Defunctionalizing Push Arrays

Bo Joel Svensson
Indiana University
joelsven@indiana.edu

Josef Svenningsson
Chalmers University of Technology
josefs@chalmers.se

Abstract

Recent work on embedded domain specific languages (EDSLs) for high performance array programming has given rise to a number of array representations. In Feldspar and Obsidian there are two different kinds of arrays, called Pull and Push arrays. Both Pull and Push arrays are deferred; they are methods of computing arrays, rather than elements stored in memory. The reason for having multiple array types is to obtain code that performs better. Pull and Push arrays provide this by guaranteeing that operations fuse automatically. It is also the case that some operations are easily implemented and perform well on Pull arrays, while for some operations, Push arrays provide better implementations. But do we really need to have more than one array representation? In this paper we derive a new array representation from Push arrays that have all the good qualities of Pull and Push arrays combined. This new array representation is obtained via defunctionalization of a Push array API.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Recent developments in high performance functional array programming has given rise to two complementary array representations, Pull and Push arrays. Pull and Push arrays are both deferred; they represent two different ways to produce values. A Pull array is a function from an indices to a values (Figure 1). A consumer of a Pull array iterates over the indices of interest and applies the function to each of those indices. Push arrays, on the other hand, encode their own iteration schema (Figure 2). A consumer of a Push array provides a *write-function* that is instantiated into the body of the Push array's predefined iteration schema.

The reason there are two types of arrays is that they complement each other. Some operations, like indexing, can be implemented efficiently for Pull arrays but not for Push arrays. Other operations, such as concatenation, are more efficient on Push arrays compared to Pull arrays. Pull and Push array implementation details are shown in sections 2.2 and 2.3.

FHPC'14, September 04, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3040-4/14/09.
http://dx.doi.org/10.1145/2636228.2636231

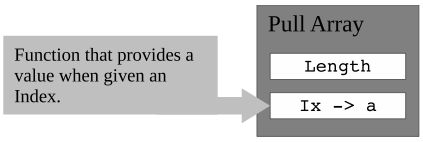

Figure 1. Pull array: a length and a function from index to value.

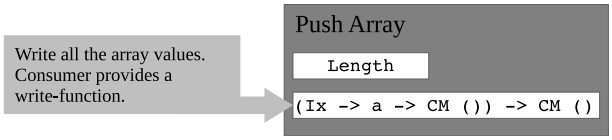

Figure 2. Push array: a higher order function that accepts a mechanism for writing (`Ix -> a -> CM ()`) an element to memory. `CM` is the code generating monad used throughout this paper.

Push and Pull arrays have the following properties:

- They are easily parallelizable. It is straightforward to generate efficient, parallel code from Pull and Push arrays, making them suitable for inclusion in high performance DSLs.
- They allow for a compositional programming style. It is easy to formulate high level, reusable combinators operating on Pull and Push arrays.
- Operations fuse aggressively. When composing two array functions the intermediate array is guaranteed to be fused away and not allocated in memory at runtime. This fusion takes place even when resulting in work duplication. The programmer is, however, given tools to control explicitly when fusion should not take place.

Despite these desirable properties, Push and Pull arrays have the following disadvantages: Given for example a Push array, it is not possible to inspect its history, how it was created. Any operation on a Push array must be implemented using its functional representation, and is effectively limited to applying it to a write-function to obtain its elements or composing operations with the function representing the Push array. If, however, there was a concrete representation of Push arrays that could be traversed and analyzed, new Push array functionality could be obtained. Furthermore, If this concrete representation supported the implementation of an indexing function that avoided computing the entire array of elements, there would be no need for a separate Pull array type. Finding this concrete representation means providing a deep embedding of Push arrays.

Fortunately there is a proven method for obtaining precisely this kind of concrete representation; it is called *defunctionalization* [23]. Applying defunctionalization to a Push array API provides a data type and an associated compiler semi-automatically (Figure

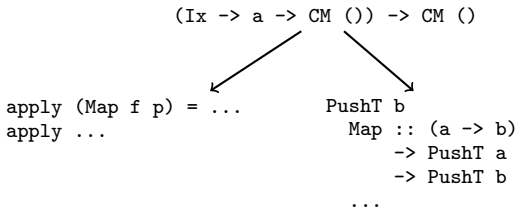

Figure 3. Using defunctionalization to go from a higher order function to a data type and apply function (compiler).

3). Moreover, the compiler gives the same fusion guarantees as the functional representation of Push arrays have.

This paper presents a unified array library which inherits the benefits of Pull and Push arrays, yet has only a single type of array.

- We present a single array type which replaces both Pull and Push arrays (section 5).
- We show how to derive our new array type by applying *defunctionalization* on push arrays (section 4).
- Our array library can support all known safe operations on Pull and Push arrays. See section 7.2 for a discussion.
- We present our new array library in the context of an embedded code-generating DSL (section 2.1). Using a code-generating DSL makes it easy to demonstrate that operations fuse. Section 7.1 outlines how to achieve fusion for our library in the context of Haskell.

Before presenting the contributions we will give an introduction to Pull and Push arrays in section 2.2 and 2.3. We will also review defunctionalization in section 3.

2. Background

This paper presents Pull and Push arrays as part of a small code-generating embedded language, a compiled EDSL. This section gives enough background material to make the paper self-contained. This section contains no new material, the embedded language techniques closely resemble those in the seminal "Compiling Embedded Languages" paper [15].

2.1 Embedded Code-Generating DSL

The embedded language used throughout this paper is compiled into a small imperative language with for loops and memory operations (allocate, write), a simple C-like language. The `Code` data type below is used to represent compiled programs.

```
type Id = String

data Code = Skip
          | Code :>>: Code
          | For Id Exp Code
          | Allocate Id Length
          | Write Id Exp Exp
```

The `:>>:` constructor is for sequential composition of programs. The empty program, `Skip` and `:>>:` allows a `Monoid` instance for `Code`.

There are also scalar expressions in the target language. These are represented by a data type called `Exp`. Here all the usual arithmetic operations, indexing and expression level conditionals are found.

```
-- Memory operations
write :: Expable a => CMMem a -> Ix -> a -> CM ()

cmIndex :: Expable a => CMMem a -> Ix -> a

-- for loop
for_ :: Expable a => Expr Int -> (a -> CM ()) -> CM ()
```

Figure 4. Memory operations and flow control

```
data Value = IntVal Int
           | FloatVal Float
           | BoolVal Bool

data Exp = Var Id
         | Literal Value
         | Index Id Exp
         | Exp :+: Exp
         | Exp :-: Exp
         | Exp :*: Exp
         | Mod Exp Exp
         | Div Exp Exp
         | Eq  Exp Exp
         | Gt  Exp Exp
         | LEq Exp Exp
         | Min Exp Exp
         | IfThenElse Exp Exp Exp
```

Phantom types provide a typed interface to expressions, while it allows a simple (not GADT based) implementation. Again this follows the presentation of the "Compiling Embedded Languages" paper [15].

```
data Expr a = E {unE :: Exp}
```

An `Expable` class sets up conversions to and from the `Exp` data type. Compilation will require that data elements are an instance of this class.

```
class Expable a where
  toExp :: a -> Exp
  fromExp :: Exp -> a
```

The Haskell class system is also used to overload the common arithmetic operators. For the expressions in this EDSL, only instances of the `Num` class are provided, overloading `+`, `-` and `*`. Other functions are given names identical to their Haskell counterpart but ending with an underscore, such as `mod_` and `div_`. Conditionals are expressed using an `ifThenElse` function. Finally, the comparison operators with boolean result are `<=*`, `==*` and `>*`.

Compilation down to `Code` is performed within a monad, a *Compile Monad* (CM).

```
newtype CM a = CM (StateT Integer (Writer Code) a)
  deriving (Monad,
            MonadState Integer,
            MonadWriter Code)
```

`CM` is a `Code` generating monad implemented as a writer/state combination. The state is used to construct new identifiers, while the writer accumulates generated code (requiring that `code` is a monoid). Figure 4 shows type signatures of some essential functions that are used in the rest of the paper. The `CMMem` data type represents arrays in memory and is defined as a length and an identifier.

The `Code` data type can be seen as a deeply embedded core language. For the purposes of this paper, consider `Code` the compiler result even though more steps are needed to actually run the code on an actual machine.

```
map :: (a -> b) -> Pull a -> Pull b
map f (Pull l ixf) = Pull l (f . ixf)

index :: Pull a -> Ix -> a
index (Pull _ ixf) i = ixf i

zipWith :: (a -> b -> c) -> Pull a -> Pull b -> Pull c
zipWith f (Pull l1 ixf1) (Pull l2 ixf2) =
  Pull (min l1 l2) (\i -> f (ixf1 i) (ixf2 i))

halve :: Pull a -> (Pull a, Pull a)
halve (Pull l ixf) = (Pull l2 ixf, Pull (l - l2) ixf')
  where l2     = l `div_` 2
        ixf' i = ixf (i + l2)

reverse :: Pull a -> Pull a
reverse (Pull l ixf) =
  Pull l (\ix -> ixf (l - 1 - ix))

rotate :: Length -> Pull a -> Pull a
rotate r (Pull l ixf) =
  Pull l (\ix -> ixf ((ix+r) `mod_` l))
```

Figure 5. Examples of operations on Pull arrays

On top of the deeply embedded `Code` language, the two array representations (Pull and Push) are implemented as shallow embeddings [24]. Lengths and indices for these arrays are represented by integer expressions.

```
type Length = Expr Int
type Ix = Expr Int
```

The following sections will show the implementation and properties of Pull and Push arrays.

2.2 Pull Arrays

A well known, and often used, representation of arrays is as a function from index to value. A value at a given index of an array is computed by applying the function to that given index. That is, pulling at an index provides a value. We refer to arrays that are implemented using this representation as *Pull* arrays.

```
data Pull a = Pull Length (Ix -> a)
```

Examples of operations on Pull arrays are shown in Figure 5. A notable thing about Pull arrays and its functions is that they are non-recursive. This has the effect that it is very easy to fuse Pull arrays such that any intermediate array is removed, in particular in the context of embedded DSLs [24]. As an example, consider the expression `zipWith f (map g a1) (map h a2)` where `a1` and `a2` are Pull arrays. When this expression is evaluated by the Haskell runtime the definition of the individual functions will be unfolded as follows:

```
   zipWith f (map g (Pull l1 ixf1)) (map h (Pull l2 ixf2))
=> zipWith f (map g (Pull l1 ixf1)) (Pull l2 (h . ixf2))
=> zipWith f (Pull l1 (g . ixf1)) (Pull l2 (h . ixf2))
=> Pull (min l1 l2) (\i -> f (g (ixf1 i)) (h (ixf2 i)))
```

The end result is a single Pull array and all the intermediate arrays have been eliminated.

Pull arrays are not named areas of memory containing data. Instead, a Pull array describes how the values can be computed. However, sometimes it is necessary to be able to compute and store the array as data in memory, for example to facilitate sharing of computed results. This can be done by adding a `force` primitive to the Pull array API.

```
force :: Expable a => Pull a -> CM (Pull a)
```

The force function is monadic (in the `CM` monad). Using the `force` function accumulates code into the writer part of the monad. The code accumulated is a program that iterates over, computes and stores all the elements of the input array to memory. Using `force` is the only way to stop operations on Pull arrays from fusing. The array returned from `force` contains the exact same data as the input, only they are now represented in memory.

2.3 Push Arrays

Push arrays are a complement to Pull arrays, first introduced in Obsidian [10]. Since then, Push arrays have also been implemented in Feldspar[1], in Nikola[2] and in meta-repa [5]. Push arrays also appear in other settings; for example, Kulkarni and Newton use push arrays as part of an embedded language for stream processing [20].

Push arrays were introduced in Obsidian and Feldspar in order to deal with very specific performance issues. In particular, array concatenation and interleaving introduces conditionals in the Pull array indexing function. When forcing an array, such conditionals can lead to bad performance on both CPUs and GPUs.

The example below shows a situation that may occur when working with pull arrays and concatenation. The code on the left executes a conditional in every iteration of the loop body. To the right, the loop is split into two separate loops, neither containing a conditional.

```
for i in 0..(m + n-1)          | for i in 0..(m-1)
  data[i] = if (i < m)         |   data[i] = ...
            then ...           | for i in 0..(n-1)
            else ...           |   data[i+m] = ...
```

Many optimizing compilers would be able to perform this particular transformation, so a natural question is whether one could rely on them instead of having two array representations. The problem is that the optimizations are heuristics by necessity and there will always be cases where they fail to transform the loop into the desired form. By using Push arrays the programmer can be certain that the right loop structure is generated, and doesn't have to second-guess the optimizer.

Another example, that occurs when flattening an array of pairs, is a loop which executes twice as many times as the array of pairs is long. In each iteration it selects the first or second component of the pair depending on whether the index is even or not.

```
for i in 0..(2*n-1)            | for i in 0..(n-1)
  data[i] = if even(i)         |   data[2*i] = fst(...)
            then fst(...)      |   data[2*i+1] = snd(...)
            else snd(...)      |
```

When working with Pull arrays, the loop structures to the left in the examples above are obtained. However, the code on the right is preferred. By switching to Push arrays the loop structures on the right can also be implemented.

Push arrays move the responsibility of setting up the iteration schema from the consumer (as with Pull arrays) to the producer. This provided, concatenation, interleaving and pair flattening can be given more efficient Push array implementations.

Just like the Pull arrays, Push arrays are added to the language as a shallow embedding.

```
data Push a = Push ((Ix -> a -> CM ()) -> CM ()) Length
```

[1] github.com/Feldspar/feldspar-language

[2] github.com/mainland/nikola/blob/master/src/Data/Array/Nikola/Repr/Push.hs

The Push array is a higher order function whose result is a monadic computation. As input, this higher order function takes a write-function (`Ix -> a -> CM ()`), that represents a way to consume the elements produced (for example by writing them to memory). In essence a Push array is a function accepting a continuation.

2.4 Push Array Library Functions

Figure 6 lists the Push array API used as basis for the defunctionalization in the upcoming sections. The selection of functions in the API is based on our experience with Push arrays from working with embedded languages.

Note that there is no arbitrary permutation, `ixMap`, in the library.

```
ixMap :: (Ix -> Ix) -> Push a -> Push a
ixMap f (Push p l) =
  Push (\k -> p (\i a -> k (f i) a)) l
```

This, somewhat naïve, `ixMap` function is dangerous and can lead to uninitialized elements in the resulting array, or race conditions where more than one element is written to the same location in the array. Instead we argue for using a set of fixed permutations, such as `reverse` and `rotate`. For the discussion in section 5, it is also important that these permutations are invertible.

2.5 Pull and Push Array Interplay

Pull and Push arrays complement one another and when programming it is nice to have both. Some functions are efficient and intuitive on Pull arrays. Function such as `zipWith` and `halve`, are *Pully* in nature, while `(++)` is more efficient on Push arrays, call it *Pushy*.

Converting a Pull array to a Push array is cheap and is also subject to fusion. The function `push` below implements this conversion.

```
push :: Pull a -> Push a
push (Pull n ixf) =
  Push (\k -> for_ n $ \i -> k i (ixf i)) n
```

Converting a Push array to a Pull array, however, requires computing and storing all elements to memory. The function `pull`, implemented below, is an example of this.

```
pull :: Push a -> CM (Pull a)
pull (Push p n) =
  do arr <- allocate n
     p $ write arr
     return $ Pull n (\i -> cmIndex arr i)
```

This encourages the following pattern when programming with Pull and Push arrays: A function takes one or several Pull arrays as arguments. These arrays are split apart and some processing is done on the individual parts. As a final step the arrays are assembled together again and produces a Push array. This Push array can then be stored to memory which then can be read back as a Pull array again, if needed. Since memory accesses are an important factor in application performance on many platforms, the number of Push to Pull conversions can be used as a crude indicator of performance. Few such conversions is likely to be better.

An example of this pattern can be seen below in the function `halfCleaner` below. For an array of size 8 it performs compare and swap on pairs of elements at the following positions (0,4), (1,5), (2,6) and (3,7). This is achieved by first splitting the array in half. Then the two halves are zipped together and compare and swap is performed on the pairs. Finally, the new halves are unzipped and concatenated together, thereby producing the final array.

A half cleaner is an integral part in bitonic sorts and a similar pattern can be used to implement the butterfly network in FFT.

```
swap (a,b) = IfThenElse (a <=* b) (a,b) (b,a)

halfCleaner :: Pull (Expr a) ->
               Push (Expr a)
halfCleaner =
  uncurry (++) . unzip . map swap . uncurry zip . halve
```

Note that the splitting and zipping must be done on a Pull array as those operations cannot be efficiently implemented using Push arrays. Mapping and unzipping can be done on either representation but in this case it is done on Pull arrays. Only the last step, concatenation, results in a Push array.

While Pull and Push arrays work well together and form powerful abstractions for expression and control of computations, having just one array representation is alluring.

3. Defunctionalization

Defunctionalization is a program transformation introduced by Reynolds [23]. It is used to convert functions to first order data types which means it can be used as a way to implement higher order languages. We work in a typed setting and will follow the presentation of Pottier and Gauthier [22].

To illustrate defunctionalization we present a small example, originally due to Olivier Danvy [13]. The following program flattens trees into lists. A naïve flattening function has a worst case quadratic time complexity, because of nested calls to append. The version below is linear by using the standard trick of John Hughes [18] to represent lists as functions from lists to lists.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

cons :: a -> ([a] -> [a])
cons x = \xs -> x : xs

o :: (b -> c) -> (a -> b) -> a -> c
f `o` g = \x -> f (g x)

flatten :: Tree t -> [t]
flatten t = walk t []

walk :: Tree t -> ([t] -> [t])
walk (Leaf x)     = cons x
walk (Node t1 t2) = walk t1 `o` walk t2
```

The function `walk` is currently higher order. One way to make it a first order function is to eta-expand it and inline `cons` and `o`. We will instead use the example to demonstrate defunctionalization and how it can be used to make a first order version of the whole program.

Defunctionalization works in three steps. First, the function space which will be defunctionalized is replaced by an algebraic data type. Second, lambda abstractions are replaced by constructors in that data type. And third, function application is replaced by a new function which interprets the algebraic data type such that the semantics of the program is preserved.

For the program above a new data type, `Lam a`, is created, which replaces the functions from lists to lists, i.e. `[a] -> [a]`. There are two lambda abstractions which will be turned into constructors of the `Lam` data type. They are underlined in the code above. When replacing lambda abstractions by constructors it is important to capture the free variables as arguments to the constructor. In the lambda abstraction occurring in `cons` there is one free variable, `x`. Therefore there will be a constructor to `Lam`, which takes one argument of type `a`. Similarly for the abstraction in `o`, there are two free variables `f` and `g`. Since they are function arguments, they will turn into elements of the data type `Lam a` and hence the constructor to replace the lambda abstraction will have two recursive arguments.

```
-- Array creation
generate :: Length -> (Ix -> a) -> Push a
generate n ixf = Push (\k -> for_ n $ \i ->
                               k i (ixf i)) n


-- Map
map :: Expable a => (a -> b) -> Push a -> Push b
map f (Push p l) = Push (\k -> p (\i a -> k i (f a))) l

imap :: Expable a => (Ix -> a -> b) -> Push a -> Push b
imap f (Push p l) =
  Push (\k -> p (\i a -> k i (f i a))) l


-- Permutations
reverse :: Push a -> Push a
reverse (Push p n) =
  Push (\k -> p (\i a -> k (n - 1 - i) a)) n

rotate :: Length -> Push a -> Push a
rotate d (Push p n) =
  Push (\k -> p (\i a -> k ((i +  d) `mod_` n) a)) n
```

```
-- Combining Push arrays
(++) :: Push a -> Push a -> Push a
(Push p1 l1) ++ (Push p2 l2) = Push r (l1 + l2)
  where r k = do p1 k
                 p2 (\i a -> k (l1 + i) a)

interleave :: Push a -> Push a -> Push a
interleave (Push p m) (Push q n) = Push r l
  where r k = do p (\i a -> k (2*i) a)
                 q (\i a -> k (2*i+1) a)
        l = 2 * min_ m n

-- Loading/storing a Push array from/to memory
use :: Expable a => CMMem a -> Push a
use mem@(CMMem _ n) = Push p n
  where p k = do for_ n $ \i ->
                   k i (cmIndex mem i)

toVector :: Expable a => Push a -> CM (CMMem a)
toVector (Push p l) =
  do arr <- allocate l
     p $ write arr
     return arr
```

Figure 6. Push array API. It contains a representative subset of the Push array libraries found in Obsidian and Feldspar.

The final step is to create the function `apply` which interprets the constructors. Since the `Lam a` type represents functions over lists the `apply` function will have type `Lam a -> ([a] -> [a])`. The function `apply` is defined with one case per constructor, where the result is the corresponding lambda abstraction in the original program which the constructor replaced. Having defined `apply`, it also needs to be inserted at the appropriate places in the program. In the running example it will be in the function `flatten` which applies the function from `walk`. It is also needed in the definition of `apply` when composing the two defunctionalized functions.

The final result of defunctionalizing the example program can be seen below.

```
data Lam a = LamCons a
           | LamO (Lam a) (Lam a)

apply :: Lam a -> [a] -> [a]
apply (LamCons x)  xs = x : xs
apply (LamO f1 f2) xs = apply f1 (apply f2 xs)

cons_def :: a -> Lam a
cons_def x  = LamCons x

o_def :: Lam a -> Lam a -> Lam a
o_def f1 f2 = LamO f1 f2

flatten_def :: Tree t -> [t]
flatten_def t = apply (walk_def t) []

walk_def :: Tree t -> Lam t
walk_def (Leaf x)     = cons_def x
walk_def (Node t1 t2) = o_def (walk_def t1) (walk_def t2)
```

A key observation in this example is that the defunctionalization is completely *local*. If the above code was contained in a module which only exported the `Tree` data type and the `flatten` function, then the defunctionalization could have been done completely independently of the rest of the program. So even though defunctionalization is in general a whole program transformation, there are programs where it can be applied locally. The defunctionalization of Push arrays in the next section depends on this fact.

4. Defunctionalizing Push Arrays

We now turn to applying defunctionalization in Push arrays. There are two potential functions to defunctionalize in the definition of Push arrays. We are going to focus on the outermost function, which takes the write function as argument and returns `CM ()` as result. Defunctionalizing any of the other functions will not help us towards the goal of a unifying array representation.

Applying defunctionalization to the outer function result in a data type (`PushT`) with constructors that represent our operations on Push arrays. We also obtain an `apply` function, which becomes a compiler for our new Push array language.

The defunctionalization of `map` and `(++)` is shown in full detail. The other functions from the API follow the same procedure. The procedure begins by investigating the body of the map function.

```
map :: Expable a => (a -> b) -> Push a -> Push b
map f (Push p l) = Push (\k -> p (\i a -> k i (f a))) l
```

The free variables in the underlined body are `p` and `f`. These two will be arguments to our `Map` constructor that is added to the `PushT` data type.

```
data PushT b where
  Map :: Expable a =>  (a -> b) -> PushT a -> PushT b
```

The implementation of the `map` function exposed to the programmer is simply the `Map` constructor.

```
map :: Expable a => (a -> b) -> PushT a -> PushT b
map = Map
```

The apply function is given from the body of the original `map` function, `\k -> p (\i a -> k i (f a))`. However, `p` is no longer a function. So an application of `apply` is needed to make the types match up.

```
apply :: (...) => PushT a -> (Ix -> a -> CM ()) -> CM ()
apply (Map f p) = \k -> apply p (\i a -> k i (f a))
```

Note that in our original definition of Push arrays, the length was stored with the array. The defunctionalized definition of Push arrays does not have this associated length. Instead the length is

found by traversing the PushT data type. This is just a stylistic choice, to make the presentation cleaner.

Defunctionalizing (++), as with map, begins by looking at the body of the function. In this case, essentially the where clause.

```
(++) :: Push a -> Push a -> Push a
(Push p1 l1) ++ (Push p2 l2) = Push r (l1 + l2)
  where r k = do p1 k
                 p2 (\i a -> k (l1 + i) a)
```

The free variables are p1, p2 and l1. The constructor Append is chosen to represent this operation and is added to the PushT data type.

```
data PushT b where
  Map :: Expable a => (a -> b) -> PushT a -> PushT b
  Append :: Ix -> PushT b -> PushT b -> PushT b
```

The apply functions gets a new case for Append.

```
apply :: (...) => PushT a -> (Ix -> a -> m ()) -> m ()
apply (Map f p) =
  \k -> apply p (\i a -> k i (f a))
apply (Append l p1 p2) =
  \k -> apply p1 k >>
        apply p2 (\i a -> k (l + i) a)
```

As with map the new (++) function is implemented directly from the Append constructor.

```
(++) :: PushT a -> PushT a -> PushT a
p1 ++ p2 = Append (len p1) p1 p2
```

This procedure is repeated for all operations in the Push array API resulting in the data type and apply function shown in figure 7.

5. A New Expressive Library

The previous section showed how to defunctionalize Push arrays. But there is seemingly no advantage to performing defunctionalization: the library still contains the same functions and they all still do the same thing. Here is the key insight: now that there is a concrete data type instead of a function, it is possible to write new functions on this data type by analyzing the tree and taking it apart. In particular, it is possible to write functions for the new library which previously belonged in the realm of Pull arrays.

Sections 2.5 showed how to convert from Pull arrays to Push arrays, which means that it is also possible to convert from Pull arrays to PushT. If there is also a way to convert from PushT to Pull arrays then all Pull array functions can be expressed used the PushT data type. Pull arrays are completely characterized by their indexing function and length. All that is needed is to implement the conversion to Pull arrays is implementing an indexing function for the PushT type.

One characteristic of Push arrays is that in order to look at a specific index, potentially all elements must be computed. On the defunctionalized Push arrays it is possible to implement an indexing function that is more efficient. No elements other than the one of interest will be computed.

```
index :: Expable a => PushT a -> Ix -> a
index (Generate n ixf) ix = ixf ix
index (Map f p) ix = f (index p ix)
index (Use l mem) ix = cmIndex mem ix
index (IMap f p) ix = f ix (index p ix)
index (Append l p1 p2) ix =
  ifThenElse (ix >* l)
    (index p2 (ix - l))
    (index p1 ix)
index (Interleave p1 p2) ix =
  ifThenElse (ix `mod_` 2 ==* 0)
    (index p1 (ix `div_` 2))
    (index p2 (ix `div_` 2))
index (Reverse p) ix =
  index p (len p - 1 - ix)
index (Rotate dist p) ix =
  index p ((ix - dist) `mod` (len p))
```

The implementation of index places restrictions on the operations used in the defunctionalization. For example the permutation functions must be invertible. This is another reason for why ixMap has been excluded from the language.

The index function allows the implementation of a Push to Pull conversion function that does not make the whole array manifest in memory before returning a Pull array.

```
convert :: Expable a => PushT a -> Pull a
convert p = Pull (\ix -> index p ix) (len p)
```

Being able to index into Push arrays and to convert them to Pull arrays, opens up for implementation of functions that are considered Pully also on Push arrays. One such function is zipWith.

```
zipWith :: (Expable a, Expable b)
         => (a -> b -> c)
         -> PushT a -> PushT b -> PushT c
zipWith f a1 a2 =
  generate (min (length a1) (length a2))
    (\i -> f (index a1 i) (index a2 i))
```

Using traditional Push arrays, the zipWith function would require one of the input arrays to be a Pull array.

6. Fusion and Compilation Output

One major benefit of Push arrays is that operations on them fuse automatically. This becomes very clear in the setting of a code generating DSL; just generate the code and count the number of loops. The first example shows that operations are fused. Here an input array is passed through three operations, map (+1), reverse and rotate 3.

```
ex1 :: (Expable b,  Num b) => PushT b -> PushT b
ex1 = rotate 3 . reverse . map (+1)
```

Compiling this program requires that the element type is instantiated, in this case to a Expr Int.

```
myVec = CMMem "input"  10

compileEx1 = runCM 0 $
               toVector ((ex1 arr) :: PushT (Expr Int))
  where arr = use myVec
```

The code generated from compiling this program allocates one array, and performs one loop over the input data. This is exactly the expected result from a completely fused program.

```
Allocate "v0" 10 :>>:
For "v1" 10 (
  Write "v0" ((((10 - 1) - v1) + 3) % 10) (input[v1] + 1)
  )
```

```
data PushT b where
  Generate :: Length
           -> (Ix -> b)
           -> PushT b

  Use :: Expable b
      => CMMem b
      -> PushT b

  Map :: Expable a
      => (a -> b)
      -> PushT a
      -> PushT b

  IMap :: Expable a
       => (Ix -> a -> b)
       -> PushT a
       -> PushT b

  Append :: Length
         -> PushT b
         -> PushT b
         -> PushT b

  Interleave :: PushT b
             -> PushT b
             -> PushT b

  Reverse :: PushT b
          -> PushT b

  Rotate  :: Length
          -> PushT b
          -> PushT b
```

```
apply :: Expable b => PushT b -> ((Ix -> b -> CM ()) -> CM ())
apply (Generate n ixf) =
  \k -> do for_ n $ \i ->
             k i (ixf i)

apply (Use mem@(CMMem _ n)) =
  \k -> do for_ n $ \i ->
             k i (cmIndex mem i)

apply (Map f p) =
  \k -> apply p (\i a -> k i (f a))



apply (IMap f p) =
  \k -> apply p (\i a -> k i (f i a))



apply (Append n p1 p2) =
  \k -> apply p1 k >>
        apply p2 (\i a -> k (n + i) a)


apply (Interleave p1 p2) =
  \k -> apply p1 (\i a -> k (2*i) a) >>
        apply p2 (\i a -> k (2*i+1) a)

apply (Reverse p) =
  \k -> apply p (\i a -> k ((len p) - 1 - i) a)

apply (Rotate n p) =
  \k -> apply p (\i a -> k ((i+n) `mod_` (len p)) a)
```

Figure 7. The PushT data type and apply function that is obtained from defunctionalization of our Push array API.

The next compilation example is the `saxpy` (Single-Precision A*X Plus Y) operation. This is an operation that typically would be implemented using Pull arrays, but it is now possible to implement it entirely using PushT.

```
saxpy :: Expr Float
       -> PushT (Expr Float)
       -> PushT (Expr Float)
       -> PushT (Expr Float)
saxpy a xs ys = zipWith f xs ys
  where f x y = a * x + y
```

We compile `saxpy` with two Push arrays that are created by a direct application of `use`.

```
i1 = CMMem "input1"  10
i2 = CMMem "input2"  10

compileSaxpy = runCM 0 $
               toVector (let as = use i1
                             bs = use i2
                         in saxpy 2 as bs)
```

This results in the following program.

```
Allocate "v0" 10 :>>:
For "v1" 10 (
  Write "v0" v1 ((2.0 * input1[v1]) + input2[v1])
  )
```

However, in the case of `saxpy` that uses the `index` function, it is more interesting to see the compiler output when at least one of the arrays is not simply created by a `use`. For example if the first example `ex1` is applied to one of the input arrays, before applying `saxpy`.

```
i1 = CMMem "input1"  10
i2 = CMMem "input2"  10

compileSaxpy = runCM 0 $
               toVector (let as = use i1
                             bs = ex1 $ use i2
                         in saxpy 2 as bs)
```

In this case the permutations and `map (+1)` from `ex1` has been completely inlined.

```
Allocate "v0" 10 :>>:
For "v1" 10 (
  Write "v0" v1
    ((2.0 * input1[v1]) +
     (input2[((10 - 1) - ((v1 - 3) % 10))] + 1.0)))
```

The code generated from the `saxpy` together with `ex1` example above is exactly the same as the code obtained when using Pull arrays. To show this the example is reimplemented using Pull array versions of the functions `reverse`, `rotate`, `map` and `zipWith`. The implementation of these functions are shown in Figure 5. The Pull array version of saxpy is identical to the Push array version, only the type signatures change.

```
saxpy :: Expr Float
      -> Pull (Expr Float)
      -> Pull (Expr Float)
      -> Pull (Expr Float)

ex1 :: Num b => Pull b -> Pull b
```

To generate code from the Pull array version of `saxpy` there needs to be a loop iterating over the elements of the resulting array. Each iteration writes a single element to memory. Here this is done by using a `freezePull` function.

```
freezePull :: Expable a => Pull a -> CM (CMMem a)
freezePull (Pull n ixf)  =
  do arr <- allocate n
     for_ n $ \i ->
       write arr i (ixf i)
     return arr
```

The listing below compiles the Pull array version of `saxpy` with `ex1` applied to one of the inputs.

```
ip1, ip2 :: Pull (Expr Float)
ip1 = Pull 10 (\ix -> (E (Index "input1" (unE ix))))
ip2 = Pull 10 (\ix -> (E (Index "input2" (unE ix))))

compileSaxpy = runCM 0 $
               freezePull $ saxpy 2 ip1 (ex1 ip2)
```

The generated code in this case looks exactly like the code generated from the Push array version of `saxpy`.

```
Allocate "v0" 10 :>>:
For "v1" 10 (
  Write "v0" v1
    ((2.0 * input1[v1]) +
     (input2[((10 - 1) - ((v1 + 3) % 10))] + 1.0)))
```

This is the desired result. When using functions of a pully character the new array representation generates the same code a Pull array program would.

7. Discussion

This paper shows that it is possible to unify Pull and Push arrays and obtain an array DSL with only a single array type, while maintaining the benefits that Pull and Push arrays bring separately. The key step is to defunctionalize the Push array library.

The main benefit of creating a concrete data type for Push array programming is that the `index` function can be implemented. This function instantly provides the programmer with all the flexibility of Pull arrays. The net effect is a library which can express all the functions provided by both Pull and Push arrays and give the same guarantees about fusion. When deriving the new library one must be extra careful with what Push array operations to start out with. They must be safe, permutations need to be invertible and all elements defined.

In this paper defunctionalization is used as a means to obtain a concrete representation of Push arrays. Now, looking at the API used as a starting point, coming up with a data type that represents those operations is not hard and could be done in a more ad hoc way. Defunctionalization, however, provides a method that is both tried and tested and in this case semi-automatically provides us with both the concrete representation and the compiler.

On a more general note, we have a mantra for dealing with programs written in continuation passing style: "Always defunctionalize the continuation!" Following that mantra almost always yields insights into programs. In some cases a defunctionalized continuation leads to opportunities to write programs that were not possible before, as we have demonstrated in this paper with the `index` function.

7.1 Embedded vs Native

This paper targets arrays for embedded languages. But how does our results translate to using a new array type natively in a language without any embedding? It is entirely possible to do so but it requires more work to provide the kind of fusion guarantees that the embedded language approach provides. The types `Push` and `Pull` are non-recursive and all functions on them are also non-recursive. Achieving fusion for these types is just a matter of inlining and beta-reductions, which are standard optimizations implemented by most compilers. However, the type `PushT` is a recursive type and the functions manipulating values of this type are also by necessity recursive. In order to achieve fusion for `PushT`, shortcut fusion or some similar technique [17] would be needed. We refrain from going into details here.

7.2 Unsafe Index Operations

Some Push array libraries [10] contain functions which permutes arrays by transforming indices, like the following:

```
ixMap :: (Ix -> Ix) -> Push a -> Push a
ixMap f (Push p l) =
  Push (\k -> p (\i a -> k (f i) a)) l
```

These kinds of functions are problematic for the new array library. The problem is that the index transformation function, `f`, which permutes the indices is not guaranteed to be a proper permutation, i.e. a bijection. If `ixMap` would have been included in the library it wouldn't have been possible to write the `index` function in Section 5, because `index` would have needed to invert the index transformation function. The library instead uses a fixed set of combinators which provide specific permutation. This not only solves the problem butx we also consider it to be a better library design. The function `ixMap` is a potentially unsafe function and would give undefined results if the programmer were to provide an index transformation function which is not a proper permutation. Another approach to dealing with functions such as `ixMap` would have been to provide a type for permutations which can be inverted and have that as an argument instead of the index transformation function.

8. Related Work

8.1 Array programming

Many operations can be implemented efficiently on Pull arrays and compose without inducing storage of data in memory. This is one of the reasons why this representation of arrays is being used in many embedded languages. Feldspar, is an example of such an embedded language for digital signal processing [6].

Another property of Pull arrays is that they are parallelizable. The elements of a Pull array are all computed independently and could be computed in any order or in parallel. This property of Pull arrays is used in the embedded language Obsidian, for general purpose GPU programming [10].

In Pan [15], a similar representation is used for images and in Repa [19], the *delayed* array is another example of the same representation. Later versions of Repa contain a more refined array representation which allows for efficiently implementing stencil convolutions [21]. Our line of work is different in that we have chosen to keep the simple Pull arrays and add Push arrays to be able to efficiently implement stencil computation.

8.2 Defunctionalization

Defunctionalization is a technique introduced by Reynolds in his seminal paper on definitional interpreters [23]. The transformation has later been studied by Danvy and Nielsen [12]. Defunctionaliza-

tion for polymorphic languages has been developed by two different groups [7, 22], and we make use of these results in this paper.

In a series of papers Olivier Danvy and his co-authors have used defunctionalization and other techniques to establish correspondences between interpreters and abstract machines, among other things [1–4, 8, 9, 11, 14]. In particular, one key step of their correspondence is to defunctionalize continuations to get a first order representation. This is very similar to the work presented here, but applied in a different context. However, we go further by looking at the defunctionalized continuation and writing new functions on this data type, functions which were not possible to write before. An example is the `index` function which was not possible to write for Push arrays before defunctionalization.

Another example of defunctionalizing continuations is presented by Filliâtre and Pottier [16]. The authors derive a very efficient, first order algorithm for generating all ideals of a forest poset as a Gray code from a functional specification.

9. Future Work

Push and Pull arrays have been central to our research in high performance array programming for a while now. This work furthers our understanding of Push arrays, and provides a way to unify functionality of Pull and Push arrays using a single array representation. However, this implementation of defunctionalized Push arrays is just a proof of concept. A natural next step is to replace Pull and Push arrays in one of our existing embedded DSLs, Obsidian or Feldspar, with this single array representation and see how well it fares under those conditions. A natural part of this work would be to generalize the arrays to higher dimensions along the lines of Repa [19].

Acknowledgments

Thanks Michal Palka, Anders Persson, Koen Claessen, Jean-Philippe Bernardy, Mary Sheeran and Lindsey Kuper for their valuable comments and feedback. We also thank the anonymous reviewers, who helped improve this paper a lot.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM, 2003.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. *From interpreter to compiler and virtual machine: a functional derivation*. BRICS, Department of Computer Science, University of Aarhus, 2003.

[3] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.

[4] M. S. Ager, O. Danvy, and J. Midtgaard. *A functional correspondence between monadic evaluators and abstract machines for languages with computational effects*. BRICS, Department of Computer Science, Univ., 2004.

[5] J. Ankner and J. D. Svenningsson. An EDSL Approach to High Performance Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. . URL `http://doi.acm.org/10.1145/2503778.2503789`.

[6] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer Verlag. ISBN 978-3-642-24275-5. URL `http://dl.acm.org/citation.cfm?id=2050135.2050143`.

[7] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 25–37, New York, NY, USA, 1997. ACM. ISBN 0-89791-918-1. . URL `http://doi.acm.org/10.1145/258948.258953`.

[8] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1):76–108, 2007.

[9] D. Biernacki and O. Danvy. From interpreter to logic engine by defunctionalization. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation*, volume 3018 of *Lecture Notes in Computer Science*, pages 143–159. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22174-6. . URL `http://dx.doi.org/10.1007/978-3-540-25938-1_13`.

[10] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM.

[11] O. Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 131–142, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. . URL `http://doi.acm.org/10.1145/1411204.1411206`.

[12] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X. . URL `http://doi.acm.org/10.1145/773184.773202`.

[13] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174. ACM, 2001.

[14] O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized interpreters for call-by-need evaluation. In *Functional and Logic Programming*, pages 240–256. Springer, 2010.

[15] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. URL `http://conal.net/papers/jfp-saig/`.

[16] J.-C. Filliâtre and F. Pottier. Producing all ideals of a forest, functionally. *Journal of Functional Programming*, 13(5):945–956, Sept. 2003. URL `http://gallium.inria.fr/~fpottier/publis/filliatre-pottier.ps.gz`.

[17] T. Harper. A library writer's guide to shortcut fusion. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 47–58, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. . URL `http://doi.acm.org/10.1145/2034675.2034682`.

[18] R. J. M. Hughes. A novel representation of lists and its application to the function reverse. *Information processing letters*, 22(3):141–144, 1986.

[19] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL `http://doi.acm.org/10.1145/1863543.1863582`.

[20] A. Kulkarni and R. R. Newton. Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs, 2013. Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13).

[21] B. Lippmeier and G. Keller. Efficient Parallel Stencil Convolution in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 59–70, New York, NY, USA, 2011. ACM.

ISBN 978-1-4503-0860-1. . URL http://doi.acm.org/10.1145/2034675.2034684.

[22] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 89–98, New York, NY, USA, 2004. ACM. . URL http://doi.acm.org/10.1145/964001.964009.

[23] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM. . URL http://doi.acm.org/10.1145/800194.805852.

[24] J. Svenningsson and E. Axelsson. Combining Deep and Shallow Embedding for EDSL. In H.-W. Loidl and R. Pea, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2013. . URL http://dx.doi.org/10.1007/978-3-642-40447-4_2.