Foundations of Software Technology and Theoretical Computer Science (Bangalore) 2008.
Editors: R. Hariharan, M. Mukund, V. Vinay; pp 383-414

Harnessing the Multicores: Nested Data Parallelism in Haskell

Simon Peyton Jones[1], **Roman Leshchinskiy**[2], **Gabriele Keller**[2], **Manuel M. T. Chakravarty**[2]

[1]Microsoft Research Ltd, Cambridge, England, simonpj@microsoft.com

[2]Programming Languages and Systems, School of Computer Science and Engineering, University of New South Wales, {rl,keller,chak}@cse.unsw.edu.au

ABSTRACT. If you want to program a parallel computer, a purely functional language like Haskell is a promising starting point. Since the language is pure, it is by-default safe for parallel evaluation, whereas imperative languages are by-default unsafe. But that doesn't make it easy! Indeed it has proved quite difficult to get robust, scalable performance increases through parallel functional programming, especially as the number of processors increases.
A particularly promising and well-studied approach to employing large numbers of processors is data parallelism. Blelloch's pioneering work on NESL showed that it was possible to combine a rather flexible programming model (nested data parallelism) with a fast, scalable execution model (flat data parallelism). In this paper we describe Data Parallel Haskell, which embodies nested data parallelism in a modern, general-purpose language, implemented in a state-of-the-art compiler, GHC. We focus particularly on the vectorisation transformation, which transforms nested to flat data parallelism.

1 Introduction

Computers are no longer getting faster; instead, we will be offered computers containing more and more CPUs, each of which is no faster than the previous generation. As the number of CPUs increases, it becomes more and more difficult for a programmer to deal with the interactions of large numbers of threads. Moreover, the physical limitations of bus bandwidth will mean that memory access times will be increasingly non-uniform (even if the address space is shared), and locality of reference will be increasingly important.

In the world of massively-parallel computing with strong locality requirements there is already a well-established, demonstrably successful brand leader, namely *data parallelism*. In a data-parallel computation one performs the *same* computation on a large collection of *differing* data values. Well-known examples of data-parallel programming environments are High Performance Fortran (HPF) [For97], the collective operations of the Message Passing Interface (MPI) [GHLL+98], NVIDIA's Compute Unified Device Architecture (CUDA) API for graphics processors [NVI07], and Google's map/reduce framework [DG04].

All these systems support only *flat* data parallelism, in which the computation that is performed on each data element must itself be (a) sequential and (b) of a similar execution time to the computation on the other data elements. In practice, this severely limits the applications of data-parallel computing, especially for sparse or irregular problems [PCS99].

FSTTCS 2008
IARCS Annual Conference on
Foundations of Software Technology and Theoretical Computer Science
http://drops.dagstuhl.de/opus/volltexte/2008/1769

```
(!:)        :: [:a:] -> Int -> a
sliceP      :: [:a:] -> (Int,Int) -> [:a:]
replicateP  :: Int -> a -> [:a:]
mapP        :: (a->b) -> [:a:] -> [:b:]
zipP        :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP    :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP     :: (a->Bool) -> [:a:] -> [:a:]

concatP     :: [:[:a:]:] -> [:a:]
concatMapP  :: (a -> [:b:]) -> [:a:] -> [:b:]
unconcatP   :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP  :: [:[:a:]:] -> [:[:a:]:]
expandP     :: [:[:a:]:] -> [:b:] -> [:b:]

combineP    :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
splitP      :: [:Bool:] -> [:a:] -> ([:a:], [:a:])
```

Figure 1: Type signatures for parallel array operations

Thus motivated, Blelloch and Sabot developed the idea of *nested* data parallelism in the early 90's, and embodied it in their language NESL [BS90].

NESL was a seminal breakthrough but, fifteen years later it remains largely un-exploited. Our goal is to adopt the key insights of NESL, embody them in a modern, widely-used functional programming language, namely Haskell, and implement them in a state-of-the-art Haskell compiler (GHC). The resulting system, Data Parallel Haskell, will make nested data parallelism available to real users.

Doing so is not straightforward. NESL a first-order language, has very few data types, was focused entirely on nested data parallelism, and its implementation is an interpreter. Haskell is a higher-order language with an extremely rich type system; it already includes several other sorts of parallel execution; and its implementation is a compiler.

This paper makes two main contributions:

- We give a tutorial, programmer's-eye view of what programming in Data Parallel Haskell is like. Rather than a series of tiny examples, we give a serious application that is very hard to fully parallelise in a flat data-parallel setting, namely the Barnes-Hut algorithm for N-body simulation.
- We give a detailed tutorial overview of the key vectorisation transformation. There are two major innovations over NESL: one is the non-parametric representation of arrays (Section 4) and one is the treatment of first-class functional values (Section 5).

All the technical innovations in this paper have appeared, piecemeal, in our earlier publications. Our hope, however, is that this paper draws together a somewhat-complex set of technical strands into a comprehensible whole.

2 The programmer's view on DPH

We begin by describing Data Parallel Haskell (DPH) purely from the point of view of the programmer, illustrating the description with a non-trivial example, the Barnes-Hut algorithm [BH86]. GHC supports other forms of concurrency besides data parallelism, but we focus here exclusively on the latter. Singh [SJ08] gives a tutorial covering a broader scope, including semi-implicit parallelism (`par`), explicit threads, transactional memory, as well as Data Parallel Haskell.

DPH is simply Haskell with the following extra features:

- A type of *parallel arrays*, denoted `[:e:]` for arrays of type `e`. These arrays are indexed by values of type `Int`. From a semantic point of view an array `[:a:]` is very similar to a list `[a]` – the difference is in the execution pragmatics. An array can contain elements of any type, including arrays and functions.
- A large number of *parallel operations* that operate collectively on entire arrays. As far as possible, these operations have the same names as Haskell's standard list functions, but with the suffix `P` added—i.e., `mapP`, `filterP`, `unzipP`, and so forth. Figure 1 lists the operations that we will use in this paper.
- Syntactic sugar, called *parallel array comprehensions*, which are similar to list comprehensions but operate on parallel arrays.

In addition to the parallel evaluation semantics, lists and parallel arrays also differ with respect to strictness: more precisely, demand for *any* element of a parallel array results in the evaluation of *all* elements.

2.1 *N*-Body Barnes-Hut Simulation Algorithm

We will demonstrate the use of DPH features using the Barnes-Hut n-body simulation algorithm as an example. We discuss the algorithm in some detail because it is a particularly striking example of the power of nested data parallelism, and of the utility of user-defined data types in data-parallel programs. We will, for the sake of clarity, restrict ourselves to two dimensions and neglect complications such as bodies that are very close to each other.

An n-body simulation computes the motion under gravitational forces of n bodies, or *particles*. A naive solution is to compute the force between every pair of particles which requires n^2 calculations in each time step. The Barnes-Hut algorithm reduces the work complexity to the order of $n \log n$ interactions by grouping together particles which are close to each other and calculating the centre of gravity, or *centroid* of the cluster. The centroids are then used to approximate the effect the particles have on other particles which are sufficiently far away. The stricter we are in determining what exactly constitutes "sufficiently far away", the more precise the final result is, and the algorithm can be parametrised accordingly.

The first phase of the algorithm determines the hierarchical grouping of the particles, computes the centroids of the clusters, and stores the result in a tree structure. To be more precise, the area is split into four subareas of equal size, the particles are grouped according to the subarea they are located in. We repeat this step for each subarea, and terminate if an area contains either none or only a single particle. Figure 2 illustrates the tree construction process for particles $p_1 \ldots p_9$. In the first iteration, the particles are split into four groups,

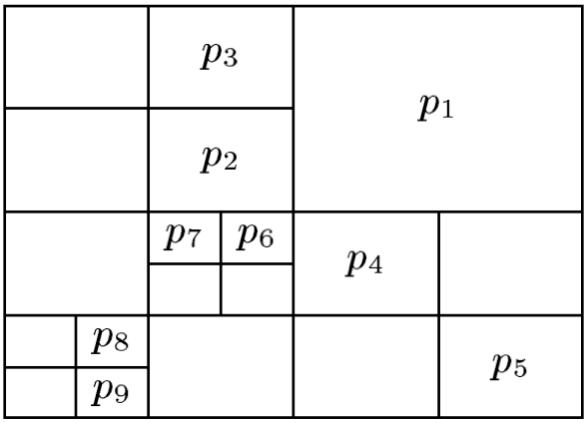

Figure 2: Subdivision of area

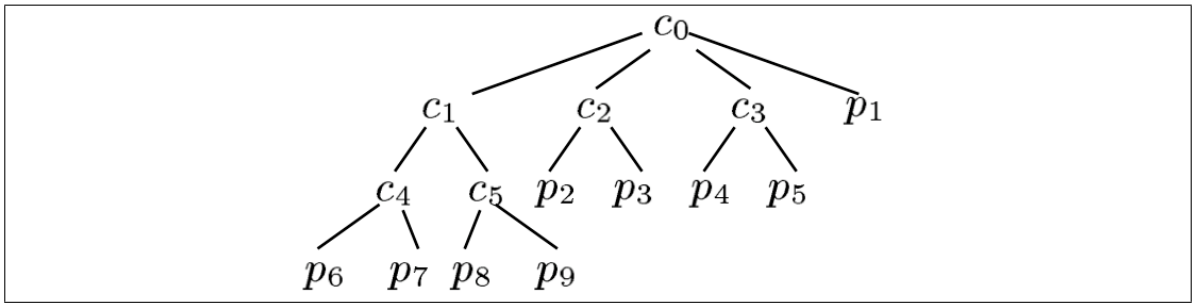

Figure 3: Rosetree

depending on which quadrant they are located in. The upper right quadrant already contains only a single particle, so it isn't divided up any further. Both the upper left and the lower right quadrant require only one more iteration, the lower left two iterations.

Figure 3 shows the resulting hierarchical tree structure: the root node contains the centroid of all particles and four subtrees (since all subareas contain at least one particle). Each of the subtrees contains the centroid of the corresponding subarea – which is the particle itself in case of a singular particle.

The second phase of the algorithm now calculates the forces that affect each particle p, by traversing the tree from the root downwards: for every subtree, if the particle p is sufficiently far away from the centroid stored in the root of that subtree, the force on p is calculated using this centroid without looking at the rest of the tree. Otherwise, we add up the forces on p from the subtrees of the current root – and so on recursively.

2.2 Encoding Barnes-Hut in DPH

Since the only way to express parallelism in DPH is to apply collective operations to parallel arrays, we need to store all data that we want to process in parallel in such an array. For instance, the function `oneStep`, which computes one step in the simulation, takes a parallel array of particles as arguments, and returns an array of the same length, with the position and velocity of each particle adjusted according to the gravitational forces:

```
-- Compute one step of the n-body simulation
oneStep :: [:Particle:] -> [:Particle:]
oneStep particles = moveParticles particles forces
  where
    tree   = buildTree initialArea particles
    forces = calcForces (lengthOf initialArea) tree particles

buildTree     :: Area -> [:Particle:] -> Tree
calcForces    :: Float -> Tree -> [:Particle:] -> [:Force:]
moveParticles :: [:Particle:] -> [:Force:] -> [:Particle:]
lengthOf      :: Area -> Float
```

The function `oneStep`, as discussed before, is comprised of three data parallel phases. First, `buildTree` decomposes the particles into sub-areas, returning the resulting `Tree`. This tree is then used by `calcForces` to compute the forces on the particles, returning a new array of forces with one element for each particle. Finally, `moveParticles` uses these forces to adjust the positions and velocities the particles.

The data types involved in the computation are defined exactly as the would be in regular Haskell. For example, a `Particle` is a record of its mass, its location, and its velocity:

```
type Vector   = (Float, Float)
type Area     = (Vector, Vector)
type Force    = Vector
type Velocity = Vector
type Location = Vector

data Particle = Particle { mass     :: Float
                         , location :: Location
                         , velocity :: Velocity}
```

Some functions are conveniently defined using the parallel array counterparts of ordinary list processing functions (see Figure 1). For example, we can define `moveParticles` like this:

```
moveParticles :: [:Particle:] -> [:Force:] -> [:Particle:]
moveParticles ps fs = zipWithP moveParticle ps fs

moveParticle :: Particle -> Force -> Particle
moveParticle (Particle { mass     = m
                       , location = loc
                       , velocity = vel })
             force
  = Particle { mass     = m
             , location = loc + vel   * timeStep
             , velocity = vel + accel * timeStep }
  where
    accel = force / m
```

Now we turn our attention to the `Tree` data type and its construction. When building and traversing a `Tree`, we want to process its sub-trees in parallel, and so we must use a parallel array for the children:

```
data Tree = Node Mass Location [:Tree:]
            -- Rose tree for spatial decomposition
```

This time, unlike the flat array of particles (which may be very long), the array of sub-trees has at most four elements at any level (recall that we are working with only 2 dimensions). To build a tree, we perform recursive descent over the area:

```
-- Perform spatial decomposition and build the tree
buildTree :: Area -> [:Particle:] -> Tree
buildTree area [: p :]   = Node (mass p) (location p) [::]
buildTree area particles = Node m l subtrees
  where
    (m,l)    = calcCentroid subtrees
    subtrees = [: buildTree a ps
                | a <- splitArea area
                , let ps = [:p | p <- particles, inArea a p:]
                , lengthP ps > 0 :]


inArea :: Area -> Particle -> Bool
inArea ((lx,ly),(hx,hy)) (Particle { location = (x,y) })
  = lx <= x && x <= hx && ly <= y && y <= hy

splitArea    :: Area -> [:Area:]
   -- splitArea returns the four sub-areas in a parallel array
calcCentroid :: [:Tree:] -> (Mass, Location)
```

The first equation deals with the case of a single particle: we simply record its mass and location. In the recursive case, the array comprehension for `subtrees` iterates in parallel over `(splitArea area)`, an array of exactly four elements. For each such area `a`, we compute the set of particles `ps` that lie inside `a` and, if that set is non-empty, we recursively call `buildTree`. The "if non-empty" test discards sub-areas which do not contain any particles at all, so the length of `subtrees` can be anything between 1 and 4. We omit the implementations of `inArea` and `calcCentroid`, since they are straightforward.

The nested comprehension in the where clause of `buildTree` makes sure that `inArea` is called on every subarea/particle combination in a single parallel step. Another source of nested parallelism in `buildTree` are the recursive calls to the parallel function `buildTree`, which are performed simultaneously on however many sub-areas contain particles (from one to four). The number of parallel steps is hence proportional to the depth of the rose tree.

Lastly, we have to write the function `calcForces`, which, given a `Tree` and an array of particles, calculates the forces applied by the `Tree` on those particles. It can do so by dividing the particles into two groups: those that are "far" from the centre of gravity of the `Tree` (as determined by a function `isFar`), and those that are "near". Here is the code:

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len (Node m l ts) ps
  = let
      far_forces     = [: forceOn p m l | p <- ps, isFar len l p :]
      near_ps        = [: p | p <- ps, not (isFar len l p) :]
      near_forces_s  = [: calcForces (len / 2) t near_ps | t <- ts :]
      near_forces    = [: sumForces p_forces
                        | p_forces <- transposeP near_forces_s :]
    in
    combineP [:isFar len l p | p <- ps:] far_forces near_forces
```

```
forceOn    :: Particle -> Mass -> Location -> Force
isFar      :: Float -> Location -> Particle -> Bool
sumForces  :: [:Force:] -> Force
```

The function `calcForces` divides the particles into two groups: those that are "near" the centroid `l` of tree, and those that are not. For the far particles, we simply use `forceOn` to compute the force on each such particle from the tree, giving `far_forces`. For particles near to `l`, `near_ps`, we recursively use `calcForces` (in parallel) to compute the force on each particle from each sub-tree giving `near_forces_s`, a short vector with one element for each sub-tree. Each element is a vector with one element for each particle, giving the force on that particle from the sub-tree. All that remains is to transpose this nested structure, and add up the forces on each particle. Finally, we must re-combine the near and far forces, using `combineP`, which interleaves two vectors as directed by a boolean mask.

2.3 Communication and locality

Here is an alternative, simpler way to write `calcForces`:

```
calcForces :: Tree -> [:Particle:] -> [:Force:]
calcForces tree ps = mapP (calc t) ps
  where
    calc (Node m l ts) p
      | isFar l p = forceOn p m l
      | otherwise = sumForces [: calc t p | t <- ts :]
```

For each particle (the `mapP`), it recurses down the tree, stopping when the centroid of the sub-tree is far away from the particle.

Which version should we prefer? Different ways of writing the code give rise to different patterns of data communication. In this latter version you can see that every particle needs a copy of (at least the top part of) the tree, so the danger here is that most of the tree ends up being copied to most of the processors. In the earlier version, the particles migrate (in smaller and smaller groups) to the tree, rather than the other way around.

It undoubtedly complicates the programmer's life to have to think about these matters, but there is no silver bullet. Parallel programming is complicated, and programmers *must* think about concurrency and communication, as well as correctness. However, one of the advantages of the data-parallel style is that it gives us a much better handle on the program's *cost model* (both computation and communication) than un-structured parallel programming [Ble96].

2.4 Summary

The algorithm we have described makes extensive use of data parallelism. For example, `buildTree` is called in parallel on the four sub-areas of the area under consideration; and for each of those sub-areas we compute the relevant subset of the particles in parallel. Similarly `calcForces` is called in parallel on the four sub-trees; and the computation of `far_forces` is done in parallel over all the particles. In each case, the recursive calls over the sub-trees express *nested* data parallelism, because the computation that is performed

on the sub-tree is itself a data-parallel computation. This is really quite difficult to express using flat data parallel frameworks; indeed tree construction is often not parallelised.

The rest of this paper uses the parallel Barnes-Hut algorithm as a running example to explain the successive steps through which the program is compiled to run efficiently on parallel shared-memory machines.

3 Compiling DPH programs

The compiler must translate high-level nested data parallel programs, as described in the previous section, into efficient low-level code. This translation consists of four main steps:

- *Desugaring* removes syntactic sugar, reducing the program to a simple lambda language. This intermediate language, GHC's "Core" language, is still strongly typed.
- *Vectorisation* transforms *nested* data parallelism into *flat* data parallelism; it is a Core-to-Core transformation.
- *Fusion* optimises the Core program, by eliminating redundant synchronisation points and intermediate arrays, thus dramatically improves locality of reference;
- *Gang parallelism* divides the parallel operations spatially into chunks, each chunk being executed by a thread from a *gang* of threads. Typically a gang contains a thread for each CPU. Gang parallelism is expressed by giving library implementations of the "vector instructions", rather than by built-in compiler support.

GHC implements these steps using a large number of Core-to-Core program transformations. Many of these transformations have been part of GHC's optimiser for a long time, in particular a sophisticated inliner, worker-wrapper unboxing, and constructor specialisation [Pey96, PM02, PL91, PTH01]. In the course of the Data Parallel Haskell project, we are adding more, array-specific transformations. Due to GHC's generic support for program transformations — specifically, the inliner and rewrite rules [PM02, PTH01] — we can implement most of these new transformations as library code, as opposed to extending the compiler itself. Indeed, apart from the vectorisation pass, the rest of the optimisation pipeline operates in ignorance of the fact that the program being optimised is a data parallel one.

In this paper we focus mainly on vectorisation, starting at Section 3.2, after taking a brief diversion to describe how array comprehensions are desugared (Section 3.1).

3.1 Desugaring array comprehensions

In the Barnes-Hut code we used both *array comprehensions*, and *ordinary functions* over parallel arrays such as `zipP` and `mapP`. However, just as in the case of list comprehensions, the former is just a convenient syntactic sugar for the latter. More precisely, Figure 4 gives rules for desugaring array comprehensions. They are quite standard [JW07], and practically identical to those for lists, so we do not discuss them further. These rules are simple, but they should be thought of as a specification rather than an implementation, because they generate somewhat inefficient code. In GHC's actual implementation we use slightly more complicated rules.

$$
\begin{array}{lrcl}
\textbf{Expressions} & e & ::= & \ldots \mid \texttt{[:}e \mid q\texttt{:]} \\
\textbf{Qualifiers} & p, q & ::= & x\texttt{<-}e \mid e \mid p\texttt{,}q \mid p \texttt{|} q
\end{array}
$$

$$\mathcal{D}[\![\texttt{[:}e \texttt{|} q\texttt{:]}]\!] = \texttt{mapP}\ (\lambda q_v.e)\ \mathcal{Q}[\![q]\!]$$

$\mathcal{Q}[\![q]\!]$ computes the parallel array of the tuples generated by q

$$
\begin{array}{rcl}
\mathcal{Q}[\![x\texttt{<-}e]\!] & = & e \\
\mathcal{Q}[\![e]\!] & = & \texttt{if}\ e\ \texttt{then [:():] else [::]} \\
\mathcal{Q}[\![p\texttt{,}q]\!] & = & \texttt{concatMapP}\ (\lambda p_v.\texttt{mapP}\ (\lambda q_v.(p_v, q_v))\ \mathcal{Q}[\![q]\!])\mathcal{Q}[\![p]\!] \\
\mathcal{Q}[\![p \texttt{|} q]\!] & = & \texttt{zipP}\ \mathcal{Q}[\![p]\!]\ \mathcal{Q}[\![q]\!]
\end{array}
$$

q_v is a tuple of the variables bound by q

$$
\begin{array}{rcl}
(x\texttt{<-}e)_v & = & x \\
(g)_v & = & () \\
(p\texttt{,}q)_v & = & (p_v, q_v) \\
(p \texttt{|} q)_v & = & (p_v, q_v)
\end{array}
$$

Figure 4: Desugaring rules for array comprehensions

3.2 Informal overview of vectorisation

The purpose of vectorisation is to take a program that uses nested data parallelism, and transform it into a program that uses only flat data parallelism. Consider this tiny example

```
f :: Float -> Float
f x = x*x + 1
```

For every such function we build its *lifted* version $\texttt{f}_\texttt{L}$ thus:

```
f_L :: [:Float:] -> [:Float:]
f_L x = (x *_L x) +_L (replicateP n 1)
        where
          n = lengthP x
```

Internally, $\texttt{f}_\texttt{L}$ uses "vector instructions" like $+_\texttt{L}$ to do its work, where

```
+_L :: [:Float:] -> [:Float:] -> [:Float:]
```

Notice that it must also replicate the constant 1 so that argument has the type that $+_\texttt{L}$ expects. So, roughly speaking (we give the true story later), to form the definition of $\texttt{f}_\texttt{L}$ we transform the body of `f` in the following way:

- Replace a constant by a call to `replicateP`.
- Replace a function by its lifted versions (e.g. + becomes $+_\texttt{L}$).
- Replace a parameter (e.g. `x`) by itself.

This new definition obeys the equation $\texttt{f}_\texttt{L}$ = `mapP f`, so it takes an array to an array. In effect, it is a specialised variant of `mapP` – specialised by fixing the function argument. The idea is that whenever we see the call (`mapP f`) we will replace it by $\texttt{f}_\texttt{L}$. But there is a problem! Suppose we have

```
g :: [:Float:] -> [:Float:]
g xs = mapP f xs
```

First we replace (`mapP f`) by f_L to get:

```
g :: [:Float:] -> [:Float:]
g xs = fL xs
```

But now we must lift `g` too, in case there are calls to (`mapP g`). If we try, we get this:

```
gL :: [:[:Float:]:] -> [:[:Float:]:]
gL xs = fLL xs
```

Not good: we need the *doubly-lifted* version of `f`! If the depth of nesting is not statically bounded (and it isn't in Barnes-Hut) then we are in trouble. Blelloch's clever solution is to observe that we can define f_{LL} in terms of f_L, thus:

```
fLL :: [:[:Float:]:] -> [:[:Float:]:]
fLL xss = unconcatP xss (fL (concatP xss))
```

That is: first concatenate all the rows of `xss` to make a single flat vector; then map `f` over that vector; then chop up the result to form a vector of vectors again, guided by the original shape of `xss`. (Note that the incoming vector might well be "ragged", so that not all the sub-vectors have the same length.) At first, this idea looks terribly inefficient, because of all the flattening and un-flattening but, as we shall see, if we choose the right data representation, `concatP` and `unconcatP` take constant time and involve no copying.

This is the core of the vectorisation transformation. We have left many details vague. What about higher order functions? What about user-defined data types? We now start to tighten our description up. We begin by discussing how to represent arrays (Section 4) and functions (Section 5) in vectorised code. These representation choices in turn drive the vectorisation transformation (Section 6). More details are given in previous work [KC98, CK00, LCK06, CLP+07].

4 Representing arrays in vectorised code

Standard arrays in Haskell are parametric; i.e., the array representation is independent of the type of array elements. This is achieved by using arrays of pointers referring to the actual element data. Such a *boxed* representation is very flexible, but it is also detrimental to performance. The indirections consume additional memory, increase memory traffic, and decrease locality of memory access. The resulting runtime penalty can be very significant.

The parallel arrays `[:a:]` offered by the DPH *source* language are also parametric, as can be seen from the polymorphic type signatures in Figure 1. One of the tasks of the vectoriser is to change the array representation, by systematically transforming a function that manipulates values of type `[:(Int,Int):]`, say, to one that manipulates values of type `PA (Int,Int)`. These new `PA` arrays have a *non-parametric* representation; that is, *the representation depends on the element type* [CK00]. For example, a value of type `PA Int` is held as a contiguous memory area containing unboxed 32-bit integer values — not as a block of pointers to `Int`-valued thunks, as is the case in vanilla Haskell.

Although `PA` is not visible to the user, such non-parametric data types are an indepenently-useful source-language feature, already implemented in GHC, which we call an *associated data type* [CKPM05]. We will therefore explain `PA` using the notation of associated data types.

Since the representation of an array depends on the type of its elements, there can be no useful polymorphic functions over `PA`. For example, we cannot define

```
lengthPA :: PA a -> Int    -- WRONG!
```

because `lengthPA`, being polymorphic in `a`, knows nothing about the representation of `PA a`. This is just what type classes are for. So we declare the type `PA` in association with a class `PAElem` that defines operations over the type, thus:

```
class PAElem a where
  data PA a
  indexPA     :: PA a -> Int -> a
  lengthPA    :: PA a -> Int
  replicatePA :: Int -> a -> PA a
  ...more operations...
```

Given a type `a` that is allowed to be an element of a parallel array, there is a corresponding data type `PA a`, and operations `indexPA`, `lengthPA`, `replicatePA`, and so on. These operations therefore have overloaded types, thus:

```
indexPA  :: PAElem a => PA a -> Int -> a
lengthPA :: PAElem a => PA a -> Int
...etc...
```

All the parametric operations of Figure 1 have `PA` variants with the same types apart from the additional `(PAElem a)` constraint. (Our concrete implementation is more complex with more operations, but the code shown here conveys the basic idea.)

An instance declaration fills in an implementation for each of these elements. For example, the instance declaration for integers takes the following form:

```
class PAElem Int where
  data PA Int = AInt ByteArray
  indexPA (AInt ba) i = indexIntArray ba i
  lengthPA (AInt ba) = lengthIntArray ba
  replicatePA n i = AInt (replicateIntArray n i)
  ...more operations...
```

We represent the array by a contiguous region of bytes (aka `ByteArray`) with primitives such as `indexIntArray` that operate on individual 32-bit integers from a `ByteArray`. (The code again simplifies the concrete implementation by omitting the use of unboxed types.)

4.1 Arrays of structured data

The `PAElem` instance for `Float`, and other primitive types, follows the same pattern. But what about more complex data structures, such as an array of pairs? It is quite unacceptable to represent it by an array of pointers to (heap-allocated) records, because the indirection costs would be too heavy. Instead, we represent it by a *pair of arrays*:

```
class (PAElem a, PAElem b) => PAElem (a, b) where
  data PA (a,b) = ATup2 Int (PA a) (PA b)
  indexPA (ATup2 _ arr1 arr2) i = (indexPA arr1 i, indexPA arr2 iv)
  lengthPA (ATup2 n _ _) = n
```

Thus, a `PA (Float,Float)` is represented by a pair of unboxed arrays, each storing a vector of floating point values. Crucially, the two arrays must have the same length; and we record that length in the `Int` field of the $\texttt{ATup}_2$ constructor. This length field is convenient, but usually redundant — but not always! Consider an array of `()` elements:

```
class PAElem () where
  data PA () = ATup0 Int
  indexPA (ATup0 _) i = ()
  lengthPA (ATup0 n) = n
```

We need no data storage to store a vector of `()` values, but we must still remember its length.

Notice that the representation is *compositional*; that is, the representation of an array of pairs is given by combining the representations of an array of the first and second elements of the pair, and so on recursively.

The representation also allows us to combine two arrays element-wise into an array of pairs *in constant time*, with unzipping being equally easy:

```
zipPA :: PAElem a => PA a -> PA b -> PA (a,b)
zipPA as bs = ATup2 (lengthPA as) as bs

unzipPA :: PA (a,b) -> (PA a, PA b)
unzipPA (ATup_2 _ as bs) = (as,bs)
```

This stands in contrast to lists, where zipping and unzipping take linear time.

Lastly, since records are converted into product types by the desugarer, the `Particle` arrays in Barnes-Hut are represented by tuples of arrays.

4.2 Nested arrays

Even more interesting is the representation of nested arrays. A classic example is that of *sparse matrices*, in which we represent a sparse matrix as a vector of rows, each row consisting of a vector of (index,value) pairs, where only the non-zero values in the row are represented. Thus

```
type SparseMatrix a = [:[:(Int,a):]:]
```

Since our ultimate goal is to eliminate nested parallelism, it is not surprising that we also want to represent nested arrays in terms of flat ones. Indeed, a nested array `PA (PA a)` can be encoded by

- a flat *data array* of type `PA a` which contains the data elements and
- a *segment descriptor* of type `PA (Int, Int)` which stores the starting position and length of the subarrays embedded in the flat data array.

This is captured by the following instance:

```
class PAElem a => PAElem (PA a) where
  data PA (PA a) = AArr (PA a) (PA (Int, Int))
  indexPA (AArr arr segd) i = slicePA arr (indexPA segd i)
  lengthPA (AArr _ seg) = lengthPA seg
```

where `sliceP` extracts a subarray from a larger array in constant time. Thus, the sparse matrix

```
[:[:(0,15),(2,9),(3,20):], [::], [:(3,46):]:]
```

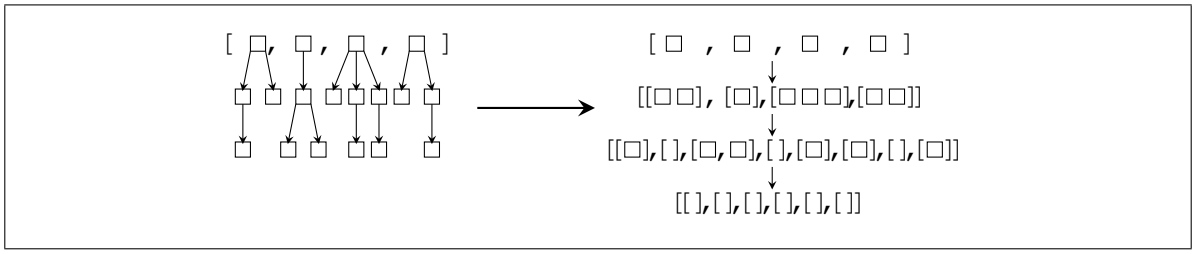

Figure 5: Value of type `[:Tree:]` and its vectorised representation

will be represented as

```
AArr (ATup2 [#0,2,3,3#] [#15,9,20,46#]) -- Data
     (ATup2 [#0,3,3#] [#3,0,1#])        -- Segment descriptor
```

where we write `[#...#]` for a literal `ByteArray`. The first `ByteArray` contains all the column indexes, the second one all the `Float`s, and the third and fourth the start indexes and lengths of the segments, respectively. Since all four `ByteArray` are unboxed, programs which process such matrices can be compiled to highly efficient code.

Remarkably, we can now give constant-time implementations of the two functions `concatPA` and `unconcatPA`, as promised in Section 3.2:

```
concatPA :: PA (PA a) -> PA a
concatPA (AArr cts _) = cts

unconcatPA :: PA (PA a) -> PA b -> PA (PA b)
unconcatPA (AArr _ shape) cts = AArr cts shape
```

4.3 Recursive types

If the array elements are recursive, the non-parametric representation of the array has to be recursive, too. For instance, arrays of `Tree` from Section 2.2 are represented as follows:

```
instance PAElem Tree where
  data PA Tree = ATree Int (PA Mass) (PA Location) (PA (PA Tree))
```

Since `Tree` is a product, the representation is similar to arrays of tuples. It stores the masses and locations of the centroids and a nested array containing the subtrees of each node. As described in the previous section, the latter is encoded by a flat array of trees together with a segment descriptor. In effect, this means that an array of trees is represented by a list with each element containing the centroids and segmentation information for one tree level. This allows all data in one level to be processed in parallel, although the levels have to be processed one after another. Figure 5 illustrates this representation. User-defined types are discussed in more detail in Section 6.4.

4.4 Polymorphism

If we were only interested in *monomorphic* code, or if we would use a whole-program compiler that specialises a polymorphic to a monomorphic program, as was the case in NESL, then life would have been much easier. We could implement the non-parametric array type

by *statically replacing* `PA Int` by `PAInt`, say, where the latter is a perfectly ordinary data type, defined as

```
data PAInt = AInt ByteArray
```

Now we do not need non-parametric types; in the vectorised code, original types `[:Int:]`, `[:(Int,Float):]`, etc, are simply replaced by `PAInt`, `PAPair PAInt PAFloat`, and so on, where all these are ordinary data types.

Alas, this does not work for *polymorphic* functions. For example, how could we translate the type of this function? What would we statically replace `[:a:]` by?

```
firstRow :: [:[:a:]:] -> [:a:]
```

We also cannot turn polymorphic functions into families of monomorphic functions, as we support separate compilation and polymorphic recursion. No — if we want polymorphism, we must use something akin to type classes, as we have described in this section. A key component of our work is the extension of the non-parametric representation idea to work in a polymorphic setting. In particular, our Core language regards `PA` as a type-level function from types to types [SCPD07].

5 Representing functions in vectorised code

Haskell is a higher order language, so we have to consider how to vectorise programs that manipulate functions. Vectorising higher-order programs raises two distinct problems.

5.1 Functions are pairs

Consider this (contrived) definition:

```
ho :: (Int->Bool) -> (Bool, [:Bool:])
ho f = (f 2, mapP f [:1,2,3:])
```

In our overview (Section 3.2), we said that we should replace `(mapP f)` with a call to $\texttt{f}_\text{L}$, the lifted version of `f`. But since `f` is lambda-bound, it is not so easy to call "the lifted version of `f`". Clearly the caller must pass the lifted version of `f` as a parameter to `ho`. But there is an ordinary, scalar call `(f x)` in the body of `ho`, so we can't pass *only* the lifted version. The obvious alternative is to pass a pair that gives *both* the lifted *and* unlifted versions. With such a representation, `mapP` can just extract the lifted variant of its argument (a pair), while the vanilla application of `f` extracts the unlifted variant.

5.2 Functions are closures

There is a second challenge. In Section 4 we discussed the efficient, non-parametric representation of data-parallel arrays. A higher order language forces us to confront the question of how to represent an array of *functions*. For example:

```
distance :: [:Float:] -> [: Float->Float :]
distance xs = mapP (λx y. sqrt (x*x + y*y)) xs

distY :: [:Float:] -> Float -> Float
distY xs y = sumP [: d y | d <- distance xs :]
```

There are more direct ways to write `distY`, of course, but arrays of functions also arise inevitably when we think about lifting. If we start with

```
f :: (Int->Int) -> Int
```

then the lifted version of `f` has the type

```
f_L :: [: Int->Int :] -> [:Int:]
```

How should we represent such an array of functions? A possible answer is "as an array of pointers to function closures". Bad answer! In data-parallel computations, all the processors are supposed to execute the same code in parallel, but if every function closure in the array is potentially different, they clearly cannot do that. Furthermore, a pointer-based representation destroys locality.

Happily, there is no need for the generality of a distinct function pointer for each array element. Consider the result of `distance`, for example. Every element of this array is a function *with the same code*, but a different value for the function's free variable `x`. This makes the solution obvious: we must represent an array of functions by a pair of a *single* code pointer and an *array* of environment records, which give the per-element bindings for the free variables.

5.3 Putting it together

Putting our two solutions together, we see that in vectorised code a function must be represented by a triple:

1. The scalar version of the function
2. The lifted version of the function
3. An environment record of the free variables of the function

To be concrete, here is the data type declaration for vectorised functions:

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

This declaration says that `(:->)` is an algebraic data type (written infix), with a single constructor `Clo`. The constructor `Clo` has an existentially-quantified type variable `e`, and three fields, `env`, $\texttt{clo}_s$, and $\texttt{clo}_l$. The vectorisation transformation will transform every function type $\tau_1\texttt{->}\tau_2$ to $\tau_1'\texttt{:->}\tau_2'$, where τ_1' is the transformed version of τ_1 and similarly for τ_2. In effect, the vectorisation transform performs closure conversion [AJ89].

With this definition in hand, we can now explain how arrays of values of type `(a :-> b)` are represented:

```
instance PAElem (a :-> b) where
  data PA (a :-> b) = forall e. PAElem e
                    => AClo { aenv  :: PA e
                            , aclo_s :: e -> a -> b
                            , aclo_l :: PA e -> PA a -> PA b }
  lengthPA (AClo env f_s f_l) = lengthPA env
  indexPA (AClo env f_s f_l) n = Clo (indexPA env n) f_s f_l
  replicatePA n (Clo env f_s f_l) = AClo (replicatePA n env) f_s f_l
```

To represent an array of functions, we keep a *single* code pointer for each of the scalar and lifted code, but have an *array* of environment records. Notice that `Clo` and `AClo` differ *only* in the type of the environment field; their $\texttt{clo}_s$ and $\texttt{clo}_l$ fields are identical.

As in the case of other types, it is worth noting that this representation supports very simple and direct implementations of indexing, replication, and so on. It does *not* efficiently support literal arrays of various different functions, such as `[:sin,cos:]`. This is quite deliberate: in a data-parallel computation all the processors should be performing the same computation at the same time. Nevertheless, such arrays can be handled, essentially using conditionals which ensure that different functions are executed one after another.

In Section 6.2, we will see how `Clo` and `AClo` are used in the transformation of the `ho` example. Before we can do so, we must first specify the vectorisation transformation more precisely.

6 Vectorisation

We are finally ready to discuss the vectorisation transformation itself. Consider a top-level function definition $f :: \tau = e$, where τ is the type of f. The *full vectorisation* transformation produces a definition for the vectorised version of f called f_V, thus:

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

Here, f_V is the *fully vectorised* variant of f, whose right-hand side is generated by the *full vectorisation transform* $\mathcal{V}\,[\![\cdot]\!]$. As we have already seen, vectorisation returns an expression of a different type to the input, so the type of f_V is obtained by vectorising the type τ, thus $\mathcal{V}_t[\![\tau]\!]$. In general, if $e :: \tau$ then $\mathcal{V}\,[\![e]\!] :: \mathcal{V}_t[\![\tau]\!]$. Figure 6 gives the functions for both type and term vectorisation. In our compiler, the transformation applies to an explicitly-typed program, but we omit all type information in Figure 6, in order to concentrate on the essentials.

Vectorisation is applied separately to each top level function in the program, so it is a whole-program transformation. In real programs, only a part will be data-parallel, while much of it is not (e.g. input/output, user interaction etc). We ignore this issue here, but in reality our compiler performs *selective* vectorisation – see Section 6.5 and [CLJK08].

The type transformation $\mathcal{V}_t[\![\tau]\!]$ transforms a source-program type to the corresponding type in the vectorised program. As can be seen in Figure 6, its effect is simple: it transforms every function arrow `(->)` to a vectorised function arrow `(:->)`, and every parametric array constructor `[::]` to a non-parametric parallel array constructor `PA`. A user-defined algebraic data type might have nested uses of `(->)` or `[::]` — for example, `Tree` does so — and for these we must generate a vectorised variant ($\texttt{Tree}_\text{V}$) of the data type itself. We elaborate this point in Section 6.4.

This type transformation forces the vectorised program to differ quite radically from the original. In particular, since a "function" is now a triple constructed with `Clo`, we need an infix application operator `$:` to extract the scalar copy:

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

and a lifted version

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

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

$$
\mathcal{L}_t[\![\tau]\!] = \mathit{PA}\ \mathcal{V}_t[\![\tau]\!]
$$

$\mathcal{V}[\![e]\!] :: \mathit{Term} \to \mathit{Term}$ is the full vectorisation transformation on terms

Invariant: if $\overline{x_i : \sigma_i} \vdash e : \tau$ then $\overline{x_i : \mathcal{V}_t[\![\sigma_i]\!]} \vdash \mathcal{V}[\![e]\!] : \mathcal{V}_t[\![\tau]\!]$

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

$$
\mathcal{V}\left[\!\!\left[\begin{array}{l} \texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3 \end{array} \right]\!\!\right] = \texttt{if}\ \mathcal{V}[\![e_1]\!]\ \texttt{then}\ \mathcal{V}[\![e_2]\!]\ \texttt{else}\ \mathcal{V}[\![e_3]\!]
$$

$\mathcal{L}[\![e]\!]\ n :: \mathit{Term} \to \mathit{Term} \to \mathit{Term}$ 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

$$
\begin{array}{rll}
\mathcal{L}[\![k]\!]\ n & = \texttt{replicatePA}\ n\ k & k \text{ is a literal} \\
\mathcal{L}[\![f]\!]\ n & = \texttt{replicatePA}\ n\ f_\mathrm{V} & f \text{ is bound at top level} \\
\mathcal{L}[\![x]\!]\ n & = x & x \text{ is locally bound (lambda, let, etc)} \\
\mathcal{L}[\![e_1\ e_2]\!]\ n & = \mathcal{L}[\![e_1]\!]\ n\ \texttt{\$:}_\mathrm{L}\ \mathcal{L}[\![e_2]\!]\ n & \\
\mathcal{L}[\![\lambda x.e]\!]\ n & = \texttt{AClo}\ \{\texttt{aenv} = \texttt{ATup}_k\ n\ y_1 \ldots y_k, & \\
 & \quad , \texttt{aclo}_s = \lambda e\, x.\, \texttt{case}\ e\ \texttt{of}\ (y_1, \ldots, y_k) \to \mathcal{V}[\![e]\!] & \\
 & \quad , \texttt{aclo}_l = \lambda e\, x.\, \texttt{case}\ e\ \texttt{of}\ \texttt{ATup}_k\ n'\ y_1 \ldots y_k \to \mathcal{L}[\![e]\!]\ n'\} & \\
 & \text{where} \{y_1, \ldots, y_k\} = \text{free variables of } \lambda x.e &
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{L}\left[\!\!\left[\begin{array}{l} \texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3 \end{array} \right]\!\!\right] n & = \texttt{combinePA}\ e'_1\ e'_2\ e'_3 & \\
 & \text{where}\ e'_1 = \mathcal{L}[\![e_1]\!]\ n & \\
 & \quad e'_2 = \texttt{case}\ ys_2\ \texttt{of}\ \texttt{ATup}_k\ n_2\ y_1 \ldots y_k \to \mathcal{L}'[\![e_2]\!]\ n_2 & \\
 & \quad e'_3 = \texttt{case}\ ys_3\ \texttt{of}\ \texttt{ATup}_k\ n_3\ y_1 \ldots y_k \to \mathcal{L}'[\![e_3]\!]\ n_3 & \\
 & \quad (ys_2, ys_3) = \texttt{splitPA}\ e'_1\ (\texttt{ATup}_k\ n\ y_1 \ldots y_k) & \\
 & \quad \{y_1, \ldots, y_k\} = \text{free variables of } e_2, e_3 & \\
\mathcal{L}'[\![e]\!]\ n & = \texttt{if}\ n\texttt{==0}\ \texttt{then emptyPA else}\ \mathcal{L}[\![e]\!]\ n &
\end{array}
$$

Figure 6: The vectorisation transformation

```
($:L) :: PA (a:->b) -> PA a -> PA b
($:L) (AClo env fs fl) = fl env
```

The transformation rules in Figure 6 are given with their type invariants, which make the rules much more comprehensible. For example, consider the rule for $\mathcal{V}[\![e_1\ e_2]\!]$. Since $e_1 : \tau_1 \text{->} \tau_2$, we know that $\mathcal{V}[\![e_1]\!] : \mathcal{V}_t[\![\tau_1]\!] \text{:->} \mathcal{V}_t[\![\tau_2]\!]$; that is why we need the application function (`$:`) to transform the application to an expression of type $\mathcal{V}_t[\![\tau_2]\!]$.

Similarly, the rule for $\mathcal{V}[\![\lambda x.e]\!]$ must produce a value of type $\mathcal{V}_t[\![\tau_1]\!] \text{:->} \mathcal{V}_t[\![\tau_2]\!]$, and that in turn must be built with a `Clo` constructor. We build an environment tuple $(y_1, \ldots, y_k)$, of the free variables of $(\lambda x.e)$. Now the type of the arguments of `Clo` tell us what functions we must build. The scalar function simply requires a recursive use of $\mathcal{V}[\![e]\!]$, while the lifted function requires us to generate a lifted version of the code for e, $\mathcal{L}[\![e]\!]\ n$.

The rules for $\mathcal{L}[\![e]\!]\ n$ can be read in the same way. The main new complication is with conditionals. First we compute in parallel e'_1, the vector of booleans (of length n) for the discriminant of the conditional. Then we use that vector to split the a vector of environment tuples into two parts, ys_2 (for which corresponding elements of e'_1 is true), and ys_3 (for which e'_1 is false). The lengths n_2, n_3 of these vectors will sum to n. Then we compute each of e'_2 and e'_3 in parallel, and finally interleave them together with `combinePA`.

Why do we need to pack and split the free variables in the conditional rule? Each free variable y_i is bound to an n-vector; but in the `then` branch we need a (shorter) n_2-vector (namely ys_2) of the elements of y_i for which e is True; and dually for the `else` branch. We must also test for n_2 or n_3 being zero (done by $\mathcal{L}'[\![e]\!]\ n$), otherwise when transforming a recursive function we would generate a program that recurses infinitely deep. The *operational* behaviour of the translated function will compute e'_1, e'_2 and e'_3 *in sequence*; as in any data-parallel machine, the "then" and "else" branches of a conditional are computed separately.

Figure 6 is the core of this paper. Our real system handles `let` expressions, `case` expressions, and constructors, and hence is a bit more complicated. But Figure 6 describes all the essential ideas.

Since $\mathcal{V}$ invokes both $\mathcal{V}$ and $\mathcal{L}$ (as does $\mathcal{L}$) you might worry about a code explosion. But notice that the $\texttt{clo}_s$ field in $\mathcal{V}$ is identical to the $\texttt{aclo}_s$ field in $\mathcal{L}$, and both are closed functions that can be named, and bound at top level; and similarly for the $\texttt{clo}_l$ and $\texttt{aclo}_l$ fields. Hence, as we will see in the examples that follow, we can avoid the code explosion simply by naming and sharing these functions.

6.1 A simple example

Here is the simplest possible example:

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

The full vectorisation transformation in Figure 6 gives us this:

```
incV :: Float :-> Float
incV = Clo () incS incL

incS :: () -> Float -> Float
incS = λe x. case e of () -> (+)V $: x $: 1
```

```
incL :: PA () -> PA Float -> PA Float
incL = λe x. case e of ATup0 n -> (+)V $:L x $:L (replicatePA n 1)
```

To aid explanation we have named $\texttt{inc}_S$ and $\texttt{inc}_L$, but otherwise we have simply applied Figure 6 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 to form the *fully vectorised version* $\texttt{inc}_V$.

The vectorised code makes use of vectorised addition `(+)`, which is part of 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 addFloatS addFloatL

(+)L :: PA () -> PA Float -> PA (Float:->Float)
(+)L = λe xs. AClo xs addFloatS addFloatL

-- Implemented in the back end
addFloatS :: Float -> Float -> Float
addFloatL :: PA Float -> PA Float -> PA Float
```

The intermediate functions $\texttt{(+)}_S$ and $\texttt{(+)}_L$ deal with partial applications of `(+)`. Finally we reach ground truth: invocations of $\texttt{addFloat}_S$ and $\texttt{addFloat}_L$, which are implemented by the back end. The former is the ordinary floating point addition instruction; the latter is a "vector instruction", which will be implemented differently on different targets. On a sequential machine it will be implemented as a loop; on a GPU it will be implemented using vector hardware; on a cluster it will be implemented using a loop on each CPU with barrier synchronisation at the end. Section 7 elaborates.

These functions look grotesquely inefficient, especially considering how trivial the original function `inc` was. Fortunately, most of the clutter is introduced to account for the *possibility* of higher order programming, and can be removed by straightforward optimisations.

For example, consider the sub-term $\texttt{(+)}_V$ `$: x $: 1` in the definition of $\texttt{inc}_S$. We can simplify it in the following way:

```
  (+)V $: x $: 1)
⟹ Inline (+)V
  (Clo () (+)S (+)L) $: x $: 1
⟹ Definition of $:
  (+)S () x $: 1
⟹ Inline (+)S
  (Clo x addFloatS addFloatL) $: 1
⟹ Definition of $:
  addFloatS x 1
```

All the intermediate closure data structures are removed. (To save generating huge intermediates during compilation, we are exploring whether the vectorisation transformation could have special cases to avoid introducing them in the first place.)

6.2 The higher order example again

It is instructive to see a case where the use of higher order functions prevents complete removal of intermediate closures. Let us return to the `ho` example of Section 5.1:

```
ho :: (Int->Bool) -> (Bool, [:Bool:])
ho f = (f 2, mapP f [:1,2,3:])
```

Again applying the vectorisation transformation blindly we get this:

```
hoV :: (Int :-> Bool) :-> (Bool, PA Bool)
hoV = Clo () hoS hoL

hoS :: () -> (Int :-> Bool) -> (Bool, PA Bool)
hoS () f = (f $: 2, mapPV $: f $: [:1,2,3:])

hoL :: PA () -> PA (Int :-> Bool) -> PA (Bool, PA Bool)
hoL (ATup_0 n) fs
  = (,)L (fs $:L replicatePA n 2)
         (replicatePA n mapPV $:L fs $:L replicatePA n [:1,2,3:])
```

We have taken a short-cut here by using optimised transformation rules for pairs:

$$\begin{array}{rcl} \mathcal{V}[\![(e_1,e_2)]\!] & = & (\mathcal{V}[\![e_1]\!], \mathcal{V}[\![e_2]\!]) \\ \mathcal{L}[\![(e_1,e_2)]\!]\ n & = & (,)_L\ n\ (\mathcal{L}[\![e_1]\!]\ n)\ (\mathcal{L}[\![e_2]\!]\ n) \end{array}$$

The reader may verify the correctness of this optimised rule by seeing what happens instead if we use the normal translation $(,)_V$ `$:` $\mathcal{V}[\![e_1]\!]$ `$:` $\mathcal{V}[\![e_1]\!]$, and the definition of $(,)_V$, which in turn is very like that for `(+)`. Because of our array representation, the lifted pairing function $(,)_L$ is a constant-time operation:

```
(,)L :: Int -> PA a -> PA b -> PA (a,b)
(,)L n xs ys = ATup2 n xs ys
```

6.3 How flattening happens

In our informal overview (Section 3.2) we said that we "replace a call (`mapP f`) by f_L". Higher order flattening takes that *static* decision and makes it *dynamic*, by representing `f` by a pair of functions, thereby allowing `mapP` to select at runtime. (With the usual compile-time optimisations when `f` is known, of course.) The code for `mapP` itself is therefore the heart of the way in which nested data parallelism is transformed to flat data parallelism. Here it is:

```
mapPV :: (a :-> b) :-> PA a :-> PA b
mapPV = Clo () mapP1 mapP2

mapP1 :: () -> (a :-> b) -> PA a :-> PA b
mapP1 _ f = Clo f mapPS mapPL
```

```
mapP2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
mapP2 _ fs = AClo fs mapPS mapPL

mapPS :: (a :-> b) -> PA a -> PA b
mapPS (Clo env fs fl) xss
  = fl (replicatePA (lengthPA xss) env) xss

mapPL :: PA (a :-> b) -> PA (PA a) -> PA (PA b)
mapPL (AClo env _ fl) xss
  = unconcatPA xss (fl (expandPA xss env) (concatPA xss))
      -- xss :: PA (PA a)
      -- env :: PA e
      -- fl  :: PA e -> PA a -> PA b
```

In `mapPS` we exploit the key observation from Section 3.2, namely that we can define the doubly-lifted function using the singly-lifted one `fl`, using constant-time reshaping operations on the data. Unfortunately, to account for free variables, we face a small complication: the environment `env` contains one element for each *subarray* of `xss`. Thus, before applying `fl` we must expand `env`, i.e., repeat each element as many times as the corresponding subarray of `xss` has elements. For top-level functions, the environment will be empty and `expandPA` performs no work.

Of course, `mapP` is not the only function that the library must implement. All of (the `PA` versions of) the functions in Figure 1 must be provided in vectorised form. For example, here is the lifted version of `zipP` (the definition of `zipPA` is given in Section 4.1):

```
zipPL :: PA (PA a) -> PA (PA b) -> PA (PA a,b)
zipPL (AArr segd xs) (AArr _ ys) = AArr segd (zipPA xs ys)
```

These library functions are the heart of flattening: they make nested data parallelism "go". Everything is organised to make their implementation, especially their lifted variants, work efficiently.

6.4 User-defined data types

One of Haskell's strengths is the ease with which programmers can declare new algebraic data types, and process them using pattern matching. DPH allows all of this expressiveness in fully-vectorised code as well. There are two main complications: occurrences of `(->)` and `[::]` in user-defined data types; and representing arrays of values drawn from such types. We discuss each in turn.

Vectorising user-defined data types

In Figure 6, the type transform $\mathcal{V}_t[\![\tau]\!]$ replaces a user-defined data type T by its vectorised counterpart T_V. But what exactly is T_V? Consider

```
data Fun = MkFun (Int -> Int)
```

Remember that in the vectorised program, each function arrow `(->)` must be replaced by a function closure `(:->)` — and of course that must also happen inside data types. So we must generate a vectorised version of `Fun`, thus:

```
data Fun_V = MkFun_V (Int :-> Int)
```

This must be done recursively: if a constructor of data type `T` mentions `Fun`, then `T` too must have a vectorised version. So the vectorised variant of each data type obtained by simply applying the $\mathcal{V}_t[\![]\!]$ transform to every type in the data type declaration. The `Tree` type of Section 2.2 is another good example, because we must replace `[::]` by `PA`:

```
data Tree_V = Node_V Mass Location (PA Tree)
```

While we *can* generate a vectorised version of every data type, it is *unnecessary* to do so for data types that do not mention functions or parallel arrays. Happily, almost all data types fall into this category; for example `Bool`, `Maybe`, lists, tuples, and so on. We quietly took advantage of this in the `Tree` example, by not transforming `Mass` to $\texttt{Mass}_V$ (and similarly `Location`) because $\texttt{Mass} = \texttt{Mass}_V$. In Section 6.5 we will see a second reason to avoid vectorising a data type unless it is absolutely necessary to do so.

Arrays of user-defined data types

The ideas of Section 4.1 can readily be extended to work for arbitrary user-defined algebraic data types. We have already seen how this works for `Tree` in Section 4.3. Here is another example, a sum type:

```
data Maybe a = Nothing | Just a
```

How can we represent an array of `Maybe Float` values? The natural dense representation is as a pair of (a) an array of booleans (`True` for `Nothing`, and `False` for `Just`), and (b) an array of `Float` containing only the `Just` values:

```
instance PAElem a => PAElem (Maybe a) where
  data PA Maybe a = AMaybe (PA Bool) (PA a)
  indexPA (AMaybe bs vs) i
    | indexPA bs i = Nothing
    | otherwise    = indexPA vs (indexPA just_indices i)
    where
      just_indices = scanPA (+) 0 (mapPA boolToInt bs)
  lengthPA (AMaybe bs _) = lengthPA bs
```

In practice, to avoid computing `just_indices` on each indexing operation we precompute the index vector, and cache it in an extra field of the `AMaybe` constructor.

In our real implementation, we avoid generating a big instance declaration for every such user-defined data type, by instead generating code to convert it to a simple sum-of-products representation, and then using a set of fixed instances for `PAElem` at those representation types.

6.5 What we have swept under the carpet

Vectorisation is a complicated transformation, and to keep it comprehensible we have simplified several aspects. In this section we briefly mention some of them.

Types and dictionaries

The alert and Haskell-savvy reader will have noticed the following discontinuity in our presentation. We described `PA` type in association with a *type class*, `PAElem`. However, type classes are dealt with by the type inference system, right at the front end of the compiler, and are completely translated out in the passage to the Core intermediate language. In this desugaring, a function with an overloaded type, such as `nub :: Eq a => [a] -> [a]` is given a second parameter which is a record, or "dictionary", of the functions that implement the operations of the `Eq` class.

In the desugared program, `nub` has type `EqD a -> [a] -> [a]`, where `EqD` is an ordinary data type, thus:

```
data EqD a = EqD { (==) :: a -> a -> Bool
                 , (/=) :: a -> a -> Bool }
```

Correspondingly, the desugarer injects an extra argument at every call to `nub`, namely the correct method suite for that particular call site.

The vectoriser generates many calls to `replicatePA`, `splitPA`, etc, which have type-class-constrained types, *yet the vectoriser runs after typechecking and desugaring are complete*. So the vectoriser cannot take advantage of the implicit injection of extra arguments; instead it must insert them itself. In the real implementation of Figure 6, the vectoriser therefore adds appropriate dictionary abstractions and applications. (In fact, since GHC's Core language is an explicitly-typed variant of System F, we also inject type abstractions and applications.) All this is tiresome but routine; showing the implicit abstractions and applications in Figure 6 would have dramatically obfuscated an already-dense figure.

Selective vectorisation

As mentioned earlier, we do not really vectorise the *whole* program; rather, we selectively vectorise parts of it. We must also generate marshaling code to allow us to "cross the border" between vectorised and unvectorised code. For example, in Barnes-Hut, we presumably want to vectorise the `oneStep` function, which will give us

```
oneStepV :: PA Particle :-> PA Particle
```

If we want to be able to call `oneStep` from ordinary scalar code, we must generate the following marshalling code:

```
oneStep :: [:Particle:] -> [:Particle:]
oneStep ps = fromPA (oneStepV $: (toPA ps))

toPA   :: PAElem a => [:a:] -> PA a
fromPA :: PAElem a => PA a -> [:a:]
```

Marshaling may also be necessary for user-defined data types.. For example, suppose we vectorise a function `f :: Int -> Fun`, so that `fV :: Int :-> FunV` (cf. Section 6.4 for the definition of `Fun`). If we want to call `f` from normal scalar code, we must generate:

```
f :: Int -> Fun
f n = case fV n of
        MkFunV tf -> MkFun (($:) tf)
```

Of course, it gets worse if the data type is recursive, because the marshaling code has to traverse the whole structure. On the other hand, no marshaling is needed for types that have have no functions or arrays inside them, which is a strong reason for exploiting that special case (Section 6.4).

Marshaling has a run-time cost. In particular, the calls to `toPA` and `fromPA` change the data representation for parallel arrays, and so are potentially *very* expensive. In fact, it is possible to choose a representation for `[:a:]` that mitigates these costs somewhat but in general, marshaling data across the border should be avoided.

The question of just which parts of the program to vectorise is therefore an interesting one. We want to vectorise code that can run in parallel; we want to reduce marshaling to a minimum; and we do not want to vectorise code where there is little or no benefit. We suggest automatic approaches in [CLJK08], but it may also be reasonable to seek help from the programmer (e.g. "vectorise module X but not module Y").

Laziness

Consider this function:

```
f :: Int -> Int
f x = h x (1/x)
```

Although `x` might be zero, let us assume that `h` only evaluates its second argument if its first argument is non-zero. Haskell's lazy evaluation therefore ensures that no divide-by-zero exception is raised.

The lifted version will look something like this:

```
f_L :: PA Int -> PA Int -> PA Int
f_L xs = h_L xs (replicatePA (lengthPA xs) 1 /_L xs)
```

The trouble is that a demand for *any* element of h_L's second argument will force *all* the elements to be evaluated, including the divisions by zero. Something very similar arises in a more local context when we have `let` expressions:

```
f x = let y = 1/x in
      if x==0 then 0 else y+1
```

Although this is something of a corner case, we do not yet have a very satisfying solution. We currently simply ignore the problem, and accept the slight change in semantics. A better solution might be to reify the exception into an exceptional value (like a IEEE NaN); but that carries an efficiency cost. Lastly, we might treat the argument as a nullary function, accepting the loss of sharing that would result.

7 Multicore execution model

The vectorisation transformation turns all nested data parallelism into parallel operations on flat arrays, as used by the instances of the `PAElem` class. The transformation is crucial to express parallel algorithms on a high-level of abstraction and in a modular fashion. However, purely functional array operations, even if restricted to flat arrays, are still a far cry from the hardware model of multicore CPUs.

In particular, the vectorised code uses many superfluous intermediate arrays, which increase the overhead of memory management and whose creation involves extra synchronisation between parallel CPUs. Even worse, the repeated traversal of large structures compromises locality of reference, and so, has a very negative effect on execution performance. Finally, we need to map the data parallel array operations onto the multi-threaded execution model of multicore CPUs by way of the Single Program Multiple Data (SPMD) model [Dar01].

In contrast to the vectorisation transformation, we can implement the mapping from flat data-parallel code to SPMD code using existing transformation and optimisation phases of GHC; in particular, we make heavy use of GHC's inliner and rewrite rules [PM02, PTH01], which enable library-specific optimisations as part of library source code, in the form of compiler pragmas. Consequently, we can implement these transformations without altering GHC's source code, which greatly simplifies experimentation with different transformations.

In the reminder of this section, we illustrate the transformation of flat data-parallel code into SPMD code by way of an example. Further details are in [KC99, CK03, CSL07, CLS07, CLP+07].

7.1 Running example

As an example, we consider the computation of the value `far_forces`, in the function `calcForces` of Section 2.2, by way of the array comprehension,

```
[: forceOn p m l | p <- ps, isFar len l p :]
```

After vectorisation and simplification to remove intermediate closure data structures, we have

```
forceOn'L (filterPS (isFar len l) ps) m l
```

The code performs two collective operations on the input array `ps` in sequence. Firstly, the application of $\texttt{filterP}_S$ to remove all particles that are not far, and secondly, a computation that corresponds to lifting `forceOn` only on its first argument (here called, $\texttt{forceOn'}_L$):

```
forceOn'L :: PA Particle -> Mass -> Location -> PA Force
```

Such pipelines of collective operations are typical for data-parallel code.

7.2 The SPMD execution model

The implementation of collective array operations, such as `mapP` and `filterP`, needs to distribute the workload evenly across the the available processing elements (PEs), such as multiple cores and CPUs. In the data-parallel model, the workload of a PE is dependent on the number of array elements residing on that PE. Hence, we balance work by suitably distributing the array elements. By default, we choose an even distribution; i.e., given p PEs and an array of length n, each PE gets about n/p array elements.

In the SPMD model, the individual PEs process local array elements until they arrive at a point in the computation where they require non-local data, and need to cooperate with other PEs. In our example, the result of $\texttt{filterP}_S$ is such a point. Even if the *input* to $\texttt{filterP}_S$ is an array that is evenly spread across the PEs, the *output* of $\texttt{filterP}_S$ might

be wildly unbalanced, depending on which elements of the array are selected by the predicate. If so, any further processing of that array would have an equally unbalanced work distribution.

To avoid a work imbalance, arrays need to be re-distributed when their size changes. Redistribution is a cooperative process in which all PEs need to coordinate. However, redistribution is not the only such operation in an SPMD implementation of data parallelism. Other prominent cooperative operations are reductions (such as `foldP`), pre-scans (such as `scanlP`), and permutation operations. Overall, a parallel program executing in SPMD-style alternates between *processing phases*, where the PEs operate independently on local data, and *communication phases*, where the processing elements interact and exchange data.

Communication phases are typically expensive because they include data exchange and blocking to allow any slower PEs to catch up. Hence, compiler optimisations that remove communication phases in favour of longer-running processing phases are often worthwhile. In particular, the redistribution of arrays after operations that change the array length, such as $\texttt{filterP}_S$, does not necessarily improve overall runtime. An inexpensive, purely local operation may be faster, even if work is not ideally balanced, than an expensive redistribution followed by the same local operation with a perfectly balanced workload.

7.3 Gang parallelism

Our implementation of the SPMD model for data parallelism is based on the coordinated execution of a *gang of threads*, with one thread per PE. GHC includes a Haskell library for concurrent programming with explicit thread forking and thread communication primitives. It forms the lowest level of abstraction in our data-parallel array framework and enables us to implement the entire library in Haskell without any special compiler support or the need to resort to C code.

We need to make the distributed nature of computations in the SPMD model explicit to further compile the code resulting from vectorisation, such as

```
forceOn'L (filterPS (isFar len l) ps) m l
```

In this context, distribution does not imply that the data is necessarily located on physically distinct memory banks, but that different threads are responsible for the processing of different portions of parallel arrays. By being explicit about distribution, we are automatically also explicit about the distinction of processing phases versus communication phases.

Our main vehicle for distinguishing between these two phases and making distribution explicit is the type `Dist a` of *distributed values.* For instance, `Dist Int`, pronounced "distributed `Int`", denotes a collection of *local* integers, such that there is one local integer value per gang thread. Arrays can be distributed, too: `Dist [:Float:]` is a collection of local array *chunks*, again one per gang member, which together make up the array. Arrays are distributed across gang members and joined back together by the following functions:

```
splitD :: PA a -> Dist (PA a)
joinD  :: Dist (PA a) -> PA a
```

Distributed values support a number of operations, most importantly mapping:

```
mapD :: (a -> b) -> Dist a -> Dist b
```

While `splitD` and `joinD` denotes communication, `mapD` is the main means of implementing parallel processing phases: the gang members concurrently apply the (purely sequential) function to their respective local values.

7.4 Inlining of gang code

Given a sequential filter function operating on a *single chunk* of a parallel array

```
filterS :: (a -> Bool) -> PA a -> PA b
```

we can define $\texttt{filterP}_\texttt{S}$ as a distributed gang computation as follows:

```
filterPS p arr = joinD (mapD (filterS p) (splitD arr))
```

Given a global parallel array, which is not distributed, `splitD` distributes the array across the gang, `mapD (filterS p)` applies the sequential filter function in parallel to all chunks of the distributed array, and finally, `joinD` combines the various chunks, which may now be of varying length, into one global array.

Similarly, $\texttt{forceOn'}_\texttt{L}$ internally consists of `mapD`s that compute the force for each particle. The force computations for the individual particles are entirely independent, so we can assume $\texttt{forceOn'}_\texttt{L}$ to have the following structure:

```
forceOn'L ps m l
  = joinD (mapD (mapS (λp. forceOn p m l)) (splitD ps))
```

where `forceOn` is the original, sequential function from the source of our Barnes-Hut implementation and `mapS` is a purely sequential array mapping function.

GHC's inliner will inline the definition of both $\texttt{filterP}_\texttt{S}$ and $\texttt{forceOn'}_\texttt{L}$; i.e., it will perform the following rewriting:

```
forceOn'L (filterPS (isFar len l) ps) m l
```

$\Longrightarrow_{\text{Inlining}}$

```
joinD (mapD (mapS (λp. forceOn p m l)) (
  splitD (joinD (mapD (filterS (isFar len l)) (splitD ps)))))
```

Of special interest here is the function `splitD` which is applied to the immediate result of `joinD` (in the second line of the resulting expression). This turns a distributed array into a global array and distributes it again. In contrast to the original array, the newly distributed one is guaranteed to be distributed evenly; hence, a `splitD/joinD` combination performs load balancing.

However, as we remarked earlier, it is often an advantage to accept some load imbalance in favour of avoiding communication phases in an SPMD computation. In GHC, we easily achieve that by specifying the following *rewrite rule*:

```
"splitD/joinD" forall xs. splitD (joinD xs) = xs
```

GHC has support for specifying such rewrite rules directly in the library source code as compiler pragmas [PTH01]. Applications of the `splitD/joinD` rule frequently produce two adjacent applications of `mapD`, which signal two adjacent purely sequential and thread-local computations. We can combine them, and hence eliminate a synchronisation point, using the well known map fusion law:

```
"mapD/mapD" forall f g xs.
  mapD f (mapD g xs) = mapD (f . g) xs
```

Applying both rules to our example, we get

```
joinD (mapD (mapS (λp. forceOn p m l)) (
  splitD (joinD (mapD (filterS (isFar len l)) (splitD ps)))))
```
$\Longrightarrow$ Apply `splitD/joinD`
```
joinD (mapD (mapS (λp. forceOn p m l)) (
  mapD (filterS (isFar len l)) (splitD ps)))
```
$\Longrightarrow$ Apply `mapD/mapD`
```
joinD (mapD (mapS (λp. forceOn p m l) . filterS (isFar len l))
            (splitD ps)))
```

At this point, the question arises whether we can combine adjacent sequential array combinators, such as `mapS` and `filterS`, to reduce the number of array traversals and intermediate data structures. Indeed, we aggressively remove such inefficiencies using a fusion framework known as *stream fusion* [CSL07, CLS07, CLP+07], but we will refrain from discussing this in detail.

This concludes our brief overview of the post-vectorisation aspects of Data Parallel Haskell. A somewhat more detailed discussion can be found in [CLP+07].

8 Related work

We discussed prior work on the implementation of language support for nested data parallelism in detail in [CLP+07]. In this paper, we will only give a brief overview of existing work, and how they compare to our approach.

The starting point for our work was the nested data parallel programming model of NESL [Ble90, BCH+94], which we extended and implemented in the context of a general-purpose language and GHC, a state-of-the-art compiler. Consequently, we have to deal with a multitude of issues not previously addressed, as for example the combination of user-defined and parallel data structures, selective vectorisation, higher-order functions, separate compilation, and aggressive cross-function optimisation.

Prins et al. worked on various aspects of the vectorisation of nested data parallel programs; see, e.g., [PP93, PPW95]. Most of their work was also in the context of a functional language, but one that like NESL lacks many of Haskell's features. Their work is largely orthogonal to ours.

The Proteus system [MNPR94] promised a combination of data and control parallelism, but Proteus had a particular focus on manual refinement of algorithms, where data parallel components were automatically vectorised, this again was a complete whole-program transformation. Moreover, the system was never fully implemented.

Manticore [FFR+07] supports a range of forms of parallelism including nested data parallelism. Manticore employs some of the same techniques that we use, but does not implement flattening yet [FRRS08]. According to the project web page, a preliminary implementation of the Manticore system should be available around the time when this paper is published.

So et al. [SGW06] developed a parallel library of immutable arrays for C/C++ supporting what they call *sub-primitive fusion*. Their choice of immutable arrays, despite working with imperative languages, is to enable aggressive program transformations, much like in

our approach. However, where we apply transformations statically during compile time, their library builds a representation of the to be executed computation at runtime. Consequently, they require less compiler support and do not have to worry about inlining and similar optimisations. However, they incur a runtime penalty by performing optimisations at runtime and need to amortise that penalty by further optimisations. Like us, they also strive for a seamless integration of data parallelism and explicit concurrency within a single program.

9 Conclusion

We are excited about Data Parallel Haskell because it gives us some chance of writing parallel programs that can in principle efficiently exploit very large parallel machines working on large data sets.

In this paper we have outlined solutions to the challenges of polymorphism, higher order functions, and user-defined data types. There is much to do, however, before we can declare victory. The very generality of Data Parallel Haskell makes it an ambitious undertaking. Many components have to work together smoothly to generate efficient code — and that is before we start to consider matters such as using SSE vector instructions or GPUs, or mapping to a distributed memory architecture. Nevertheless, we regard nested data parallelism general, and Data Parallel Haskell in particular, as a very promising and exciting approach to harnessing the multicores.

Acknowledgements

We gratefully acknowledge the help of Max Bolingbroke, Ryan Ingram and John Reppy, whose comments led to real improvements in the paper. Thank you.

References

[AJ89] 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*, pages 293–302, New York, NY, USA, 1989. ACM Press.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BH86] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[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.

[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.

[CK03] Manuel M. T. Chakravarty and Gabriele Keller. An approach to fast arrays in haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Lecture notes for The Summer School and Workshop on Advanced Functional Programming 2002*, number 2638 in Lecture Notes in Computer Science, 2003.

[CKPM05] Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM Press, 2005.

[CLJK08] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial vectorisation of Haskell programs. In *Proc ACM Workshop on Declarative Aspects of Multicore Programming*, San Francisco, January 2008. ACM Press.

[CLP+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

[CSL07] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, pages 50–64. Springer-Verlag, January 2007.

[Dar01] Frederica Darema. The spmd model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, December 2004.

[FFR+07] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The manticore project. In *2007 ACM SIGPLAN Workshop on ML*. ACM Press, 2007.

[For97] High Performance Fortran Forum. High performance fortran language specification version 2.0. Technical report, Rice University, 1997.

[FRRS08] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. ACM Press, 2008.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference (Vol. 2)*. The MIT Press, 1998.

[JW07] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions: comprehensions with order by and group by. In *Haskell Workshop 2007*, pages 61–72, Frieburg, Germany, September 2007.

[KC98] Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470

in Lecture Notes in Computer Science, pages 709–719, Berlin, 1998. Springer-Verlag.

[KC99] Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In José Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in LNCS, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[LCK06] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In *Third International Workshop on Practical Aspects of High-level Parallel Programming (PAPP 2006)*, number 3992 in LNCS. Springer-Verlag, 2006.

[MNPR94] P. Mills, L. Nyland, J. Prins, and J. Reif. Software issues in high-performance computing and a framework for the development of hpc applications. In *Computer Science Agendas for High Perfromance Computing*. ACM Press, 1994.

[NVI07] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 1.1*, 2007. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[PCS99] Jan F. Prins, S. Chatterjee, and M. Simons. Irregular computations in Fortran — expression and implementation strategies. *Scientific Programming*, 7:313–326, 1999.

[Pey96] SL Peyton Jones. Compilation by transformation: a report from the trenches. In *European Symposium on Programming*, volume 1058 of *LNCS*, pages 18–44. Springer Verlag, 1996.

[PL91] SL Peyton Jones and J Launchbury. Unboxed values as first class citizens. In RJM Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston, 1991. Springer.

[PM02] SL Peyton Jones and S Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, 2002. First published at Workshop on Implementing Declarative Languages, Paris, Sept 1999.

[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.

[PPW95] Daniel Palmer, Jan Prins, and Stephan Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95)*. IEEE Press, 1995.

[PTH01] Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*. ACM SIGPLAN, September 2001.

[SCPD07] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.

[SGW06] Byoungro So, Anwar Ghuloum, and Youfeng Wu. Optimizing data parallel

operations on many-core platforms. In *First Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2006. `http://www.isi.edu/~kintali/stmcs06/prog.html`.

[SJ08] Satnam Singh and Simon Peyton Jones. *A Tutorial on Parallel and Concurrent Programming in Haskell*. Lecture Notes in Computer Science. Springer Verlag, Nijmegen, Holland, May 2008.