Defunctionalizing Push Arrays

Josef Svenningsson  Bo Joel Svensson
Chalmers University of Technology
[josefs, joels]@chalmers.se

Abstract
Recent work on domain specific languages (DSLs) 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 and in Repa there is a even higher number of different array types. The reason for having multiple array types is to obtain code that performs better. Pull- and Push arrays, that are present in Feldspar and Obsidian, 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. Repa has Pull arrays, called delayed arrays for the same reason and so-called cursored arrays which are important for performance in stencil operations. 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 obtain 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 is the traditional type $\mathbb{I} \times \rightarrow \mathbb{V}$; a function from index to a value. Push arrays, on the other hand, are programs parameterised on a write function that take an index and a value. Pull- and Push array implementation details are shown in sections 2.2 and 2.4. These two representations have many advantages:

- 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.
- They support fusion. When composing two array functions the intermediate array is guaranteed to be fused away and not allocated in memory at runtime. Having such guarantees makes the compositional programming style particularly attractive, as it comes without any performance overhead.

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.

But why do we need two types of arrays? It would certainly be easier for the programmer if there was only a single type to keep track of.

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 subsumes 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 DSL (section 2.1) because compiled DSLs easily provide fusion guarantees. 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 2.4. We will also review defunctionalization in section 3.

2. Background
In this section we will present enough background material to make the paper self-contained. Nothing here is new material.

2.1 Preliminaries
In this paper we present Pull- and Push arrays as part of a small code generating embedded language, a compiled EDSL [15]. The embedded language is compiled into a small imperative language with for loops, memory operations (allocate, write) and conditionals, a simple C like language. The data type Code below is used to represent compiled programs.

```haskell
type Id = String
data Code = Skip
  | Code :>>: Code
  | For Id Exp Code
  | Allocate Id Length
  | Write Id Exp Exp
  | Cond Exp Code Code
```

Copyright notice will appear here once 'preprint' option is removed.
are represented by a data type called an underscore, such as mod_.

Functions in the embedded language that do not mesh well with the existing type class system are given a name ending with function.

There are also scalar expressions in the target language. These data type can be seen as a deeply embedded core language. For the purposes of this paper, we consider Code the compiler result even though more steps are needed to actually run the code on an actual machine.
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 into Obsidian [10]. Since then, Push arrays have also been implemented in Feldspar², in Nikola² and in meta-repa [5]. Push arrays are also used in reference [20], as part of an embedded language for stream processing.

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.

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

Another example, that occurs when flattening an array of pairs, is a loop that 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.

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

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.

### 2.4 A Push Array Library

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

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

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 write an element to memory. The Push array can then use this write-function any number of times. There is also a connection between this representation of Push arrays and continuations (see figure 3).

Figure 4, 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.

Perceptive readers will see that this is related to continuations as follows:

```haskell
data Cont r a = (a -> r) -> r
```

Then Push arrays can be implemented as:

```haskell
type Push a b = Cont (CM ()) (Ix,a)
```

In this way we can understand Push arrays as computations which non-deterministically generates index-value-pairs, implemented using a continuations monad.

### 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 `zip` and `pair`, are `Pully` in nature, while `unPair` is more efficient on Push arrays, call it `Pushy`.

There is also a connection between the concepts of `scattergather` operations and Push/Pull arrays. Pull arrays allow implementation of operations that gather, while Push arrays enable scatter operations.

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

```haskell
push (Pull n ixf) =
  Push (
     \k -> p (\i a -> k (f i) a)) 1
```

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

```haskell
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.
--- Array creation

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

--- 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

(++) :: Expable a => Push a -> Push a -> Push a
(Push p1 l1) ++ (Push p2 l2) = Push r (l1 + l2)
  where r k = do p1 k
              p2 \(\i a -> k (11 + i) a)\)

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

--- Create Push array from data in memory

use :: Expable a => CMMem a -> Length -> Push a
use mem l = Push p 1
  where
    p k = for_ l \$ \ix ->
      k ix (cmIndex mem ix)
toVector :: Expable a => Push a -> CM (CMem a)
toVector (Push p l) =
  do
    arr <- allocate l
    p \$ write arr
    return arr

Figure 4. Our Push array API. The functions are representative of what we use when programming with Push arrays in Obsidian and Feldspar.

--- End of programming with Push arrays

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

\[
\text{swap} (a, b) = \text{IfThenElse} (a :<: b) (a, b) (b, a)
\]

\[
\text{halfCleaner} :: \text{Pull} (\text{Expr} a) \rightarrow \text{Push} (\text{Expr} a)
\]

\[
\text{halfCleaner} = \text{uncurry} (++) \cdot \text{unzip} \cdot \text{map} \cdot \text{uncurry} \cdot \text{zip} \cdot \text{halve}
\]

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 [22]. 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 [21].

To illustrate defunctionalization we present a small example, originally due to Olivier Danvy [13]. The following program flattens trees into lists. A naive 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.

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

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 ++ 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 \(\text{cons}\) and \(o\). We will instead use defunctionalization to make a first order version of the whole program.

Defunctionalization works in three steps. First, the function space we wish to defunctionalize 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 we create a new data type \(\text{Lam}\) which replaces the functions from lists to lists, i.e. \([a] \rightarrow [a]\). There are two lambda abstractions which will be turned into constructors of the \(\text{Lam}\) 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 \(\text{cons}\) there is one free variable, \(x\). We will therefore create a constructor \(\text{Lam}\), 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.

Finally, we need to create the function apply which interprets
the constructors. Since our data type PushT a 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, we
also need to insert it at the appropriate places in the program.
In our case it will be in the function flatten which applies the
function from walk. We will also need it in the definition of apply
when composing the two defunctionalized functions.

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

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

The apply function is given from the body of the original map
function, \( \lambda k \rightarrow p (\lambda i \rightarrow k (f \ a)) \). 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) = \lambda k \rightarrow apply p (\lambda i \rightarrow k (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.

To defunctionalize (++) we begin as with map by looking at the
body of the function. In this case, essentially the where clause.

(++) :: Monad m => Push a -> Push a -> Push a
(Push p1 l1) ++ (Push p2 l2) = Push r (l1 + l2)
where r k = do p1 k
  p2 (\l -> k (l1 + l1) 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 :: (a -> b) -> PushT a -> PushT b
map = Map

The apply functions gets a new case for Append.

apply :: (...) => PushT a -> (Ix -> a -> m ()) -> m ()
apply (Map f p) = \lambda k \rightarrow apply p (\lambda i \rightarrow k (f \ a))
apply (Append p1 p2) = \lambda k \rightarrow apply p1 k
  apply p2 (\l -> k (l + l) 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 our API resulting
in the data type and apply function shown in figure 6.

Figure 5. Using defunctionalization, we go from a higher order
function to a data type and apply function.
5. A New Expressive Library

The previous section showed how to defunctionalize Push arrays. But we haven’t really gained anything, the library still contains the same functions and they all still do the same thing. Here is the key insight: now that we have a concrete data type instead of a function, we can write new functions on this data type by analysing the value and taking them apart. In particular, we will show how to write functions for our new library which previously belonged in the realm of Pull arrays.

We already know from previous sections that it is possible to convert from Pull arrays to Push arrays, which means that we can also convert from Pull arrays to PushT. If we could also find a way to convert from PushT to Pull arrays it would mean that we can express any function on Pull arrays using PushT. We will demonstrate the conversion by implementing an indexing function.

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.

Having 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 :: 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 pullly also on Push arrays. One such function is zipWith.

\[
\text{zipWith :: (Expable a, Expable b)} \\
\quad \Rightarrow (a \rightarrow b \rightarrow c) \\
\quad \rightarrow \text{PushT } a \\
\quad \rightarrow \text{PushT } b \\
\quad \rightarrow \text{PushT } c \\
\text{zipWith } f \text{ a1 a2 = } \\
\text{generate } (\min (\text{length a1}) (\text{length a2})) \\
(\langle i \rightarrow f (\text{index a1 } i) (\text{index a2 } i) \rangle)
\]

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

6. Example Programs and Compilation Output

6.1 Fusion

The big 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.

\[
ex1 :: (\text{Expable b, Num b}) \Rightarrow \text{PushT } b \\
ex1 = \text{rotate 3 . reverse . map (+1)}
\]

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

\[
\text{myVec = CMMem "input" 10} \\
\text{compileEx1 = runCM 0 $} \\
\text{toVector ((ex1 arr) :: PushT (Expr Int)) where arr = use myVec}
\]

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

\[
\text{Allocate "v0" 10 :>>;} \\
\text{For "v1" 10 (} \\
\text{\text{Write "v0" v1 (((10 - 1) - v1) \times 3 \div 10) (input[v1] + 1)}) }
\]

6.2 Saxpy

The next compilation example is the saxpy operation. This is an operation that we typically would not implement entirely on Push arrays, but this has now been made possible.

\[
saxpy :: \text{Expr Float} \\
\quad \Rightarrow \text{PushT (Expr Float)} \\
\quad \Rightarrow \text{PushT (Expr Float)} \\
\quad \Rightarrow \text{PushT (Expr Float)} \\
saxpy a x s y = \text{zipWith } f x s y \\
\text{where} \\
\quad f x y = a \times x + y
\]

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

\[
\text{i1 = CMMem "input1" 10} \\
\text{i2 = CMMem "input2" 10} \\
\text{compileSaxpy = runCM 0 $} \\
\text{toVector (let as = use i1} \\
\text{bs = ex1 $ use i2} \\
\text{in saxpy 2 as bs)}
\]

This results in the following program.

\[
\text{Allocate "v0" 10 :>>;} \\
\text{For "v1" 10 (} \\
\text{\text{Write "v0" v1 ((2.0 \times \text{input1}[v1]) + \text{input2}[v1])}) }
\]

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.

\[
i1 = \text{CMMem "input1" 10} \\
i2 = \text{CMMem "input2" 10}
\]

\[
\text{compileSaxpy = runCM 0 $} \\
\text{toVector (let as = use i1} \\
\text{bs = ex1 $ use i2} \\
\text{in saxpy 2 as bs)}
\]

In this case the permutations and map (+1) from ex1 is inlined into the indexing into index2. This is precisely what the generated code would have looked like if both input arrays had been of Pull array type.

\[
\text{Allocate "v0" 10 :>>;} \\
\text{For "v1" 10 (} \\
\text{\text{Write "v0" v1 ((2.0 \times \text{input1}[v1]) + \text{input2}(((10 - 1) - (v1 - 3) \times 10)) + 1.0)})}
\]

7. Discussion

This paper answers the question if 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. By applying defunctionalization to a Push array API we obtain that goal.

The main benefit of creating a concrete data type for Push array programming is that the \text{index} function can be implemented. This function instantly provides the programmer with all the flexibility of Pull arrays. The crux is that the Push array operations must be safe, permutations need to be invertible and all elements defined. Here we used defunctionalization as a means to obtain a concrete representation of Push arrays. Now, looking at the API we 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. But by using defunctionalization, we cut the amount of thinking necessary to implement compilation of the defunctionalized Push array language down to almost zero.

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 to program. In some cases a defunctionalized continuation leads to new opportunities to write programs which were not possible before, as we have demonstrated in this paper.

7.1 Embedded vs Native

In this paper we have targeted arrays for embedded languages. But what if we wanted to use the new array type natively in a language without any embedding? It is entirely possible to do so but we will have to work harder 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 are 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 we would have to use shortcut fusion or some similar technique [17]. We refrain from going into details here.

7.2 Unsafe Index Operations

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

\[
\text{ixMap} :: (Ix \rightarrow Ix) \rightarrow \text{Push a} \rightarrow \text{Push a}
\]

\[
\text{ixMap f (Push p l)} =
\text{Push \langle k \rightarrow p \langle \lambda a \rightarrow k (f i) a \rangle \rangle l}
\]

These kinds of functions are problematic for our new array library. The problem is that the index transformation function, \( f \), which permutes the indexes is not guaranteed to be a proper permutation, i.e. a bijection. If we had included \( \text{ixMap} \) in our library we wouldn’t have been able to write the index function in Section 5, because we would have needed to invert the index transformation function. In our library we have instead opted for a fixed set of combinators which provide specific permutation. This not only solves our problem but we also consider it to be a better library design. The function \( \text{ixMap} \) 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 \( \text{ixMap} \) 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 for 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]. 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 [24].

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 [7]. 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 [22]. The transformation has later been studied by Danvy and Nielsen [12]. Defunctionalization for polymorphic languages has developed have been developed by two different groups [7, 21], 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 we have presented here, but applied in a different context. However, we go further by looking at the defunctionalized continuation and write new functions on this data type, functions which were not possible to write before.

Another example of defunctionalizing continuations is presented by Fillitre 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-Phillipe Bernardy and Mary Sheeran for important insights, feedback and support.

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


