}
\]

\[
\frac{\Gamma \vdash \text{reduce}_{\tau, \text{if}} e_1 e_2 e_3^{\tau} \text{ok}}{\Gamma \vdash e_1^{(\tau, \tau) \rightarrow \tau} \text{ok} \quad \Gamma \vdash e_2^{\tau} \text{ok} \quad \Gamma \vdash e_3^{[\tau; \nu]} \text{ok}}
\]

Figure 4.4: Well-formedness judgment on terms (2 of 2).
\[ N \rho = \begin{cases} \{ \rho' \ ; \ nd(\nu) \} & \text{where} \ \rho = \{ \rho' \ ; \ \nu \} \\ (N \rho_1, N \rho_2) & \text{where} \ \rho = (\rho_1, \rho_2) \end{cases} \]

\[ g \downarrow g \quad \{ g \downarrow \{ g ; \lf \} \]
Figure 4.7: Flattening transformation (2 of 2).

\[
\begin{align*}
(e_{1,2}^{\tau}) & \searrow e'_{1,2}^{((\tau,\tau)/(\rho_1,\rho_2))} & (e_{1,2}^{\tau}) & \searrow e'_{1,2}^{((\tau,\tau)/(\rho_1,\rho_2))} \\
(fst e)^{\tau_1} & \searrow (fst e')^{\tau_1/\rho_1} & (snd e)^{\tau_1} & \searrow (snd e')^{\tau_2/\rho_2}
\end{align*}
\]

\[
\begin{align*}
e_{1,2}^{\tau} & \searrow \phi^{\tau/\rho} \quad e_{1,2}^{\tau} & \searrow \phi^{\tau/(\rho, \text{int})} \quad \phi : (\rho, \text{int}) \rightarrow \rho' \\
(e_1 \cdot e_2)^{\tau} & \searrow (e_1' \cdot e_2')^{\tau/\rho'}
\end{align*}
\]

\[
\begin{align*}
\tau \searrow \rho & \quad \tau \searrow \rho' & \quad i \in \{1 \ldots n\} & \quad e_i^{\tau} \searrow e_i^{\tau/(\tau/\rho)} \quad [\tau] \searrow \rho' \\
\end{align*}
\]

\[
\begin{align*}
e_{1,2}^{\tau} & \searrow \phi^{\tau/\rho} & \phi = \{\rho : \text{int}\} \curlyvee \{\rho'\}; \text{int}(0, n))
\end{align*}
\]

\[
\begin{align*}
\text{map}_{\tau_1, \tau_2} & \quad e_{1,2}^{\tau_1} \searrow \text{map}_{\rho_1, \rho_2, \nu} e_{1,2}^{\tau_2} \\
\text{map}_{\tau_1, \tau_2} & \quad e_{1,2}^{\tau_1} \searrow \text{map}_{\rho_1, \rho_2, \nu} e_{1,2}^{\tau_2}
\end{align*}
\]

\[
\begin{align*}
\text{filt}_{\rho, \nu} e_{1,2}^{\tau_1} & \searrow \text{filt}_{\rho, \nu} e_{1,2}^{\tau_1} \\
\text{filt}_{\rho, \nu} e_{1,2}^{\tau_1} & \searrow \text{filt}_{\rho, \nu} e_{1,2}^{\tau_1}
\end{align*}
\]

\[
\begin{align*}
\text{reduce}_{\tau} e_{1,2,3}^{\tau} & \searrow \text{reduce}_{\rho, \nu} e_{1,2,3}^{\tau} \\
\text{reduce}_{\tau} e_{1,2,3}^{\tau} & \searrow \text{reduce}_{\rho, \nu} e_{1,2,3}^{\tau}
\end{align*}
\]

\[
\begin{align*}
\text{fl}_g & = \text{id} \\
\text{fl}_{\rho_1, \rho_2} & = \text{id} \\
\text{fl}_{\rho_1, \rho_2} & = (\text{fl}_{\rho_1} \otimes \text{fl}_{\rho_2}) \circ \text{unzip}' \\
\text{fl}_{\rho, \nu} & = \text{cat}' \rho \circ \text{map}' \text{fl}_{\rho}
\end{align*}
\]

Figure 4.8: Type-indexed family of data-flattening operators \(\text{fl}_{\rho}\).

\[
\begin{align*}
\text{nest}_g & = \text{id} \\
\text{nest}_{\rho_1, \rho_2} & = \text{id} \\
\text{fl}_{\rho_1, \rho_2} & = \text{zip}' \circ (\text{nest}_{\rho_1} \otimes \text{nest}_{\rho_2}) \\
\text{fl}_{\rho, \nu} & = \text{map}' \text{nest}_\rho \circ \text{uncat}' \rho
\end{align*}
\]

Figure 4.9: Type-indexed family of nesting operators (inverse of data-flattening operators) \(\text{nest}_{\rho}\).
\[
g \sim g = \text{id}_g
\]
\[
[\tau] \sim \{\tau; \ t \ f\} = \text{toFA}_\tau
\]
\[
\{\tau; \ t \ f\} \sim [\tau] = \text{fromFA}_\tau
\]
\[
\{\tau; \ t \ f\} \sim \{\tau; \ t \ f\} = \text{cat}_{\tau}
\]
\[
\{\tau; \ t \ f\} \sim \{\tau; \ t \ f\} = \text{seg}_{\tau}
\]
\[
\{\tau; \ t \ f\} \sim \{\tau; \ t \ f\} = \text{unzip}_{\tau}
\]
\[
\{\tau; \ t \ f\} \sim \{\tau; \ t \ f\} = \text{zip}_{\tau}
\]

\[
\nu = \nu_1 = \nu_2
\]

Figure 4.10: Coercion operators.

\[
\text{id}_\tau \circ \text{id}_\tau = \text{id}_\tau
\]
\[
\text{toFA}_\tau \circ \text{fromFA}_\tau = \text{id}_{\{\tau; \ t \ f\}}
\]
\[
\text{fromFA}_\tau \circ \text{toFA}_\tau = \text{id}_{[\tau]}
\]
\[
\text{cat}_{\tau} \circ \text{seg}_{\tau} = \text{id}_{\{\tau; \ t \ f\}}
\]
\[
\text{seg}_{\tau} \circ \text{cat}_{\tau} = \text{id}_{\{\tau; \ t \ f\}}
\]
\[
\text{unzip}_{\{\tau_1, \tau_2\} ; \nu} \circ \text{zip}_{\{\tau_1, \tau_2\} ; \nu} = \text{id}_{\{\tau_1, \tau_2\} ; \nu}
\]
\[
\text{zip}_{\{\tau_1, \tau_2\} ; \nu} \circ \text{unzip}_{\{\tau_1, \tau_2\} ; \nu} = \text{id}_{\{\tau_1, \tau_2\} ; \nu}
\]

Figure 4.11: Cancelling coercion operators.
CHAPTER 5
IMPLEMENTATION

In this section, we describe the realization of the flattening transformation 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 or deactivated with a command-line control when running pmlc. In practice, it has turned out to be both feasible and profitable to apply the flattening transformation to nested-data parallel PML programs.

5.1 Implementation Overview

Programs that are fed to the PML compiler progress through a standard sequence of intermediate representations. The source file is first parsed into an untyped parse tree (PT). The parse tree is then passed to a typechecker which either produces a typed abstract syntax tree (AST) or fails (in the case of an ill-typed program). The typed AST is then 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. Next, pattern matches are checked and simplified. Then the AST is translated into a lower-level normalized typed language, BOM. From there, 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.

This process can be depicted as follows. The phases past BOM are elided, since flattening does not operate past that point. See Figure 5.1.

We integrate the flattening transformation into this compilation chain as follows. First, the compilation of implicitly-threaded constructs (labeled “it” above) is modified such that parallel arrays persist beyond it. Second, a flattening phase is inserted between AST and BOM. The phase applied flattening operators to parallel array valued expressions as needed. See Figure 5.2.
Flattening entails traversing the given program to change the representation of every parallel array, according to rules similar to those given in Chapter 4. Every parallel array has a different type after flattening than before. Since variables are typed in the AST representation, every variable bound to a parallel array, whether it be in a val binding or as part of a function parameter, must be replaced with a fresh variable with the appropriate flattened type. The transformation of types and representation changes propagate all throughout the program. Every context expecting a value of any parray type now must be altered to expect a new value of a different type.

The flattening transformation itself occurs in phases. The first phase is abstract flattening. In the abstract flattening phase, every expression that affected by flattening is identified and marked with a new abstract flattened type. The abstract flattening phase follows the theoretical flattening system presented in Chapter 4. It is called “abstract” because parallel arrays are transformed to flattened arrays without the compiler’s committing to any particular concrete representation thereof. The phase’s actions consist of replacing certain core parallel array operators with type-indexed
flattened operators, as well as inserting flattening operators to change representations of nested structures where necessary. The second phase is an optimizing flatten-operator fusion phase. In this phase, the compiler searches for flattening operations of no effect and discards them when possible. The third and last phase is concrete flattening, where the compiler does at last commit to particular concrete representations of flattened arrays and operations. The phases of flattening in pmlc are described in detail in the sections that follow.

### 5.2 Abstract Flattening

Abstract flattening is first and foremost concerned with altering the representation of parallel arrays; transformation of functions that consume them follows as a natural consequence.

The parallel arrays in a given program can be divided into two groups. First, there are arrays that are directly typed, one expression at a time, into the surface program. Second, there are arrays that are constructed through syntactic forms (comprehensions and ranges) and calls to basis library functions (tabulate).

In the first case, when an array is constructed expression-wise, the compiler determines what its post-flattening type should be, and inserts into the program an application of the appropriate flattening operator to that array. When an explicitly-enumerated parallel array contains only literals, or can be rewritten through compile-time transformations to contain only literals, then its flattening need not be delayed till run time; it can be flattened completely at compile time. This optimization presents no particular technical difficulties, but it is at present not implemented in pmlc.

In the second case, where arrays are built from syntactic forms or tabulations, we take care to catch special cases where we can apply powerful optimizations. By inspecting parallel comprehensions, we can identify opportunities for tab flattening (Section 3.5.2). At this stage in compilation, parallel comprehensions exist as their own kind of AST node, as do ranges. Neither one has yet been translated into function calls or otherwise eliminated. If every generator in a nested paral-
lel comprehension is a range, then that comprehension may be replaced by a flattened tab call. (This is a sufficient rather than a necessary condition.) To increase the number of parallel comprehensions we can recruit for tab flattening at this stage, we inline range expressions into parallel comprehensions to meet this condition more often.

With respect to functions that operate on parallel arrays, we view them as combinations of a group of core parallel array operations that we consider “blessed” and about which the compiler has special knowledge. The blessed operators are the following: `PArray.length`, `PArray.sub` (also written `{ in PML}), `PArray.tabulate and its variants`, `PArray.map`, `PArray.reduce`, `PArray.range` and `PArray.app`. Note that each operators is equipped with a working implementation in the PML basis library. If the flattening transformation is not applied, then those original implementations are used in all cases. However, if the flattening transformation is active, those implementations will all be discarded and replaced by appropriate flattened implementations wherever they are applied.

### 5.3 Flatten Operator Fusion

Here is the family of flattening operators as a PML datatype, as it appears in the compiler.

```plaintext
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
```

Our formulation of the flattening transformation sometimes introduces operators (identity operators, compositions of identity operators, \textit{etc.}) that can be safely discarded at compile time. The present compiler phase identifies unnecessary flattening operations and removes them. It does so by applying the following rules repeatedly until it reaches a fixpoint.

- The application of any operator `ID` to any term `e` is rewritten to `e`. 
Any operator of the form \( \text{Map}(ID, \_ ) \) 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.

## 5.4 Concrete Flattening

Concrete flattening is the compilation phase where abstract flattening operations — essentially placeholders — are replaced with their specific implementations. In concrete flattening, the compiler commits to particular representations of data structures and the operations that work on them.

PML flattening has been designed so that concrete flattening need not take place in only one way. In the current implementation, there is a particular concrete flattened array type to which all nested arrays are transformed (the \( \text{farray} \)), but in the same framework it would be possible to use different representations to explore their performance characteristics.

In this phase, the parallel array value \([ [ 1, 2, 3 ] ]\) will still exist as an abstract parallel array, which is its own variant in the AST datatype, and in itself makes no commitment to any particular runtime representation. The concrete flattening phase will process that value by building a rope of its data elements and constructing a shape tree of its nesting structure information.

Also during this phase, “flattening operators” are realized. Flattening operators are a subset of the coercion operators defined in Chapter 4: they are those coercion operators who convert their arguments to “flat” values for the definition of “flat” given above. As the coercion operators are defined inductively, being either primitives or constructed out of coercion operators, flattening operators either have concrete implementations in the basis library (corresponding to the primitives in Figure 4.10), or are synthesized by composition of other coercion operators whose implementations are themselves either primitive or recursively synthesized.

Array operators themselves are also synthesized during concrete flattening. In the following
code fragment, a \texttt{PArray.map} is applied to an array of pairs of ground terms:

\begin{verbatim}
  val f : int * bool -> int = ...
  val a = PArray.map f [|(1, true), (2, false), ...|]
\end{verbatim}

Since we are going to both unzip and monomorphize the array of pairs as part of the flattening transformation, we need a version of \texttt{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. Such \texttt{map}s represent an infinite family of operators; it is neither desirable nor possible to implement them all in advance and select them out of the basis library as needed. In the present case, we need an operator of the following type:

\begin{verbatim}
  map_ibi : int_farray * bool_farray -> int_farray
\end{verbatim}

The compiler is able to synthesize the map operators it needs based on the types. In such cases, the code follows a standard pattern, so synthesis of such operators is a simple mechanical process.

### 5.5 Optimizations

Applying the flattening transformation allows us to employ various powerful optimizations, including monomorphization and tab flattening, both of which are discussed here.

#### 5.5.1 Monomorphization

When possible, the compiler chooses a monomorphic representation for flattened arrays. This is manifest in the rope component of the flat arrays. The polymorphic rope datatype (Section 3.3) point to sequences of boxed, heap-allocated values at its leaves, whereas monomorphic ropes such as \texttt{IntRope} and \texttt{DoubleRope} point to contiguous sequences of unboxed values at their leaves. Monomorphically ropes are smaller and faster to compute with than polymorphic ones, and we observe
performance benefits when using them even in benchmarks when no proper flattening (concatenation of data in nested structures) is involved (see Section 6.1 for an example).

Unzipping tuples leads to more opportunities for monomorphizing arrays. An array of pairs of integers, when unzipped, becomes a pair of arrays of integers, each of which will eventually be represented by a flattened array based on an IntRope. In this sense tuple unzipping and monomorphization go hand in hand. In the past, this was in fact necessary, since all flattened arrays were what we would now call monomorphic — such were the machines for which the flattening transformation was originally conceived.

At present, when it is no longer necessary to convert everything into a flat vector of ground types, we can choose whether or not we wish to, say, unzip tuples to expose more opportunities for monomorphization. It might, in fact, not be worth it, as we have discussed above.

5.5.2 Tab Flattening

With flattening disabled, nested parallel comprehensions over ranges are compiled to nested tabulations. With flattening enabled, they are compiled to flattened tabulations, which yield notable performance improvements.

In one dimension.

Start with this.

\[
[| e | i \text{ in } [| f \text{ to } t \text{ by } s |] |]
\]

Construct this.

\[
\text{fun } g \ i = e
\]

Also this.

\[
\text{fun indexMap1D } (f, t, s) = (\text{fn } k \Rightarrow f + (k * s))
\]

Finally this.

53
fun tab1D ((f,t,s), g) = let
  val ix = indexMap1D (f,t,s)
  fun g’ k = g (ix k)
  val n = nElts (f,t,s)
  val data = Rope.tabulate (n, g’)
  val shape = Shape.regularShape [(f,t,s)]
  in
    FArray (data, shape)
  end

In two dimensions.

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

In m dimensions.

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

Where possible, the PML compiler rewrites parallel comprehensions to calls to PArray.tabulate.

The simplest case to rewrite is

[| e | i in [| 0 to n |] |]

assuming n is bound to a positive integer value. This is rewritten to

PArray.tabulate (n, fn i => e)

Ranges need not begin at zero, of course, and their steps may be of any size, as optionally specified
in the range’s by clause. To represent a general one-dimensional parallel comprehension, we must
include a range that starts and ends at arbitrary points, and furthermore takes steps of arbitrary size.
The translation of this parallel comprehension is still to a call to \texttt{tabulate}, but it is now necessary to perform arithmetic to map the natural numbers onto the elements in the sequence as specified by our range semantics. In order to call \texttt{tabulate} correctly, we first compute the number of elements in the range according to the range semantics specified above. In PML, the number of elements is given by the function \texttt{nElts}, defined as follows:

\begin{verbatim}
fun nElts (f, t, s) = 1 + Int.max (0, (t-f) div s)
\end{verbatim}

(Here, \texttt{f}, \texttt{t} and \texttt{s} are mnemonic names for “from,” “to” and “step.”) Note that if \texttt{s} is 0, the computation of the number of elements will raise an exception when it attempts to divide by 0; this is fitting, as no meaningful answer can be computed with step size 0 in any case.

[[ John asks: “is that the semantics of ranges?” not exactly. I need to spell this out more. ]]}

Making use of \texttt{nElts} to compute the number of elements, we call \texttt{PArray.tabulate} with a function that has been constructed to map the first \texttt{n} natural numbers onto the range’s implied sequence.

\begin{verbatim}
let
  val n = nElts (f, t, s)
  fun g k = let val i = f + k * s in e end
in
  PArray.tabulate (n, g)
end
[[ John: “use fonts to distinguish between vars and metavars” ]]}

In the PML compiler, this technique is generalized to higher dimensions, as long as the parallel comprehension being compiled adheres to a few criteria: first, at every dimension it must be traversing a range expression, and it must be free of filters. In such cases its precise shape can be easily computed before evaluation and the necessary index arithmetic can be performed according to a simple pattern.
Let us consider the two-dimensional case before generalizing to higher dimensions. Here is a two-dimensional parallel comprehension, ripe for tabulation. (We bind its ranges to names for clarity.)

```
let
  val iRng = [| a to b by c |]
  val jRng = [| d to e by f |]
in
  [| [| i*j | j in jRng |] | i in iRng |]
end
```

To compile this parallel comprehension, we insert calls to `nElts` to compute how many elements are in each of its two dimensions, we construct a function `g` which performs index math on its single integer argument, then we call `tab2D` on `g` with the number of elements in the whole two-dimensional structure, which is simply the product of the number of elements in each of its dimensions.

```
let
  val iDim = nElts (a, b, c)
  val jDim = nElts (d, e, f)
  fun g k = let val (i,j) = (k div jDim, k mod jDim)
in i*j end
in
  PArray.tab2D (iDim*jDim, g)
end
```

### 5.5.3 Range Inlining

Our implementation of tab flattening looks for particular syntactic forms in the AST: parallel comprehensions whose inner parallel arrays are ranges.

Consider the following code fragment.

```
val r = [| e1 to e2 by e3 |]
val c = [| e | i in r |]
```

Our range inlining pass transforms this code into

```
val rLo = e1
val rHi = e2
val rStep = e3
val r = [| rLo to rHi by rStep |]
val c = [| e | i in [| rLo to rHi by rStep |] |]
```
The compiler recognizes the transformed c as an opportunity to use a tabulation rather than a map.

notes:

• subcomputations are not hoisted if they are constants or variables

• hoisting the range subcomputations and binding them in this order preserves order of evaluation while eliminating duplication of computation

• if range value is unused elsewhere, it will be discarded in later contraction phase
CHAPTER 6
EVALUATION

This section demonstrates that, across a variety of benchmarks, PML programs compiled with the flattening transformation scale very well on many processors (up to forty-eight in our tests), and compare favorably, with respect to performance, to comparable programs written in mature sequential languages: C and SML. In all cases our benchmark programs are written in a natural style: that is, PML programs make use of nested comprehensions to build nested arrays, where C programs use nested loops to populate arrays. In other words, we have not tuned our results by forcing C or SML programs to abandon their standard idioms to imitate too closely the behavior of the PML programs they are compared with; rather, we have let those languages do the same essential computations in the ways they were intended to do so. As such, we are matching strength to strength. We demonstrate here that, even though the PML compiler at present has standard well-known optimizations remaining to be implemented, it is still able to run within reasonable factors of optimized sequential code, and run much faster than sequential code on many processors.

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 AMD Opteron 6172 “Magny Cours” processor, 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.

We ran each experiment 49 times and we report the average (arithmetic mean) results in the plots that follow. Standard deviations were small across the board (single digit percents) and as such they are not depicted in the plots that follow.
code goes here...

Figure 6.1: PML code for flat-arith.

Figure 6.2: Speedup of flat-arith, with flattening transformation applied, vs. itself at one processor.

6.1 Flat Arithmetic

This benchmark demonstrates that for a program where no flattening needs to be done, the monomorphization entailed in our flattening pass yields good improvements nonetheless.

The flat arithmetic benchmark performs a fixed number of arithmetic operations for every element of a parallel array of integers. There is no flattening per se to be done in compiling this benchmark, but monomorphization of the integer array improves performance by a factor of about three. We compare PML to C here. The sequential C program is more than eight times as fast as the monomorphized PML program run on a single processor, but the PML program scales well (whether flattened or not) and the faster flattened PML program outperforms the C program at about 9(?) processors. On forty-eight processors, flattened PML is faster than C by a factor of about 3.48.


Figure 6.3: Speedup of flat-arith vs. sequential C baseline.

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

Figure 6.4: PML code for mandelbrot.

### 6.2 Mandelbrot

Computing the Mandelbrot set begins to show the real power of the flattening transformation. In PML, computation is performed over a two-dimensional array of integers. At each pair-indexed location in that array, the pair is mapped to a complex number, and a recursive iterative function is called on that number to determine its membership in the Mandelbrot set. From one pair to the next, the amount of work to be done is unpredictable; the boundary of the Mandelbrot set is well-known as a fractal (possessing the characteristic of “infinite brokenness”). The PML program is written idiomatically as a nested comprehension. The comprehension is subject to the tab flattening compilation technique discussed above.

The C program we compare this to executes a nested loop, and, to write idiomatically and
Figure 6.5: Speedup of mandelbrot, with flattening transformation applied, vs. itself at one processor.

maximize performance, runs a while loop to determine Mandelbrot set membership (where PML employed a recursive function).

Flat Mandelbrot scales beautifully up to forty-eight processors, as shown in Figure ???. Unflattened Mandelbrot does not scale well. Both PML programs perform well compared to the C program.

### 6.3 Ray Tracing

Our ray tracing benchmark computes the image of a scene graph consisting of $n$ spheres with transparency and reflection. The amount of work needed to compute each pixel varies unpredictably. We compare PML to SML compiled with MLton.

Once again, flat PML scales beautifully, unflat PML does not, and PML performs well compared to its sequential counterpart.
fun raytracer sz = let
  ...
  fun trace (i, j) = tracepixel (world, lights, i, j, r0, sx, sy)
  val side = [| 0 to (sz-1) |]
  val scene = [| [| trace (i, j) | j in side |] | i in side |]
  ...

Figure 6.7: PML code for raytracer.

6.4 Mandelbulb

The Mandelbulb [42] is a fractal-like three dimensional volume. Each voxel bounded by a cube around the origin is either in the Mandelbulb or not, and its membership is decided by an iterative computation (similar to the Mandelbrot set).

As the Mandelbrot and ray tracer benchmarks have demonstrated the effectiveness of tab flattening at two dimensions, Mandelbulb demonstrates its effectiveness at three. The beauty of tab flattening is that it makes all nested iterations, no matter how deeply nested, look alike.

Once again, flat PML scales beautifully, unflat PML does not, and PML performs well compared to its sequential C counterpart.
Figure 6.8: Speedup of raytracer, with flattening transformation applied, vs. itself at one processor.

6.5 Sparse Matrix Vector Multiplication

smvm
Figure 6.9: Speedup of raytracer vs. sequential SML baseline.

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

Figure 6.10: PML code for mandelbulb.
Figure 6.11: Speedup of mandelbulb, with flattening transformation applied, vs. itself at one processor.

Figure 6.12: Speedup of mandelbulb vs. sequential C baseline.
CHAPTER 7
CONCLUSION

We have formalized a flattening transformation on a model nested data parallel language. Our flattening transformation is based on compilation to a novel flattened-array representation which pairs flat data vectors with shape trees. We have used that formalization to implement a flattening transformation into the compiler for PML, a full-featured, realistic parallel functional language with a broad assortment of parallel constructs. We have exploited the flattening transformation to implement various optimizations, including unboxing of rope elements (monomorphization) and tab flattening. We have demonstrated substantial performance improvements of flattened programs vs. non-flattened ones. Notably, our techniques are especially effective applied to programs that are written in a declarative NDP style. In other words, flattening improves performance while supporting PML’s natural programming idioms. Our experimental results show that flattened programs significantly outperform their non-flattened counterparts and scale linearly and monotonically up to forty-eight cores. As we had conjectured in earlier work [3], flattening and the lazy tree splitting scheduling technique are an effective combination.

Flattened PML execution times compare favorably to those of optimized C and SML programs. For example, computing the Mandelbrot set on forty-eight cores with a PML program is currently a better than twentyfold speedup over a corresponding sequential C program.

7.1 Future Work

7.1.1 Theory

The model presented here could be extended to include DPH-style datatype flattening.

Definitions of some specific hybrid flattening transformations.

Model extended with a formal cost semantics.
7.1.2 Implementation

SSE instructions.

Vector machines.

Implementation: perform FT at a later phase in compilation. As it stands, range inlining happens on AST pre-FT. But really it should be part of the standard inlining phase at BOM.

Experiment with farrays with sequence types other than ropes. (Concrete flattening.)

Some form of metaprogramming would be useful in the implementation of synthesis of operators like specialized maps, tabs, etc.
APPENDIX A

PROOFS

Theorem 4.2.3: Given: \( e^\tau \) ok and \( \tau \downarrow \rho \) and \( e^\tau \downarrow e^{(\tau/\rho)} \).

Then \( e^{(\tau/\rho)} \) ok.

Proof. The proof proceeds by induction on —.

- \( e \) is a constant \( b \).

  Given: \( b^g \) ok, \( g \downarrow g \) (by \( \downarrow \) on types), and \( b^g \downarrow b^g \) (by \( \downarrow \) on terms).

  Goal: \( b^g \) ok.

  The goal is established in the premise.
REFERENCES


