Vectorisation Avoidance

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy
Ben Lippmeier[†] Simon Peyton Jones[‡]

[†]School of Computer Science and Engineering
University of New South Wales, Australia
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Abstract

Flattening nested parallelism is a vectorising code transform that converts irregular nested parallelism into flat data parallelism. Although the result has good asymptotic performance, flattening thoroughly restructures the code. Many intermediate data structures and traversals are introduced, which may or may not be eliminated by subsequent optimisation. We present a novel program analysis to identify parts of the program where flattening would only introduce overhead, without appropriate gain. We present empirical evidence that avoiding vectorisation in these cases leads to more efficient programs than if we had applied vectorisation and then relied on array fusion to eliminate intermediates from the resulting code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; Polymorphism; Abstract data types

General Terms Languages, Performance

Keywords Nested data parallelism, Haskell, Program transformation

1. Introduction

Data Parallel Haskell (DPH) is an extension to the Glasgow Haskell Compiler (GHC) that implements *nested data parallelism*. DPH is based on a vectorising program transformation called *flattening* [5, 6, 14, 15, 17], which converts irregular and nested data parallelism into regular traversals over multiple flat arrays. Flattening simplifies load balancing and enables SIMD parallelism together with cache-friendly, linear traversals of unboxed arrays.

Unfortunately, without subsequent aggressive optimisation, flattened code is grossly inefficient on current computer architectures. A key aspect of flattening is to convert *scalar operations* into aggregate *array operations*, such as turning floating-point addition into the element-wise addition of two arrays of floats. However, *intermediate* scalar values in the source code are also converted to arrays, so values that were once stored in scalar registers are now shuffled to and from memory between each array operation.

Haskell'12, September 13, 2012, Copenhagen, Denmark.

The observation that flattening increases memory traffic is not new [7, 10, 11, 19], and DPH uses array fusion [12, 13] to combine successive array operations back into a single traversal. While this works for small kernels, relying on fusion alone turns out to have serious drawbacks:

1. Array fusion can be fragile, because it depends on many enabling code transformations.
2. Specifically, fusion depends critically on inlining — which must be conservative in the presence of sharing to avoid work duplication, and cannot handle recursive definitions. Aggressive inlining leads to large intermediate programs, and hence long compile times.
3. Even if fusion goes well, GHC's current back-end code generators cannot properly optimise fused code, which leads to excessive register use in the resulting machine code.

Tantalisingly, we have identified many common situations in which vectorisation provides no benefit, though the overheads introduced are very much apparent. Thus motivated, we present a program analysis that allows us to completely avoid vectorising parts of the program that do not require it. To our knowledge, this is the first attempt to guide flattening-based vectorisation so that vectorisation is avoided where it is not needed, instead of relying on a subsequent fusion stage to clean up afterwards.

In summary, we make the following contributions:

- We characterise those parts of DPH programs where vectorisation introduces overheads without appropriate gain (Section 2 & 3).
- We introduce a program analysis to identify subexpressions that need not be vectorised, and modify the flattening transform to lift entire expressions to vector-space, so that its intermediates are not converted to array values. (Section 4).
- We present empirical evidence supporting our claim that vectorisation avoidance is an improvement over plain array fusion, at least for our benchmarks (Section 5)

In our previous work [11] we introduced *partial vectorisation*, which allows vectorised and unvectorised code to be combined in the same program. With partial vectorisation, the unvectorised portion fundamentally *cannot* be vectorised, such as when it performs an IO action. In contrast, this paper presents *vectorisation avoidance*, where the unvectorised portion is code that *could* be vectorised, but we choose not to for performance reasons. We discuss further related work in Section 6.

2. Too Much Vectorisation

Vectorisation for higher-order programs is a complex transformation, but we only need the basics to appreciate the problem addressed in this paper. In this section, we walk though a simple example that highlights the problem with intermediate values, and sketch our solution. A more comprehensive description of the vectorisation transform can be found in Peyton Jones et al. [17].

2.1 Parallel force calculation

We use a parallel gravitation simulation as a running example. The simulation consists of many massive points (bodies), each with a mass, a location in two dimensions, and a velocity vector:

```
type Mass      = Float
type Vector    = (Float, Float)
type Location  = Vector
type Velocity  = Vector
type Accel     = Vector
data MassPoint = MP Mass Location Velocity
```

At each time step we compute the acceleration due to gravitational force between each body, and use the sum of accelerations to update each body's velocity. The following `accel` function computes the acceleration a body experiences due to the gravitational force between it and a second body. The `eps` (epsilon) parameter is a smoothing factor that prevents the acceleration from approaching infinity when the distance between the two bodies approaches zero.

```
accel :: Float -> MassPoint -> MassPoint -> Accel
accel eps (MP _ (x1, y1) _) (MP m (x2, y2) _)
 = let dx   = x1 - x2
       dy   = y1 - y2
       rsqr = dx * dx + dy * dy + eps * eps
       r    = sqrt rsqr
       aabs = m  / rsqr
   in  (aabs * dx / r , aabs * dy / r)
```

In DPH we express nested data parallelism using bulk operations on *parallel arrays*. The type of parallel arrays is written `[:e:]`, for some element type `e`. Parallel array operators behave like Haskell's standard list operators (but on arrays), and are identified by a suffix `P` — for example, `mapP`, `unzipP`, `sumP`, and so on. We also use *parallel array comprehensions*, which behave similarly to list comprehensions [17]. The crucial difference between Haskell lists and parallel arrays is that the latter have a parallel evaluation semantics. Demanding any element in a parallel array results in them all being computed, and on a parallel machine we expect this computation to run in parallel.

Using parallel array comprehensions, we implement a naïve $O(n^2)$ algorithm that takes a parallel array of bodies, and computes the gravitational acceleration acting on each one:

```
allAccels :: Float -> [:MassPoint:] -> [:Accel:]
allAccels eps mps
 = [: (sumP xs, sumP ys)
    | mp1 <- mps
    , let (xs, ys) = unzipP [: accel eps mp1 mp2
                             | mp2 <- mps :]
    :]
```

The degree of parallelism in `allAccels` is n^2: for each element `mp1` of `mps` the inner comprehension computes `accel` for every other element `mp2` in `mps`. We also sum the individual accelerations on each body using $2 * n$ parallel sums in parallel. In the full simulation we would then use the resulting accelerations to compute new velocities and positions for each body.

Although this is a naïve algorithm, it presents the same challenges with respect to vectorisation of intermediates as the more sophisticated Barnes-Hut algorithm. We stick to the simpler version to avoid complications with the irregular tree structures that Barnes-Hut uses, which are orthogonal to the ideas discussed in this paper.

2.2 Lifting functions into vector space

Vectorising `accel` from the previous section and simplifying yields $\texttt{accel}_L$ shown below. The subscript L is short for "lifted", and we describe vectorisation as *lifting a function to vector space*.

```
accel_L :: PArray Float
        -> PArray MassPoint -> PArray MassPoint
        -> PArray Accel
accel_L epss (MP_L _ (xs1,ys1) _) (MP_L ms (xs2,ys2) _)
 = let dxs   = xs1 -_L xs2
       dys   = ys1 -_L ys2
       rsqrs = dxs *_L dxs +_L dys *_L dys +_L epss *_L epss
       rs    = sqrt_L rsqrs
       aabss = ms  /_L rsqrs
   in  (aabss *_L dxs /_L rs , aabss *_L dys /_L rs)
```

In the type of $\texttt{accel}_L$, the `PArray` constructor is our internal version of `[::]` which only appears in vectorised code. A value of type `(PArray a)` contains elements of type `a`, which are stored in a type-dependent manner. For example, we store an array of pairs of integers as two arrays of unboxed integers. This "unzipped" representation avoids the intermediate pointers that would otherwise be required in a completely boxed representation. Similarly, we store an array of `MassPoint` as five separate arrays: an array of masses, two arrays for the x and y components of their locations, and two arrays for the x and y components of their velocities. We can see this in the above code, where we add a suffix "`s`" to variable names to highlight that they are now array-valued.

Lifting of `accel` itself is straightforward, because it consists of a single big arithmetic expression with no interesting control flow. The lifted versions of primitive functions correspond to their originals, except that they operate on arrays of values. For example:

```
(*_L) :: PArray Float -> PArray Float -> PArray Float
```

Pairwise multiplication operates as $\texttt{(*}_L\texttt{)} = \texttt{zipWithP (*)}$ and is provided by the back-end DPH libraries, along with other lifted primitives.

Returning to the definition of $\texttt{accel}_L$ we see the problem with intermediate values. Whereas the variables `rsqr`, `r`, `dx`, `dy` and so on were scalar values in the source code, in the lifted version they have become array values. *Without array fusion, each of these intermediate arrays will be materialised in memory*, whereas with the unvectorised program they would exist only transiently in scalar registers.

The original `accel` function does not use nested parallelism, branching, or anything else that justifies an elaborate transformation, so in fact this simpler code would be far better:

```
accel_L epss (MP_L _ (xs1,ys1) _) (MP_L ms (xs2,ys2) _)
 = let f eps x1 y1 x2 y2 m
         = let dx   = ...; dy = ...;
               rsqr = ...; r  = ...; aabs = ...
               ... ⟨as in original definition of accel⟩ ...
           in  (aabs * dx / r , aabs * dy / r)
   in  zipWithPar6 f epss xs1 ys1 xs2 ys2 ms
```

Here, `zipWithPar6` is a 6-ary parallel-array variant of `zipWith`. This simpler version of $\texttt{accel}_L$ performs a single simultaneous traversal of all six input arrays, *with no array-valued intermediates*.

This simpler version of $\texttt{accel}_L$ works only because the original version *does not call any parallel functions*, directly or indirectly. This property ensures that the `f` passed to `zipWithPar6` does not introduce any nested parallelism itself. If the original `accel` had itself used a data-parallel function, then that function would need to be called for each pair of points — and this nested invocation of data-parallel functions is precisely what flattening is supposed to eliminate.

So the question is this: can we identify *purely-sequential subcomputations* (such as `accel`) that will not benefit from vectorisation? If so, we can generate simple, efficient `zipWith`-style loops for these computations, instead of applying the general vectorisation transformation and hoping that fusion then eliminates the intermediate arrays it introduces.

2.3 Maximal sequential subexpressions

The situation in the previous section was the ideal case. With `accel` we were able to avoid vectorising its entire body because it did not use any data parallel functions. This made `accel` completely sequential. We may not be that lucky, and the function we want to vectorise may contain parallel subexpressions as well as sequential ones. In general, to perform vectorisation avoidance we need to identify *maximal sequential subexpressions*.

To illustrate this idea, consider a variant of `accel`, where the smoothing factor is not passed as an argument, but instead determined by a parallel computation that references `dx` and `dy`:

```
accel' (MP _ (x1, y1) _) (MP m (x2, y2) _)
 = let dx   = x1 - x2
       dy   = y1 - y2
       eps  = ⟨a data-parallel computation with dx and dy⟩
       rsqr = dx * dx + dy * dy + eps * eps
       r    = sqrt rsqr
       aabs = m / rsqr
   in  (aabs * dx / r , aabs * dy / r)
```

This time we cannot lift `accel'` by wrapping the entire body in a parallel zip, as this would yield a nested parallel computation. Instead, we proceed as follows: (1) identify all maximal sequential subexpressions; (2) lambda lift each of them, and (3) use `zipWithPar`n to apply each lambda-lifted function element-wise to its lifted (vectorised) free variables. Doing this to the above function yields:

```
accel'_L (MP_L _ (xs1, ys1) _) (MP_L ms (xs2, ys2) _)
  = let dxs   = zipWithPar (λx1 x2. x1 - x2) xs1 xs2
        dys   = zipWithPar (λy1 y2. y1 - y2) ys1 ys2
        epss  = ⟨lifted data-parallel computation with dxs and dys⟩
    in  zipWithPar4 (λm dx dy eps.
          let rsqr = dx * dx + dy * dy + eps * eps
              r    = sqrt rsqr
              aabs = m / rsqr
          in  (aabs * dx / r, aabs * dy / r))
          ms dxs dys epss
```

The resulting lifted function $\texttt{accel'}_L$ contains three parallel array traversals, whereas $\texttt{accel}_L$ only had one. Nevertheless, that is still much better than the 13 traversals that would be in the vectorised code without vectorisation avoidance.

We were able to encapsulate the bindings for `rsqr`, `r`, and `aabs` by the final traversal because after the `eps` binding, there are no more parallel functions. In general, we get the best results when intermediate sequential bindings are floated close to their use sites, as this makes them more likely to be encapsulated along with their consumers.

2.4 Conditionals and recursive functions

The function `accel` did not have any control-flow constructs. For an example with a conditional, consider a division function that tests for a zero divisor:

```
divz :: Float -> Float -> Float
divz x y = if (y == 0)
           then 0
           else x `div` y
```

Without vectorisation avoidance, the lifted version of `divz` is as follows:

```
divz_L :: PArray Float -> PArray Float -> PArray Float
divz_L xs ys
 = let n                = lengthPA ys
       flags            = (ys ==_L (replicatePA n 0))
       (xs_then, xs_else) = splitPA flags xs
       (ys_then, ys_else) = splitPA flags ys
   in  combinePA flags (replicatePA (countPA flags) 0)
                       (xs_else `div`_L ys_else)
```

We first compute an array of `flags` indicating which branch to take for each iteration. We use these flags to split (using `splitPA`) the array-valued free variables into the elements for each branch, and apply the lifted version of each branch to just those elements associated with it. Finally, `combinePA` re-combines the results of each branch using the original array of flags. Using these extra intermediate arrays ensures that the parallel execution of $\texttt{divz}_L$ will be load-balanced, as each intermediate array can be redistributed such that each worker is responsible for the same number of elements. To see this, suppose that performing a division on our parallel machine is more expensive than simply returning a constant 0. If we were to evaluate multiple calls to `divz` in parallel without using our `splitP`/`combineP` technique, there is a chance that one processor would need to evaluate more divisions than another, leading to work imbalance. However, in the code above we split the input arrays (`xs` and `ys`) into the elements corresponding to each branch, and use *all* processors to compute the results for both branches. As all processors are used to evaluate both branches, we can be sure the computation is perfectly balanced.

Unfortunately, both `splitP` and `combineP` are expensive. If we were to execute the unvectorised version of `divz` then the arguments for `x` and `y` would be passed in scalar registers. The flag indicating which branch to take would also be computed in a register. Once again, when we vectorise `divz` these intermediate values are converted to arrays, which reifies them in memory. For this example, the extra memory traffic introduced will far exceed the gain from improved load balancing; it would be better to avoid vectorising the conditional entirely and instead generate:

```
divz_L xs ys
 = zipWithPar2
     (λx y. if (y == 0) then 0 else x `div` y)
     xs ys
```

The situation becomes even more interesting when recursion is involved, such as in this familiar function:

```
fac :: Int -> Int -> Int
fac acc n
 = if n == 0 then acc
             else fac (n * acc) (n - 1)
```

Lifting this function without vectorisation avoidance will use `splitP` and `combineP` to flatten the conditional, and the recursive call will be to the lifted version. The consequence is excessive data movement for each recursion and an inability to use the tail recursion optimisation.

In contrast, using vectorisation avoidance we can confine the recursive call to the sequential portion of the function:

```
fac_L :: PArray Int -> PArray Int -> PArray Int
fac_L accs ns
 = let f acc n
       = if n == 0 then acc
                   else f (n * acc) (n - 1)
   in  zipWithPar2 f accs ns
```

This code risks severe load imbalance if the depth of the recursion varies widely among the instances computed on each processor. At the same time, it also avoids the high overheads of the fully vectorised version. Which is the lesser evil?

In our experience so far, using vectorisation avoidance for purely sequential recursive functions has always produced faster code. We intend to provide a pragma to override this default approach in exceptional cases.

2.5 Vectorisation avoidance in a nutshell

In summary, vectorisation avoidance consists of three steps. First, we identify the maximal sequential subexpressions. Second, we lambda lift those subexpressions. Third, we use `zipWithPar`n to apply each lambda-lifted function (in parallel) to each valuation of its free variables. As a result we generate fewer array traversals and fewer intermediate arrays, though we may sacrifice some load balancing by doing so. We will formalise these steps in Section 4, but first we pause to review the vectorisation transformation itself.

3. Vectorisation Revisited

This section reviews the features of vectorisation that are central to subsequently adding vectorisation avoidance in Section 4. For a complete description of vectorisation, see our previous work [17].

3.1 Parallel arrays, maps, and the `Scalar` type class

As mentioned in Section 2.2, in vectorised code we store parallel array data in a type-dependent manner, so the representation of `PArray a` depends on the element type `a`. Haskell, we realise this using type families [8, 9] thus:

```
data family   PArray a
data instance PArray Int        = PInt (Vector Int)
data instance PArray (a, b)     = P2 (PArray a) (PArray b)
data instance PArray (PArray a) = PNested VSegd (PDatas a)
... more instances ...
```

For arrays of primitive values such as `Int` and `Float` we use an unboxed representation, specifically the unboxed arrays from the `vector` package.[1] Using unboxed arrays improves absolute performance over standard boxed arrays, and admits a simple load balancing strategy when consuming them. For structured data we hoist the structure to top-level: representing an array of structured values as a tree with with unboxed arrays of scalars at its leaves. The instance for pairs is shown above.

The `PArray` type family also supports flattening of nested arrays, and arrays of sum types [10, 16, 17]. A partial definition of nested arrays is shown above, but we leave discussion of the `VSegd` segment descriptor and `PDatas` data blocks to [16].

For vectorisation avoidance, the important point is that we can only bail out to a sequential function if the elements that function consumes can be extracted from their arrays in constant time. As discussed in [16], it is not possible to select a nested array from an array of arrays in constant time, due to time required to cull unused segments from the resulting segment descriptor. Now, consider the $\texttt{accel}_L$ function back in Section 2.2. The body of its sequential part `f` *does* run in constant time, so `zipWithPar6` can't take any longer than this to extract its arguments — otherwise we will worsen the asymptotic complexity of the overall program.

We side-step this complexity problem by restricting vectorisation avoidance to subexpressions that process scalar values only. These are the primitive types such as `Int` and `Float`, as well as tuples of primitive types, and enumerations such as `Bool` and `Ordering`. We collect these types into the `Scalar` type class, which is the class of types that support constant time indexing. When we determine maximal subexpressions as per Section 2.3, all free variables, along with the result, must be `Scalar`.

[1] `http://hackage.haskell.org/package/vector`

Now that we have the `Scalar` class, we can write down the full types of the n-ary parallel mapping functions we met back in Section 2.2.

```
mapPar :: (Scalar a, Scalar b)
       => (a -> b) -> PArray a -> PArray b

zipWithPar2 :: (Scalar a, Scalar b, Scalar c)
            => (a -> b -> c)
            -> PArray a -> PArray b -> PArray c

zipWithPar3
  :: (Scalar a, Scalar b, Scalar c, Scalar d)
  => (a -> b -> c -> d)
  -> PArray a -> PArray b -> PArray c -> PArray d
⟨and so on⟩
```

These *scalar mapping functions* apply their worker element-wise, and in parallel to their argument arrays. They are provided by the DPH back-end library.

The key goal of vectorisation avoidance is to maximise the work done by these mapping functions per corresponding tuple of array elements. The sequential subexpressions from Section 2.3 are subexpressions that can be mapped by the scalar mapping functions, and we maximise the work performed by finding subexpressions that are as big as possible.

3.2 Vectorising higher-order functions

Vectorisation computes a *lifted* version of every function that is involved in a parallel computation, which we saw in Section 2.2. Previously, we simplified the discussion by statically replacing expressions of the form `mapP f xs` (equivalently `[:f x| x <- xs:]`) by $\texttt{f}_L$ `xs` — an application of the lifted version of `f`. This static code rewriting is not sufficient for a higher-order language. Statically, we cannot even determine which worker functions will be passed to `mapP`, and hence need a lifted version. Instead, we closure-convert lambda abstractions and dynamically dispatch between the original and lifted versions of a function [17].

In GHC, the vectorisation transformation itself operates on the Core intermediate language, which is an extension of System F [21]. Figure 1 displays the transformation on a slightly cut-down version of Core, which contains all elements that are relevant to our discussion of vectorisation avoidance. The central idea is that given a top-level function definition $f :: \tau = e$, the *full vectorisation transformation* produces a definition for the *fully vectorised* version of f named f_V:

$$f_\mathrm{V} :: \mathcal{V}_t[\![\tau]\!] = \mathcal{V}[\![e]\!]$$

We vectorise types with $\mathcal{V}_t[\![\cdot]\!]$ and values with $\mathcal{V}[\![\cdot]\!]$. In general, if $e :: \tau$, then $\mathcal{V}[\![e]\!] :: \mathcal{V}_t[\![\tau]\!]$. The vectorisation of types and values goes hand in hand.

3.2.1 Vectorising types

To vectorise types with $\mathcal{V}_t[\![\cdot]\!]$, we replace each function arrow `(->)` by a *vectorised closure* `(:->)`. We also replace each parallel array `[::]` by the type-dependent array representation `PArray` from Section 3.1. For example:

$$\mathcal{V}_t[\![\texttt{Int -> [:Float:] -> Float}]\!] = \texttt{Int :-> PArray Float :-> Float}$$

Figure 1 also shows vectorisation of data types, which we use for the arrays of tuples described in Section 3.1.

$\mathcal{V}_t[\![\tau]\!] :: \mathit{Type} \to \mathit{Type}$ is the vectorisation transformation on types

$$
\begin{array}{rcll}
\mathcal{V}_t[\![\tau_1 \texttt{->} \tau_2]\!] & = & \mathcal{V}_t[\![\tau_1]\!] \texttt{ :-> } \mathcal{V}_t[\![\tau_2]\!] & \text{Functions} \\
\mathcal{V}_t[\![\texttt{[:}\tau\texttt{:]}]\!] & = & \mathcal{L}_t[\![\tau]\!] & \text{Parallel arrays} \\
\mathcal{V}_t[\![\texttt{Int}]\!] & = & \texttt{Int} & \text{Primitive scalar types} \\
\mathcal{V}_t[\![\texttt{Float}]\!] & = & \texttt{Float} & \\
\mathcal{V}_t[\![T\ \tau_1 \ldots \tau_n]\!] & = & T_\mathrm{V}\ \mathcal{V}_t[\![\tau_1]\!] \ldots \mathcal{V}_t[\![\tau_n]\!] & \text{Algebraic data types (e.g. lists)} \\
\\
\mathcal{L}_t[\![\tau]\!] & = & \texttt{PArray}\ \mathcal{V}_t[\![\tau]\!] &
\end{array}
$$

$\mathcal{V}[\![e]\!] :: \mathit{Expr} \to \mathit{Expr}$ is the full vectorisation transformation on terms
Invariant: if $\overline{x_i : \sigma_i} \vdash e : \tau$ then $\overline{x_i : \mathcal{V}_t[\![\sigma_i]\!]} \vdash \mathcal{V}[\![e]\!] : \mathcal{V}_t[\![\tau]\!]$

$$
\begin{array}{rll}
\mathcal{V}[\![k]\!] & = k & k \text{ is a literal} \\
\mathcal{V}[\![f]\!] & = f_\mathrm{V} & f \text{ is bound at top level} \\
\mathcal{V}[\![x]\!] & = x & x \text{ is locally bound (lambda, let, etc)} \\
\mathcal{V}[\![C]\!] & = C_\mathrm{V} & C \text{ is data constructor with } C :: \tau \text{ and } C_\mathrm{V} :: \mathcal{V}[\![\tau]\!] \\
\mathcal{V}[\![e_1\ e_2]\!] & = \mathcal{V}[\![e_1]\!]\ \texttt{\$:}\ \mathcal{V}[\![e_2]\!] & \\
\mathcal{V}[\![\lambda x.e]\!] & = \texttt{Clo}\ \{\texttt{env} = (y_1, \ldots, y_k) & \\
 & \quad , \texttt{clo}_s = \lambda env\, x.\, \texttt{case}\, e\, \texttt{of}\ (y_1, \ldots, y_k) \to \mathcal{V}[\![e]\!] & \\
 & \quad , \texttt{clo}_l = \lambda env\, x.\, \texttt{case}\, e\, \texttt{of}\ \texttt{ATup}_k\ n\ y_1 \ldots y_k \to \mathcal{L}[\![e]\!]\ n\} & \\
 & \text{where} \{y_1, \ldots, y_k\} = \text{free variables of } \lambda x.e & \\
\mathcal{V}[\![\texttt{if}\, e_1 \texttt{then}\, e_2 \texttt{else}\, e_3]\!] & = \texttt{if}\, \mathcal{V}[\![e_1]\!]\, \texttt{then}\, \mathcal{V}[\![e_2]\!]\, \texttt{else}\, \mathcal{V}[\![e_3]\!] & \\
\mathcal{V}[\![\texttt{let}\, x \texttt{=}\, e_1 \texttt{in}\, e_2]\!] & = \texttt{let}\, x \texttt{=}\, \mathcal{V}[\![e_1]\!]\, \texttt{in}\, \mathcal{V}[\![e_2]\!] & \\
\mathcal{V}[\![\texttt{case}\, e_1\, \texttt{of}\, C\, x_1 \ldots x_k\ \to\ e_2]\!] & = \texttt{case}\, \mathcal{V}[\![e_1]\!]\, \texttt{of}\, C_\mathrm{V}\, x_1 \ldots x_k\ \to\ \mathcal{V}[\![e_2]\!] &
\end{array}
$$

$\mathcal{L}[\![e]\!]\ n :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}$ is the lifting transformation on terms
Invariant: if $\overline{x_i : \sigma_i} \vdash e : \tau$ then $\overline{x_i : \mathcal{L}_t[\![\sigma_i]\!]} \vdash \mathcal{L}[\![e]\!]\ n : \mathcal{L}_t[\![\tau]\!]$
where n is the length of the result array

$$
\begin{array}{rll}
\mathcal{L}[\![k]\!]\ n & = \texttt{replicatePA}\ n\ k & k \text{ is a literal} \\
\mathcal{L}[\![f]\!]\ n & = \texttt{replicatePA}\ n\ f_\mathrm{V} & f \text{ is bound at top level} \\
\mathcal{L}[\![x]\!]\ n & = x & x \text{ is locally bound (lambda, let, etc)} \\
\mathcal{L}[\![e_1\ e_2]\!]\ n & = \mathcal{L}[\![e_1]\!]\ n\ \texttt{\$:}_\mathrm{L}\ \mathcal{L}[\![e_2]\!]\ n & \\
\mathcal{L}[\![C]\!]\ n & = \texttt{replicatePA}\ n\ C_\mathrm{V} & C \text{ is a data constructor} \\
\mathcal{L}[\![\lambda x.e]\!]\ n & = \texttt{AClo}\ \{\texttt{aenv} = \texttt{ATup}_k\ n\ y_1 \ldots y_k, & \\
 & \quad , \texttt{aclo}_s = \lambda env\, x.\, \texttt{case}\, env\, \texttt{of}\ (y_1, \ldots, y_k) \to \mathcal{V}[\![e]\!] & \\
 & \quad , \texttt{aclo}_l = \lambda env\, x.\, \texttt{case}\, env\, \texttt{of}\ \texttt{ATup}_k\ n'\ y_1 \ldots y_k \to \mathcal{L}[\![e]\!]\ n'\} & \\
 & \text{where} \{y_1, \ldots, y_k\} = \text{free variables of } \lambda x.e & \\
\mathcal{L}[\![\texttt{if}\, e_1 \texttt{then}\, e_2 \texttt{else}\, e_3]\!]\ n & = \texttt{combinePA}\ e_1'\ e_2'\ e_3' & \\
 & \text{where}\ e_1' = \mathcal{L}[\![e_1]\!]\ n & \\
 & \quad e_2' = \texttt{case}\ ys_2\ \texttt{of}\ \texttt{ATup}_k\ n_2\ y_1 \ldots y_k \to \mathcal{L}'[\![e_2]\!]\ n_2 & \\
 & \quad e_3' = \texttt{case}\ ys_3\ \texttt{of}\ \texttt{ATup}_k\ n_3\ y_1 \ldots y_k \to \mathcal{L}'[\![e_3]\!]\ n_3 & \\
 & \quad (ys_2, ys_3) = \texttt{splitPA}\ e_1'\ (\texttt{ATup}_k\ n\ y_1 \ldots y_k) & \\
 & \quad \{y_1, \ldots, y_k\} = \text{free variables of } e_2, e_3 & \\
\mathcal{L}'[\![e]\!]\ n & = \texttt{if}\ n\texttt{==}0\ \texttt{then emptyPA else}\ \mathcal{L}[\![e]\!]\ n & \\
\mathcal{L}[\![\texttt{let}\, x \texttt{=}\, e_1 \texttt{in}\, e_2]\!]\ n & = \texttt{let}\, x \texttt{=} \mathcal{L}[\![e_1]\!]\ n\, \texttt{in}\, \mathcal{L}[\![e_2]\!]\ n & \\
\mathcal{L}[\![\texttt{case}\, e_1\, \texttt{of}\, C\, x_1 \ldots x_k\ \to\ e_2]\!]\ n & = \texttt{let}\, v \texttt{=} \mathcal{L}[\![e_1]\!]\ n & \text{scrutiniser cast to representation type} \\
 & \quad \texttt{in case cast}\, v'\, \texttt{of \_}\, x_1 \ldots x_k\ \to\ \mathcal{L}[\![e_2]\!]\ n &
\end{array}
$$

Figure 1. The vectorisation transformation without avoidance

3.2.2 Vectorised closures

In general, in a higher-order language it is not possible to statically determine which functions may end up as `f` in a specific data-parallel application `mapP f xs`. Therefore, all values of function type must include both a scalar as well as a lifted version of the original function. To make matters worse, functions may be put into arrays — indeed, the vectorisation transformation must do this itself when lifting higher-order functions.

The trouble with arrays of functions is that (a) we *do* want to use standard array operations, such as `filterP`, but (b) do *not* want to represent `PArray (a -> b)` as a boxed array of function pointers. If we were to use a boxed array of function pointers then we would sacrifice much data parallelism, as we explained in previous work [17]. With this in mind, we represent single vectorised functions as explicit closures, containing the following:

1. the scalar version of the function,
2. the parallel (lifted) version of the function, and
3. an environment record of the free variables of the function.

As we will see in Section 3.3, arrays of functions are represented similarly, by using an array for the environment instead of a single record.

Concretely, we use the following data type for vectorised closures:

```
data (a :-> b) = forall e. PA e =>
   Clo { env  :: e
       , clo_s :: e -> a -> b
       , clo_l :: PArray e -> PArray a -> PArray b }
```

The two fields $\texttt{clo}_s$ and $\texttt{clo}_l$ contain the scalar and lifted version of the function respectively. The field `env` has the existentially quantified type `e`, and is the environment of the closure. We bundle up useful operators on arrays of type `PArray e` into the existential `PA e` type class, so that consumers can process the environment of a deconstructed `Clo`. During vectorisation, all lambdas in the source code are closure converted [1] to this form.

In Figure 1, we see closure conversion in action, where $\mathcal{V}[\![\cdot]\!]$ replaces lambda abstractions in the original program by explicit closures. Consequently, it also replaces function application by closure application, which is defined as:

```
($:) :: (a :-> b) -> a -> b
($:) (Clo env fs fl) arg = fs env arg
```

Closure application extracts the scalar version of the function (`fs`) and applies it to the environment and function argument. The lifted version of the function (`fl`) is produced by the lifting transformation $\mathcal{L}[\![\cdot]\!]\cdot$ from Figure 1. It is used to handle nested parallelism when a vectorised closure happens to be the `f` in `mapP f` — though we need to cover more ground before we can discuss the implementation.

3.3 Arrays of functions

Vectorisation turns functions into explicit closures, so type vectorisation gives us $\mathcal{V}_t[\![\texttt{[:a -> b:]}]\!] = \texttt{PArray (a :-> b)}$. The corresponding `PArray` instance is given below. Arrays of functions are represented as a slightly different form of plain closures, which we call *array closures*:

```
data instance PArray (a :-> b) = forall e. PA e =>
  AClo { aenv  :: PArray  e
       , aclo_s :: e -> a -> b
       , aclo_l :: PArray e -> PArray a -> PArray b }
```

The difference between plain closures and array closures is that with the latter, the environment is array valued. As with plain closures, array closures come with a matching application operator:

```
($:_L) :: PArray (a:->b) -> PArray a -> PArray b
($:_L) (AClo env fs fl) = fl env
```

This lifted function application operator is used to implement application in the lifting transformation $\mathcal{L}[\![\cdot]\!]\cdot$ from Figure 1.

3.4 A simple example

Before we get into any more detail, let us run through the vectorisation of a simple example:

```
inc :: Float -> Float
inc = λx. x + 1
```

Applying the full vectorisation transformation in Figure 1 yields:

```
inc_V :: Float :-> Float
inc_V = Clo () inc_S inc_L

inc_S :: () -> Float -> Float
inc_S = λe x. case e of () -> (+)_V $: x $: 1

inc_L :: PArray () -> PArray Float -> PArray Float
inc_L = λe x. case e of
  ATup_o n -> (+)_V $:_L x $:_L (replicatePA n 1)
```

To aid explanation we have named $\texttt{inc}_S$ and $\texttt{inc}_L$, but otherwise simply applied Figure 1 blindly. Notice the way we have systematically transformed `inc`'s type, replacing `(->)` by `(:->)`. Notice too that this transformation neatly embodies the idea that we need two versions of every top-level function `inc`, a *scalar version* $\texttt{inc}_S$ and a *lifted version* $\texttt{inc}_L$. These two versions paired together form the *fully vectorised version* $\texttt{inc}_V$.

The vectorised code makes use of vectorised addition $(+)_V$, which is provided by a fixed, hand-written library of vectorised primitives:

```
(+)_V :: Float :-> Float :-> Float
(+)_V = Clo () (+)_S (+)_L

(+)_S :: () -> Float -> Float :-> Float
(+)_S = λe x. Clo x addFloat_S addFloat_L

(+)_L :: PArray ()
      -> PArray Float -> PArray (Float :-> Float)
(+)_L = λe xs. AClo xs addFloat_S addFloat_L

addFloat_S :: Float -> Float -> Float
addFloat_S = Prelude.(+)

addFloat_L :: PArray Float -> PArray Float
           -> PArray Float
addFloat_L = zipWithPar2 Prelude.(+)
```

The intermediate functions $(+)_S$ and $(+)_L$ handle partial applications of `(+)`. Finally we reach ground truth: invocations of $\texttt{addFloat}_S$ and $\texttt{addFloat}_L$, implemented by the DPH back-end library. The former is ordinary floating point addition; the latter is defined in terms of `zipWithPar2` from Section 3.1.

This is important! It is only here at the bottom of a pile of nested closures that the old, full vectorisation transformation finally uses the scalar mapping functions. Considering how trivial the original function `inc` was, the result of vectorisation looks grotesquely inefficient. Most of this clutter is introduced to account for the *possibility* of higher order programming, and in many cases, it can be removed by subsequent optimisations.

However, even when GHC's optimiser can remove all the clutter, it still has a real cost: compile time. With vectorisation avoidance, we can completely avoid vectorising `inc` and save all of this overhead.

3.5 Using lifted functions

In Section 3.2.2, we asserted that lifted code is ultimately invoked by `mapP` (and its cousins `zipWithP`, `zipWith3P`, and so on). The code for `mapP` itself is where nested data parallelism is finally transformed to flat data parallelism:

```
mapP_V :: (a :-> b) :-> PArray a :-> PArray b
mapP_V = Clo () mapP_1 mapP_2

mapP_1 :: () -> (a :-> b) -> PArray a :-> PArray b
mapP_1 _ f = Clo f mapP_S mapP_L

mapP_2 :: PArray () -> PArray (a :-> b)
       -> PArray (PArray a :-> PArray b)
mapP_2 _ fs = AClo fs mapP_S mapP_L

mapP_S :: (a :-> b) -> PArray a -> PArray b
mapP_S (Clo env fs fl) xss
  = fl (replicatePA (lengthPA xss) env) xss
```

```
mapP_L :: PArray (a :-> b)
       -> PArray (PArray a) -> PArray (PArray b)
mapP_L (AClo env _ fl) xss
  = unconcatPA xss (fl env (concatPA xss))
      -- xss :: PArray (PArray a)
      -- env :: PArray e
      -- fl  :: PArray e -> PArray a -> PArray b
```

The function $\texttt{mapP}_L$ implements a nested map. It uses a well known observation that a nested map produces the same result as a single map, modulo some shape information:

```
concat . (map (map f)) = (map f) . concat
```

The implementation of $\texttt{mapP}_L$ exploits this fact to flatten nested parallel maps. It eliminates one layer of nesting structure using `concatPA`, applies the simply lifted function $\texttt{f}_L$ to the resulting array, and re-attaches the the nesting structure information using `unconcatPA`.

3.6 Conditionals and case expressions

GHC's Core language includes a case expression with shallow (non-nested) patterns. Conditionals are desugared by using that case expression on a Boolean value. Given the syntactic complexity of case expressions, we have opted for a simplified presentation in Figure 1. We have an explicit conditional `if-then-else` to capture dynamic control flow, together with a one pattern case construct to capture data constructor decomposition by pattern matching. The lifting of conditionals by $\mathcal{L}\,[\![\cdot]\!]\,\cdot$ proceeds as per the discussion in Section 2.4.

4. Transformation Rules

We will now formalise our intuition about where and how to avoid vectorisation. The formalisation consists of three parts:

1. *Labelling:* a static code analysis that identifies maximal sequential subexpressions (Section 4.1).
2. *Encapsulation:* a code transformation that isolates and lambda-lifts the maximal sequential subexpressions (Section 4.2).
3. *Vectorisation:* a slight modification of the vectorisation transformation that uses labelling information to avoid vectorisation (Section 4.3).

The trickiest step is the first. Once the maximal sequential subexpressions have been identified, it is reasonably straightforward to use that information in the vectorisation transform itself.

4.1 Labelling

Labelling performs an initial pass over the program to be vectorised. To decide which parts are maximally sequential, we need labels for each subexpression, as well as the current context. Figure 2 defines $\mathcal{A}_T\,[\![\cdot]\!]$ to label types, and $\mathcal{A}\,[\![\cdot]\!]\,\cdot$ to label expressions. These functions produce four possible labels:

- $\mathbf{p}$ — the labelled expression (or type) *may contain* a parallel subexpression (or a parallel array subtype);
- $\mathbf{s}$ — the labelled expression (or type) *does not contain* any parallel subexpression (or parallel array subtype) and the type of the expression (or the type itself) *is* a member of type class `Scalar`;
- $\mathbf{c}$ — the labelled expression (or type) *does not contain* any parallel subexpression (or parallel array subtype), but the type of the expression (or the type itself) *is not* a member of type class `Scalar`; and
- $\mathbf{e}$ — the labelled expression is an encapsulated lambda abstractions whose body should not be vectorised.

The type labeller maps a type to a label:

$$\mathcal{A}_T\,[\![\cdot]\!] :: \mathit{Type} \to \{\mathbf{s}, \mathbf{p}, \mathbf{c}\}$$

In the present formalisation, we omit polymorphic types as they don't add anything interesting. Our implementation in GHC works fine with polymorphism and all of GHC's type extensions.

The intuition behind $\mathcal{A}_T\,[\![\tau]\!]$ is that it produces $\mathbf{s}$ for types in the class `Scalar`, $\mathbf{p}$ for types containing parallel arrays, and $\mathbf{c}$ in all other cases. It depends on a set P_{ts} (parallel types) that contains all type constructors whose definition either directly includes `[::]` or indirectly includes it via other types in P_{ts}.

Labelling uses a right associative operator $\triangleright$ (combine) that combines labelling information from subexpressions and subtypes. It is biased towards its first argument, except when the second argument is $\mathbf{p}$, in which case $\mathbf{p}$ dominates. We assume $\triangleright$ is overloaded to work on labels and labelled expressions. In the latter case, it ignores the expression and considers the label only.

The expression labeller takes an expression to be labelled, and a set of variables P (parallel variables) which may introduce parallel computations. It produces a *labelled expression*, being a pair of an expression and a label, which we denote by *LExpr*.

$$\mathcal{A}\,[\![\cdot]\!]\,\cdot :: \mathit{Expr} \to \{\mathit{Var}\} \to \mathit{LExpr}$$

We assume P initially includes all imported variables that are bound to potentially parallel computations. Labelling of expressions produces one label, and also rewrites the expression so that each subexpression is labelled as well.

The expression labeller operates as follows. Literals are labelled $\mathbf{s}$. Variables in the set P are labeled $\mathbf{p}$, otherwise the label depends on their type, as is the case with data constructors. For applications, the label depends on their type, unless one subexpression is $\mathbf{p}$. The same holds for conditionals.

If the body of a lambda expression or its argument type are $\mathbf{p}$, then the whole expression will be labelled $\mathbf{p}$, otherwise it is handled slightly differently. The type of a sequential lambda expression is never in `Scalar`, so tagging it $\mathbf{c}$ does not add any information. What we are interested in is the tag of its body (once stripped of all lambda abstractions), as this is the information that we need to decide whether we need to vectorise the expression.

For let-expressions, we add the variable to the set P if the bound expression is labelled $\mathbf{p}$ and the variable's type is not in `Scalar`. If the type *is* in `Scalar`, then our unboxed array representation ensures that demanding the value of this expression in the vectorised program cannot trigger a suspended parallel computation — remembering that GHC core is a lazy language. Note that if the bound expression turns out to be labelled $\mathbf{p}$ then we may need to re-run the analysis to handle recursive lets. First, we assume the expression is not $\mathbf{p}$, so we do not include the bound variable in P. Under this assumption, if we run the analysis and the expression *does* turn out to be $\mathbf{p}$, then we need to run it again with the set $P \cup \{x\}$, as we may have tagged subexpressions incorrectly. In GHC core, there is a distinction between recursive and non-recursive let-bindings, and this extra step is only necessary for recursive bindings. If the bound expression is not tagged $\mathbf{p}$, we proceed as usual.

4.2 Encapsulation

Once we have the labelled tree, we traverse it to find the maximal sequential subexpressions we do not want to vectorise. As per Section 2.3, we avoid vectorisation by first lambda lifting the expression, and then applying the resulting function to all its free variables. This is done by the following encapsulation function, where x_i are the free variables of exp.

$$\begin{array}{l}\mathit{encapsulate\ exp} = \\ \quad ((\lambda x_1.(\ldots \lambda x_k.\mathit{exp}, \mathbf{e}), \ldots), \mathbf{e})(x_1, \mathbf{s}) \ldots (x_k, \mathbf{s}), \mathbf{e})\end{array}$$

Definition of right associative operator $\triangleright$, overloaded to work on labels and labelled expressions

$$\begin{array}{rcl} t \triangleright \mathbf{p} & = & \mathbf{p} \\ t_1 \triangleright t_2 & = & t_1, \text{if } t_2 \neq \mathbf{p} \end{array}$$

$$\mathcal{A}_T \, [\![\cdot]\!] :: \mathit{Type} \to \{\mathbf{p}, \mathbf{c}, \mathbf{s}\}$$

$$\begin{array}{rcll} \mathcal{A}_T \, [\![\tau_1 \to \tau_2]\!] & = & \mathbf{c} \triangleright \mathcal{A}_T \, [\![\tau_1]\!] \triangleright \mathcal{A}_T \, [\![\tau_2]\!] & \\ \mathcal{A}_T \, [\![\texttt{[:}\tau\texttt{:]}]\!] & = & \mathbf{p} & \\ \mathcal{A}_T \, [\![T \; \tau_1 \ldots \tau_n]\!] & = & \mathbf{s} & \text{if } T \; \tau_1 \ldots \tau_n \in \texttt{Scalar}, \text{ where } n \geq 0 \\ & = & \mathbf{p} & \text{if } T \in P_{ts} \\ & = & \mathbf{c} \triangleright \mathcal{A}_T \, [\![\tau_1]\!] \triangleright \cdots \triangleright \mathcal{A}_T \, [\![\tau_n]\!] & \text{otherwise} \end{array}$$

$$\mathcal{A} \, [\![\cdot]\!] \cdot :: \mathit{Expr} \to \{\mathit{Var}\} \to \mathit{LExpr}$$

$$\begin{array}{rcll} \mathcal{A} \, [\![k :: \tau]\!] \; P & = & (k, \mathbf{s}) & k \text{ is a literal} \\ \mathcal{A} \, [\![x :: \tau]\!] \; P & = & (x, \mathbf{p}) \quad \text{if } x \in P & x \text{ is a variable} \\ & & (x, \mathcal{A}_T \, [\![\tau]\!]) \quad \text{otherwise} & \\ \mathcal{A} \, [\![C :: \tau]\!] \; P & = & (C, \mathcal{A}_T \, [\![\tau]\!]) & C \text{ is data constructor} \\ \mathcal{A} \, [\![e_1 \; e_2 :: \tau]\!] \; P & = & ((\mathcal{A} \, [\![e_1]\!] \; P)(\mathcal{A} \, [\![e_2]\!] \; P), & \\ & & \mathcal{A}_T \, [\![\tau]\!] \triangleright \mathcal{A} \, [\![e_1]\!] \; P \triangleright \mathcal{A} \, [\![e_2]\!] \; P) & \\ \mathcal{A} \, [\![(\lambda x :: \tau . e)]\!] \; P & = & (\lambda x . (\mathcal{A} \, [\![e]\!] \; P), \;\; \mathcal{A} \, [\![e]\!] \; P \triangleright \mathcal{A}_T \, [\![\tau]\!]) & \\ \mathcal{A} \, [\![(\texttt{if} \; e_1 \texttt{then} \; e_2 \; \texttt{else} \; e_3) :: \tau]\!] \; P & = & (\texttt{if} \; \mathcal{A} \, [\![e_1]\!] \; P \; \texttt{then} \; \mathcal{A} \, [\![e_2]\!] \; P \; \texttt{else} \; \mathcal{A} \, [\![e_3]\!] \; P, & \\ & & \mathcal{A}_T \, [\![\tau]\!] \triangleright \mathcal{A} \, [\![e_1]\!] \; P \triangleright \mathcal{A} \, [\![e_2]\!] \; P \triangleright \mathcal{A} \, [\![e_3]\!] \; P) & \\ \mathcal{A} \, [\![(\texttt{let} \; x :: \tau_1 \texttt{=} \; e_1 \; \texttt{in} \; e_2) :: \tau]\!] \; P & = & (\texttt{let} \; x \texttt{=} \; \mathcal{A} \, [\![e_1]\!] \; (P \cup \{x\}) \; \texttt{in} \; \mathcal{A} \, [\![e_2]\!] \; (P \cup \{x\}), \mathbf{p}) & \text{if } \mathcal{A} \, [\![e_1]\!] \; P = (e_1', \mathbf{p}) \\ & & & \text{and } \tau_1 \notin \texttt{Scalar} \\ & = & (\texttt{let} \; x \texttt{=} \; \mathcal{A} \, [\![e_1]\!] \; P \; \texttt{in} \; \mathcal{A} \, [\![\, e_2]\!] \; P, & \text{otherwise} \\ & & \mathcal{A}_T \, [\![\tau]\!] \triangleright \mathcal{A} \, [\![e_1]\!] \; P \triangleright \mathcal{A} \, [\![e_2]\!] \; P) & \\ \mathcal{A} \, [\![(\texttt{case} \; e_1 \; \texttt{of} \; C \; \overline{x_i :: \tau_i} \; \to \; e_2) :: \tau]\!] \; P & = & (\texttt{case} \; \mathcal{A} \, [\![e_1]\!] \; P \; \texttt{of} \; C \; \overline{x_i :: \tau_i} \; \to \; \mathcal{A} \, [\![e_2]\!] \; (P \cup \overline{x_i}), & \text{if } \mathcal{A} \, [\![e_1]\!] \; P = (e_1', \mathbf{p}) \\ & & \mathbf{p}) & \text{and } \overline{\tau_i} \notin \texttt{Scalar} \\ & = & (\texttt{case} \; \mathcal{A} \, [\![e_1]\!] \; P \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; \mathcal{A} \, [\![e_2]\!] \; P, & \text{otherwise} \\ & & \mathcal{A}_T \, [\![\tau]\!] \triangleright \mathcal{A} \, [\![e_1]\!] \; P \triangleright \mathcal{A} \, [\![e_2]\!] \; P) & \end{array}$$

Figure 2. Static code analysis determining the subexpressions that need to be vectorised

$$\mathit{fvs}(e) \quad = \quad \forall v :: \tau \in \mathit{FreeVars}(e) . \mathcal{A}_T \, [\![\tau]\!] = \mathbf{s}$$

$$\mathcal{E} \, [\![\cdot]\!] :: \mathit{LExpr} \to \mathit{LExpr}$$

$$\begin{array}{rcll} \mathcal{E} \, [\![(k, l)]\!] & = & (k, l) & \\ \mathcal{E} \, [\![(x, l)]\!] & = & (x, l) & \\ \mathcal{E} \, [\![(C, l)]\!] & = & (C, l) & \\ \mathcal{E} \, [\![(e_1 \; e_2, \mathbf{s})]\!] & = & \mathit{encapsulate}(e_1 \; e_2), & \text{if } \mathit{fvs}(e_1 \; e_2) \\ \mathcal{E} \, [\![(e_1 \; e_2, l)]\!] & = & (\mathcal{E} \, [\![e_1]\!] \; \mathcal{E} \, [\![e_2]\!] \, , l) & \\ \mathcal{E} \, [\![(\lambda \overline{x_i} . e, \mathbf{s})]\!] & = & \mathit{encapsulate}(\lambda \overline{x_i} . e) & \text{if } \mathit{fvs}(e) \\ \mathcal{E} \, [\![(\lambda \overline{x_i} . e, l)]\!] & = & (\lambda \overline{x_i} . \mathcal{E} \, [\![e]\!]), l) & \\ \mathcal{E} \, [\![(\texttt{if} \; e_1 \texttt{then} \; e_2 \texttt{else} \; e_3, \mathbf{s})]\!] & = & \mathit{encapsulate}(\texttt{if} \; e_1 \texttt{then} \; e_2 \texttt{else} \; e_3) & \text{if } \mathit{fvs}(e_1 \; e_2) \\ \mathcal{E} \, [\![(\texttt{if} \; e_1 \texttt{then} \; e_2 \texttt{else} \; e_3, l)]\!] & = & (\texttt{if} \; \mathcal{E} \, [\![e_1]\!] \; \texttt{then} \; \mathcal{E} \, [\![e_2]\!] \; \texttt{else} \; \mathcal{E} \, [\![e_3]\!] \, , l) & \\ \mathcal{E} \, [\![(\texttt{let} \; x \texttt{=} \; e_1 \texttt{in} \; e_2, \mathbf{s})]\!] & = & \mathit{encapsulate}(\texttt{let} \; x \texttt{=} \; e_1 \texttt{in} \; e_2) & \text{if } \mathit{fvs}(\texttt{let} \; x \texttt{=} \; e_1 \texttt{in} \; e_2) \\ \mathcal{E} \, [\![(\texttt{let} \; x \texttt{=} \; e_1 \texttt{in} \; e_2, l)]\!] & = & (\texttt{let} \; x \texttt{=} \; \mathcal{E} \, [\![e_1]\!] \; \texttt{in} \; \mathcal{E} \, [\![e_2]\!]), l) & \\ \mathcal{E} \, [\![(\texttt{case} \; e_1 \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; e_2, \mathbf{s})]\!] & = & \mathit{encapsulate}(\texttt{case} \; e_1 \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; e_2) & \text{if } \mathit{fvs}(\texttt{case} \; e_1 \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; e_2) \\ \mathcal{E} \, [\![(\texttt{case} \; e_1 \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; e_2, l)]\!] & = & (\texttt{case} \; \mathcal{E} \, [\![e_1]\!] \; \texttt{of} \; C \; x_1 \ldots x_k \; \to \; \mathcal{E} \, [\![e_2]\!] \, , l) & \end{array}$$

Figure 3. Encapsulation of maximal sequential subexpressions

As usual top-level bindings are not included in the free variables. Note how *encapsulate* labels all new lambda expressions as well as the encapsulated expression *exp* with **e**.

Figure 3 defines $\mathcal{E} \, [\![\cdot]\!] :: \mathit{LExpr} \to \mathit{LExpr}$, which, given a labelled expression, encapsulates all subexpressions that (1) perform some work (are larger than an individual literal, variable, or data constructor), (2) are marked as **s**, and (3) whose free variables are also marked as **s**. These are our sequential subexpressions. Since $\mathcal{E} \, [\![\cdot]\!]$ defines a top down transformation, it encapsulates the first subexpression it encounters that meets the requirements. Hence, it encapsulates maximal sequential subexpressions.

$$\begin{aligned}\mathcal{VE}\,[\![(\lambda x_1 \ldots x_k.e, \mathbf{e})]\!] \quad &= \texttt{Clo}\ \{\texttt{env} = () \\ &\quad , \texttt{clo}_s = \lambda env\, x_1 \ldots x_k.\, e \\ &\quad , \texttt{clo}_l = \lambda env\, x_1 \ldots x_k.\, \texttt{case}\, env\, \texttt{of ATup}\ n\ \rightarrow (\texttt{zipWithPar}k(\lambda x_1 \ldots x_k.e))\}\end{aligned}$$

Figure 4. Modified vectorisation rules for lambda abstractions

4.3 Modifying vectorisation

After labelling an expression with $\mathcal{A}\,[\![\cdot]\!]\,\cdot$ and encapsulating all maximal sequential subexpressions with $\mathcal{E}\,[\![\cdot]\!]$, we only need a slight addition to the rules of vectorisation from Figure 1 to avoid vectorisation. Firstly, the vectorisation and lifting transform, $\mathcal{V}\,[\![\cdot]\!]$ and $\mathcal{L}\,[\![\cdot]\!]\,\cdot$ need to be adapted to process labelled expressions. That adaptation is trivial as the existing rules all just ignore the label and operate on the expression as usual.

Secondly, we add one additional rule for vectorising lambda expressions that is shown in Figure 4. If a lambda expression is labelled $\mathbf{e}$, we know without needing any further checks that it is safe to lift it by simply using a scalar mapping function, `zipWithPar`k,[2] instead of the full lifting transformation $\mathcal{L}\,[\![\cdot]\!]\,\cdot$. In this case, we do not need to check for free variables, as lambda abstractions marked $\mathbf{e}$ are closed.

5. Performance

In this section, we provide empirical evidence that vectorisation avoidance yields a net improvement in both compile time and runtime, when compared to full vectorisation in combination with array fusion. Our measurements are restricted to the implementation of vectorisation in GHC (the only existing implementation of higher-order flattening known to us) and to multicore CPUs without considering SIMD vector instructions. The results may be different for vector hardware, such as SIMD instructions or GPUs, but currently we have no DPH backend to investigate these architectures.

We chose those relatively simple benchmark programs, as they expose the effect of vectorisation avoidance well. The impact on the performance of the more complex DPH benchmarks, like Barnes-Hut, is as expected given the numbers below, taking into account the ratio between code which is affected by the optimisation and code which is not.

All benchmarks have been executed on a quadcore 3.4 GHz Intel Core i7 running OS X with the current development version of GHC (version 7.5). With the exception of the benchmarks concerned with load balancing, vectorisation avoidance ought to improve program performance independent of whether the program is executed sequentially or in parallel on multiple cores. We include parallel runtimes throughout, to support that claim, but only for up to four cores as there are no interesting scalability questions.

5.1 Additional lambdas

The encapsulation of maximal sequential subexpressions in lambda abstractions by vectorisation avoidance arguably complicates the program. Does this introduce overheads? In our benchmarks it didn't appear to.

This is not surprising. These lambda abstractions are processed twice by vectorisation: to produce a scalar and a lifted version of the abstraction when creating a vectorised closure (c.f., Section 3.2.2). In the case of the lifted code, the lifting transformation $\mathcal{L}\,[\![\cdot]\!]\,\cdot$ introduces additional abstractions anyway.

In the scalar code, the situation is less obvious. Encapsulation introduces an expression of the form

```
(λx₁ ... xₙ. expr) x₁ ... xₙ
```

This turns into `Clo{...} $: x₁ $: ··· $: xₙ`, which GHC's simplifier reliably simplified by inlining `($:)`, case simplification, and beta reduction in our experiments.

5.2 Simple arithmetic operations

Our first two benchmark programs investigate the case where stream fusion [12, 13] is able to completely fuse a chain of array traversals. Specifically, we measure the runtime of the following two functions when used in parallel — that is, we measure `zipWith pythagoras xs ys` and `zipWith distance xs ys`, where `xs` and `ys` are vectors containing 10^8 `Double` values:

```
pythagoras x y
  = sqrt (x * x + y * y + 2 * x * y)

distance (xo, yo) ((x1, y1), (x2, y2))
  = (x1 - xo) * (y2 - yo) - (y1 - yo) * (x2 - xo)
```

In our current implementation, the `Scalar` class does not yet have an instance for pairs. Hence, vectorisation avoidance cannot use `zipWith_scalar` on the entire body of `distance`. Instead it encapsulates the body of the innermost case expression (performing pattern matching on the pairs); so, the code has the following structure after encapsulation:

```
distance xy0 xy =
 case xy0 of (x0, y0) ->
  case xy of (xy1, xy2) ->
   case xy1 of (x1, y1) ->
    case xy2 of (x2, y2) ->
     (λ x0 y0 x1 y1 x2 y2.
       (x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0))
      x0 y0 x1 y1 x2 y2
```

Given that the additional array traversals introduced by vectorisation of `pythagoras` and `distance` can be completely eliminated by array fusion, we might expect that vectorisation avoidance does not provide any additional benefit. However, the graph displayed in Figure 5 shows that vectorisation avoidance does improve performance slightly. This is because fusion leads to slightly more complex loops than vectorisation avoidance.

According to the graph, the two benchmarks do not scale particularly well. We believe that this is because the code is memory bound — i.e., the full floating-point performance of the processor is not exploited because the processor has to wait for the memory-subsystem to fetch the operands.

5.3 Fusion and vectorisation avoidance together

In the previous benchmark, we measured the performance of `zipWithP pythagoras xs ys` by itself (and the same with `distance`). Next, we study that same code sandwiched between a producer (enumerating the arrays consumed by the `zipWithP`) and a consumer (summing up the result array with `sumP`); so, we have got

```
sumP (zipWithP pythagoras
              (enumFromToP 1 (10^8))
              (enumFromToP 1 (10^8)))
```

Ideally, we would like the whole pipeline to fuse into a single loop that computes the final sum without creating any intermediate arrays. Looking at the graph in Figure 6 that does happen for

[2] We assume `zipWithPar1 = mapPar`.

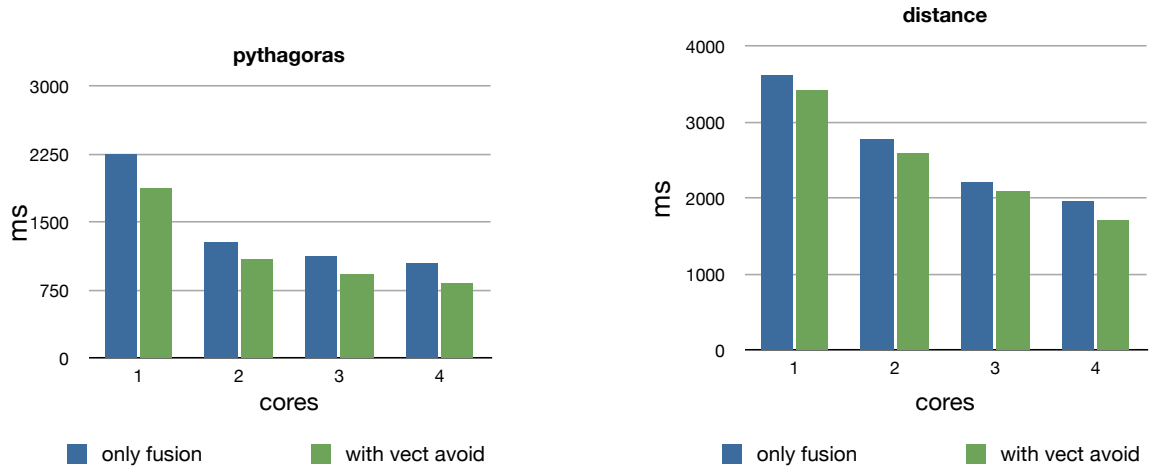

Figure 5. Runtimes of the `pythagoras` and `distance` functions on vectors of 10^8 floating-point numbers

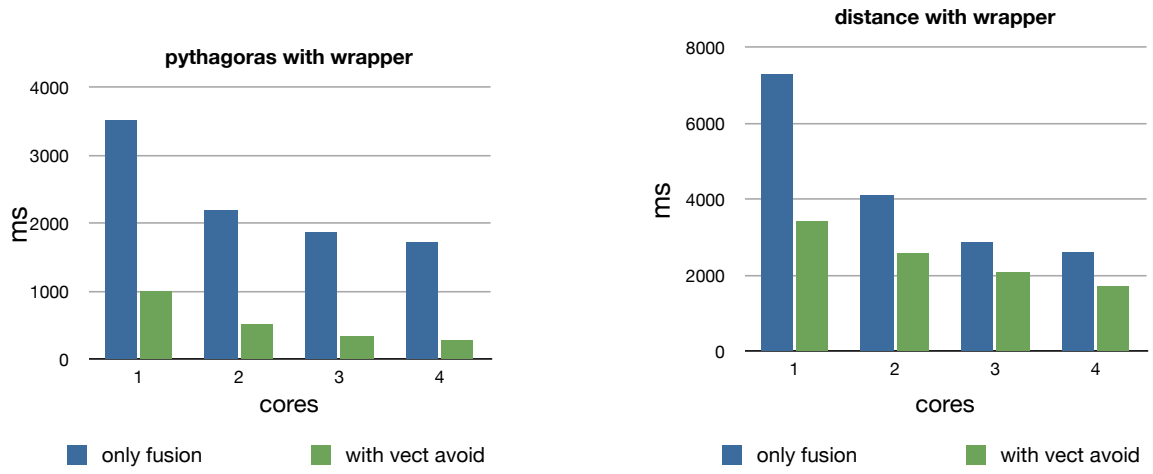

Figure 6. Runtimes of the `pythagoras` and `distance` functions *with wrappers* on vectors of 10^8 floating-point numbers

the `pythagoras` benchmark with vectorisation avoidance enabled. With fusion alone, performance is worse by more than a factor of 3. Array fusion by itself did not manage to eliminate the entire pipeline, whereas the combination of array fusion with vectorisation avoidance did so successfully, leading to a dramatic performance improvement. Once the pipeline fuses completely, the code also scales better — since there are no more intermediate structures, the memory-access bottleneck vanishes.

Why is fusion by itself not successful at removing all intermediate structures? In the lifted code, the arrays produced by the enumerations are shared, because the arguments to `pythagoras` are used multiple times in the body. This hampers inlining, and hence, stream fusion. With vectorisation avoidance, all sharing is in the code that is not vectorised, so this problem does not arise.

In the case of `distance`, vectorisation avoidance is also a clear improvement. However, it is less dramatic as the remaining pattern matching of the argument pairs (discussed in the previous subsection) prevents fusion of the entire pipeline.

5.4 Conditionals

Fully vectorising conditionals is expensive due to the `splitPA` and `combinePA` operations. On the other hand, vectorised conditionals balance load well, whereas if vectorisation is avoided, we might suffer from load imbalance. To assess the impact of vectorisation avoidance on conditionals, we measured `mapP simpleCond xs` and `mapP sinCond xs`, for arrays with 10^8 elements, where

```
simpleCond x = if (x `mod` 2 == 0)
               then 3 * x
               else 2 * x

sinCond x n = if (x < n / 2)
              then  x
              else sin (sin (sin (2 * x)))
```

We chose the input array to contain the increasing sequence of 1 to the size of the array. Hence, for `simpleCond`, we have no load imbalance with vectorisation avoidance, whereas `sinCond` has a severe load imbalance as we execute the then-branch on the first half of elements and the else-branch on the other half.

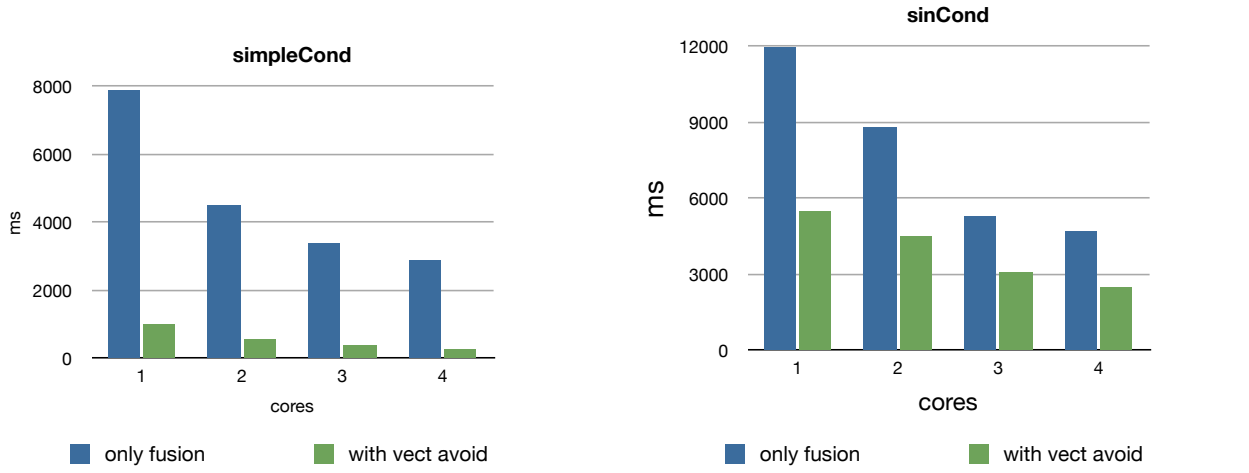

Figure 7. Runtimes of the `simpleCond` and `sinCond` functions on vectors of 10^8 `Int`s and `Double`s, respectively

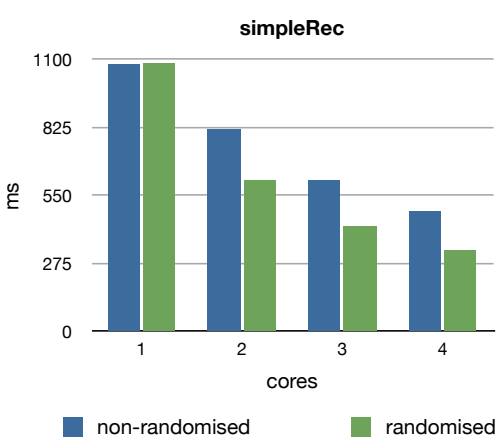

Figure 8. Runtimes of `simpleRec` on vectors of 10^5 `Double`s with and without randomisation

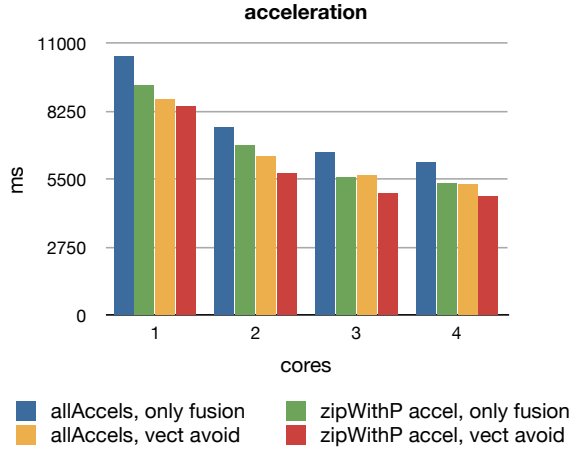

Figure 9. Runtimes of `allAccels` and `zipWithP accels` computing 10^8 interactions

As expected, the graph in Figure 7 shows that vectorisation avoidance for conditionals is a big improvement when there is no load imbalance. However, even with a severe load imbalance, vectorisation avoidance is still an advantage for small numbers of cores. Nevertheless, with vectorisation avoidance, scalability suffers in case of load imbalance; hence, it would be worthwhile to enable the programmer to determine the behaviour of the vectoriser with a pragma.

5.5 Recursive Functions

The question of load imbalance becomes even more pressing when the work complexity of a function depends on the array element it is applied to, as in

```
simpleRec x = if (x < 5)
              then x
              else simpleRec (x - 5)
```

Interestingly, in the case of a tail recursive function, the benefit of vectorisation avoidance is even greater, because vectorisation prevents the code generator from compiling the recursion into a simple loop. For the admittedly extreme case of `simpleRec`, where there is no work in the recursive steps, the fully vectorised version is two orders of magnitude slower than that using vectorisation avoidance.

Unfortunately, when mapping `simpleRec` over an array containing the increasing sequence of 1 to the size of the array, load imbalance is also significant. Vectorisation of `simpleRec` provides load balancing, but, in this example, with a prohibitively expensive constant factor. An alternative to full vectorisation in such cases is to apply vectorisation avoidance, but to randomise the input vector. The graph in Figure 8 suggests that this is a worthwhile strategy. Currently, a programmer needs to do it explicitly, but we plan to investigate automatic randomisation.

5.6 Calculating accelerations

Figure 9 displays the running times for computing the acceleration of 10^8 mass point interactions (the example we used in Section 2) with and without vectorisation avoidance. It does so in two ways. Firstly, by using `allAccels` on 10^4 mass points (it implements a quadratic algorithm); and secondly, by directly using `zipWithP accel` on two arrays of 10^8 mass points. The main difference between these two computations is that `allAccels` also

computes $2 * 10^4$ parallel sums. Hence, it is not surprising that `allAccels` is slower than `zipWithP accel` across the board.

However, it is interesting to see that the gain due to vectorisation avoidance is higher in the case of `allAccels`, where it is 11% on a single core, than for `zipWithP accel`, where it is 3%. The reason is as for `pythagoras` and `distance` with wrappers. Fusion of the acceleration computation with `sumP` is more effective with vectorisation avoidance.

Note also that, as previously mentioned, the `Scalar` class in our current implementation does not yet include pairs. Hence, the vectorisation of `accel` cannot yet be entirely avoided (c.f., the explanation for `distance` in Section 5.2). We expect the gain to be even more pronounced once support for tuples is added.

5.7 Compilation time

We argued that, apart from producing faster code, vectorisation avoidance also produces simpler code which requires fewer subsequent optimisations. This, in turn, should result in shorted compilation times. For the examples presented in this section, overall compilation was about 25% faster when vectorisation avoidance was enabled.

6. Related Work

We are not aware of any other work that attempts to selectively avoid vectorisation in higher-order programs. However, in a recent port of Blelloch's original NESL system to GPUs [2], the code for NESL's virtual vector machine, called VCODE, is analysed to fuse sequences of lifted arithmetic operations. This is selectively *undoing* some vectorisation, albeit in a first-order setting. Much like our old stream fusion system, it cannot undo the vectorisation of conditionals or recursive functions.

Manticore is an implementation of nested data parallelism which uses a technique called *hybrid flattening* leaving the code mostly intact, and which relies on dynamic methods, such as work stealing and lazy tree splitting [3, 18]. Similarly, Blelloch et al. [4, 20] investigated alternatives to flattening based on multi-threading. Based on the scheduling strategy, they were able to establish asymptotic bounds on time and space for various forms of parallelism, including nested data parallelism.

Overall, there are two general approaches to implementing nested data parallelism. Approaches based on multi-threading naturally fit the execution model of MIMD machines, such as multicore processors. However, they need to perform dynamic load balancing (e.g., by using work stealing) and to agglomerate operations on successive array elements (to get efficient loops). The alternative is vectorisation, which produces pure SIMD programs. This approach must make efforts to increase locality (e.g., by array fusion). It seems that a hybrid approach may work best, and the question is whether to start from the MIMD or SIMD end. We chose to start from SIMD and relax that using vectorisation avoidance, as discussed here, complemented by a more flexible array representation as discussed in a companion paper [16].

Acknowledgements. This work was supported in part by the Australian Research Council under grant number LP0989507.

References

[1] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1989.

[2] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *ICFP'12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2012. Forthcoming.

[3] L. Bergstrom, J. Reppy, M. Rainey, A. Shaw, and M. Fluet. Lazy tree splitting. In *ICFP'10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2010.

[4] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the Association for Computing Machinery*, 46(2), 1999.

[5] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8, 1990.

[6] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In *ICFP'00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2000.

[7] M. M. T. Chakravarty and G. Keller. Functional array fusion. In *ICFP'01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2001.

[8] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ICFP'05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2005.

[9] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Sysposium on Principles of Programming Languages*. ACM Press, 2005.

[10] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[11] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial vectorisation of Haskell programs. In *DAMP 2008: Workshop on Declarative Aspects of Multicore Programming*, 2008.

[12] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP 2007: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2007.

[13] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting haskell strings. In *PADL 2007: Practical Aspects of Declarative Languages 8th International Symposium*. Springer-Verlag, Jan. 2007.

[14] G. Keller and M. M. T. Chakravarty. Flattening trees. In *Euro-Par'98, Parallel Processing*, number 1470 in LNCS. Springer-Verlag, 1998.

[15] R. Leshchinskiy, M. M. T. Chakravarty, and G. Keller. Higher order flattening. In *PAPP 2006: Third International Workshop on Practical Aspects of High-level Parallel Programming*, number 3992 in LNCS. Springer-Verlag, 2006.

[16] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher order vectorisation. In *ICFP'12: Proceedings of the ACM SIGPLAN International Conference on Functional Programming(to appear)*. ACM Press, 2012.

[17] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS 2008: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, LIPIcs. Schloss Dagstuhl, 2008.

[18] A. Shaw. *Implementation Techniques For Nested-data-parallel Languages*. Phd thesis, Department Of Computer Science, The University Of Chicago, 2011.

[19] B. So, A. Ghuloum, and Y. Wu. Optimizing data parallel operations on many-core platforms. In *STMCS'06: First Workshop on Software Tools for Multi-Core Systems*, 2006.

[20] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. *J. Funct. Program.*, 20(5-6), 2010. ISSN 0956-7968.

[21] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI'07: ACM SIGPLAN International Workshop on Types in Language Design and Implementation*. ACM, 2007.