PREPRINT: To be presented at PPoPP '13, February 23–27, 2013, Shenzhen, China.

Data-Only Flattening for Nested Data Parallelism

Lars Bergstrom
John Reppy
Stephen Rosen
Adam Shaw
University of Chicago
{larsberg,jhr,sirosen,ams}@cs.uchicago.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Mike Rainey
Max Planck Institute for Software Systems
mrainey@mpi-sws.org

Abstract

Data parallelism has proven to be an effective technique for high-level programming of a certain class of parallel applications, but it is not well suited to irregular parallel computations. Blelloch and others proposed *nested data parallelism* (NDP) as a language mechanism for programming irregular parallel applications in a declarative data-parallel style. The key to this approach is a compiler transformation that *flattens* the NDP computation and data structures into a form that can be executed efficiently on a wide-vector SIMD architecture. Unfortunately, this technique is ill suited to execution on today's multicore machines. We present a new technique, called *data-only flattening*, for the compilation of NDP, which is suitable for multicore architectures. Data-only flattening transforms nested data structures in order to expose programs to various optimizations while leaving control structures intact. We present a formal semantics of data-only flattening in a core language with a rewriting system. We demonstrate the effectiveness of this technique in the Parallel ML implementation and we report encouraging experimental results across various benchmark applications.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (Functional) Programming, Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors – Compilers, Optimization

Keywords multicore, NESL, nested data parallelism, compilers

1. Introduction

Data-parallel computations are ones in which a function is applied to the elements of a collection (e.g., set or sequence) in parallel. Data parallelism is an effective technique to take advantage of parallel hardware and is especially suited to large-scale parallelism [10], but most languages that support data parallelism limit that support to *flat data parallelism* (FDP), where the computation being mapped over the collection does not contain nested data parallel computation. While FDP is very effective for many regular-parallel applications, it is not well-suited for irregular parallel applications. To address this weakness, Blelloch and others proposed *nested data parallelism* (NDP) [4, 5, 8, 20].

The basic operation in both flat and nested data parallelism is the *parallel map operation*, which applies a function to the elements of a collection in parallel. What distinguishes NDP from FDP is that elements of the collection may themselves be collections, and the mapped computation may itself involve parallel maps over the nested collections. Because the nested collections in an NDP computation may vary in size, it is difficult to ensure a balanced partitioning of work across multiple processors and it is difficult to execute the parallel computation with Single-Instruction-Multiple-Data (SIMD) architectures.

Blelloch addressed these challenges with an approach that he called *flattening*. Flattening (also called *vectorisation*) is a technique for converting irregular nested computations into regular computations on flat arrays. This approach, which was invented for first-order NDP by Sabot and Blelloch [2, 5] and extended to full-featured higher-order functional languages by Keller, Chakravarty, and others [7, 8, 14, 16], transforms both the data representations and the code so that the computation can be executed by a SIMD machine. An alternative approach, which is used in the Manticore system, is to execute the parallel map operations as fork-join parallelism and to rely on efficient work-stealing techniques to handle load balancing [1, 23].

In this paper, we introduce a new approach to implementing NDP constructs that is based on the idea of flattening the nested data representations, but not vectorising the code [25]. In our prototype implementation, we build on the Manticore compiler for PML [11], a parallel dialect of Standard ML [17].

This paper makes the following contributions:

1. We introduce a novel approach to implementing NDP that is well suited to execution on MIMD architectures, such as modern multicore processors.
2. We provide a formalization of our approach using a core calculus.
3. We present empirical evidence that our approach improves the performance of code that executes over irregular data while also preserving performance on regular data.

2. Full flattening for NDP

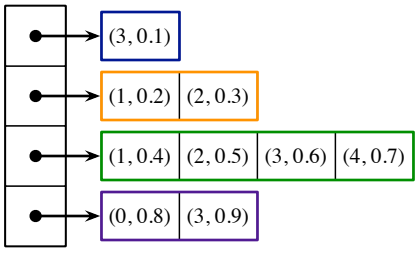
As a motivating example, consider the NDP code to implement sparse-matrix times dense-vector multiplication. We represent a dense vector as a parallel array of floating-point values:

```
type dense_vec = float parray
```

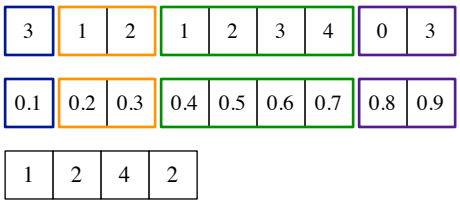
and a sparse vector is a parallel array of index-value pairs:

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.1 & 0.0 \\ 0.0 & 0.2 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.4 & 0.5 & 0.6 & 0.7 \\ 0.8 & 0.0 & 0.0 & 0.9 & 0.0 \end{bmatrix}$$

(a) a matrix



(b) its sparse representation



(c) its flat representation

Figure 1. Representations of a sparse matrix

```
type sparse_vec = (int * float) parray
```

In this representation, we record only the non-zero entries of the vector, paired with their indices. A sparse matrix is a parallel array of sparse vectors:

```
type sparse_mat = sparse_vec parray
```

Figure 1(a) and (b) illustrate a matrix and its sparse representation.

The `sumP` operator is a parallel reduction that computes the sum of the elements of a parallel array of floats:

```
val sumP : float parray -> float
```

We define multiplying a dense vector by a sparse-matrix as the following high-level NDP function:

```
fun smvm (sm : sparse_mat, v : dense_vec) =
  [| sumP [| x * v!i | (i,x) in sv |] | sv in sm |]
```

The technical challenge is to find the meeting point between this elegant declarative code and an efficient implementation that can exploit powerful multicore architectures.

As one solution to this challenge, Blelloch and Sabot introduced the NESL language and the *flattening transformation* to compile nested-data-parallel programs for wide-vector parallel machines [2, 5]. NESL is an ML-like language with nested-data-parallel features. NESL's provides only scalars and associated operators, sequences, simple datatypes, conditionals, let bindings, top-level function definitions, and a parallel *apply-to-each* construct. Blelloch's flattening transformation then substantially changes both the data structures and code. This transformation is especially effective in treating irregular parallelism: certain operations like the parallel *segmented sum* operation, where sums are computed over an array of arrays of numbers, complete in the same number of steps regardless of the irregularity of the shapes of the segments.

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

datatype shape
  = Lf of int * int
  | Nd of shape rope

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

Figure 2. Datatypes `rope`, `shape` and `farray`.

The flattening transformation yields flattened arrays consisting of two components: one or more flat data vectors (more than one in the case of unzipped tuples), containing the elements of the nested array in left-to-right order, and one or more *segment-descriptors*. In NESL, a segment-descriptor is always a flat vector of integers, and a flattened array carries with it one segment-descriptor for each level of nesting. Figure 1(c) illustrates the flat representation of the example sparse matrix.

2.1 Challenges for full flattening

Full flattening, while a successful innovation, can produce inefficient code in common cases, including conditionals and certain regular nested parallel programs.

When full flattening is applied to conditionals, the resulting code generates many intermediate vectors, as we demonstrate here in a simplified example. Here is a function that replaces the zeroes in a vector with ones:

```
fun g (xs : dense_vec) =
  [| if x=0.0 then 1.0 else x | x in xs |]
```

The transformed code (which uses standard `split` and `combine` operations [2]) builds intermediate vectors to handle partitioning the data elements, performs the appropriate conditional work on each partition of the data, and then combines them:

```
fun g_full_flat (xs : dense_vec) = let
  val flags = [| x=0.0 | x in xs |]
  val (zs, nzs) = split (xs, flags)
  val zs' = [| 1.0 | z in zs |]
in
  combine (zs', nzs, flags)
end
```

This approach makes sense for SIMD architectures with large penalties for failing to keep the vector registers full, but on multicore machines those benefits can be overwhelmed by the large number of memory operations. There are many more extended examples of this kind of transformation in the literature [2, 13]. Our approach, by contrast, will transform `g` to operate on a flattened dense vector, but will not otherwise transform the code; to the point, it will not necessitate generating any intermediates.

With respect to regular nested parallel programs, problems similar to dense matrix multiplication suffer from a polynomial space increase under full flattening, which is a known serious problem [26] also due to excessive data copying. The present work describes a system designed to address these problems with traditional full flattening by avoiding extra data copies, both due to splitting of conditional branches and duplication of vectors within nested parallel applications.

3. Data-Only Flattening

We present data-only flattening in the context of a broader system for *hybrid flattening*. Hybrid flattening is a program representation that allows both flat and nested representations of parallel arrays and which has coercions for transforming between representations.

This representation allows flexibility in choosing when it is profitable to use a flat representation vs. a nested representation. Hybrid flattening does not itself express or embody a particular transformation policy.

Whereas Manticore without flattening compiles nested parallel arrays to nested ropes [6], with flattening it compiles nested parallel arrays to flattened arrays. Flattened arrays, like nested arrays as compiled by NESL, consist of two pieces: a flat data vector, and a value representing the structure of the nested array called a *shape tree*. By means of standard unzipping transformations, nested arrays of tuples are compiled to tuples of flattened arrays. In our implementation, flattened arrays are represented by the polymorphic `farray` datatype. To represent the flat data vector part of flattened arrays, we use ropes, exploiting Manticore's existing rope infrastructure to compute in parallel with them. Figure 2 presents the PML datatype definitions for `rope`, `shape`, for shape trees, and `farray`. Note that ropes and shape trees are internal representations; the programmer uses them only indirectly.

Shape trees are our adaptation of segment descriptors in the NESL tradition. A `shape` is an n-ary tree whose leaves store integer pairs. Each leaf contains the starting index and the index of the element following the segment of data in an `farray`. The shape `Shape.Lf (i, i+n)` describes a length-`n` segment starting at `i` and ending at the last position before `i+n`.[1]

A simple, flat parallel array of integers such as

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

has the following `farray` representation:[2]

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

The data in the original sequence appears here in the original order as a `Rope.Leaf` and the accompanying `shape` indicates 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:

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

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

Literal values can be flattened at transformation time directly, with the representation change incurring no runtime cost. In general, however, the compiler must cope with arbitrarily nested arrays whose dimensions are known only at runtime. In such cases, the compiler needs to arrange for flattening to take place at runtime. We handle this issue, by inserting coercion operators that perform runtime flattening. When nested arrays are transformed into flattened arrays, all operations applied to those array values must be correspondingly transformed. Our approach to this problem is to provide a core group of type-indexed families of array operators, each of which performs its operation at every array type in its family.

[1] In practice, this choice of bounds is a convenient convention that guards against common fencepost errors.

[2] For brevity, we present the sequence in the `Rope.Leaf` node and the sequence of leaves in the `Shape.Nd` with list syntax.

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

Figure 3. Flatland: types.

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

Figure 4. Flatland: terms.

This group contains parallel array subscripting and parallel maps, filters, and reductions over parallel arrays. All operations on parallel arrays are either members of this core group or are built from members of this group. As such, transformation of the type-indexed operators matching the transformations of data structures is sufficient to preserve the program's behavior.

4. Formalization

The system presented here, *Flatland*, consists of a model language and a variety of rewriting systems and judgments. Its model language is an explicitly-typed, monomorphic, strict, pure functional language. Flattened and non-flattened terms commingle in Flatland: there is no inherent distinction between source language and target language.

Figure 3 presents Flatland's types, ranged over by the metavariable τ. We use subscript indices (τ_i) and overbars ($\overline{\tau}$) to distinguish types from one another. The type language consists of ground types int and bool, pairs, functions, parallel arrays, and flattened parallel arrays. Flattened-parallel types include *shape types* as subcomponents. Shape types, ranged over by the metavariable ν (for "nesting"), record an array's nesting depth.

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

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

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

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

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

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

$$\frac{\vdash \tau \rhd \overline{\tau}\ ok}{\Gamma \vdash (\tau \rhd \overline{\tau})^{\tau \to \overline{\tau}}\ ok} \qquad \ldots$$

Figure 5. Well-typedness of terms (selected rules).

$$\frac{}{\vdash \tau \rhd \tau\ ok}$$

$$\frac{}{\vdash [\tau] \rhd \{\tau\ ;\ lf\}\ ok} \qquad \frac{}{\vdash \{\tau\ ;\ lf\} \rhd [\tau]\ ok}$$

$$\frac{}{\vdash [(\tau_1,\tau_2)] \rhd ([\tau_1],[\tau_2])\ ok} \qquad \frac{}{\vdash ([\tau_1],[\tau_2]) \rhd [(\tau_1,\tau_2)]\ ok}$$

$$\frac{\vdash \tau_1 \rhd \tau_2\ ok \qquad \vdash \tau_2 \rhd \tau_3\ ok}{\vdash \tau_1 \rhd \tau_3\ ok} \qquad \ldots$$

Figure 6. Well-formedness of coercions (selected rules).

Figure 4 contains Flatland's term language. Every term t includes an explicit type as a superscript. The metavariable b ranges over constants, and x ranges over variables. As in the type language, parallel-array types are written with square brackets, and flattened-array types with curly braces. Every flattened array includes a shape tree. Since the language is monomorphic, array operators — in Flatland these are subscript, map, filter, and reduce — do not have polymorphic implementations. Instead, we assume there is a type-indexed family for each one.

Flattening and unflattening operators are represented in our system by *coercions*. For the coercion that transforms values of type τ_1 into values of type τ_2, we write $\tau_1 \rhd \tau_2$. Except for identity coercions, these are potentially expensive representation-changing operations; during compilation, we try to eliminate as many of them as possible.

Well-formedness of coercions, so that nonsensical coercions like $int \rhd (int, int)$ are rejected, is given in Figure 6. Well-typedness of terms is in Figure 5, and the rules for calculating shape types appear in Figure 7.[3] For array type τ, we use the notation

[3] Due to space limitations, only selected rules are presented; the full rule sets appear in [25].

$$\frac{\Gamma \vdash {e_1}^{int}\ ok \qquad \Gamma \vdash {e_2}^{int}\ ok}{\Gamma \vdash \mathbf{lf}({e_1}^{int}, {e_2}^{int}) : lf}$$

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

Figure 7. Calculation of shape types.

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

Figure 8. Array-type subscripting.

$$\frac{}{\tau\ \mathbf{L}\ \tau}$$

$$\frac{\tau_1\ \mathbf{L}\ \tau_2}{\tau_2\ \mathbf{L}\ \tau_1} \qquad \frac{\tau_1\ \mathbf{L}\ \tau_2 \qquad \tau_2\ \mathbf{L}\ \tau_3}{\tau_1\ \mathbf{L}\ \tau_3}$$

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

$$\frac{}{\{\{\tau\ ;\ \nu\}\ ;\ lf\}\ \mathbf{L}\ \{\tau\ ;\ nd(\nu)\}}$$

$$\frac{}{\{\{\tau\ ;\ \nu\}\ ;\ nd(\nu')\}\ \mathbf{L}\ \{\{\tau\ ;\ nd(\nu)\}\ ;\ \nu'\}}$$

$$\frac{\tau_1\ \mathbf{L}\ \tau_2}{[\tau_1]\ \mathbf{L}\ [\tau_2]} \qquad \frac{\tau_1\ \mathbf{L}\ \tau_2}{[\tau_1]\ \mathbf{L}\ \{\tau_2\ ;\ lf\}}$$

$$\frac{\tau_1\ \mathbf{L}\ \tau_2}{\{\tau_1\ ;\ \nu\}\ \mathbf{L}\ \{\tau_2\ ;\ \nu\}}$$

$$\frac{\tau_1\ \mathbf{L}\ \tau_1' \qquad \tau_2\ \mathbf{L}\ \tau_2'}{(\tau_1,\tau_2)\ \mathbf{L}\ (\tau_1',\tau_2')} \qquad \frac{\tau_1\ \mathbf{L}\ \tau_1' \qquad \tau_2\ \mathbf{L}\ \tau_2'}{\tau_1 \to \tau_2\ \mathbf{L}\ \tau_1' \to \tau_2'}$$

$$\frac{\tau_1\ \mathbf{L}\ [\tau_2]}{\tau_1\ \mathbf{A}\ \tau_2}$$

Figure 9. Definition of $\tau_1\ \mathbf{A}\ \tau_2$ and its auxiliary relation $\tau_1\ \mathbf{L}\ \tau_2$.

$(!\,\tau)$ to mean the type of the element selected by subscript out of a value of type τ. The return type of a particular subscript operator is calculated from its domain. Figure 8 gives the definition of $(!\,\tau)$. If $(!\,\tau)$ cannot be computed from these rules, it is undefined.

In order to understand the typing of the type-indexed operators, one needs to understand the relation $\mathbf{A}$. For a type τ, we cannot definitively write $[\tau]$ to mean "array of τ," because $\{\tau\ ;\ lf\}$ is also an array of τ, albeit in a different representation. Furthermore, if τ is a pair type (τ_1, τ_2), then $[(\tau_1, \tau_2)]$ and $([\tau_1], [\tau_2])$ (and more) are also arrays of τ, and so on. Thus we appeal to the relation $\tau_1\ \mathbf{A}\ \tau_2$ for "τ_1 is an array of τ_2." For arrays of (int, int), for example, all

$$\frac{}{e^{\tau} \mapsto (\tau \triangleright \tau)\ e^{\tau}} \qquad \frac{}{(\tau \triangleright \tau)\ e^{\tau} \mapsto e^{\tau}}$$

$$\frac{\vdash \tau_1 \triangleright \tau_2\ ok \qquad \vdash \tau_2 \triangleright \tau_3\ ok}{\tau_1 \triangleright \tau_3 \mapsto (\tau_2 \triangleright \tau_3) \circ (\tau_1 \triangleright \tau_2)} \qquad \frac{}{(\tau_2 \triangleright \tau_3) \circ (\tau_1 \triangleright \tau_2) \mapsto \tau_1 \triangleright \tau_3}$$

$$\frac{}{(\mathbf{let}\ x = t_1\ \mathbf{in}\ (\overline{\tau} \triangleright \tau)\ t_2)^{\tau} \mapsto ((\overline{\tau} \triangleright \tau)\ (\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2))^{\tau}}$$

$$\frac{\vdash \overline{\tau} \triangleright \tau\ ok}{x^{\tau} \mapsto ((\overline{\tau} \triangleright \tau)\ \overline{x}^{\overline{\tau}})^{\tau}} \qquad \ldots$$

Figure 10. Coercion distribution (selected rules), by which coercions are introduced, eliminated and otherwise manipulated.

of the following are established by $\mathbf{A}$:

$$\begin{array}{rcl} [(int, int)] & \mathbf{A} & (int, int) \\ ([int], [int]) & \mathbf{A} & (int, int) \\ \{(int, int)\ ;\ lf\} & \mathbf{A} & (int, int) \\ (\{int\ ;\ lf\}, \{int\ ;\ lf\}) & \mathbf{A} & (int, int) \end{array}$$

These types correspond to arrays of integer pairs and all equivalent representations under well-formed coercions. The relation $\mathbf{A}$ is defined in Figure 9. $\mathbf{A}$ is defined by defining an auxiliary relation $\mathbf{L}$ that verifies that two types are both representations of the same scalar or array type. We use the name $\mathbf{L}$ for the relation because it tells us that one type is "on the same level as" another. All arrays of type τ can be coerced between one another. Formally, if $\tau_1\ \mathbf{A}\ \tau$, then $\tau_2\ \mathbf{A}\ \tau \Leftrightarrow \vdash \tau_1 \triangleright \tau_2\ ok$ (proven elsewhere [25].)

We specify array operators by writing their type indices in a subscript. $!_{\tau}$ is the operator that selects elements from an array type τ. $\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_4)}$ takes two arguments, a function of type $\tau_1 \to \tau_2$ and a term of array type τ_3, and produces a term of array type τ_4. **filt** and **red** carry similar type subscripts. The typing rules of **map**, **filt** and **red** all appeal to $\mathbf{A}$.

To perform a flattening step, we insert a coercion. A standard step in flattening transformations is to unzip arrays of pairs of scalars—that is, to reshape an array of pairs into a pair of arrays. The coercion that unzips an array of integer pairs is written $[(int, int)] \triangleright ([int], [int])$. Its inverse coercion $([int], [int]) \triangleright [(int, int)]$ is also part of the language of coercions. If it is ever the case that a pair of inverse coercions, like these two, are successively applied to a value, they may be rewritten to an identity coercion, which can in turn be removed from the program.

Well-formed programs are guaranteed to remain well-formed under any legal transformation in our rewriting system. The top-level type of the whole program remains fixed under transformation, but the types of the subexpressions within a program may change. The well-formedness guarantee is maintained by introducing inverse coercions for every coercion introduced into the program, to maintain the stability of the types in the program. For example, every time a coercion is applied to the value to which a variable *x* is bound, the inverse coercion is introduced at every use of coerced *x*. This ensures that no clients of *x* are put in a position to compute with a term of the wrong type, post-coercion. Type-indexed operators such as **map** and **filt** have the ability to absorb type coercions, since, for each one, there are many implementations from which to choose, and this helps reduce the number of type coercions in the transformed program.

To provide intuition about how this machinery works, here is an example of *Flatland* in action. Consider the following program:

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

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

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

$$\frac{\overline{\tau}_3\ \mathbf{A}\ \tau_1 \qquad \overline{\tau}_4\ \mathbf{A}\ \tau_2}{\begin{array}{l}(\mathbf{map}_{(\tau_1,\tau_2,\tau_3,\tau_4)}\ (t_1, t_2))^{\tau_4} \\ \quad \mapsto ((\overline{\tau}_4 \triangleright \tau_4)\ (\mathbf{map}_{(\tau_1,\tau_2,\overline{\tau}_3,\overline{\tau}_4)}\ (t_1, (\tau_3 \triangleright \overline{\tau}_3)\ t_2)))^{\tau_4}\end{array}}$$

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

Figure 11. Coercion introductions with type-indexed operators. Note how the indexing types change to accommodate the coercions introduced.

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

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

Figure 12. Coercion propagation rules. Coercions introduced at binding sites necessitate substitutions accordingly.

$$\begin{array}{rl} & \mathbf{let}\ ns = [1,2,3]\ \mathbf{in}\ (ns\ !_{[int]}\ 0) \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ (\uparrow\ \overline{ns})\ !_{[int]}\ 0 \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ (int \triangleright int)\ ((\downarrow\ (\uparrow\ \overline{ns}))\ !_{\{int\ ;\ lf\}}\ 0) \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ (\downarrow\ (\uparrow\ \overline{ns}))\ !_{\{int\ ;\ lf\}}\ 0 \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ ((\downarrow \circ \uparrow)\ \overline{ns})\ !_{\{int\ ;\ lf\}}\ 0 \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ ((\{int\ ;\ lf\} \triangleright \{int\ ;\ lf\})\ \overline{ns})\ !_{\{int\ ;\ lf\}}\ 0 \\ \mapsto & \mathbf{let}\ \overline{ns} = \downarrow\ [1,2,3]\ \mathbf{in}\ (\overline{ns}\ !_{\{int\ ;\ lf\}}\ 0) \end{array}$$

Figure 13. Transforming a program by $\mapsto$.

For brevity, we let

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

and

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

Think of $\downarrow$ as "flatten" and $\uparrow$ as "unflatten." The step-by-step transformation of this let-expression appears in Figure 13. The parallel array $[1,2,3]$ is transformed to its flattened array equivalent, $(\downarrow [1,2,3])$, which evaluates to $\{1,2,3; \mathbf{lf}(0,3)\}$. Furthermore, by rewriting, we exchange one type-indexed subscript operator for another, thereby eliminating coercion operations. The coercions $\downarrow$ and $\uparrow$ are inverse coercions. Note by the fourth rule in Figure 10 we have

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

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

Figure 14. Type flattening.

$$
\frac{\{\} \vdash {e^\tau}^\tau \searrow (\overline{\tau} \triangleright \tau) \diamond \overline{e}^{\overline{\tau}}}{{e^\tau}^\tau \Downarrow ((\overline{\tau} \triangleright \tau)\ \overline{e}^{\overline{\tau}})^\tau}
$$

Figure 15. Top-level data-only flattening.

so the composition of $\downarrow$ and $\uparrow$ is mutually annihilating. Post-transformation, the representation of the array bound to *ns* is coerced exactly once, to the differently-typed fresh variable $\overline{ns}$.

4.1 Formal data-only flattening

We now present data-only flattening in the context of the *Flatland* system, as a transformation from a source language to a target language. A *source type* is a type that is neither a flattened-array type, nor contains any flattened-array types, generated by the grammar

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

We define *source programs* as a term e^τ for source type τ, all of whose subterms have source types and contain no coercions. By this definition, we have made it illegal to write down flattened arrays anywhere in a source program. The transformation will introduce all flattened-array values and all coercions.

The target language is defined in terms of *flat types*. A type τ is *flat* if

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

If a type is not flat, we say it is *nonflat*. Note that source types and nonflat types are not the same. For example, $[(\mathit{int}, \mathit{bool})]$ is a source type, and the related type $(\{\mathit{int}\ ;\ \mathit{lf}\}, \{\mathit{bool}\ ;\ \mathit{lf}\})$ is a flat type. But the related type $\{(\mathit{int}, \mathit{bool})\ ;\ \mathit{lf}\}$ is neither a source type nor a flat type; it is disqualified as a source type since it is a flattened-array type and disqualified as a flat type since it includes a pair inside an array. A *target program* is an expression whose outermost type is a source type, yet all of whose subexpressions have flat types. The restriction on its outermost type is a consequence of the type-preservation property of *Flatland*'s rewriting. The restriction is enforced by the application of one last "unflattening" coercion to the transformed program at the top level. Within the program, all subexpressions are flattened.

Flattening of whole programs is written as a type-preserving relation $\Downarrow$. Figure 15 gives the sole judgment for $\Downarrow$, which immediately delegates its work to an auxiliary relation $\searrow$. Whole-program

$$
\frac{}{\Delta \vdash {b^\tau}^\tau \searrow (\tau \triangleright \tau) \diamond b^\tau}
$$

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

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

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

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

$$
\frac{\begin{array}{c}\Delta \vdash {e_1}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond \overline{e}_1{}^{\overline{\tau}_1} \\ \Delta \vdash {e_2}^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond \overline{e}_2{}^{\overline{\tau}_2}\end{array}}{\begin{array}{l}\Delta \vdash ({e_1}^{\tau_1}, {e_2}^{\tau_2})^{(\tau_1, \tau_2)} \\ \quad \searrow ((\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2)) \diamond (\overline{e}_1{}^{\overline{\tau}_1}, \overline{e}_2{}^{\overline{\tau}_2})^{(\overline{\tau}_1, \overline{\tau}_2)}\end{array}}
$$

$$
\frac{\Delta \vdash e^{(\tau_1, \tau_2)} \searrow ((\overline{\tau}_1, \overline{\tau}_2) \triangleright (\tau_1, \tau_2)) \diamond \overline{e}^{(\overline{\tau}_1, \overline{\tau}_2)}}{\Delta \vdash (\pi_1\ e^{(\tau_1, \tau_2)})^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond (\pi_1\ \overline{e}^{(\overline{\tau}_1, \overline{\tau}_2)})^{\overline{\tau}_1}} \qquad (\pi_2 \text{ sim.})
$$

Figure 16. Data-only flattening, group 1.

flattening consists of transforming a program e^τ of source type τ to a program $\overline{e}^{\overline{\tau}}$ of flat type $\overline{\tau}$ and then coercing the transformed program to original type τ at the top level. Note that in the cases where τ is, for example, a ground type (see Figure 13), the outermost coercion is an identity coercion and has no effect.

The auxiliary relation of data-only flattening is given in Figures 16 and 17. The syntax of $\searrow$ is as follows:

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

Δ is a finite map from variable terms to variable terms; it is used to implement propagations through let-expressions and functions. On the right-hand side of the relation, a diamond ($\diamond$) is used to construct a pair out of a coercion and a transformed expression. The relation produces an unflattening coercion, along with the transformed expression, for use in one of the following ways. If the expression transformed is the whole program, the $\Downarrow$ relation applies the coercion to preserve the program's original type (as per the rule in Figure 15). If the expression is not the whole program, the accompanying coercion is used as a building block for further coercions as program transformation proceeds outward.

The important work in data-only flattening takes place at array terms; see the second rule in Figure 17 (which in turn appeals to

$$\frac{\Delta \vdash {e_1}^{\tau_1 \to \tau_2} \searrow (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2) \diamond {\overline{e}_1}^{\overline{\tau}_1 \to \overline{\tau}_2} \quad \Delta \vdash {e_2}^{\tau_1} \searrow (\overline{\tau}_1 \triangleright \tau_1) \diamond {\overline{e}_2}^{\overline{\tau}_1}}{\Delta \vdash ({e_1}^{\tau_1 \to \tau_2}\ {e_2}^{\tau_1})^{\tau_2} \searrow (\overline{\tau}_2 \triangleright \tau_2) \diamond ({\overline{e}_1}^{\overline{\tau}_1 \to \overline{\tau}_2}\ {\overline{e}_2}^{\overline{\tau}_1})^{\overline{\tau}_2}}$$

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

$$\frac{\Delta \vdash {e_1}^{\tau_2 \to \tau_3} \searrow (\overline{\tau}_2 \to \overline{\tau}_3 \triangleright \tau_2 \to \tau_3) \diamond {\overline{e}_1}^{\overline{\tau}_2 \to \overline{\tau}_3} \quad \Delta \vdash {e_2}^{\tau_1 \to \tau_2} \searrow (\overline{\tau}_1 \to \overline{\tau}_2 \triangleright \tau_1 \to \tau_2) \diamond {\overline{e}_2}^{\overline{\tau}_1 \to \overline{\tau}_2}}{\Delta \vdash (e_1 \circ e_2)^{\tau_1 \to \tau_3} \searrow (\overline{\tau}_1 \to \overline{\tau}_3 \triangleright \tau_1 \to \tau_3) \diamond (\overline{e}_1 \circ \overline{e}_2)^{\overline{\tau}_1 \to \overline{\tau}_3}}$$

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

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

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

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

Figure 17. Data-only flattening, group 2.

type flattening in Figure 14.) This rule introduces coercions from parallel arrays to flattened arrays. The variable rule (second rule, Figure 16) substitutes typed variables for their flattened replacements per the map carried by Δ. The array-operator rules exchange operators indexed by source type to operators indexed by the corresponding flat types. The other rules are administrative, recursively propagating transformations through expressions. There is exactly one rule for every distinct syntactic form, so the rules describes both a semantic specification and an algorithm.

5. Implementation

Data-only flattening in PML is accomplished in three successive phases: an *abstract flattening* phase, whereby abstract flattening operations — symbolic values that stand in for actual implementations — are inserted throughout the code; a *fusion* phase where canceling coercions (adjacent coercions that undo one another's work) are eliminated; and a *concrete flattening* phase, where symbolic flattening operations are replaced by monomorphic code. Due to space limitations, the detailed operations of these phases is not presented here and may be found elsewhere [25].

5.1 Optimizations

Flattened PML programs are amenable to various optimizations that cannot be applied to non-flattened ones. In this section, we discuss several such optimizations, each of which is responsible, in part, for the performance improvements we report in our benchmark results.

Monomorphization. Monomorphization is an optimization whereby a polymorphic data structure containing uniformly-represented (*i.e.*, boxed) elements is transformed to a representation containing raw (unboxed) elements in their place. The flattening transformation, by virtue of unzipping arrays of tuples, exposes more opportunities for monomorphization than otherwise. In PML, arrays of `double` pairs, for example, become pairs of `double` arrays, which in turn become `farrays`, each containing a specialized rope of `doubles` as its flat data vector. Monomorphization is well known to be valuable even outside the context of nested data parallel compilation. MLton [18], an optimizing whole-program SML compiler, performs monomorphization to generate better-performing sequential code. PML stands to benefit from monomorphization even without flattening (PML currently does no monomorphization unless flattening is enabled), although it will never be the case that, without unzipping tuples, non-flattened PML will have as many opportunities to do it.

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

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

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

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

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

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

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

Tabulating over integer intervals requires no intermediate data structures, and realizes a performance improvement over the build-and-map strategy outlined above.

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

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

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

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

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

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

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

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

The shape tree in rectangular cases has a simple regular structure as well, and be computed from the dimensions of a regular array in a straightforward way. Tab flattening operation scales to any number of dimensions for regular nested arrays.

Segmented reductions. NESL's fast segmented operations are an important element of NESL's ability to perform well on irregular nested data parallel programs, and an important one for PML to emulate. NESL's segmented sum operation, for example, is able to compute the sums of a nested array of numbers in a fixed number of steps regardless of the irregularity of the array's structure. This operation is critical to the performance of sparse-matrix/vector multiplication (see below). Here is an example of an irregular sum computation in PML:

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

If this parallel comprehension is rewritten such that the `sum` operation is simply mapped over the array-valued elements of `nss`, the irregularity of the structure of `nss`, if there is wide variation in the lengths of its elements, is bound to affect load balancing adversely. As such, the compiler replaces nested reductions like these

```
val ss = [| PArray.reduce add 0.0 ns | ns in nss |]
```

with "segmented reductions"

```
val ss = PArray.segreduce add 0.0 nss
```

The non-flattened implementation of `PArray.segreduce oper ident` is simply `PArray.map (PArray.reduce oper ident)`. As part of the flattening transformation, the non-flattened `segreduce` implementation is replaced by a faster `segreduce` written to exploit the shape of `farray` data structures.

6. Evaluation

These benchmarks compare the performance and scalability of Manticore's current implementation of parallel arrays [1] against data-only flattening. The baseline performance is from the sequential version of the benchmark, which runs on a single processor and eliminates all overhead from the parallel language constructs and associated runtime features. The sequential version does not take advantage of any of the data transformations provided by data-only flattening, relying on the default polymorphic array representation.

These benchmarks were selected to show that we gain performance on code that executes over irregular data, as in SMVM, while preserving most of the performance of benchmarks over regular data. Flattening happens at compile time in the following sense: nested arrays such as those in the main expressions of `mandelbrot` and `raytracer` are created flat in the first place, not constructed nested and then flattened.

6.1 Experimental method

Our benchmark machine is a Dell PowerEdge R815 server, outfitted with 48 cores and 128 GB physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-42. The 48 cores are provided by four 12 core AMD Opteron 6172 "Magny Cours" processors. Each core operates at 2.1 GHz and has 64 KB each of instruction and data L1 cache and 512 KB of L2 cache. There are two 6 MB L3 caches per processor, each of which is shared by six cores, for a total of 48 MB of L3 cache.

We ran each experiment configuration 30 times, and we report the average performance results in our graphs and tables. Times are reported in seconds.

6.2 Mandelbrot

We compute the Mandelbrot set by means of a function `elt` which consumes a pair of integers and produces an integer. The argument to `elt` represents a location in the complex plane. Its return value is the number of iterations required, according to the standard iterating Mandelbrot set membership test, for a given point to diverge outside the set (by having a modulus greater than 2). A point is a member of the Mandelbrot set if it fails to diverge before reaching a fixed upper limit of iterations (we use 1000). We execute this simple function in parallel over a 2048×2048 range using the following PML code:

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

Figure 18*(a)* shows PML speedups, with and without flattening, against the sequential baseline. Due to the relatively small amount of computation at each element, the benefits of the data-only flattening transformation provide only a 5% speedup at 48 cores. These benefits come from the reduced amount of memory traffic when using a monomorphic array representation, avoiding an extra allocation per result element and associated garbage collector pressure.

6.3 Raytracer

Our ray tracing benchmark computes the image of a scene graph consisting of a group of overlapping spheres with transparency and reflection. The code is translated from a parallel program in the implicitly-parallel language Id90 [19]. It is a brute-force implementation and does not use any acceleration data structures. The Raytracer benchmark renders a 2048×2048 image in parallel as a two-dimensional sequence.

Similar to the Mandelbrot benchmark, we write the body of the main function as a nested parallel comprehension:

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

In all versions of the program, the Manticore compiler transforms this nested parallel comprehension into a two-dimensional tabulation over fixed ranges, avoiding creation of an intermediate array of values to iterate over. In the data-only version of this program, that two-dimensional tabulation is instead performed as a

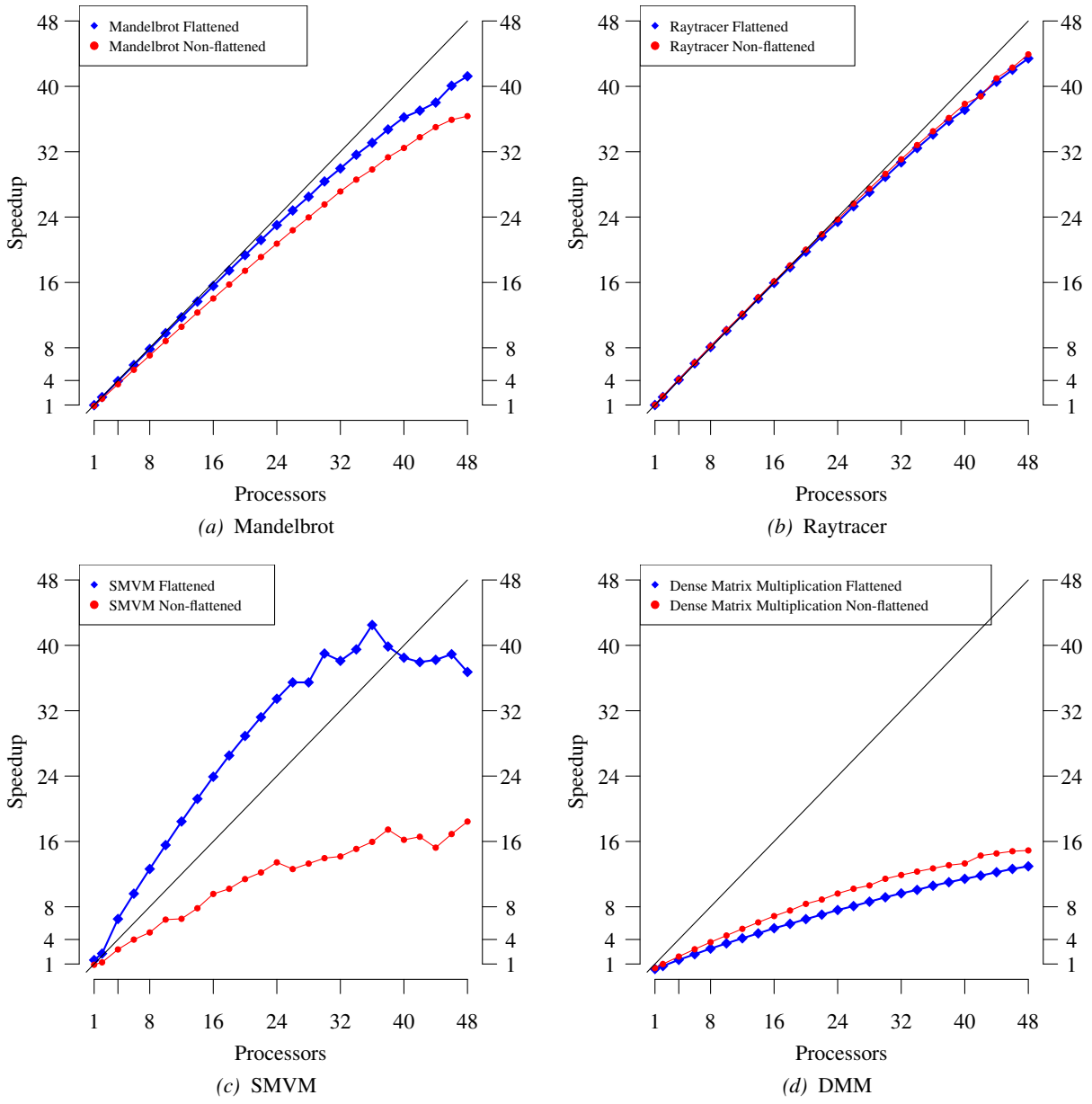


Figure 18. Comparison of flattened and non-flattened performance.

one-dimensional tabulation with adjusted indices. As is shown in Figure 18*(b)*, this transformation slightly reduces the overhead required by the work-stealing scheduler to evenly balance the remaining work, since work on the one-dimensional data structure requires only splitting the index in half instead of the default finger-tree splitting required in the two-dimensional version [1].

6.4 Sparse-Matrix Vector Multiplication

Among our benchmarks, sparse-matrix vector multiplication profits most from the data-only flattening transformation. Sparse-matrix vector multiplication is expressed concisely in the following PML code:

```
fun smvm (sm, v) = let
  fun add (a,b) = a+b
  in
    [| PArray.reduce add 0.0
      [| x*(v!i) | (i,x) in sv |]
    | sv in sm |]
  end
```

The inner parallel array expression computes the dot-product of the sparse matrix and the vector. The outer parallel array then computes the sum of each of those resulting dot-products. The summation is implemented on top of `PArray.reduce`, which allows the runtime to processes the reduction in parallel.

In both versions of the benchmark, the Manticore compiler automatically optimizes the parallel reduction over multiple values

into a single segmented reduction. Segmented operators have been shown by Blelloch and others to result in more balanced chunks of parallel work, across a variety of platforms [3, 24]. All versions are transformed into the following PML code:

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

The flattened version of `smvm` uses monomorphic vectors for both the intermediate representation of the dot-products and the final result of the segmented reduction. Further, the implementation of segmented reduction over monomorphic vectors takes advantage of the layout, performing the segmented reduction with far less overhead than the sequential and non-flattened versions.

Figure 18*(c)* gives the speedups of PML over its sequential baseline, both with and without flattening. The sparse matrix is $10,000 \times 10,000$, with a random number of entries between 100 and 500 in each row. Flattened `smvm` is substantially faster than non-flattened `smvm` for all numbers of processors up to 48, and furthermore has much better performance with respect to the sequential baseline. The super-linear speedups are due to the relatively small amount of work performed on each element compared to the improvement due to the representation change. Above 36 processors, our performance improvement flattens due to having insufficient data to take advantage of the processors. Unfortunately, limitations in the Manticore runtime currently prevent us from further increasing the size of the data.

6.5 Dense Matrix Matrix Multiplication

The dense matrix multiplication (DMM) benchmark is a dense-matrix by dense-matrix multiplication in which each matrix is 600×600. As mentioned in Section 2, this benchmark has traditionally had extremely poor performance under flattening. As shown in Figure 18*(d)*, our approach does still result in a slowdown in performance due to the creation of some intermediate arrays (resulting in a factor of 3 increase in memory usage). This penalty is roughly 13% and could be reduced through the introduction of additional fusion operations to avoid those intermediate arrays.

6.6 Conclusion

These benchmarks demonstrate that the data-only flattening transformation significantly improves a benchmark with irregular data (SMVM), does not experience the polynomial blowup typical to full flattening on DMM, and does not dramatically change the performance of other programs.

7. Related work

The incremental extension of NESL's foundation to a more feature-rich platform has ultimately taken the form of Data Parallel Haskell [9]. Chakravarty *et al.* first present the language Nepal in 2001 [8], characterized as a version of Haskell including nested data parallelism. Nepal is succeeded by Chakravarty *et al.*'s Data Parallel Haskell [9], bringing together Nepal-style nested data parallelism with Haskell as implemented in the Glasgow Haskell Compiler (GHC) [12]. In 2008, Peyton Jones *et al.* give a thorough overview of the Data Parallel Haskell language and its compilation [22], including an updated account of how Data Parallel Haskell uses the flattening transformation in their implementation. The especially germane question is this: what happens after a system has compiled a Data Parallel Haskell program in the manner of NESL, and yet executes on an SMP rather than a SIMD machine? Their answer to this technical problem is to adapt the `split` and `join` mechanisms originally presented in Keller's dissertation [13] to implement NESL-style operations across the multiple processing elements on a multicore computer. Parallel computations are `split` across processing elements and subsequently `join`ed on completion. To eliminate unnecessary synchronization points, GHC uses rewrite rules [21] to erase successive applications of `split` and `join` (per the identity equivalence rule originally given by Keller). (In addition to this, various advanced fusion techniques are employed to streamline the resulting post-flattened program, an overview of which is given in their paper.)

Though Data Parallel Haskell represents broad advances in NESL-style flattening in numerous ways, and although it has been adapted to run on multicore machines, its compilation strategy continues to reflect the SIMD orientation of its predecessors, though there has been recent work on vectorisation in DPH to avoid excessive flattening [15]. This work differs from that line of research by never performing the full vectorisation transformation on the code, though both approaches flatten nested data.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-0811389 and CCF-1010568, and upon work performed in part while John Reppy was serving at the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[1] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. In *ICFP '10*, pages 93–104, New York, NY, September 2010. ACM.

[2] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[3] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[4] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, Mar. 1996.

[5] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *JPDC*, 8(2):119–134, 1990.

[6] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *SP&E*, 25(12):1315–1330, Dec. 1995. ISSN 0038-0644.

[7] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In *ICFP '00*, pages 94–105, New York, NY, Sept. 2000. ACM.

[8] M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – nested data parallelism in Haskell. In *Euro-Par '01*, volume 2150 of *LNCS*, pages 524–534, New York, NY, Aug. 2001. Springer-Verlag.

[9] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, pages 10–18, New York, NY, Jan. 2007. ACM.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, Berkeley, CA, Dec. 2004. USENIX Association.

[11] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *ML '07*, pages 15–24, New York, NY, Oct. 2007. ACM.

[12] GHC. The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc`.

[13] G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 1999.

[14] G. Keller and M. M. T. Chakravarty. Flattening trees. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 709–719, London, UK, 1998. Springer-Verlag.

[15] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones. Vectorisation Avoidance. In *HASKELL '12*, New York, NY, Sept. 2012. ACM. Forthcoming.

[16] R. Leshchinskiy, M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *ICCS '06*, number 3992 in LNCS, pages 920–928, New York, NY, May 2006. Springer-Verlag.

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[18] MLton. The MLton Standard ML compiler. Available at `http://mlton.org`.

[19] R. S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[20] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *FoMPP5*, pages 186–193, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[21] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233, Sept. 2001.

[22] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *APLAS '08*, pages 138–138, New York, NY, Dec. 2008. Springer-Verlag.

[23] M. Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, Aug. 2010. Available from `http://manticore.cs.uchicago.edu`.

[24] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07*, pages 97–106, Aire-la-Ville, Switzerland, Aug. 2007. Eurographics Association.

[25] A. Shaw. *Implementation techniques for nested-data-parallel languages*. PhD thesis, University of Chicago, Aug. 2011. Available from `http://manticore.cs.uchicago.edu`.

[26] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *ICFP '08*, pages 253–264, New York, NY, Sept. 2008. ACM.