An Approach to Fast Arrays in Haskell

Manuel M. T. Chakravarty and Gabriele Keller

The University of New South Wales
School of Computer Science & Engineering
Sydney, Australia
{chak,keller}@cse.unsw.edu.au

Abstract Many array-centric algorithms from computational science and engineering, especially those based on dynamic and irregular data structures, can be coded rather elegantly in a purely functional style. The challenge, when compared to imperative array languages, is performance. These lecture notes discuss the shortcomings of Haskell's standard arrays in this context and present an alternative approach that decouples array from list processing and is based on program transformation and generic programming. In particular, we will present (1) an array library that uses type analysis to achieve unboxing and flattening of data structures as well as (2) equational array fusion based on array combinators and compiler-driven rewrite rules. We will make use of a range of advanced language extensions to Haskell, such as multi-parameter type classes, functional dependencies, rewrite rules, unboxed values, and locally state-based computations.

1 Motivation

Let us start with something simple, namely the dot product of two vectors:

$$\overline{v} \cdot \overline{w} = \sum_{i=0}^{n-1} v_i w_i$$

In Haskell 98, using the standard array library, we can define the function rather nicely as

```
type Vector = Array Int Float

(·)   :: Vector → Vector → Float
v · w = sum [v!i * w!i | i ← indices v]
```

Unfortunately, this elegance comes at a considerable price. Figure 1 (next page) graphs the running time in dependence on the length of the input vectors. The figures were obtained by running the code on a 1.2GHz Pentium IIIM under GNU/Linux compiled with GHC 5.04.1 and optimisation level `-O2`. At 100,000 elements, the code needs $3\mu s$ per vector element, which seems slow for a 1.2GHz machine.

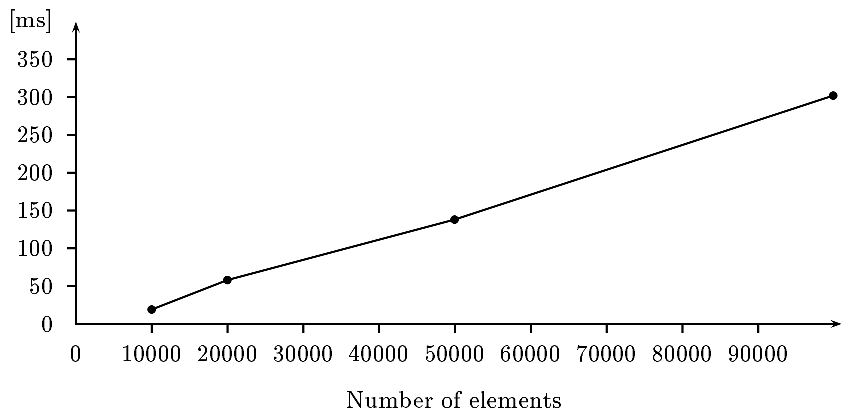

Figure1. Running time of computing the dot product in Haskell (in ms)

This suspicion is easily verified by timing the corresponding C function, which is as follows:

```
float dotp (float *v1, float *v2, int n)
{
  int   i;
  float sum = 0;

  for (i = 0; i < n; i++)
    sum += v1[i] * v2[i];
  return sum;
}
```

On 100,000 element vectors, the C code runs about 300 times faster![1]

The remainder of these lecture notes will (1) look into the reasons for this huge performance difference, (2) propose a slightly different approach to array programming that avoids some of the inherent inefficiencies of the standard Haskell approach, and (3) discuss an optimising implementation scheme for the new form of array programs. In particular, the presentation includes the detailed treatment of an array library that makes use of type analysis (aka generic programming) to achieve unboxing and flattening of data structures. The library also optimises array traversals by way of GHC's rewrite rules.

2 Where Do the Inefficiencies Come From?

Why is the Haskell code for the dot product so slow? In this section, we will study three functions over arrays, which are of increasing sophistication; each function will demonstrate one of the shortcomings of Haskell 98's standard arrays.

[1] This is compiled with GCC 3.2 using `-O2`.

2.1 Vector Dot Product: Lists are Slow

Looking at the expression $sum\ [v!i * w!i \mid i \leftarrow indices\ v]$ from the dot product code, two properties stand out: most of the operations involved are actually list operations and it creates a superfluous intermediate structure, namely the list produced by the list comprehension and consumed by *sum*. The suspicion that these two properties are the main culprits is easily verified by measuring the running time of an explicitly deforested [Wad88] version of the dot product:

```
v · w = loop 0 0
  where
    loop i acc | i > n     = acc
               | otherwise = loop (i + 1) (v!i * w!i + acc)
    n                      = snd (bounds v)
```

This optimisation leads to a dramatic improvement: the explicit loop is by a factor of about 25 faster than the original Haskell code. On one hand, this is good news, as it means that we can optimise quite effectively within Haskell. On the other hand, we would prefer the compiler to automatically perform the conversion from the comprehension-based to the loop-based code.

In summary, the lesson learnt from studying the dot product implementation is that the use of lists to drive array computations is highly detrimental to performance. But this is not the whole story as we will see during the discussion of the next benchmark.

2.2 Matrix Vector Multiplication: Boxing is Slow

Let us look at a slightly more complicated function than the dot product, namely matrix-vector multiplication $\overline{w} = A\overline{v}$ where

$$w_i = \sum_{j=0}^{m-1} A_{ij} v_j, 0 \leq i \leq n-1$$

Using the convenient, but slow, list-based interface to arrays, we have

```
type Matrix = Array (Int, Int) Float

mvm     :: Matrix → Vector → Vector
mvm a v = listArray (0, n') [sum [a!(i,j) * v!j | j ← [0..m']] | i ← [0..n']]
  where
    (n', m') = snd (bounds a)
```

Unlike in the case of the dot product, we can only partially avoid the intermediate lists by using an explicit loop in *mvm*. More precisely, we can transform the inner list comprehension together with the application of *sum* into a loop that is very much like that for the dot product, but we cannot as easily remove the outer list comprehension—to create the result array, Haskell 98 forces us to go via a

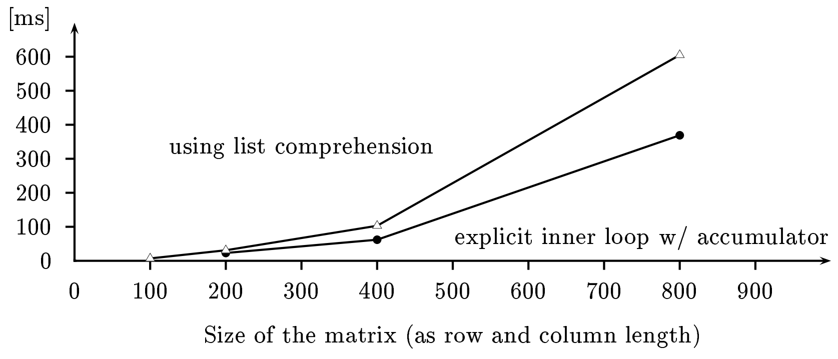

Figure2. Running time of matrix-vector multiplication in Haskell (in ms)

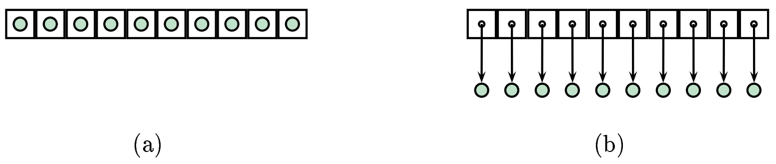

Figure3. Different array representations: (a) unboxed and (b) boxed

list. This proves to be rather limiting for *mvm*. For a range of square matrices, Figure 2 graphs the running time of *mvm* for both the version displayed above as well as the version where the inner list comprehension is replaced by an explicit loop. The version using an explicit loop is clearly superior, but by a less significant factor than in the dot product benchmark. A comparison with the corresponding C function reveals that this is not because the list-based Haskell code is better, but because optimising the inner loop alone is far from sufficient. In brief, for an 800×800 matrix, the C code runs more than a 200 times faster than the comprehension-based Haskell code and about 120 times faster than the Haskell code that uses an explicit loop instead of the inner list comprehension.

Part of the inefficiencies derive from the parametricity of Haskell arrays; i.e., the fact that Haskell arrays use the same representation regardless of the type of elements contained in the array. Hence, even arrays of primitive values, such as integer and floating-point numbers, use a boxed element representation. This affects performance negatively for two reasons: the code needs a larger number of memory accesses and cache utilisation is worse. The difference in layout between boxed and unboxed arrays is depicted in Figure 5. The inefficiencies are caused by the additional indirection in the boxed array implementation. In addition, these boxed arrays are lazy, which further increases the number of memory accesses, reduces cache utilisation, and also makes branch prediction harder for the CPU.

```
mvm a v = runST (do
                ma ← newArray_ (0, n)
                outerLoop ma 0
                unsafeFreeze ma
              )
  where
    outerLoop :: STUArray s Int Float → Int → ST s ()
    outerLoop ma i | i > n       = return ()
                   | otherwise = do
                                   writeArray ma i (loop i 0 0)
                                   outerLoop ma (i + 1)
    loop i j acc   | j > m       = acc
                   | otherwise = loop i (j + 1) (acc + a!(i,j) * v!j)
```

Figure4. Optimised matrix vector multiplication using the *ST* monad

GHC's libraries provide an extended implementation of the standard array interface, which supports unboxed arrays for basic types.[2] When we modify the version of *mvm* that is based on an explicit loop to use unboxed arrays, the running time is reduced by approximately 50%. This is a significant improvement, but it is still more than a factor of 50 slower than the C code. A contributor to the remaining gap is the lack of inlining a critical function in the libraries of GHC 5.04.1. We gain more than another factor of 4 by forcing the inlining of that function. Finally, by replacing the outer loop and the call to *listArray* by a state-based loop using GHC's mutable arrays, as provided in the *ST*-monad,[3] we gain yet another factor of nearly two, as we avoid the list-based interface of *listArray*.

Overall, if we also disable bounds checking, the Haskell code based on the *ST* monad, brings us within a factor of 7–8 of the performance of the C code. This is good news. The bad news is that the resulting code, which is displayed in Figure 4, is even less readable than the C code.

In summary, we can infer that as soon as new arrays are constructed, unboxed arrays are to be preferred for best performance. If we combine this with using a stateful programming style for array construction, we get good performance at the expense of code clarity.

2.3 Sparse Matrix Vector Multiplication: Nesting is Slow

Interestingly, there are applications that are tricky to handle in both plain C code and in the *ST* monad with unboxed arrays. An example of such an application is the multiplication of a *sparse* matrix with a vector. A popular representation

[2] Unboxed arrays are implemented by the data constructor *UArray* from the module *Data.Array.Unboxed* (in GHC 5.04 upward).

[3] See GHC's interfaces for *Data.Array.MArray* and *Data.Array.ST*. An introduction to the *ST* monad is contained in [PL95].

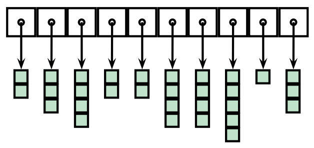
Figure5. Standard layout of irregular nested arrays

of sparse matrices is the so-called *compressed sparse row* format [DER86]. This format represents a sparse row by an array of column-index/value pairs, where each pair represents a non-zero element of the sparse matrix. An array of such sparse rows implements a sparse matrix.

$$\begin{array}{ll}\textbf{type}\ \mathit{SparseRow} & = \mathit{Array\ Int}\ (\mathit{Int},\ \mathit{Float}) \quad \text{-- column index, value} \\ \textbf{type}\ \mathit{SparseMatrix} & = \mathit{Array\ Int\ SparseRow}\end{array}$$

As an example, consider the following matrix and its compressed array representation:

$$\begin{bmatrix}5&0&0&0\\0&0&0&7\\3&4&0&0\\0&0&0&0\end{bmatrix} \Longrightarrow \begin{array}{rl}\mathit{listArray}\ (0,4)[\mathit{listArray}\ (0,0) & [(0,\ 5\ )], \\ \mathit{listArray}\ (0,0) & [(3,\ 7\ )], \\ \mathit{listArray}\ (0,1) & [(0,\ 3\ ),(1,\ 4\ )], \\ \mathit{listArray}\ (0,-1) & []]\end{array}$$

The numbers that represent actual non-zero values in the matrix are highlighted in the array representation.

Based on standard Haskell arrays, we can denote the multiplication of a sparse matrix with a dense vector, resulting in a new dense vector, as follows:

$$\begin{array}{l}\mathit{smvm} \qquad\quad :: \mathit{SparseMatrix} \to \mathit{Vector} \to \mathit{Vector} \\ \mathit{smvm\ sm\ vec} = \mathit{listArray\ bnds} \\ \quad [\mathit{sum}\ \underbrace{[x * (\mathit{vec}!\mathit{col}) \mid (\mathit{col},\ x) \leftarrow \mathit{elems\ row}]}_{\text{products of one row}} \mid \mathit{row} \leftarrow \mathit{elems\ sm}] \\ \qquad \textbf{where} \\ \qquad\quad \mathit{bnds} = \mathit{bounds\ sm}\end{array}$$

This code again is nice, but lacking in performance. The trouble is that it is not clear how to use unboxed arrays to improve the code. GHC provides unboxed arrays for primitive element types, such as floating-point and integer numbers. However, the sparse matrix representation uses an array of pairs and an array of arrays of pairs.

A quite obvious idea is to represent an array of pairs by a pair of arrays. Less clear is how to handle nested arrays efficiently. In the case of dense matrices, we did avoid using an array of arrays by choosing an array with a two-dimensional index domain. For sparse matrices, it is more difficult to find an efficient representation. As depicted in Figure 5, the subarrays are of varying size; hence, a compact representation will need to include an indexing scheme that takes the irregular structure into account. To achieve this, we will discuss a method for separating the actual matrix elements from the structure of the sparse matrix in the next section.

In summary, we can draw two conclusions from the sparse matrix example: firstly, irregular, nested array structures are more difficult to represent efficiently than simple flat arrays; and secondly, irregular, nested structures also hit on limitations of imperative languages. Thus, irregular structures seem to be where we might be able to provide a serious advantage over conventional array languages by using a functional language in combination with a suitable set of program transformations.

3 Parallel Arrays and the Flattening Transformation

So far, it appears as if we can have either beauty or performance. On one hand, standard Haskell arrays in combination with list comprehensions and list combinators enable elegant, but slow, formulations of array algorithms. On the other hand, unboxed arrays and the ST monad enable fast, but inelegant implementations. The rest of these lecture notes explore an approach to reconciling beauty and performance, while simultaneously optimising nested structures, such as those needed to represent sparse matrices.

For optimal convenience, the whole set of program transformations discussed in the following should be automated by a compiler. However, our current implementation is only partial, which means that some transformations need to be performed manually.

3.1 An Alternative Array Notation

The conclusions drawn in the previous section prompt us to consider an alternative array interface. Firstly, from a semantic viewpoint, the use of unboxed, instead of boxed arrays implies a change. More precisely, arrays are no longer lazy, but instead have a parallel evaluation semantics, where all elements are computed as soon as any of them is needed. Hence, we call the new form of arrays *parallel arrays*.[4] Secondly, we want to entirely avoid the reliance on list operations and drop parametrisation of the index domain in favour of simple integer indexes. To emphasise the use of the new array interface, we introduce a new notation for array types and array comprehension. The new type corresponds to Haskell arrays as follows:

type [:α:] = *Array Int* α -- parallel arrays

[4] In fact, these arrays are also well suited for a parallel implementation, but we will not cover this aspect in these lecture notes.

Moreover, we have the following correspondence between the new and the standard form of array comprehensions:

$$[{:}e_1 \mid p \leftarrow e_2, q{:}] \;=\; \mathit{listArray}\ \mathit{bnds}\ [e_1 \;\mid\; p \;\leftarrow\; \mathit{elems}\ e_2,\ q']$$

where the other qualifiers q are treated similarly and *bnds* depends on the size of the list produced by the list comprehension. We assume that for each list operation of the Haskell Prelude (that generates finite lists), there exists a corresponding operation on the array type $[{:}\cdot{:}]$. In particular, we require the existence of functions *lengthP*, *zipP*, *filterP*, *replicateP*, and *concatP*.

3.2 An Overview of the Transformations

The implementation scheme that we discuss in the following comprises three program transformations:

1. *Code vectorisation:* The code is transformed such that it facilitates processing entire arrays in one sweep. Such collective array operations enable the compiler to generate code that has a structure similar to loop-based code in C.
   We will discuss this transformation in detail in the present section.
2. *Structure flattening:* As we saw in the previous section, boxed arrays are too expensive. Hence, we need to find a method that transforms arrays of structured types into an alternative representation that makes use of unboxed arrays. We call this a *representation transformation.*
   We will discuss some of the basic ideas of this transformation at a later point in this section when we illustrate the implementation technique at the example of *smvm*.
3. *Array Fusion:* To avoid superfluous intermediate structures and improve cache performance, consecutive array traversals need to be amalgamated. This is especially important as, as an artifact of the previous two transformations, we will end up with a multitude of simple array operations. Executing them as they are would lead to extremely poor cache behaviour and a large number of intermediate structures.
   Array fusion will be discussed in Section 5.

In addition to these three general transformations, we need to optimise special cases, as for some common operations we can do much better than the general rules due to knowledge about the algebraic properties of some functions. In these lecture notes, we will not discuss the treatment of special cases in much detail, but only illustrate some basic ideas when discussing the implementation of *smvm*.

3.3 Code Vectorisation

The essential idea behind code vectorisation is the lifting of code that operates on individual elements to code that processes entire arrays in bulk. The approach

Function definitions	$\mathsf{D} \to \mathsf{V}\ \mathsf{V}_1 \cdots \mathsf{V}_n = \mathsf{E}$	$(n \geq 1)$
Expressions	$\mathsf{E} \to \mathsf{C}$	(constant)
	$\mid \mathsf{V}$	(variable)
	$\mid \mathsf{E}\ \mathsf{E}_1 \cdots \mathsf{E}_n$	(application, $n \geq 1$)
	$\mid \mathbf{let}\ \mathsf{V} = \mathsf{E}_1\ \mathbf{in}\ \mathsf{E}_2$	(local binding)
	$\mid \mathbf{case}\ \mathsf{E}\ \mathbf{of}\ \triangleleft\mathsf{V}_1 \to \mathsf{E}_1;\ \triangleright\mathsf{V}_2 \to \mathsf{E}_2$	(selection)
	$\mid [{:}\mathsf{E}_1 \mid \mathsf{V} \leftarrow \mathsf{E}_2{:}]$	(array comprehension)

Figure6. Grammar of a simple functional programs

to vectorisation that we take was introduced in [BS90,CK00]. Here we will only explain the core idea of the transformation at the example of a restricted set of functional programs, in such a way that vectorisation can be performed manually. Nevertheless, the transformation, as presented in the following, is sufficient to handle typical array codes, such as the sparse matrix vector multiplication from Section 2.3.

Figure 6 displays the grammar of the restricted set of functional programs that we consider in this subsection. We allow only top-level function definitions and restrict array comprehensions to single generators without filter expressions. The later is not really a restriction as multiple generators and filter expressions can be expressed by using the standard functions *zipP* or *filterP*, respectively. There are neither partial applications nor local function definitions. Moreover, we restrict ourselves to a two-way **case** construct that scrutinises binary sum types defined as

$$\mathbf{data}\ a + b = \triangleleft\, a \mid \triangleright\, b$$

which corresponds to Haskell's *Either* type. All this may seem restrictive, but it simplifies the presentation of the vectorisation transformation significantly.

Function definitions. For each function f that occurs, directly or indirectly, in an array comprehension, we need to generate a *vectorised* variant $f^{\uparrow}$ by the following scheme that makes use of the *lifting transformation* $\mathcal{L}[\![\cdot]\!]^{\cdot}$:

$$\begin{array}{lll} f\ x_1\ \ldots\ x_n &= e & \text{-- original definition} \\ f^{\uparrow}\ x_1\ \ldots\ x_n &= \mathcal{L}[\![e]\!]^{[x1,\ldots,x_n]} & \text{-- vectorised version} \end{array}$$

We assume that for primitive operations, such as $(+)$, $(*)$, and so on, vectorised versions are already provided. For example, $(+^{\uparrow})$ adds the elements of two arrays pairwise:

$$[{:}1,2,3{:}] +^{\uparrow} [{:}5,4,3{:}] = [{:}6,6,6{:}]$$

Vectorised functions, in essence, correspond to mapping the original function over an array or, in case of a function with multiple arguments, zipping the arguments and mapping the function over the result.

As a next step, we replace each array comprehension by the lifted version of the body expression of the comprehension, where we bind the array generator expression (ge) to the generator variable (x):

$$[{:}\,be \mid x \leftarrow ge{:}] \quad = \quad \textbf{let}\ x\ =\ ge\ \textbf{in}\ \mathcal{L}[\![be]\!]^{[x]}$$

Both the introduction of vectorised functions and the removal of array comprehensions makes use of the lifting transformation $\mathcal{L}[\![\cdot]\!]^{\cdot}$, which does most of the actual work of vectorising code. An application $\mathcal{L}[\![e]\!]^{vs}$ of the lifting transformation lifts the expression e into vector space within the *vectorisation context* vs. This essentially means that all scalar operations are turned into array operations and all array operations are adjusted to work on irregular collections of arrays. The vectorisation context contains all those free variables of the lifted expression that are already lifted. The transformation function $\mathcal{L}[\![\cdot]\!]^{\cdot}$ operates syntax-oriented as described in the following.

Constant values. Constant values are lifted by generating an array that has the same length as the arrays bound to the variables in the vectorisation context:

$$\mathcal{L}[\![c]\!]^{(v:\_)} \ =\ \mathit{replicateP}\ (\mathit{lengthP}\ v)\ c$$

Variables. We distinguish two cases when lifting variables: (1) If the variable is in the vectorisation context, it is already lifted as it was bound by the generator of a comprehension or is a function argument; (2) otherwise, we need to vectorise the value represented by the variable by treating it like a constant value:

$$\begin{array}{lll} \mathcal{L}[\![w]\!]^{(v:vs)} & \mid\ w\ \in\ (v : vs) & =\ w \\ & \mid\ \mathit{otherwise} & =\ \mathit{replicateP}\ (\mathit{lengthP}\ v)\ w \end{array}$$

Bindings. Lifting of **let** bindings is straight forward. The only interesting aspect is that we need to include the variable name of the bound variable into the vectorisation context for the body expression.

$$\mathcal{L}[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!]^{vs}\ =\ \textbf{let}\ v\ =\ \mathcal{L}[\![e_1]\!]^{vs}\ \textbf{in}\ \mathcal{L}[\![e_2]\!]^{(v:vs)}$$

Function application. Lifting of function applications depends on whether the applied function is already a vectorised function—i.e., whether the name is of the form $f^{\uparrow}$. In the case where the applied function is not vectorised, we replace the function by its vectorised variant and lift the function arguments. More precisely, we rewrite $\mathcal{L}[\![f\ e_1\ \cdots\ e_n]\!]^{vs}$ to $f^{\uparrow}\ (\mathcal{L}[\![e_1]\!]^{vs})\ \cdots\ (\mathcal{L}[\![e_n]\!]^{vs})$.

Much more interesting is, however, the case where the applied function is already vectorised. In fact, the treatment of this case is one of the central points of the flattening transformation, so we need to look into it more closely. Let us, for a moment, pretend that we are dealing with list operations. We mentioned earlier that vectorising a function is like mapping the function over an array. Lifting an already vectorised function is therefore like mapping a mapped function. Moreover, we know that the following equation holds for every function f and every list xs:

$$\mathit{concat} \circ \mathit{map}\ (\mathit{map}\ f) \ = \ \mathit{map}\ f \circ \mathit{concat} \tag{1}$$

This equation implies that, in some situations, an expression mapping a mapped function can be replaced by a simple mapping. To exploit this property for lifting of function applications, we also need to consider the function

$$\mathit{segment} :: [[\alpha]] \to [\beta] \to [[\beta]]$$

which extracts the nesting structure from a nested list and applies this structure to another list, such that the following holds:[5]

$$\mathit{segment}\ xs \circ \mathit{concat}\ \$\ xs \ = \ xs \tag{2}$$

For example, we have $\mathit{segment}\ [['a', 'b'], [], ['c']]\ [1, 2, 3] \ = \ [[1, 2], [], [3]]$.

By combining the two equations, we can derive an alternative implementation for mapping a mapped function as follows:

$$\begin{array}{ll} & \mathit{map}\ (\mathit{map}\ f)\ \$\ xs \\ = & \{\text{Equation (2)}\} \\ & \mathit{segment}\ xs \circ \mathit{concat} \circ \mathit{map}\ (\mathit{map}\ f)\ \$\ xs \\ = & \{\text{Equation (1)}\} \\ & \mathit{segment}\ xs \circ \mathit{map}\ f \circ \mathit{concat}\ \$\ xs \end{array}$$

What exactly does this mean for the lifting of function applications? It means that instead of having to vectorise vectorised functions, we can do with plain vectorised functions if we use *concatP* to strip a nesting level of all function arguments and replace this nesting level by using *segmentP* on the result of the function application. Hence, we overall get the following transformation rule for lifting function applications:

$$\begin{array}{l} \mathcal{L}[\![f\ e_1 \cdots e_n]\!]^{vs} \mid f \text{ is not vectorised} = f^{\uparrow}\ \mathit{el}_1 \cdots \mathit{el}_n \\ \qquad\qquad\qquad \mid \mathit{otherwise} \qquad\quad = \\ \quad \mathit{segmentP}\ \mathit{el}_1\ (f\ (\mathit{concatP}\ \mathit{el}_1) \cdots (\mathit{concatP}\ \mathit{el}_n)) \\ \\ \quad \textbf{where} \\ \qquad \mathit{el}_1 = \mathcal{L}[\![e_1]\!]^{vs} \\ \qquad\qquad \vdots \\ \qquad \mathit{el}_n = \mathcal{L}[\![e_n]\!]^{vs} \end{array}$$

[5] In Haskell, the dollar operator ($) denotes function application with very low operator precedence.

There is one caveat, however. Since the transformation introduces *concatP* and *segmentP* operations, we have to ensure that they are efficient. As we will see later, by choosing an appropriate representation for nested array, it is possible for the two operations to execute in constant time.

Conditionals. To lift a conditional, we have to determine, for each array element, to which branch it belongs and, based on this, construct different versions of each array for the two branches. The function $\mathit{getInlFlags} :: [{:}\alpha + \beta{:}] \rightarrow [{:}\mathit{Bool}{:}]$ computes a flag array for an array of sum types $\alpha + \beta$, such that the flag array contains *True* at each position that corresponds to a value of type α and *False* at all the other positions in the array (*getInrFlags* computes the inverse of this array).

Moreover, the function $\mathit{packP} :: [{:}\mathit{Bool}{:}] \rightarrow [{:}\alpha{:}] \rightarrow [{:}\alpha{:}]$ drops all elements of an array that correspond to a *False* entry in the flag vector. For example, we have $\mathit{packP}\ [{:}\mathit{False},\ \mathit{True},\ \mathit{False}{:}]\ [1,\ 2,\ 3] = [2]$. Finally, the function

$$\mathit{combineP} :: [{:}\mathit{Bool}{:}] \rightarrow [{:}\alpha{:}] \rightarrow [{:}\alpha{:}] \rightarrow [{:}\alpha{:}]$$

combines two arrays based on a flag vector. For example, we have

$$\mathit{combineP}\ [{:}\mathit{False},\ \mathit{True},\ \mathit{False}{:}]\ [2]\ [1,\ 3] = [1,\ 2,\ 3]$$

Based on these functions, we can lift **case** constructs as follows:

```
$\mathcal{L}[\![\textbf{case}\ e\ \textbf{of}\ \triangleleft v_1 \rightarrow e_1;\ \triangleright v_2 \rightarrow e_2]\!]^{vs} =$
  $\textbf{let}$
    $e' \quad\ \ = \mathcal{L}[\![e]\!]^{vs}$
    $\mathit{lflags} = \mathit{getInlFlags}\ e'$
    $\mathit{rflags} = \mathit{getInrFlags}\ e'$
    $e_1' \quad\ = (\mathcal{L}[\![e_1]\!]^{v_1:vs})[v_1\ /\ (\mathit{packP}\ \mathit{lflags}\ v_1)]$
    $e_2' \quad\ = (\mathcal{L}[\![e_2]\!]^{v_2:vs})[v_2\ /\ (\mathit{packP}\ \mathit{rflags}\ v_2)]$
  $\textbf{in}$
  $\mathit{combineP}\ \mathit{lflags}\ e_1'\ e_2'$
```

Here the notation $e_1[x/e_2]$ means to substitute all occurrences of x in e_1 by e_2.

3.4 Vectorising the Sparse Matrix Vector Multiplication

Next, let us use the previously introduced transformation rules to vectorise the *smvm* program. We start from the following definition of the sparse matrix vector multiplication:

```
type SparseRow   = [:(Int, Float):]
type SparseMatix = [:SparseRow:]
type Vector      = [:Float:]
```

$$
\begin{array}{l}
\mathit{smvm} \quad :: \mathit{SparseMatrix} \rightarrow \mathit{Vector} \rightarrow \mathit{Vector} \\
\mathit{smvm}\ \mathit{sm}\ \mathit{vec} = \\
\quad [{:}\mathit{sumP}\ [{:}\ (\mathit{snd}\ \mathit{cx}) * (\mathit{vec}\ !{:}\ (\mathit{fst}\ \mathit{cx}))\ \mid\ \mathit{cx} \leftarrow \mathit{row}{:}] \mid \mathit{row} \leftarrow \mathit{sm}{:}]
\end{array}
$$

We begin the code transformation by lifting the inner array comprehension. The calculation is as follows:

$$
\begin{array}{cl}
 & [{:}(\mathit{snd}\ \mathit{cx}) * (\mathit{vec}\ !{:}\ (\mathit{fst}\ \mathit{cx}))\ \mid\ \mathit{cx} \leftarrow \mathit{row}{:}] \\
= & \{\text{Replacing array comprehensions \& substituting } \mathit{row} \text{ for } \mathit{cx}\} \\
 & \mathcal{L}[\![(\mathit{snd}\ \mathit{row}) * (\mathit{vec}\ !{:}\ (\mathit{fst}\ \mathit{row}))]\!]^{[\mathit{row}]} \\
= & \{\text{Lift function application}\} \\
 & (\mathcal{L}[\![\mathit{snd}\ \mathit{row}]\!]^{[\mathit{row}]}) *^{\uparrow} (\mathcal{L}[\![\mathit{vec}\ !{:}\ (\mathit{fst}\ \mathit{row}))]\!]^{[\mathit{row}]}) \\
= & \{\text{Lift function application and variables}\} \\
 & (\mathit{snd}^{\uparrow}\ \mathit{row}) *^{\uparrow} (\mathcal{L}[\![\mathit{vec}]\!]^{[\mathit{row}]})\ !{:}^{\uparrow}\ (\mathcal{L}[\![\mathit{fst}\ \mathit{row}]\!]^{[\mathit{row}]}) \\
= & \{\text{Lift function application, bound variables, and free variables}\} \\
 & (\mathit{snd}^{\uparrow}\ \mathit{row}) *^{\uparrow} ((\mathit{replicateP}\ (\mathit{lengthP}\ \mathit{row})\ \mathit{vec})\ !{:}^{\uparrow}\ (\mathit{fst}^{\uparrow}\ \mathit{row}))
\end{array}
$$

We successfully replaced the array comprehension by the use of vectorised functions, but unfortunately there is a serious problem with this code. The expression *replicateP* (*lengthP row*) *vec* produces, for each element of the array *row*, a complete copy of *vec*—this is definitely not what we want! It is in situations like this that we need additional optimisation rules, as previously mentioned. The index operation, when lifted and applied to a replicated array, can be replaced by a reverse permutation operation that honours the following equation:

$$
(\mathit{replicateP}\ \mathit{xs}\ (\mathit{lengthP}\ \mathit{inds}))\ !{:}^{\uparrow}\ \mathit{inds}\ =\ \mathit{bpermuteP}\ \mathit{xs}\ \mathit{inds} \qquad (3)
$$

For example, we have *bpermuteP* [1, 2, 3] [0, 1, 0, 2] = [1, 2, 1, 3]. The function *bpermuteP* enjoys another nice property in the context of lifting. This property essentially states that, instead of applying a lifted backpermute operation to a replicated value, we can also do with the simple backpermute:

$$
\begin{array}{r}
\mathit{bpermuteP}^{\uparrow}\ (\mathit{replicateP}\ (\mathit{lengthP}\ \mathit{inds})\ \mathit{xs})\ \mathit{inds} = \qquad\qquad\qquad \\
\mathit{segmentP}\ \mathit{inds}\ (\mathit{bpermuteP}\ \mathit{xs}\ (\mathit{concatP}\ \mathit{inds})) \qquad (4)
\end{array}
$$

We can now continue vectorisation of *smvm* by removing the outer list comprehension (where we already insert the result from the calculation that vectorised the inner comprehension):

$$
\begin{array}{cl}
 & [{:}\mathit{sumP}\ ((\mathit{snd}^{\uparrow}\ \mathit{row}) *^{\uparrow} ((\mathit{replicateP}\ (\mathit{lengthP}\ \mathit{row})\ \mathit{vec})\ !{:}^{\uparrow}\ (\mathit{fst}^{\uparrow}\ \mathit{row}))) \\
 & \mid\ \mathit{row} \leftarrow \mathit{sm}{:}] \\
= & \{\text{Equation (3)}\} \\
 & [{:}\mathit{sumP}\ ((\mathit{snd}^{\uparrow}\ \mathit{row}) *^{\uparrow} (\mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ \mathit{row})))\ \mid\ \mathit{row} \leftarrow \mathit{sm}{:}] \\
= & \{\text{Replacing array comprehension \& substituting } \mathit{sm} \text{ for } \mathit{row}\} \\
 & \mathcal{L}[\![\mathit{sumP}\ ((\mathit{snd}^{\uparrow}\ \mathit{sm}) *^{\uparrow} (\mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ \mathit{sm})))]\!]^{[\mathit{sm}]} \\
= & \{\text{Lift function application}\} \\
 & \mathit{sumP}^{\uparrow}\ (\mathcal{L}[\![(\mathit{snd}^{\uparrow}\ \mathit{sm}) *^{\uparrow} (\mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ \mathit{sm}))]\!]^{[\mathit{sm}]})
\end{array}
$$

$= \{\text{Lift function application (x2) and bound variable; Equation (2)}\}$
$\quad \mathit{sumP}^{\uparrow} \circ \mathit{segmentP}\ \mathit{sm}\ \$$
$\qquad (\mathit{snd}^{\uparrow}\ (\mathit{concatP}\ \mathit{sm})) *^{\uparrow} (\mathit{concatP}\ (\mathcal{L}[\![\mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ \mathit{sm})]\!]^{[\mathit{sm}]}))$
$= \{\text{Lift function application (x2) and variables; Equation (2) \& (4)}\}$
$\quad \mathit{sumP}^{\uparrow} \circ \mathit{segmentP}\ \mathit{sm}\ \$$
$\qquad (\mathit{snd}^{\uparrow}\ (\mathit{concatP}\ \mathit{sm})) *^{\uparrow} \mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ (\mathit{concatP}\ \mathit{sm}))$

Overall, the vectorisation transformation leads us to the following version of *smvm*, which has all array comprehensions removed:

$$\begin{aligned}&\mathit{smvm}\ \mathit{sm}\ \mathit{vec}\ =\ \mathit{sumP}^{\uparrow} \circ \mathit{segmentP}\ \mathit{sm}\ \$\\ &\quad (\mathit{snd}^{\uparrow}\ (\mathit{concatP}\ \mathit{sm})) *^{\uparrow} \mathit{bpermuteP}\ \mathit{vec}\ (\mathit{fst}^{\uparrow}\ (\mathit{concatP}\ \mathit{sm}))\end{aligned}$$

We will see in the next subsection that a suitable choice of representation for nested arrays provides us with implementations for *segmentP*, *concatP*, $\mathit{fst}^{\uparrow}$, and $\mathit{snd}^{\uparrow}$ whose runtime complexity is constant. Moreover, in Section 5, we will see how the application of *sumP*, $(*^{\uparrow})$, and *bpermuteP* can be translated into a single nested loop over unboxed arrays by means of a set of array fusion rules.

3.5 Array Representation

We mentioned before that collection-oriented array operations, and in particular lifted operations, are executed most efficiently on flat array structures. Moreover, we concluded in the previous subsection that operations that alter the nesting structure of arrays, such as *segmentP* and *concatP*, need to execute in constant time. The flattening transformation achieves both goals by decomposing complex data structures into the primitive data values contained in a structure and the information that determines the structure of that data. The exact rules of this representation transformation are discussed in Section 4, but we like to sketch the basic idea here and illustrate it at the example of the sparse matrix vector multiplication.

The array representation transformation proceeds by choosing an array representation in dependence on the element type stored in an array. The simplest case is that of arrays of values of unit type, which can simply be represented by the length of the array—such arrays do not contain any other information. Moreover, arrays of primitive types can be directly represented as unboxed arrays. Slightly more interesting are arrays of pairs, which we represent as a pair of arrays of equal length. The representation of arrays of sum type requires, in addition to two arrays containing the component values, a flag array that indicates for each array element to which alternative of the sum the element belongs; this is exactly the flag array that the function *getInlFlags*, mentioned in the vectorisation rule for the **case** construct, returns. Finally, nested arrays are represented by a flat array that collects the elements of all subarrays plus a so-called *segment descriptor,* which contains the length of all subarrays.[6] With

[6] In fact, to improve the efficiency of some array operations, concrete implementations of segment descriptors usually contain more information than just the lengths of all

this representation for nested arrays, *concatP* and *segmentP* correspond to the removal and addition of a segment descriptor, respectively. As this does not require a traversal of the structure itself, these operations have indeed constant time complexity.

To illustrate the representation transformation, let us look at the representation of sparse matrices:

$$\begin{aligned}&\textbf{type}\ \mathit{SparseRow} &&= [{:}(\mathit{Int},\ \mathit{Float}){:}]\\ &\textbf{type}\ \mathit{SparseMatrix} &&= [{:}\mathit{SparseRow}{:}]\end{aligned}$$

Since *SparseMatrix* is a nested array, it is represented by a pair consisting of a segment descriptor and the representation type of *SparseRow*, which in turn, is an array of pairs, and will therefore be represented by a pair of type ([:*Int*:], [:*Float*:]). Overall, the matrix

$$\begin{bmatrix}5&0&0&0\\0&0&0&7\\3&4&0&0\\0&0&0&0\end{bmatrix}$$

corresponds to the structure

```
([:1, 1, 2, 0:],       -- (simplified) segment descriptor
 ([:0, 3, 0, 1:],      -- position of the values in each subarray
  [:5, 7, 3, 4:]))     -- values of non-zero elements
```

which consists only of flat arrays that have an immediate unboxed representation. Interestingly, this transformation can even be applied to recursive occurrences of array types. The usefulness of such a representation transformation for arrays is repeatedly mentioned in the literature [BS90,PP93,HM95,CPS98,CK00].

4 A Representation Transformation for Parallel Arrays

In this section, we will formalise the representation transformation that was informally introduced at the end of the previous section. Guided by the element type of an array, the transformation separates structural information from the primitive data values contained in a structure. The result is an array representation that minimises the occurrence of pointers in arrays, in favour of the use of unboxed arrays.

4.1 Parallel Arrays as a Polytypic Type

In generic programming, a type constructor whose concrete representation depends on its type argument is called a *type-indexed type* or *polytypic type*. The

the subarrays. However, to keep the presentation simple, we ignore this for the moment.

$$\begin{array}{lll}
\mathbf{type}\ [{:}\,\mathit{Unit}{:}] & = \mathit{Int} & \\
\quad [{:}\rho{:}] \mid \rho \in \{\mathit{Int},\ \mathit{Float},\ \ldots\} & = \mathit{UArr}\ \rho & \text{-- unboxed basic array} \\
\quad [{:}\tau_1 :\!*\!: \tau_2{:}] & = [{:}\tau_1{:}] :\!*\!: [{:}\tau_2{:}] & \\
\quad [{:}\tau_1 :\!+\!: \tau_2{:}] & = \mathit{Sel} :\!*\!: [{:}\tau_1{:}] :\!*\!: [{:}\tau_2{:}] & \\
\quad [{:}[{:}\tau{:}]{:}] & = \mathit{Segd} :\!*\!: [{:}\tau{:}] & \\
 & & \\
\mathbf{type}\ \mathit{Sel} & = [{:}\mathit{Bool}{:}] & \text{-- selector} \\
\mathbf{type}\ \mathit{Segd} & = [{:}\mathit{Int}{:}] & \text{-- segment descriptor}
\end{array}$$

Figure7. Polytypic definition of parallel arrays

choice of representation for a polytypic type naturally affects the implementation of functions operating on that type, too. Such functions, whose concrete implementation depends on a type argument, are called *type-index functions* or *polytypic* functions. Due to these polytypic types and functions, generic programming achieves extra generality on both the value and the type level. Harper & Morrisett [HM95], in their work on implementing polymorphism by way of intensional type analysis, realise type-indexed types and functions by a typecase construct; i.e., by type and value expressions that choose between a number of alternatives on the basis of a type argument. In the case of parallel arrays, this type argument would be the element type of the array. Hinze [Hin00] suggests an alternative implementation scheme based on the compile-time specialisation of all polytypic functions and types, which is partly related to the dictionary-passing implementation of Haskell type classes.

The elimination of pointers, and hence boxed data, from array elements constitutes a *representation transformation,* which we can specify by defining the array type $[{:}\cdot{:}]$ as a type-indexed type [CK00]. This type-indexed type inspects its type index—the element type of the parallel array—by way of a typecase and makes its concrete representation dependent on the structure of the type index. In other words, we usually regard parametric type constructors as free functions over types; typecase allows us to define type constructors that implement a more sophisticated mapping.

Type-indexed definitions determine types and values by matching the type-index against a range of elementary type constructors; in particular, they distinguish the unit type, basic types (such as, *Char*, *Int*, *Float*, etc.), binary products, binary sums, the function space constructor, and in our case, also the parallel array constructor. The latter needs to be considered for nested arrays, as the representation of array nesting is independent of the concrete representation of the inner array.

Figure 7 displays a polytypic definition of parallel arrays, which selects a concrete array representation in dependence on the array element type. Here *Unit*, (:∗:), and (:+:) are the type constructors representing units, products, and sums in generic definitions.[7] Unit arrays are simply represented by their length.

[7] These are from GHC's *Data.Generics* module.

Arrays of pairs are represented by pairs of arrays. Arrays of sums get an extra selector component *Sel* that determines for each element in which of the two flattened subarrays the data resides. Finally, nested arrays are represented by a segment descriptor *Segd* together with a flattened array.

4.2 Polytypic Types as Multi-parameter Type Classes

Given that we are aiming at an implementation in Haskell, the following question arises: how can we implement the representation transformation encoded in the polytypic array definition of Figure 7 in Haskell? Moreover, we need to define the basic operations on parallel arrays as type-indexed functions that adapt their behaviour to the representation type in dependence on the type index. A heavy-weight solution would be to extend the language with a typecase construct, as for example done in the TILT compiler [TMC$^+$96]. An alternative, implementation-intensive approach would be implement general compile-time specialised polytypic functions and types, as done in the Generic Haskell system [CL02]. However, in the interest of simplicity, we prefer to avoid sophisticated language extensions.

Cheney & Hinze [CH02] recently demonstrated that polytypic functions can also be implemented in Haskell 98 extended with existential types by encoding representation types on the value level and using type classes to infer representations for user-defined data types. This is a flexible and general approach to generic programming, but it comes at the expense of extra runtime costs, as the presence of existential types preempts some central optimisations performed by a compiler like GHC. An alternative implementation of polytypism in Haskell by way of a slight language extension has been proposed in [HP01]. However, the proposal in its current form comes with a number of limitations that make it unsuitable for our purposes. In particular, it currently neither supports multi-parameter type classes nor generic types of a kind other than $\star$.

As pointed out in [HJL02], multi-parameter type classes in combination with functional dependencies [Jon00] can be coerced into implementing some forms of type-indexed types. Generally, this approach has its limitations: firstly, it works only for type constructors of some specific kinds and, secondly, the marshalling between user-defined data types and the product-sum representation used in polytypic definitions has to be manually coded for all instances. The first restriction does not affect us, as parallel arrays belong to the polytypic types that can be defined via multi-parameter type classes; and we will see in Subsection 4.6 that we can work around the second restriction to some extent.

In the following, we will explore the encoding of type-indexed data types by way of multi-parameter type classes with functional dependencies. We start by discussing the concrete type mapping needed to implement parallel arrays. Afterwards, we shall have a look at a general approach to overcome the requirement for manually coded instances for the type classes that encode type-indexed types.

4.3 The Concrete Implementation of the Type Mapping

A type-indexed data type can be implemented by a binary type class that relates the type index to the representation type [HJL02]. In other words, to define a type-indexed type TI of kind κ with type index τ, namely $TI\langle\tau :: \star\rangle :: \kappa$, we introduce a type class

$$\textbf{class}\ TI\ \tau\ \rho\ \mid\ \tau \rightarrow \rho$$

where $\rho :: \kappa$. The type class TI essentially denotes a mapping from types $\tau :: \star$ to types $\rho :: \kappa$ and, hence, encodes a type-dependent representation transformation. This mapping is populated by instance declarations for the class. More precisely, for each defining equation of the form $TI\langle t\rangle = s$ of the type-indexed type, we define a class instance

$$\textbf{instance}\ TI\ t\ s$$

Finally, the elementary operations on the type-indexed type TI, which need to be type-indexed functions, are implemented as methods of the class TI.

Following this scheme, we can implement $[:\cdot:]$ using a type class

$$\textbf{class}\ PArray\ e\ arr\ \mid\ e \rightarrow arr,\ arr \rightarrow e$$

$PArray\ e\ arr$ is a bijection between array element types e and array representation types arr for those elements. In other words, e is a type index and arr the representation type for arrays that contain values of the type index as elements. The dependence of the representation type on the type of array elements is captured by the functional dependency $e \rightarrow arr$, as in the scheme illustrated previously by TI. Interestingly, we also need to establish a functional dependency in the opposite direction; i.e., the dependence $arr \rightarrow e$. This has serious implications for the generality of the definition of $PArray$; we shall return to the reasons for this functional dependency as well as its implications in the next subsection.

Given the $PArray$ class declaration, we need to fix the representation types and associate them with the corresponding element types by way of instance declarations. For example, for arrays of unit values, where it suffices to store the length of the array (as all values are the same anyway), we can use

```
newtype PAUnit = PAUnit Int    -- length of unit array
instance PArray Unit PAUnit
```

This instance declaration corresponds to the first equation of Figure 7. Moreover, arrays of products can be defined as follows:

```
data PAProd l r = PAProd l r    -- array of pairs as pair of arrays
instance (PArray a aarr, PArray b barr) ⇒
  PArray (a :*: b) (PAProd aarr barr)
```

Arrays of basic types have a direct representation as unboxed arrays. Let us assume the existence of a type constructor *UArr* that may be parametrised with a range of basic types; i.e., *UArr Int* denotes arrays of unboxed integer values. With this, we can define a *PArray* instance for basic types as

```
newtype PAPrim e = PAPrim (UArr e)
instance PArray Int (PAPrim Int)    -- etc for other basic types
```

Before completing the set of required instances, let us step back for a moment and consider the elementary operations that we need to define as class methods of *PArray*.

4.4 Immutable Versus Mutable Arrays

We have seen in Section 2 that for optimal performance, array codes need to be implemented with mutable arrays in the *ST* monad. Our goal here is to define *PArray* such that array algorithms defined in terms of *PArray*, after inlining and similar optimisations, expand into *ST* monad-based code. Nevertheless, the user-level interface of the library should remain purely functional. Hence, we may use mutable arrays internally, but must convert them into immutable arrays once they are fully defined and returned to user-level code.

Overall, most array-producing operations proceed as follows:

1. Allocate a mutable array of sufficient size.
2. Populate the mutable array with elements using a loop that executes within the *ST* monad.
3. Coerce the fully defined mutable array into an immutable array.

Provided that the mutable array is not altered anymore, the last step can (unsafely) coerce the type without actually copying the array.

A detailed introduction to the *ST* monad, including the outlined strategy for implementing immutable by mutable arrays, is provided by Peyton Jones & Launchbury [PL95]. Here we will constrain ourselves to the example displayed in Figure 8. The code defines the function *replicateU*, which produces an unboxed array of given size where all elements are initialised to the same value. To understand the details, we need to have a look at the interface to unboxed arrays, which defines the abstract data type *UArr e* for immutable, unboxed arrays of element type *e* and *MUArr s e* for mutable, unboxed arrays, where *s* is the state type needed for mutable structures in the *ST* monad. The type constraint that enforces that unboxed arrays can only contain basic types is implemented by the type class *UAE* (which stands for "unboxed array element"). On immutable arrays, we have the following basic operations:

```
lengthU :: UAE e ⇒ UArr e → Int
indexU  :: UAE e ⇒ UArr e → Int → e
```

These two functions obtain an array's length and extract elements, respectively.

```
replicateU     :: UAE e => Int -> e -> UArr e
replicateU n e =
  runST (do
    ma <- newMU n
    fill0 ma
    unsafeFreezeMU ma n)
  where
    fill0 ma = fill 0
    where
      fill off | off == n = return ()
               | otherwise = do
                               writeMU ma off e
                               fill (off + 1)
```

Figure8. Typical use of mutable arrays to define an immutable array

On mutable arrays, we have

```
lengthMU :: UAE e => MUArr s e              -> Int
newMU    :: UAE e => Int                    -> ST s (MUArr s e)
readMU   :: UAE e => MUArr s e -> Int       -> ST s e
writeMU  :: UAE e => MUArr s e -> Int -> e -> ST s ()
```

to create new arrays as well as index and update them. Finally, we can convert mutable into immutable arrays with

```
unsafeFreezeMU :: UAE e => MUArr s e -> Int -> ST s (UArr e)
```

where the second argument provides the length of the immutable array. This may be less than the length with which the mutable array was originally allocated. The structure of this interface was inspired by the unboxed array support of GHC's extension libraries [HCL02].

4.5 From Representation Types to Representation Constructors

Given that we need to handle mutable and immutable arrays, we need to rethink the instance declarations for *PArray* provided in Section 4.3. If we use

```
newtype PAPrim e = PAPrim (UArr e)
instance PArray e (PAPrim e)
```

as proposed before, the *PArray* class is restricted to immutable arrays, as the type constructor *UArr* appears explicitly in the definition of *PAPrim*. As a consequence, we have to define a second, structurally identical class with almost identical instance declarations for mutable arrays; i.e., instance declarations for types that replace *UArr* by *MUArr*. Such duplication of code is obviously not

```
type PArr      (arr :: ⋆ → (⋆ → ⋆) → ⋆) e = arr e UArr
type MPArr s   (arr :: ⋆ → (⋆ → ⋆) → ⋆) e = arr e (MUArr s)

-- Operations that apply to all parallel arrays
class PArray e arr | e → arr, arr → e where
  lengthP :: PArr arr e                  → Int
    -- Yield the length of a parallel array (if segmented, number of segments)
  indexP  :: PArr arr e → Int            → e
    -- Extract an element out of an immutable parallel array
  sliceP  :: PArr arr e → Int → Int → PArr arr e
    -- Extract a slice out of an immutable parallel array

-- Operations that apply only to flat parallel arrays
class PArray e arr ⇒ FArray e arr where
  newMP          :: Int                        → ST s (MPArr s arr e)
    -- Allocate a mutable parallel array of given length
  writeMP        :: MPArr s arr e → Int → e → ST s ()
    -- Update an element in a mutable parallel array
  unsafeFreezeMP :: MPArr s arr e → Int        → ST s (PArr arr e)
    -- Convert a mutable into an immutable parallel array
```

Figure9. Definition of the parallel array classes

desirable. A more elegant solution is to parametrise the definition of the *PArray* instances with the type constructor of the base array.

In other words, we make *PArray e arr* into a bijection between element types $e :: \star$ and *array representation constructors* $arr :: \star \to (\star \to \star) \to \star$. The representation constructor gets two arguments: (1) an array element type of kind $\star$ and (2) a type constructor specifying the base array, which is of kind $\star \to \star$. The main reason for passing the array element type is to avoid ambiguities in type signatures, which we will discuss in more detail later. The type constructor for the base array can be either *UArr* or *MUArr s* and it determines whether we obtain a mutable or an immutable array representation.

Figure 9 introduces two type synonyms *PArr* and *MPArr* to simplify the use of these generalised array constructors. Moreover, it contains the complete class definition of *PArray* including all the elementary functions of the class. Given the previous discussion of the interface for unboxed arrays, most of these functions should be self-explanatory. The only new function is *sliceP*, which extracts a subarray, specified by its start index and length, from an immutable array.

In addition, the class *PArray* is split into two classes. *PArray* itself defines the type-indexed operations that apply to all forms of parallel arrays, whereas *FArray* defines those type-indexed operations that apply only to flat arrays; i.e., to parallel arrays that are not segmented. We will discuss the reason for this distinction in more detail below, in conjunction with the representation of nested arrays.

In the following, we discuss the concrete implementation of the individual equations defining $[: \cdot :]$ in Figure 7 by means of instance declarations of the class *PArray*. In all instances, the concrete element type e will be used as a phantom type; i.e., as a type argument to the array representation constructor that is not used on the right-hand side of the constructor's type definition. The purpose of this type argument is to ensure that the concrete element type appears in signatures for functions that map arrays to arrays, such as *sliceP*, and would otherwise not mention the element type. The presence of the element type avoids ambiguities that would otherwise arise during type checking of overloaded array operations.

To keep the presentation reasonably compact, we will not discuss the definition of the methods for the various instances. However, Appendix A describes how to obtain a library implementation that covers all details.

Arrays of units. In this new setting, the instance declaration for *Unit* reads

```
newtype PAUnit e (ua :: ⋆ → ⋆) = PAUnit Int
instance PArray Unit PAUnit                         -- also for FArray
```

For reasons of uniformity, *PAUnit* is parametrised over the base array type ua; although ua is not used on the right-hand side of the definition, as we only store the length of arrays of units.

Arrays of primitives. In contrast to arrays of units, the definition for primitive types makes use of the base array type:

```
newtype PAPrim r e (ua :: ⋆ → ⋆) = PAPrim (ua r)
 instance PArray Char (PAPrim Char)   -- also for FArray
instance PArray Int   (PAPrim Int)    -- also for FArray
...
```

The constructor of base arrays, ua, is applied to the primitive representation type r. Instances are provided for all types in *UAE*.

Arrays of products. More interesting is the case of products:

```
data PAProd l r e (ua :: ⋆ → ⋆) = forall le re. PAProd (l le ua) (r re ua)
instance (PArray a aarr, PArray b barr) ⇒
   PArray (a :*: b) (PAProd aarr barr)          -- also for FArray
```

Here the base array type ua is passed down into the component arrays constructors l and r, which are being obtained from the type context in the instance declaration. The base array type will finally be used when the component arrays contain elements of primitive type. The use of the existential types *le* and *re*,[8]

[8] Due to the covariance of functions, existentials are introduced by universal quantification denoted **forall**.

which represent the components of the concrete product type, may be surprising. These existentials are necessary as place holders, as we have no means to decompose the type e into its components.

Arrays of sums. We treat sums similar to products, but, in addition to the component arrays, we also provide a selector array (see Figure 7), which is parametrised with the base array *ua* in the same way as the component arrays are.

```
data PASum l r e (ua :: ⋆ → ⋆) = forall le re.
                                   PASum (Sel ua) (l le ua) (r re ua)
instance (PArray a aarr, PArray b barr) ⇒
  PArray (a :*: b) (PASum aarr barr)          -- also for FArray
```

Arrays of arrays. As outlined in Section 3.5, we represent nested arrays by a flat array combined with an extra structure, a segment descriptor, that encodes the partitioning of the flat array into subarrays. Hence, we have a concrete representation and instance as follows:

```
data PAPArr arr e (ua :: ⋆ → ⋆) = forall e'. PAPArr (Segd ua) (arr ua)
instance PArray e arr ⇒ PArray (PArr arr) (PAPArr arr)
```

In contrast to all the previous cases, we cannot provide an *FArray* instance for *PAPArr*. This is essentially as the operations *newMP* and *writeMP* of the *FArray* class need a more complex signature to deal with array segmentation. For example, the length of the segmented array (i.e., the number of segments) is not sufficient for *newMP* to allocate a segmented array structure, as we also need to know the total number of elements across all segments to determine the storage requirements. Moreover, we cannot simply pass the total number of elements to *newMP* as this does not place an upper bound on the storage requirements of the segment descriptor; after all, there may be an arbitrarily large number of empty segments. Hence, we need to introduce a more complex operation

```
newMSP :: FArray r arr ⇒ Int → Int → ST s (MSPArr s arr e)
```

that receives both the number of segments as well as the total number of elements across all segments to allocate a segmented array. Similarly, we provide

```
nextMSP :: FArray r arr
           ⇒ MSPArr s arr e → Int → Maybe r → ST s ()
```

as a replacement for *writeMP*. If the third argument to *nextMSP* is *Nothing*, a new segment (working from left to right) is being initialised. All following calls to *nextMSP* write to the new segment. Alternatively, segmented arrays can be created by constructing a flat array first, and then combining it with a segment descriptor.

The big picture. It is worthwhile to reflect some more on the generalisation of *PArray* that allows us to cover immutable and mutable array representations with a single type mapping. The essential point is the change of the array representation type *arr*, in the mapping *PArray e arr*, from being a manifest type of kind $\star$ to becoming a type constructor of kind $\star \rightarrow (\star \rightarrow \star) \rightarrow \star$, and hence from being an array representation type to being an array representation *constructor.* With this change, an element type no longer implies an array representation, but instead a blueprint that enables the construction of an array representation from a type of base arrays. For example, the element type (*Int* :∗: *Float*) maps to the type constructor *PAProd* (*PAPrim Int*) (*PAPrim Float*). If this type constructor is applied to *UArr*, we get the representation type for immutable arrays of pairs of integer and float values; however, if it is applied to *MUArr s*, we get the representation for mutable arrays of pairs of integer and float values. Hence, *PAProd* (*PAPrim Int*) (*PAPrimFloat*) encodes the type structure for flattened arrays of integer/float pairs, while leaving the type of the underlying unboxed arrays unspecified.

As a consequence, the bijection enforced by the functional dependencies in the type class *PArray* is not violated, even though a single element type relates to the type of both a mutable and an immutable array representation. On the level of the array representation constructor, the mapping is still one-to-one.

4.6 Embedding Projection Pairs

In the encoding of polytypic arrays into type classes, we only provided instances for elementary type constructors (products, sums, and so on). Obviously, this by itself is not sufficient to use arrays of user-defined data types, despite the earlier claim that all algebraic data types can be represented by a combination of these elementary type constructors. In fact, the situation is worse: given our definition of the class *PArray*, it is impossible to use the same array representation for two isomorphic element types. In other words, we cannot represent [:():] and [:*Unit*:] in the same way.

The reason is the bijection required by the functional dependencies in

```
class PArray e arr | e → arr, arr → e
```

The dependency *arr* → *e* asserts that any given array representation is used for exactly one element type. Hence, *PAUnit* can only be used to represent either [:():] or [:*Unit*:], but not both. This naturally leads to the question of whether the dependency *arr* → *e* is really needed. The answer is yes. If the dependency is omitted, some class methods do not type check for recursive types, such as products and sums. The exploration of the precise reason for this is left as an exercise, which requires to study the library implementation referenced in Appendix A.

Fortunately, we can regain the ability to use a single array representation for multiple element types via a different route. The idea is to regard the element type of *PArray* merely as the representation type of the actual element type.

We relate the actual element type with its representation type via another type class that defines a so-called *embedding projection pair (or EP, for short),* which is a pair of functions, *from* and *to*, that map elements from the actual to the representation type and back. So, we have

```
class EP t r | t → r where
  from :: t → r
  to   :: r → t
```

where t is the actual type and r its representation type. Note that we have a functional dependency from the actual to the concrete type only. This implies that for each actual type, we uniquely define a representation type. However, as we do not have a functional dependency in the opposite direction, a single representation type may represent many actual types. In particular, we have, as the simplest cases,

```
instance EP Unit Unit where
  from = id
  to   = id
instance EP () Unit where
  from () = Unit
  to Unit = ()
```

Equipped with *EP*, we can now use *PArray* for a wide range of element types by combining the two type classes in a type context (*EP e r*, *PArray r arr*). This may be read as the actual element type e is represented by the representation type r, which in turn uniquely maps on an array representation *arr*; as illustrated in the following diagram:

$$\begin{tikzcd} e \arrow[r, "EP"'] & r \arrow[r, leftrightarrow, "PArray"'] & arr \end{tikzcd}$$

In other words, we use *EP* to convert between user-level data types and canonical representations as product-sum types. Arrays are, then, defined over the canonical representation.

5 Array Fusion

So far, we have not discussed the third transformation listed in Section 3.2. Array programs that are expressed by means of collective array combinators have a tendency to produce intermediate arrays that are immediately consumed by the operation following their generation. For example, the vectorised version of *smvm*, from Section 3.4, essentially constitutes a three stage pipeline of array traversals. The stages are implemented by *bpermuteP*, *zipWithP* (∗), and *sumSP*.

Such a decomposition of complex computations into pipeline stages makes the code more readable, but it also limits performance. The intermediate arrays consume resources and affect cache locality negatively. In [CK01], we introduce an approach to equational array fusion that is based in GHC's rewrite rules [PHT01]. In the following, we will provide an overview of this approach.

5.1 Loops

There exists a plethora of array combinators; so, any attempt to consider the fusion of all possible pairs of combinators would lead to an unmanageable number of transformation rules. Hence, as in the case of list fusion (i.e., shortcut deforestation), we need a small set of elementary combinators that serve as building blocks for the others; in the case of lists, these combinators are *build* and *foldr* [GLP93]. Then, all that remains is to define fusion for these elementary combinators. In combination with inlining, this also takes care of the non-elementary combinators that are defined in terms of the elementary ones.

As in the case of *build* and *foldr* for lists, we need an array constructing and an array consuming function. As the constructing function, we use

$$\mathit{replicateP} :: (\mathit{EP}\ e\ r,\ \mathit{FArray}\ r\ \mathit{arr}) \Rightarrow \mathit{Int} \rightarrow e \rightarrow \mathit{PArr}\ \mathit{arr}\ e$$

It generates an array of given length where all elements are initialised to the same value. However, the array consuming function is more involved. As is the case of *foldr* for lists, we require that the function can describe both mapping operations as well as reductions. Moreover, it needs to be able to deal with a running accumulator. All this functionality is integrated into a generalised loop combinator:

```
loopP :: (EP e r, PArray r arr, EP e' r', FArrayr' arr')
         ⇒ (acc → e → (acc, Maybe e'))     -- mapping & folding elements
         → acc                              -- initial accumulator value
         → PArr arr e                       -- consumed array
         → (PArr arr' e', acc)
```

The versatility of *loopP* becomes clear when considering the implementation of mapping, reduction, prescan, and filtering in Figure 10. Moreover, the definition of *enumFromToP* (in the same figure) demonstrates how *loopP* can implement complex generators when combined with *replicateP*. This particular combination of *loopP* and *replicateP* may appear wasteful due to the intermediate array generated by *replicateP*. However, the representation transformation from Section 4.3 assigns to arrays of unit type the concrete definition *PAUnit*, which represents unit arrays simply by their length. Thus, the array created by *replicateP* $(\mathit{to} - \mathit{from} + 1)$ *Unit* is represented by nothing other than a plain number, which serves as an upper bound for the iteration encoded in *loopP*.

5.2 Fusion Rules

In the following, we denote rewrite rules as follows:

$$\langle\text{rule name}\rangle\ \forall v_1\ \ldots\ v_n.\ \mathit{exp}_1 \mapsto \mathit{exp}_2$$

where the v_i are the free variables in the rules. These rules should be read as *replace every occurrence of* exp_1 *by* exp_2. GHC permits to embed such rules directly into library code [PHT01].

$$
\begin{array}{lll}
\mathit{mapP} & :: & (\mathit{EP}\ e\ r,\ \mathit{PArray}\ r\ \mathit{arr},\ \mathit{EP}\ e'\ r',\ \mathit{FArray}\ r'\ \mathit{arr}') \\
 & \Rightarrow & (e \to e') \to \mathit{PArr}\ \mathit{arr}\ e \to \mathit{PArr}\ \mathit{arr}'\ e' \\
\mathit{mapP}\ f & = & \mathit{fst} \circ \mathit{loopP}\ (\lambda\_\ e \to (\mathit{Unit},\ \mathit{Just}\ \$\ f\ e))\ \mathit{Unit}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{foldlP} & :: & (\mathit{EP}\ a\ r,\ \mathit{PArray}\ r\ \mathit{arr}) \Rightarrow (b \to a \to b) \to b \to \mathit{PArr}\ \mathit{arr}\ a \to b \\
\mathit{foldlP}\ f\ z & = & \mathit{snd} \circ \mathit{loopP}\ (\lambda a\ e \to (f\ e\ a,\ \mathit{Nothing}))\ z
\end{array}
$$

$$
\begin{array}{lll}
\mathit{scanlP} & :: & (\mathit{EP}\ a\ r,\ \mathit{PArray}\ r\ \mathit{arr}) \Rightarrow (b \to a \to b) \to b \to \mathit{PArr}\ \mathit{arr}\ a \to \mathit{arr} \\
\mathit{scanlP}\ f\ z & = & \mathit{fst} \circ \mathit{loopP}\ (\lambda a\ e \to (f\ e\ a,\ \mathit{Just}\ a))\ z
\end{array}
$$

$$
\begin{array}{lll}
\mathit{filterP} & :: & (\mathit{EP}\ e\ r,\ \mathit{FArray}\ r\ \mathit{arr}) \Rightarrow (e \to \mathit{Bool}) \to \mathit{PArr}\ \mathit{arr}\ e \to \mathit{PArr}\ \mathit{arr}\ e \\
\mathit{filterP}\ p & = & \mathit{fst} \circ \mathit{loopP}\ (\lambda\_\ e \to (\mathit{Unit},\ \mathit{if}\ p\ e\ \mathit{then}\ \mathit{Just}\ e\ \mathit{else}\ \mathit{Nothing}))\ \mathit{Unit}
\end{array}
$$

$$
\begin{array}{l}
\mathit{enumFromToP} \qquad :: (\mathit{Enum}\ e,\ \mathit{EP}\ e\ r,\ \mathit{FArray}\ r\ \mathit{arr}) \Rightarrow e \to e \to \mathit{PArr}\ \mathit{arr}\ e \\
\mathit{enumFromToP}\ \mathit{from}\ \mathit{to} = \\
\quad \mathit{fst} \circ \mathit{loopP}\ (\lambda a\ \_ \to (\mathit{succ}\ a, \mathit{Just}\ a))\ \mathit{from} \circ \mathit{replicateP}\ (\mathit{to} - \mathit{from} + 1)\ \$\ \mathit{Unit}
\end{array}
$$

Figure10. Standard combinators in terms of *loopP*

The first and simplest fusion rule encodes an optimisation that is related to the discussion of *enumFromToP* from Figure 10 in the previous subsection. We can transform any occurrence of a *replicateP* followed by a *loopP* into a modified *loopP*/*replicateP* combination where *replicateP* only produces a unit array, which effectively eliminates the overhead of the intermediate array:

$$
\begin{array}{l}
\langle\text{loopP/replicateP fusion}\rangle\ \forall\ \mathit{mf}\ \mathit{start}\ n\ e\ . \\
\quad \mathit{loopP}\ \mathit{mf}\ \mathit{start}\ (\mathit{replicateP}\ n\ e) \mapsto \\
\quad\quad \mathit{loopP}\ (\lambda \mathit{acc}\ \_ \to \mathit{mf}\ \mathit{acc}\ e)\ \mathit{start}\ (\mathit{replicateP}\ n\ \mathit{Unit})
\end{array}
$$

Fusion of two consecutive loops is more involved. It requires to combine the two mutator functions (first argument to *loopP*) into one. This is achieved by the following rule:

$$
\begin{array}{l}
\langle\text{loopP/loopP fusion}\rangle\ \forall\ \mathit{mf}_1\ \mathit{start}_1\ \mathit{mf}_2\ \mathit{start}_2\ \mathit{arr}. \\
\quad \mathit{loopP}\ \mathit{mf}_2\ \mathit{start}_2\ (\mathit{loopArr}\ (\mathit{loopP}\ \mathit{mf}_1\ \mathit{start}_1\ \mathit{arr})) \mapsto \\
\quad \mathbf{let} \\
\quad\quad \mathit{mf}\ (\mathit{acc}_1,\ \mathit{acc}_2)\ e = \\
\quad\quad \mathbf{case}\ \mathit{mf}_1\ e\ \mathit{acc}_1\ \mathbf{of} \\
\quad\quad\quad (\mathit{acc}_1', \mathit{Nothing}) \to ((\mathit{acc}_1',\ \mathit{acc}_2),\ \mathit{Nothing}) \\
\quad\quad\quad (\mathit{acc}_1', \mathit{Just}\ e') \quad \to \mathbf{case}\ \mathit{mf}_2\ e'\ \mathit{acc}_2\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{acc}_2',\ \mathit{res}) \to ((\mathit{acc}_1',\ \mathit{acc}_2'),\ \mathit{res}) \\
\quad \mathbf{in} \\
\quad \mathit{loopSndAcc}\ (\mathit{loopP}\ \mathit{mf}\ (\mathit{start}_1,\ \mathit{start}_2)\ \mathit{arr})
\end{array}
$$

The accumulator of the combined loop maintains the two components of the original loops as a product and sequences the two mutators. The function *loopSndAcc* drops the accumulator result that corresponds to first loop.

$$\begin{array}{ll} \mathit{loopSndAcc} & :: (\mathit{arr}, (\mathit{acc}_1, \mathit{acc}_2)) \rightarrow (\mathit{arr}, \mathit{acc}_2) \\ \mathit{loopSndAcc}\,(\mathit{arr}, (\mathit{acc}_1, \mathit{acc}_2)) = & (\mathit{arr}, \mathit{acc}_2) \end{array}$$

Further rules are needed to handle loops separated by *zipP* as well as loops over segmented operations. For details, see [CK01].

6 Concluding Remarks

We discussed an approach to purely functional array programming that uses program transformation to turn elegant high-level code into efficient low-level implementations. In particular, we covered code vectorisation in combination with an array representation transformation based on techniques from generic programming. The resulting code is much faster than what can be achieved with Haskell's standard array library and, for simple examples, comes within a factor of 3–4 within the performance of C programs.

We have omitted concrete benchmarks figures for parallel arrays on purpose. The appendix contains a set of programming exercises, which include the implementation of the benchmarks from Section 2 using parallel arrays. We did not want to anticipate the solution to these exercises and instead included comparative benchmarks as part of these exercises.

As of the writing of these lecture notes, code vectorisation has to be manually performed for Haskell programs. In contrast, the array representation transformation and array fusion are automated by a library for the Glasgow Haskell Compiler. The appendix contains instructions on how to obtain this library as well as a set of programming exercises that illustrate how to use the techniques introduced in this text.

Acknowledgements. We like to thank the participants of the Summer School on Advanced Functional Programming 2002, and especially Hal Daume III and Simon Peyton Jones, for feedback on these lecture notes.

A Exercises

The following programming exercises reinforce the concepts and approach presented in these lecture notes. Moreover, they include the setup needed to perform comparative benchmarks that demonstrate the performance improvement of parallel arrays over standard Haskell arrays as well as indicate how much efficiency is sacrificed by using Haskell with parallel arrays instead of plain C code. The support software, installation instructions, and solutions to the exercises are available from the following website:

`http://www.cse.unsw.edu.au/~chak/afp02/`

The website also includes references to further material as well as extensions to what is described in these lecture notes.

A.1 Warm Up Exercise

After installing the support software from the website, start by interactively exploring the types and functions of the `parr` library in GHCi. The website contains some exploratory exercises as well as instructions on library features that were added after the writing of these lecture notes.

The high-level array computation $\mathit{sumP}\ [{:}x * x \mid x \leftarrow [{:}1..n{:}]{:}]$ can be implemented by the following function definition:

```
PArray> let sumSq :: Int -> Int = sumP . mapP (\x -> x * x) . enumFromToP 1
PArray> sumSq 100
338350
```

This function is a nice example for the gains that fusion promises in the extreme case.[9] More about this is on the website.

A.2 A Simple Benchmark

As a first benchmark consider the computation of the dot product:

$$\overline{v} \cdot \overline{w} = \sum_{i=0}^{n-1} v_i w_i$$

To save you the effort required to write benchmark support routines, some code is provided in the directory `labkit/` of the support software. In particular, the module `BenchUtils.hs` contains benchmarking support and `DotP.hs` contains a skeleton for the dot-product benchmark. Just fill in the missing pieces under the heading "Benchmarked code". Then, compile and link the code with the command at the top of `DotP.hs`. This command contains all the options needed to put GHC's optimisation engine into high gear. Run the executable `dotp` to execute your dot-product code on vectors between 100,000 to 500,000 elements.

If you like to compare the execution time with that of a whole range of dot product implementations in Haskell plus one in C, compile and run the code provided in the files `DotProd.hs`, `dotprod.h`, and `dotprod.c`. For instructions on how to compile and run the files, see the file headers. Note that the C function is called from the Haskell code via the *foreign function interface*; i.e., you need to compile the C code before compiling and linking the Haskell module.

A.3 Matrix Vector Multiplication

The following definition of the multiplication of a sparse matrix with a vector is given the high-level array notation of Section 3.1:

$$\begin{aligned}
&\mathit{type}\ \mathit{SparseRow} &&= [{:}(\mathit{Int},\ \mathit{Double}){:}]\\
&\mathit{type}\ \mathit{SparseMatix} &&= [{:}\mathit{SparseRow}{:}]\\
&\mathit{type}\ \mathit{Vector} &&= [{:}\mathit{Double}{:}]
\end{aligned}$$

[9] The example, in fact, appears on the first page of Wadler's deforestation paper.

$$\begin{array}{ll} \mathit{smvm} & :: \mathit{SparseMatrix} \to \mathit{Vector} \to \mathit{Vector} \\ \mathit{smvm}\ \mathit{sm}\ \mathit{vec} & = [{:}\mathit{sumP}\ [{:}x * \mathit{vec}\ !{:}\ \mathit{col} \mid (\mathit{col}, x) \leftarrow \mathit{row}{:}] \mid \mathit{row} \leftarrow \mathit{sm}{:}] \end{array}$$

Section 3.4 demonstrates how to vectorise the above, resulting in code that can be implemented by way of the parallel array library. Implement the vectorised version of *smvm* by extending the file `labkit/smvm/SMVM_fusion.hs`, which together with the other files in the same directory and `BenchUtils.hs` implements four versions of the code (two using Haskell arrays and two using parallel arrays). Compile these files using the same optimisation options as for the dot product benchmark. Note that the module `SMVM_optimal.hs` implements the code that corresponds to the situation where the fusion rules optimise the vectorised code optimally.

Hint: Use `unzipP` where the vectorised code in the lecture notes uses $\mathit{fst}^{\uparrow}$ and $\mathit{snd}^{\uparrow}$.

A.4 Advanced Exercises

Prime numbers. The following implementation of the Sieve of Eratosthenes is interesting as it constitutes a data parallel algorithm that parallelises both loops of the sieve at once:

```
primes             :: Int → [:Int:]
primes n| n ≤ 2    = [::]
        | otherwise =
  let
    sqrPrimes = primes (ceiling (sqrt (fromIntegral n)))
    sieves    = concatP [:[:2 * p, 3 * p..n − 1:] | p ← sqrPrimes:]
    sieves'   = zipP sieves (replicateP (lengthP sieves) False)
    flags     = bpermuteDftP n (const True) sieves'
  in
  dropP 2 [:i | i ← [:0..n − 1:] | f ← flags, f:]
```

Implement this code with the help of the parallel array library.

Quicksort. The quicksort algorithm can be expressed in terms of array comprehensions quite conveniently:

```
qsort    :: Ord α ⇒ [:α:] → [:α:]
qsort [::] = [::]
qsort xs = let
             m      = xs !: (lengthP xs ‘div‘ 2)
             ss     = [:s | s ← xs, s < m:]
             ms     = [:s | s ← xs, s == m:]
             gs     = [:s | s ← xs, s > m:]
             sorted = [:qsort xs' | xs' ← [:ss, gs:]:]
           in
           (sorted !: 0) + ms + (sorted !: 1)
```

Try to vectorise this code using the scheme introduced in Section 3. Interestingly, vectorisation turns the tree-shaped call graph of *qsort* into a linear one, so that all invocations that are on one level of the tree are executed by one invocation to the vectorised variant $qsort^{\uparrow}$ operating on segmented arrays. Implement the vectorised code using the parallel array library. This may require to implement some additional combinators in terms of *loopP* and *loopSP*.

References

[BS90] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[CH02] James Cheney and Ralf Hinze. Poor man's dynamics and generics. In Manuel M. T. Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104. ACM Press, 2002.

[CK00] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In Philip Wadler, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.

[CK01] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In Xavier Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 205–216. ACM Press, 2001.

[CL02] Dave Clarke and Andres Löh. Generic haskell, specifically. In *Proceedings of the IFIP WG2.1 Working Conference on Generic Programming*, 2002.

[CPS98] S. Chatterjee, Jan F. Prins, and M. Simons. Expressing irregular computations in modern Fortran dialects. In *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[DER86] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986.

[GLP93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232. ACM, 1993.

[HCL02] Haskell core libraries (base package). http://haskell.org/ghc/docs/latest/html/base/index.html, 2002.

[Hin00] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.

[HJL02] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed datatypes. In *Proceedings of the 6th Mathematics of Program Construction Conference.* Springer-Verlag, 2002.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.

[HP01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 2001.

[Jon00] Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[PHT01] Simon Peyton Jones, Tony Hoare, and Andrew Tolmach. Playing by the rules: rewriting as a practical optimisation technique. In *Proceedings of the ACM SIGPLAN 2001 Haskell Workshop*, 2001.

[PL95] Simon Peyton Jones and John Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[PP93] Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM Press.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[Wad88] P. Wadler. Deforestation. In *Proceedings of the European Symposium on Programming'88*, number 300 in Lecture Notes in Computer Science, 1988.