# Higher Order Flattening[⋆]

Roman Leshchinskiy[⋆⋆], Manuel M. T. Chakravarty, and Gabriele Keller

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

**Abstract** We extend the *flattening transformation,* which turns nested into flat data parallelism, to the full higher-order case, including lambda abstractions and data parallel arrays of functions. Our central observation is that flattening needs to transform the closures used to represent functional values. Thus, we use closure conversion before flattening and introduce *array closures* to represent arrays of functional values.

## 1 Introduction

Nested data parallelism [Ble96] enables the concise specification of irregular parallel computations involving sparse data structures (e.g., trees and sparse matrixes) and comes with a simple language-based cost model. Following [CKLP01], we add nested data parallelism to Haskell by including a type of *parallel arrays*, denoted $[{:}\alpha{:}]$, such that we can use variants of all special list syntax and *Prelude* list operations that only involve finite lists. To distinguish arrays from lists, we add colons to the square brackets (as in $[{:}1, 2, 3{:}]$) and append the suffix $P$ to function names (e.g., $\mathit{mapP}\ (+1)\ [{:}1, 2, 3{:}]$ adds 1 to each element of $[{:}1, 2, 3{:}]$). The main difference between lists and arrays is that the latter have a *parallel evaluation semantics;* i.e., all elements are evaluated as soon as one is demanded.

Given an array of arrays, $\mathit{xs} :: [{:}[{:}\mathit{Float}{:}]{:}]$, we may evaluate the sum of each subarray by $\mathit{mapP}\ \mathit{sumP}\ \mathit{xs}$. As $\mathit{sumP}$ itself is a parallel operation, we overall have a nested parallel computation, where $\mathit{lengthP}\ \mathit{xs}$ parallel summations are performed in parallel. A nested array, such as $\mathit{xs}$, generally constitutes an irregular structure as the subarrays may be of varying length; hence it goes beyond Fortran's notion of an array. Nevertheless, we want to avoid a pointer-based representation, where the outer array is an array of pointers to the subarrays, and prefer a flat representation, which stores the elements of all subarrays in one contiguous block of memory and keeps the information about subarray boundaries in a separate structure called a *segment descriptor.* For example, $[{:}[{:}1,\ 2{:}],\ [{:}{:}],\ [{:}3,\ 4,\ 5{:}]{:}]$ would be represented by the pair $([{:}2, 0, 3{:}],\ [{:}1,\ 2,\ 3,\ 4,\ 5{:}])$ of segment descriptor and flat value array. This improves locality of reference, and hence, increases cache utilisation as well as decreases communication on a distributed-memory parallel machine.

⋆ Partly funded by the Australian Research Council under grant number DP0211203.
⋆⋆ The first author was, in part, at the Technische Universität Berlin, Germany.

*Flattening* is a program transformation that turns nested data structures into flat representations (of the form just described) and vectorises the code that operates on these nested structures [Ble90]. For example, *mapP sumP xs* turns into $sumP^{\uparrow}$ *xs*, where $sumP^{\uparrow}$ is the *lifted* variant of *sumP* that simultaneously computes the sum of all segments of a flat representation of a nested array. In previous work [CK00], we showed how flat data representations and vectorised code can be derived for functional programs operating on nested arrays of arbitrary recursive product-sum types. However, we could not flatten arrays of functions nor vectorise partial function applications. This effectively restricted the approach to first-order programs and precluded a seamless integration of nested data parallelism into Haskell or any other functional language. The present paper closes this gap. We show how to extend flattening to the higher-order case by introducing an explicit notion of closures, which leads to the concepts of *array closures* and *closure vectorisation.* Consequently, we need to apply *closure conversion* [MMH96] to a program before flattening.

In summary, our main contributions are threefold:

– a flat representation of arrays of functions and closures (Sect. 3);
– a method to combine and concatenate arrays of functions (Sect. 4); and
– a new approach to flattening nested *mapP*s (Sect. 5).

Before addressing these technical contributions, Sect. 2 explains the shortcomings of previous approaches to flattening with respect to arrays of functions and partial applications. Space constraints restrict us to an illustration of the core ideas by example in this paper; more details are available in [Les05].

## 2 Why Flattening Higher-Order Functions Is Hard?

The flattening transformation comprises (1) a *flattened array representation* and (2) code *vectorisation.* The flattened array representation turns arrays containing arbitrary tree structures into tree structures that contain arrays of *unboxed primitive* types.[1] As an example, an array of rose trees storing floats in their leaves is flattened to a list of arrays of floats. The length of the list corresponds to the height of the tallest rose tree, and each list element (i.e., array) contains the floats of one level of the original rose trees.

Code vectorisation pairs every function $f$ with a version $f^{\uparrow}$ *lifted* into vector space—think of $f^{\uparrow}$ as the data-parallel version of $f$. Code vectorisation also replaces all computations of the form *mapP* $f$ by $f^{\uparrow}$. It turns out that code vectorisation transforms a program exactly such that it can directly operate on a flattened representation of its array data.

Now for some examples. The code of the identity function $id = \lambda n.n$ remains the same after lifting, but where *id*'s type is $\alpha \rightarrow \alpha$, that of $id^{\uparrow}$ is $[:\alpha:] \rightarrow [:\alpha:]$. The increment function $inc = \lambda n.1 + n$ is slightly more interesting. Assuming its type is $Int \rightarrow Int$, the type of of $inc^{\uparrow}$ is $[:Int:] \rightarrow [:Int:]$. The code of $inc^{\uparrow}$ is

[1] Arrays of unboxed primitive types are C- or Fortran-style arrays that are essentially represented as chunks of memory full of integer and floating-point values.

$$\lambda ns.(\mathit{replicateP}\ (\mathit{lengthP}\ ns)\ 1) +^{\uparrow} ns \quad \text{, where } \mathit{replicateP}\ n\ e\ =\ [{:}e,\ \ldots,\ e{:}]$$

(i.e., $e$ repeated $n$ times) and $(+^{\uparrow})$ denotes the pairwise addition of two arrays.

Slightly more involved is the treatment of sum types. In Haskell, we have

$$\begin{array}{ll}
\mathbf{data}\ \mathit{Either}\ \alpha\ \beta = \mathit{Left}\ \alpha \mid \mathit{Right}\ \beta & \\
\mathit{either} & :: (\alpha \to \gamma) \to (\beta \to \gamma) \to \mathit{Either}\ \alpha\ \beta \to \gamma \\
\mathit{either}\ f\ g\ x & = \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Left}\ x' \to f\ x'; \mathit{Right}\ x' \to g\ x'\}
\end{array}$$

The flat representation of an array of sums $[{:}\mathit{Either}\ \alpha\ \beta{:}]$ is a product of a *selector* and two arrays containing the values of the two alternatives $([{:}\mathit{Bool}{:}], [{:}\alpha{:}], [{:}\beta{:}])$; e.g., $[{:}\mathit{Left}\ 5,\ \mathit{Right}\ 4,\ \mathit{Left}\ 2{:}]$ becomes $([{:}\mathit{True},\ \mathit{False},\ \mathit{True}{:}],\ [{:}5,\ 2{:}],\ [{:}4{:}])$.

Let us combine these two examples into a larger one:

$$\begin{array}{ll}
\mathit{foo} & :: \mathit{Either\ Int\ Int} \to \mathit{Int} \\
\mathit{foo}\ x = & \mathit{either\ inc\ id}\ x
\end{array}$$

We lift this into vector space as $\mathit{foo}^{\uparrow}\ xs\ =\ \mathit{either}^{\uparrow}\ \mathit{inc}^{\uparrow}\mathit{id}^{\uparrow}\ xs$ with

$$\mathit{either}^{\uparrow}\ f\ g\ (\mathit{sel},\ \mathit{left},\ \mathit{right})\ =\ \mathit{combineP}\ \mathit{sel}\ (f\ \mathit{left})\ (g\ \mathit{right})$$

The array primitive $\mathit{combineP} :: [{:}\mathit{Bool}{:}] \to [{:}\alpha{:}] \to [{:}\alpha{:}] \to [{:}\alpha{:}]$ merges its two $[{:}\alpha{:}]$ arguments, arranging the elements in the order determined by the selector of type $[{:}\mathit{Bool}{:}]$. For example, if we apply *foo* to $[{:}\mathit{Left}\ 5,\ \mathit{Right}\ 4,\ \mathit{Left}\ 2{:}]$, *combineP* will be invoked as $\mathit{combineP}\ [{:}\mathit{True},\ \mathit{False},\ \mathit{True}{:}]\ [{:}6,\ 3{:}]\ [{:}4{:}]$, resulting in $[{:}6,\ 4,\ 3{:}]$.

In its full glory, the story is slightly more complicated as we need to replace a function $f$ by a pair of its original and lifted version $(f, f^{\uparrow})$ and adjust all function applications by adding appropriate projections. However, these details are secondary for demonstrating the core difficulty in flattening higher-order functions; please refer to [CK00,CK03] for a more detailed introduction to flattening.

Now, let us turn to the core of the problem that we solve in this paper. We vary the example function *foo* to get

$$\begin{array}{ll}
\mathit{bar} & :: (\mathit{Either\ Int\ Int},\ \mathit{Int}) \to \mathit{Int} \\
\mathit{bar}\ (x,\ y) = & \mathit{either}\ ((+)\ y)\ \mathit{id}\ x
\end{array}$$

The first argument to *either*, namely $(+)\ y$, is a partial application of addition to the new argument $y$, replacing the previously added constant 1 with the variable $y$. It might seem as if we can lift *bar* in essentially the same way as *foo*:

$$\mathit{bar}^{\uparrow}\ (xs,\ ys) =\ \mathit{either}^{\uparrow}\ ((+^{\uparrow})\ ys)\ \mathit{id}^{\uparrow}\ xs$$

Unfortunately, we will see that this gets us into trouble quickly. We flatten an array of products $[{:}(\alpha,\ \beta){:}]$ as a product of arrays $([{:}\alpha{:}],\ [{:}\beta{:}])$. For example, $[{:}(\mathit{Left}\ 5,\ 1),\ (\mathit{Right}\ 4,\ 2),\ (\mathit{Left}\ 2,\ 7){:}]$ is represented by $(([{:}\mathit{True},\ \mathit{False},\ \mathit{True}{:}],\ [{:}5,\ 2{:}],\ [{:}4{:}]),\ [{:}1,\ 2,\ 7{:}])$. We can now calculate as follows:

$$\begin{array}{ll}
& \mathit{bar}^{\uparrow}(([{:}\mathit{True}, \mathit{False}, \mathit{True}{:}], [{:}5, 2{:}], [{:}4{:}]), [{:}1, 2, 7{:}]) \\
= & \{\text{Unfolding } \mathit{bar}^{\uparrow}\} \\
& \mathit{either}^{\uparrow}\ ((+^{\uparrow})\ [{:}1, 2, 7{:}])\ \mathit{id}^{\uparrow}\ ([{:}\mathit{True}, \mathit{False}, \mathit{True}{:}], [{:}5, 2{:}], [{:}4{:}]) \\
= & \{\text{Unfolding } \mathit{either}^{\uparrow}\} \\
& \mathit{combineP}\ [{:}\mathit{True}, \mathit{False}, \mathit{True}{:}]\ ((+^{\uparrow})\ [{:}1, 2, 7{:}]\ [{:}5, 2{:}])\ (\mathit{id}^{\uparrow}\ [{:}4{:}])
\end{array}$$

Now we have a bogus subexpression: $(+^{\uparrow})$ [:1, 2, 7:] [:5, 2:]. The length of the two arguments to $(+^{\uparrow})$ does not match! This is not only a matter of choosing an appropriate prefix of the longer array (as with Haskell's *zipWith*). We would need to evaluate $(+^{\uparrow})$ [:1, 7:] [:5, 2:] to achieve the correct result; i.e., drop the middle element of the first array. Why does the first argument have to be [:1, 7:]? Because the selector [:*True*, *False*, *True*:] determines that only the first and third element may be used by the first argument of *either*. Unfortunately, the partial application $(+^{\uparrow})$ [:1, 2, 7:] drags the entire array [:1, 2, 7:] into the body of *either*, completely disregarding the selector, instead of narrowing the array according to the selector, as in *packP* [:*True*, *False*, *True*:] [:1, 2, 7:] = [:1, 7:].

Similar problems arise from local function definitions with free variables; e.g., if we replace $(+)\ y$ in *bar* by $\lambda x'.y\ +\ x'$, we arrive at the same situation as above. As we will see next, key to solving these problems is an appropriate flattened representation of arrays of functions.

## 3 Flattening Closures

The crucial observation of last section was that $either^{\uparrow}$ needs to *packP* its functional arguments, or rather the arrays embedded in these arguments. Hence, we need to represent function values in a form that permits array operations (such as *packP*) to inspect these values and modify them. The standard method to separate code from data, and thus make data embedded in function values explicit, is *closure conversion* [MMH96]. It replaces all function values by *closures*, which are pairs of a closed binary function and its *environment*. The latter is a value for the first argument of the binary function and contains all the data items embedded in the function value represented by the closure.

With flattening, each closure contains two binary functions: the standard version and its lifted counterpart—we'll discuss later why this is crucial to our approach. Thus, we denote closures as expressions $\langle\!\langle (f, f^{\uparrow}),\ e \rangle\!\rangle$ of type $\alpha \Rightarrow \beta$ if $f :: (\gamma, \alpha) \rightarrow \beta$, $f^{\uparrow} :: ([:\gamma:],\ [:\alpha:]) \rightarrow [:\beta:]$ and $e :: \gamma$; e.g., we represent $(+)\ 1$ by the closure $\langle\!\langle (\lambda(v_1, v_2).v_1\ +\ v_2,\ \lambda(vs_1, vs_2).vs_1 +^{\uparrow} vs_2),\ 1 \rangle\!\rangle$, or just $\langle\!\langle ((\underline{+}),\ (\underline{+}^{\uparrow})),\ 1 \rangle\!\rangle$, where $\underline{f} :: (\alpha,\ \beta) \rightarrow \gamma$ is the uncurried version of $f :: \alpha \rightarrow \beta \rightarrow \gamma$. The application of a closure of type $\alpha \Rightarrow \beta$ to a value $x :: \alpha$ is defined as $\langle\!\langle (f, f^{\uparrow}),\ e \rangle\!\rangle \dagger x\ =\ f\ (e,\ x)$.

To solve our problem with lifting partial applications, we need a suitable representation of partial applications of lifted functions, such as $(+^{\uparrow})$ [:1, 2, 7:]. Is a closure of the form $\langle\!\langle ((\underline{+}^{\uparrow}),\ (\underline{+}^{\uparrow\uparrow})),\ [:1,\ 2,\ 7:] \rangle\!\rangle$, where $(\underline{+}^{\uparrow\uparrow})$ is addition lifted twice, sufficient? Unfortunately, no! In particular, it prevents typing closure converted, flattened programs. After all, the type of *packP* is $[:Bool:] \rightarrow [:\alpha:] \rightarrow [:\alpha:]$; so, it's second argument should be an array. Hence, if we want to apply *packP* to a closure value, that value should better be the flattened representation of an array type. Not surprisingly, it turns out that packable closures are the flattened representation of arrays of functions. They differ slightly from vanilla closures, as their environment is always an array. In other words, an *array closure* has the form $\langle\!\langle :(f,\ f^{\uparrow}),\ es: \rangle\!\rangle$ of type $\alpha \Rrightarrow \beta$ if $f :: (\gamma, \alpha) \rightarrow \beta$, $f^{\uparrow} :: ([:\gamma:], [:\alpha:]) \rightarrow [:\beta:]$, and $es :: [:\gamma:]$. The type $\alpha \Rrightarrow \beta$ represents arrays of standard closures $[:\alpha \Rightarrow \beta:]$.

For example, we represent both the partial application $(+^{\uparrow})$ $[{:}1,\ 2,\ 7{:}]$ and the local function $\lambda xs'.ys\ +^{\uparrow}\ xs'$, where $ys = [{:}1,\ 2,\ 7{:}]$, by the array closure $\langle\!\langle{:}((\underline{+}),\ (\underline{+}^{\uparrow})),\ [{:}1,\ 2,\ 7{:}]{:}\rangle\!\rangle$. The closure's environment fixes the length of arrays acceptable as the second argument to $(\underline{+}^{\uparrow})$. We define the application of an array closure of type $\alpha \Rrightarrow \beta$ to a value $xs :: [{:}\alpha{:}]$ as $\langle\!\langle{:}(f,\ f^{\uparrow}),\ es{:}\rangle\!\rangle \ddagger xs\ =\ f^{\uparrow}\ (es,\ xs)$.

As we will see in the rest of the paper, this representation directly supports the full range of array operations. For instance, standard closures can be replicated to array closures simply by replicating the environment:

$$\mathit{replicateP}\ n\ \langle\!\langle(f, f^{\uparrow}),\ e\rangle\!\rangle\ =\ \langle\!\langle{:}(f, f^{\uparrow}),\ \mathit{replicateP}\ n\ e{:}\rangle\!\rangle$$

Conversely, we obtain a standard closure from an array closure by indexing:

$$\langle\!\langle{:}(f, f^{\uparrow}), es{:}\rangle\!\rangle\ !{:}\ n\ =\ \langle\!\langle(f, f^{\uparrow}),\ es\ !{:}\ n\rangle\!\rangle \qquad \text{-- !: is indexing of parallel arrays}$$

NB: To convert between standard closures and array closures, it is crucial that both closure types include the standard and the lifted version of the function.

Pivotal to our running example is the ability to *packP* array closures:

$$\mathit{packP}\ bs\ \langle\!\langle{:}(f, f^{\uparrow}),\ es{:}\rangle\!\rangle\ =\ \langle\!\langle{:}(f, f^{\uparrow}),\ \mathit{packP}\ bs\ es{:}\rangle\!\rangle$$

More generally, an array closure $\langle\!\langle{:}(f, f^{\uparrow}),\ es{:}\rangle\!\rangle$ represents $\mathit{mapP}\ (\mathit{curry}\, f)\ es$ and, therefore, satisfies the usual parametric *map* laws [Wad89]. Thus, a wide range of polymorphic array operations, including permutations, can be implemented analogously to *packP* by propagating the operation to the environment.

Now, let us return to the problematic function *bar* of the previous section. This time, we replace the two function arguments to $\mathit{either}^{\uparrow}$ by array closures:

$$\begin{aligned}
&\mathit{bar}^{\uparrow}\ (xs,\ ys)\ =\ \mathit{either}^{\uparrow}\ \langle\!\langle{:}((\underline{+}),\ (\underline{+}^{\uparrow})),\ ys{:}\rangle\!\rangle\ \langle\!\langle{:}(id_0,\ id_0^{\uparrow}),\ [{:}(),\ (),\ (){:}]{:}\rangle\!\rangle\ xs\\
&\qquad \textbf{where}\quad id_0\ =\ \lambda(v_1, v_2).v_2\\
&\qquad\qquad\qquad id_0^{\uparrow}\ =\ \lambda(vs_1, vs_2).vs_2
\end{aligned}$$

and re-define $\mathit{either}^{\uparrow}$ such that it packs its first two arguments:

$$\begin{aligned}
&\mathit{either}^{\uparrow}\ f\ g\ (\mathit{sel},\ \mathit{left},\ \mathit{right})\ =\\
&\qquad \mathit{combineP}\ \mathit{sel}\ ((\mathit{packP}\ \mathit{sel}\, f)\ \ddagger\ \mathit{left})\ ((\mathit{packP}\ (\mathit{mapP}\ \mathit{not}\ \mathit{sel})\ g)\ \ddagger\ \mathit{right})
\end{aligned}$$

Now, we can redo the calculation for the *bar* example:

$$\begin{aligned}
&\quad \mathit{bar}^{\uparrow}(([{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}], [{:}5, 2{:}], [{:}4{:}]), [{:}1, 2, 7{:}])\\
&= \{\text{Unfolding } \mathit{bar}^{\uparrow}\}\\
&\quad \mathit{either}^{\uparrow}\ \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1, 2, 7{:}]{:}\rangle\!\rangle\ \langle\!\langle{:}(id_0, id_0^{\uparrow}), [{:}(), (), (){:}]{:}\rangle\!\rangle\\
&\qquad ([{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}], [{:}5, 2{:}], [{:}4{:}])\\
&= \{\text{Unfolding } \mathit{either}^{\uparrow}\}\\
&\quad \mathit{combineP}\ [{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}]\\
&\qquad ((\mathit{packP}\ [{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}]\ \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1, 2, 7{:}]{:}\rangle\!\rangle)\ \ddagger\ [{:}5, 2{:}])\\
&\qquad ((\mathit{packP}\ [{:}\mathit{False}, \mathit{True}, \mathit{False}{:}]\ \langle\!\langle{:}(id_0, id_0^{\uparrow}), [{:}(), (), (){:}]{:}\rangle\!\rangle)\ \ddagger\ [{:}4{:}])\\
&= \{\text{Applying } \mathit{packP}\}\\
&\quad \mathit{combineP}\ [{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}]\\
&\qquad \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1, 7{:}]{:}\rangle\!\rangle\ \ddagger\ [{:}5, 2{:}])\ (\langle\!\langle{:}(id_0, id_0^{\uparrow}), [{:}(){:}]{:}\rangle\!\rangle\ \ddagger\ [{:}4{:}])\\
&= \{\text{Unfolding } \ddagger\}\\
&\quad \mathit{combineP}\ [{:}\,\mathit{True}, \mathit{False}, \mathit{True}{:}]\ ((\underline{+}^{\uparrow})\ ([{:}1, 7{:}], [{:}5, 2{:}]))\ (id_0^{\uparrow}\ ([{:}(){:}], [{:}4{:}]))
\end{aligned}$$

$$\begin{aligned}
&= \{\text{Applying } \underline{+}^{\uparrow} \text{ and } id_0^{\uparrow}\} \\
&\quad \mathit{combineP}\ [{:}\mathit{True}, \mathit{False}, \mathit{True}{:}]\ [{:}6, 9{:}]\ [{:}4{:}] \\
&= \{\text{Applying } \mathit{combineP}\} \\
&\quad [{:}6, 4, 9{:}]
\end{aligned}$$

In summary, to flatten higher-order programs, we first apply *closure conversion* to make closures explicit, hence admitting the manipulation of closures as first-class values. All remaining function values (which now only occur inside closures) are supercombinators; i.e, closed functions, which also only use supercombinators inside. Moreover, we use array closures as the representation of arrays of functional values, so that we can perform standard array operations, such as *packP*, on lifted partial applications and arrays of functions.

## 4 Combining Closures

Data parallelism dictates that an array closure $\langle\!\langle{:}(f, f^{\uparrow}),\ e{:}\rangle\!\rangle$ represents the partial application of a single function to multiple arguments. Many array operations, such as *replicateP*, *mapP*, and *packP*, maintain this property, but unfortunately some don't. For instance, consider

$$\mathit{combineP}\ [{:}\mathit{True}, \mathit{False}, \mathit{True}{:}]\ \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1, 2{:}]{:}\rangle\!\rangle\ \langle\!\langle{:}((\underline{*}), (\underline{*}^{\uparrow})), [{:}3{:}]{:}\rangle\!\rangle$$

The result is of type $\mathit{Int} \Rrightarrow \mathit{Int}$ and represents the partial applications of both addition and multiplication. In a data parallel environment, can we express this as a partial application of a *single* function?

The following observation suggests a solution: Elementwise application of a combined closure to an argument array is equivalent to splitting the argument array, applying the two closures individually, and combining the results:

$$\begin{aligned}
&\quad (\mathit{combineP}\ [{:}\mathit{True}, \mathit{False}, \mathit{True}{:}]\ \langle\!\langle{:}((+), (+^{\uparrow})), [{:}1, 2{:}]{:}\rangle\!\rangle\ \langle\!\langle{:}((*), (*^{\uparrow})), [{:}3{:}]{:}\rangle\!\rangle) \\
&\quad \ddagger\ [{:}4, 5, 6{:}] \\
&= \{\text{Splitting the argument array}\} \\
&\quad \mathit{combineP}\ [{:}\mathit{True}, \mathit{False}, \mathit{True}{:}] \\
&\qquad (\langle\!\langle{:}((+), (+^{\uparrow})), [{:}1, 2{:}]{:}\rangle\!\rangle \ddagger [{:}4, 6{:}])\ (\langle\!\langle{:}((*), (*^{\uparrow})), [{:}3{:}]{:}\rangle\!\rangle \ddagger [{:}5{:}]) \\
&= \{\text{Merging the results}\} \\
&\quad [{:}5, 15, 8{:}]
\end{aligned}$$

This scheme is easily encoded by a closure $\langle\!\langle{:}(\mathit{capp}, \mathit{capp}^{\uparrow}), \mathit{es}{:}\rangle\!\rangle$ whose environment *es* contains the two original closures along with the selector. The function $\mathit{capp}^{\uparrow}$, defined below, uses the selector to split the argument and combines the result of applying the two closures.

Crucially, this approach fits nicely with the rest of the flattening transformation, as the environment can be directly represented by a parallel array of type $[{:}\mathit{Either}\ (\mathit{Int} \Rightarrow \mathit{Int})\ (\mathit{Int} \Rightarrow \mathit{Int}){:}]$. As described in Section 2, this is flattened to the product $([{:}\mathit{Bool}{:}], \mathit{Int} \Rrightarrow \mathit{Int}, \mathit{Int} \Rrightarrow \mathit{Int})$ which stores precisely the selector and the two closures and is constructed by *combineP* as follows:

$$\mathit{combineP}\ \mathit{bs}\ \mathit{fs}\ \mathit{gs}\ =\ \langle\!\langle{:}(\mathit{capp}, \mathit{capp}^{\uparrow}), (\mathit{bs}, \mathit{fs}, \mathit{gs}){:}\rangle\!\rangle$$

where we define the split/combine procedure as follows

$$\begin{aligned}&\mathit{capp}^{\uparrow} :: (([:\mathit{Bool}:], \alpha \Rrightarrow \beta, \alpha \Rrightarrow \beta), [:\alpha:]) \to [:\beta:]\\&\mathit{capp}^{\uparrow}\ ((\mathit{bs}, \mathit{fs}, \mathit{gs}), \mathit{xs}) =\\&\qquad \mathit{combineP}\ \mathit{bs}\ (\mathit{fs} \ddagger \mathit{packP}\ \mathit{bs}\ \mathit{xs})\ (\mathit{gs} \ddagger \mathit{packP}\ (\mathit{mapP}\ \mathit{not}\ \mathit{bs})\ \mathit{xs})\end{aligned}$$

Our definition of $\mathit{capp}^{\uparrow}$ sequentialises the two applications of *fs* and *gs*. This is a natural consequence of data parallelism, as *fs* and *gs* will generally be entirely different computations. This corresponds closely to $\mathit{either}^{\uparrow}$ from Section 3.

The result of *combineP* is a regular array closure and, thus, naturally supports all array operations. For instance, packing clearly has the desired semantics by propagating the operation into the environment. Moreover, indexing extracts a single element from the environment by, depending on the index, selecting it from either of the two closures. Depending on which closure is selected, the result will be embedded into a *Left* or *Right* constructor. Applying the resulting closure to an argument invokes *capp* which is implemented as

$$\begin{aligned}&\mathit{capp} &&:: (\mathit{Either}\ (\alpha \Rightarrow \beta)\ (\alpha \Rightarrow \beta),\ \alpha) \to \beta\\&\mathit{capp}\ (h,\ x) &&= \textbf{case}\ h\ \textbf{of}\ \{\mathit{Left}\ f\ \to\ f \dagger x;\ \mathit{Right}\ g\ \to\ g \dagger x\}\end{aligned}$$

In fact, $\mathit{capp}^{\uparrow}$ is, as expected, precisely the lifted version of *capp*; i.e., we can derive $\mathit{capp}^{\uparrow}$ from *capp* by vectorisation.

This idea can be extended to other joining operations, such as concatenation, which is a special case of combining, but we only have space to discuss *combineP*.

## 5 Nesting Closures

The purpose of the flattening transformation is the elimination of nested parallelism, of which the archetypal example is the nested application of *mapP*, as in $\mathit{mapP}\ (\mathit{mapP}\ (1+))\ [:[:1,2:], [::], [:3,4,5:]:]$, which increments every integer element while preserving the array's nesting structure. As discussed in Section 1, the nested array of our example is implemented by the flattened representation $([:2,0,3:], [:1,2,3,4,5:])$, where the the segment descriptor $[:2,0,3:]$ encodes the nesting structure. In the flat representation, we can increment all integer elements without affecting the nesting structure by simply applying the non-nested $\mathit{mapP}\ (+1)$ to the second component of the representation. So, we have

$$\mathit{mapP}\ (\mathit{mapP}\ f)\ (\mathit{segd},\ \mathit{xs})\ =\ (\mathit{segd},\ \mathit{mapP}\ f\ \mathit{xs})$$

In the following, we discuss how to realise this with array closures.

After closure conversion, *mapP* expects a closure as its first argument, which we simply replicate to get an array closure that we can apply as follows:

$$\begin{aligned}&\mathit{mapP} &&:: (\alpha \Rightarrow \beta) \to [:\alpha:] \to [:\beta:]\\&\mathit{mapP}\ c\ \mathit{xs} &&= \mathit{replicateP}\ (\mathit{lengthP}\ \mathit{xs})\ c \ddagger \mathit{xs}\end{aligned}$$

The partial application $\mathit{mapP}\ (1+)$ in our nested example ultimately evaluates to $\langle\!\langle(\underline{\mathit{mapP}}, \underline{\mathit{mapP}^{\uparrow}}), \langle\!\langle((\underline{+}), (\underline{+^{\uparrow}})), 1\rangle\!\rangle\rangle\!\rangle$, which we map over the nested array:

$$\begin{aligned}&\quad \mathit{mapP}\ (\langle\!\langle(\underline{\mathit{mapP}}, \underline{\mathit{mapP}^{\uparrow}}), \langle\!\langle((\underline{+}), (\underline{+^{\uparrow}})), 1\rangle\!\rangle\rangle\!\rangle)\ ([:2,0,3:], [:1,2,3,4,5:])\\&= \{\text{Unfolding } \mathit{mapP}\}\\&\quad \mathit{replicateP}\ 3\ \langle\!\langle(\underline{\mathit{mapP}}, \underline{\mathit{mapP}^{\uparrow}}), \langle\!\langle((\underline{+}), (\underline{+^{\uparrow}})), 1\rangle\!\rangle\rangle\!\rangle \ddagger ([:2,0,3:], [:1,2,3,4,5:])\\&= \{\text{Unfolding } \mathit{replicateP} \text{ twice}\}\end{aligned}$$

$$
\begin{aligned}
&\langle\!\langle{:}(\underline{mapP}, \underline{mapP^{\uparrow}}), \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1,1,1{:}]{:}\rangle\!\rangle{:}\rangle\!\rangle \ddagger ([{:}2,0,3{:}], [{:}1,2,3,4,5{:}])\\
= &\{\text{Unfolding } \ddagger\}\\
&\underline{mapP^{\uparrow}}\ (\langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1,1,1{:}]{:}\rangle\!\rangle, ([{:}2,0,3{:}], [{:}1,2,3,4,5{:}]))
\end{aligned}
$$

Not surprisingly, the two nested applications of *mapP* are resolved to a single application of $mapP^{\uparrow}$. The implementation of the latter can be derived by lifting the definition of *mapP*; here, we present a slightly simplified version:

$$
\begin{aligned}
&mapP^{\uparrow}\ :: (\alpha \Rrightarrow \beta) \to ([{:}Int{:}], [{:}\alpha{:}]) \to ([{:}Int{:}], [{:}\beta{:}])\\
&mapP^{\uparrow}\ \langle\!\langle{:}(f, f^{\uparrow}), es{:}\rangle\!\rangle\ (segd, xs)\ =\ (segd,\ \langle\!\langle{:}(f, f^{\uparrow}),\ expandP\ segd\ es{:}\rangle\!\rangle \ddagger xs)\\
&\quad \textbf{where}\\
&\qquad expandP\ [{:}n_1, \ldots, n_k{:}]\ [{:}x_1, \ldots, x_k{:}]\ =\ [{:}\underbrace{x_1, \ldots, x_1}_{n_1 \text{ times}}, \ldots, \underbrace{x_k, \ldots, x_k}_{n_k \text{ times}}{:}]
\end{aligned}
$$

Conceptually, the mapped array closure contains one element for each subarray of the nested array. Hence, we blow the array closure up, such that each element is replicated as often as the length of the corresponding subarray dictates—this is the job of *expandP*. Afterwards, we can simply apply the expanded array closure to the representation of the nested array. The nesting structure remains invariant under this operation. Returning to our example, we now have:

$$
\begin{aligned}
&mapP^{\uparrow}\ \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1,1,1{:}]{:}\rangle\!\rangle\ ([{:}2,0,3{:}], [{:}1,2,3,4,5{:}])\\
= &\{\text{Unfolding } mapP^{\uparrow}\}\\
&([{:}2,0,3{:}],\ \langle\!\langle{:}((\underline{+}), (\underline{+}^{\uparrow})), [{:}1,1,1,1,1{:}]{:}\rangle\!\rangle \ddagger [{:}1,2,3,4,5{:}])\\
= &\{\text{Unfolding } \ddagger\}\\
&([{:}2,0,3{:}],\ [{:}2,3,4,5,6{:}])
\end{aligned}
$$

As expected, the result is the flat representation of $[{:}[{:}2,3{:}], [{:}{:}], [{:}4,5,6{:}]{:}]$; moreover, it has been computed in one parallel step.

## References

[Ble90] G. E. Blelloch. *Vector Models f. Data-Parallel Computing.* MIT Press, 1990.

[Ble96] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, 1996.

[CK00] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In P. Wadler, editor, *5th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.

[CK03] M. M. T. Chakravarty and G. Keller. An approach to fast arrays in Haskell. In J. Jeuring and S. Peyton Jones, editors, *Lecture notes for The Summer School on Advanced Functional Programming '02*, LNCS 2638, 2003.

[CKLP01] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal—Nested data parallelism in Haskell. In R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing*, number 2150, pages 524–534. Springer-Verlag, 2001.

[Les05] R. Leshchinskiy. *Higher-Order Nested Data Parallelism: Semantics and Implementation.* PhD thesis, Technische Universität Berlin, 2005.

[MMH96] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *POPL '96: Proc. of the 23rd ACM SIGPLAN-SIGACT Sym. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[Wad89] P. Wadler. Theorems for free! In *FPCA'89: Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, New York, 1989. ACM Press.