

Recycle Your Arrays!

Roman Leshchinskiy

Programming Languages and Systems, School of Computer Science and Engineering,
University of New South Wales, r1@cse.unsw.edu.au

Abstract. Purely functional arrays are notoriously difficult to implement and use efficiently due to the absence of destructive updates and the resultant frequent copying. Deforestation frameworks such as stream fusion achieve significant improvements here but fail for a number of important operations which can nevertheless benefit from elimination of temporaries. To mitigate this problem, we extend stream fusion with support for in-place execution of array operations. This optimisation, which we call *recycling*, is easy to implement and can significantly reduce array allocation and copying in purely functional array algorithms.

Key words: Deforestation, Optimisation, Array Programming, Functional Programming

1 Introduction

Functional languages such as Haskell are wonderful because they allow programs to be written at a high level of abstraction. However, this places a significant burden on the compiler which must incorporate a large number of sophisticated optimisations to achieve satisfactory performance. One such optimisation, *fusion* or *deforestation* [1], removes temporary data structures and combines traversals when possible. A classical example is the transformation $map\ f \circ map\ g \mapsto map\ (f \circ g)$ which eliminates a temporary list by performing the two maps in lockstep instead of one after the other.

For the most part, research has concentrated on fusion for inductive data structures, in particular lists [2–4]. In comparison, fusion of purely functional array operations has received relatively little attention with *functional array fusion* [5] and *stream fusion* [6–8] being notable recent exceptions. Unfortunately, even these frameworks treat arrays as list-like sequences, concentrating on regular traversals like *map* or *filter* and completely neglecting operations that rely on efficient indexing, such as updates, sorting etc. But often, these operations are the very reason for choosing arrays over lists!

To understand the problem, consider the bulk update operation (*//*) which yields an array obtained by updating an existing array with a list of index/value pairs. For instance, $\langle a, b, c, d, e \rangle // [(0, x), (2, y)] = \langle x, b, y, d, e \rangle$.

Without optimisation, the term $xs // ps // qs$ is evaluated by allocating a temporary array $ys = xs // ps$ and then creating another array $zs = ys // qs$ which is the final result of the computation. Obviously, we would like to perform the second update in-place, thus *recycling* ys and eliminating zs (we cannot

simply update xs without losing referential transparency). Existing fusion frameworks are of no help here, though. They only implement loop fusion and these two loops cannot be fused!

This example highlights the difference between loop fusion and the optimisation we call recycling. Although both eliminate temporary arrays, the former does so by reducing the number of loops in a program whereas the latter removes unnecessary array allocation and copying by executing operations in-place. Of course, fusing loops is usually preferable but, as we have seen, not always possible.

In this paper, we extend the stream fusion framework with recycling capabilities with the goal of executing as many array operations as possible in-place even when loop fusion fails. Our approach relies on rewriting and is much less powerful than techniques based on static analysis [9–11] or linear types [12] but significantly easier to implement. It applies to all operations that rely on destructive updates. For simplicity, we only consider bulk updates in this paper and base our development on three representative use cases shown below. Note that here and in the rest of the paper, map , $filter$ etc. denote the corresponding array operations, not standard Haskell list functions.

Term	Expected evaluation strategy
(1) $xs // ps // qs$	Perform the two updates in-place on a copy of xs .
(2) $map (+1) xs // ps$	Store the result of the map in a new array and update it in-place.
(3) $map (+1) (xs // ps)$	Update a copy of xs and then increment each element in-place.

The rest of the paper is structured as follows. Section 2 describes a simple implementation of arrays in Haskell and introduces stream fusion. In Section 3, we develop a framework capable of optimising the first two use cases by providing a pure interface to destructive array initialisation and integrating it with stream fusion. In Section 4, we tackle the third example which requires more advanced mechanisms for performing array operations in-place. Section 5 demonstrates the feasibility of our approach on a simple algorithm and quantifies the performance gains. Finally, in Section 6 we suggest future research directions and conclude.

2 Setting the stage

The optimisations introduced in this paper operate on a fairly low-level representation of arrays. This means that they cannot operate directly on the standard Haskell *Array* type. It is too abstract, does not provide access to its underlying implementation and also supports rather sophisticated index spaces which would significantly complicate the development.

Instead, we introduce our own array type *Vector*, a thin wrapper over the low-level array primitives provided by the Glasgow Haskell Compiler (GHC) which supports the mechanisms required by our framework. We also give a quick introduction to stream fusion and show its implementation for *Vector*.

— abstract data types

data *Vector* *a*

data *MutableVector* *s a*

— monadic operations

newMV :: *Int* → *ST s (MutableVector s a)*

readMV :: *MutableVector s a* → *Int* → *ST s a*

writeMV :: *MutableVector s a* → *Int* → *a* → *ST s ()*

unsafeFreezeMV :: *MutableVector s a* → *ST s (Vector a)*

— pure operations on mutable vectors

lengthMV :: *MutableVector s a* → *Int*

sliceMV :: *MutableVector s a* → *Int* → *Int* → *MutableVector s a*

— pure operations

index :: *Vector a* → *Int* → *a*

length :: *Vector a* → *Int*

Fig. 1. Basic vector operations

2.1 Arrays in Haskell

On the lowest level, arrays as provided by GHC live through two distinct phases. In the first phase, a *mutable* array is allocated and initialised by means of destructive updates. This code is necessarily monadic. Once initialisation is complete, the mutable array is *frozen*, i.e., converted to an *immutable* read-only array which can then be freely used in purely functional code.

Figure 1 shows the basic data types and operations implementing such arrays. Effects are encapsulated by the state transformer monad *ST* [13] which can be embedded in pure code with the operator *runST*:

$$\text{runST} :: (\text{forall } s. \text{ST } s \ a) \rightarrow a$$

The state token *s* ties a *MutableVector* to a particular *ST* computation. This is crucial for ensuring referential transparency.

To get a feel for how to use these primitive operations, consider the following implementation of *map* for *Vector*:

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow \text{Vector } a \rightarrow \text{Vector } b \\ \text{map } f \ xs &= \text{runST } \$ \ \mathbf{do} \ v \leftarrow \text{newMV } n && \text{— allocate} \\ & \quad \text{mapM}_- (\text{put } v) [0 .. n - 1] && \text{— initialise} \\ & \quad \text{unsafeFreezeMV } v && \text{— freeze} \end{aligned}$$

where

$$\begin{aligned} n &= \text{length } xs \\ \text{put } v \ i &= \text{writeMV } v \ i \ (f \ (\text{index } xs \ i)) \end{aligned}$$

This code illustrates the standard pattern of allocating a *MutableVector*, initialising it and then freezing it to a *Vector*. Freezing is an unsafe operation since for the sake of efficiency, it does not copy the array. Instead, the original *MutableVector* and the frozen *Vector* share the same block of memory. This

implies that subsequent destructive writes to the mutable vector would change the value of the immutable one, thus violating referential transparency. It is the programmer's responsibility to ensure that this does not happen. Typically, the mutable vector will not be used after freezing, as in the above example.

Later, we will rely on the support for constant time slicing provided by *MutableVector*. The function *sliceMV v i n* extracts *n* elements starting from index *i* from *v*. Again, the elements are not copied but, rather, aliased by the slice such that updating the slice will also change the original vector and vice versa.

2.2 Stream fusion

Of course, the primitive vector operations are much too imperative for our taste. We really want to program in terms of familiar combinators such as *map*, *filter*, *zip* etc. As the previous definition of *map* demonstrates, they are easily implemented on top of the primitive interface. However, although correct, this implementation is not very efficient when used in *pipelines* of computations. For instance, when evaluating *map f (map g xs)* the result of *map g xs* is stored in a temporary vector and then *f* is mapped over it in a second traversal. We would like to fuse the two loops, in effect computing *map (f ∘ g) xs* and thus eliminating the temporary.

Stream fusion achieves this by providing a coinductive, functional view of an array, which we call a *stream*:

```
data Step s a = Yield a s | Skip s | Done
data Stream a = ∃s. Stream (s → Step s a) s Int
```

A stream is made up of three components: a *stepping function*, a *seed* and a *size hint* which gives an upper bound on the number of elements in the stream. Streams are traversed by repeatedly applying the stepping function to the current seed. In each step, the function can yield the next element and a new seed (*Yield*), return just a new seed without producing an element (*Skip*) or signal the end of the stream (*Done*). We can easily provide *Stream* versions of standard combinators such as *map* or *filter*. In the following, we use the suffix *S* to distinguish them from their vector counterparts. We only show the implementation of *mapS* here and refer the reader to [7] for other stream combinators.

```
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream next s n) = Stream next' s n
where
  next' s = case next s of
    Yield x s' → Yield (f x) s'
    Skip s'    → Skip s'
    Done      → Done
```

Like all stream producers, *mapS* is not recursive. This is crucial since it allows pipelines of stream transformers, like *mapS f ∘ mapS g*, to be fused and

optimised by general-purpose transformations such as inlining and constructor specialisation which GHC already implements [14, 15]. The need to avoid recursion is also the motivation for *Skip* which is necessary for filtering out elements.

Stream fusion itself relies on two functions which convert between streams and vectors. Obtaining a *Stream* from a *Vector* is straightforward:

```
stream :: Vector a → Stream a
stream xs = Stream next 0 n
  where
    n                = length xs
    next i | i < n  = Yield (index xs i) (i + 1)
           | otherwise = Done
```

The inverse operation constructs a new *Vector* following the usual pattern of allocation, initialisation and freezing identified in the previous section:

```
unstream :: Stream a → Vector a
unstream (Stream next s n) = runST $
  do v ← newMV n
     n' ← fill v s 0
     unsafeFreezeMV (sliceMV v 0 n')
  where
    fill v s i = case next s of Yield x s' → do writeMV v i x
                                                fill v s' (i + 1)
              Skip s'   → fill v s' i
              Done      → return i
```

While enough space for n elements is allocated initially, that is only an upper bound on the actual length of the stream. After consuming the entire stream, the exact number of elements becomes known and only the corresponding slice of the vector is frozen. Note that in contrast to *mapS* and *stream*, *unstream* is recursive. This does not interfere with optimisation since it is a pure consumer, i.e., it does not produce a stream.

Based on the functions introduced in this section, we can implement typical vector operations in terms of operations on streams:

```
map :: (a → b) → Vector a → Vector b
map f = unstream . mapS f . stream
```

The last missing ingredient in the fusion framework is a mechanism for eliminating unnecessary conversions from and to streams. GHC allows this to be implemented as part of the library by specifying a *rewrite rule* which is applied whenever possible during optimisation [16]:

```
<stream/unstream> ∀s. stream (unstream s) ↦ s
```

The semantics is straightforward: instead of creating a temporary vector from the stream s and then converting it back to a stream, we can use s directly. The following transformation sequence demonstrates this rule in action:

```

    map f (map g xs)
= {inline}
  unstream (mapS f (stream (unstream (mapS g (stream xs))))))
= {apply stream/unstream}
  unstream (mapS f (mapS g (stream xs)))

```

The resulting code only has one loop (*unstream*) and does not create any temporary vectors, operating on streams instead. As mentioned above, after another round of inlining GHC’s optimiser is capable of completely eliminating any stream-related overheads, producing a tight, efficient loop which executes the two *mapS* in lockstep.

3 Basic recycling

Stream fusion only works for combinators which can be implemented in terms of streams. Unfortunately, some crucial vector operations cannot be written in this way. For instance, the bulk update operation (*//*) described in the introduction has no efficient stream counterpart. We can, of course, implement (*//*) directly by destructively updating a mutable copy of a vector:

```

(//) :: Vector a → [(Int, a)] → Vector a
xs // ps = runST $ do v ← newMV n
                mapM_ (copy v) [0..n-1]
                mapM_ (put v) ps
                unsafeFreezeMV v

```

where

```

n          = length xs
copy v i   = writeMV v i (index xs i)
put v (i, x) = writeMV v i x

```

However, this implementation is, again, not optimal since it introduces superfluous temporary vectors when used in pipelines. In particular, none of the three use cases from the introduction are evaluated as desired. For instance, as discussed previously, *xs // ps // qs* unnecessarily copies the result of *xs // ps* into a new vector before updating it with *qs*. This is a great opportunity for recycling!

3.1 Combining initialisers

While this particular example can be simply rewritten to *xs // (ps ++ qs)*, we are, of course, interested in a more general solution which is applicable to all array operations that can benefit from recycling. Our approach is based on a data type which encapsulates the allocation and initialisation of a *MutableVector*:

```

data New a = New (∀s. ST s (MutableVector s a))

```

New simply wraps a monadic initialiser which produces a *MutableVector*. Constructing a *Vector* from it is straightforward:

```

new :: New a → Vector a
new (New init) = runST $ do { v ← init; unsafeFreezeMV v }

```

We can also define an operation which produces a fresh, mutable copy of a *Vector*:

```

clone :: Vector a → New a
clone xs = New $ do v ← newMV n
              mapM_ (copy v) [0..n-1]
              return v

```

where

```

n      = length xs
copy v i = writeMV v i (index xs i)

```

With these definitions in hand, we are now in the position to introduce the core technique of our approach. First, we define all array operations that can benefit from recycling (e.g. bulk update) as functions on *New*.

```

update :: New a → [(Int, a)] → New a
update (New init) ps = New $ do v ← init
              mapM_ (put v) ps
              return v

```

where

```

put v (i, x) = writeMV v i x

```

The corresponding operations on *Vector* are now easily obtained with the help of *clone* and *new*, as the following definition of (*//*) shows:

```

xs // ps = new (update (clone xs) ps)

```

The new definition has a crucial advantage: it makes array copying explicit and is much more amenable to rewriting than the original monadic one. In fact, all we need to do is eliminate unnecessary conversions between *Vector* and *New*. This principle is quite similar to stream fusion, as is the rule implementing it:

<clone/new> $\forall p. \text{clone} (\text{new } p) \mapsto p$

The rule encodes the basic idea of recycling: there is no need to copy a mutable vector if it is immediately discarded. This simple mechanism is already sufficient to handle the first use case from the introduction:

```

xs // ps // qs
= {inline}
  new (update (clone (new (update (clone xs) ps))) qs)
= {apply clone/new}
  new (update (update (clone xs) ps) qs)

```

The resulting code performs the two updates in-place and does not unnecessarily copy vectors. As with stream fusion, inlining and other standard optimisations further improve its performance.

It is important to realise that the correctness of the rewrite rule crucially depends on the fact that it does *not* operate directly on a vector but rather on a computation which constructs one. This allows the *clone/new* pair to be safely eliminated even if the computation is shared since it produces a new vector each time it is executed.

3.2 Integrating stream fusion

The framework developed in the previous section is capable of eliminating temporaries from adjacent applications of (*//*) and similar operations. But what about *map*, *filter* and other combinators which benefit from loop fusion as opposed to just recycling? Fortunately, it turns out that stream fusion can be seamlessly integrated with our approach.

The key observation is that both *unstream* and *clone* can be implemented in terms of a more primitive combinator which initialises a vector from a *Stream*:

$$\textit{fill} :: \textit{Stream } a \rightarrow \textit{New } a$$

Its definition is easily derived from the implementation of *unstream* given in Section 2.2 – all we need to do is replace *runST* by the constructor *New* and refrain from freezing the *MutableVector*. Of course, there is no need to duplicate this code as we can now use *fill* in the definition of *unstream*:

$$\begin{aligned} \textit{unstream} &:: \textit{Stream } a \rightarrow \textit{Vector } a \\ \textit{unstream } s &= \textit{new } (\textit{fill } s) \end{aligned}$$

Analogously, *clone* can be easily rewritten to use the new combinator:

$$\begin{aligned} \textit{clone} &:: \textit{Vector } a \rightarrow \textit{New } a \\ \textit{clone } xs &= \textit{fill } (\textit{stream } xs) \end{aligned}$$

With these definitions, *unstream* and *clone* are no longer primitive with respect to fusion. We need to reformulate our rewrite rules to account for this:

$$\begin{aligned} \langle \mathbf{fusion} \rangle & \quad \forall s. \textit{stream } (\textit{new } (\textit{fill } s)) \mapsto s \\ \langle \mathbf{recycling} \rangle & \quad \forall p. \textit{fill } (\textit{stream } (\textit{new } p)) \mapsto p \end{aligned}$$

The rules are obtained from **stream/unstream** and **clone/new** simply by expanding the definitions of the respective combinators. It is instructive to contrast the roles played by the two rules:

- **fusion** is derived from the original stream fusion rule and eliminates temporary *immutable* vectors by fusing loops;
- **recycling** eliminates unnecessary copying of *mutable* vectors during the initialisation phase.

The following example shows that the first rule implements stream fusion in the new system:

$$\begin{aligned}
& \text{map } f \text{ (map } g \text{ } xs) \\
= & \text{\{inline\}} \\
& \text{new (fill (mapS } f \text{ (stream (new (fill (mapS } g \text{ (stream } xs)))))) \\
= & \text{\{apply } \mathbf{fusion}\}} \\
& \text{unstream (mapS } f \text{ (mapS } g \text{ (stream } xs)))
\end{aligned}$$

It is also instructive to see how the recycling functionality developed in the previous section is still provided by the second rule:

$$\begin{aligned}
& xs \ // \ ps \ // \ qs \\
= & \text{\{inline\}} \\
& \text{new (update (fill (stream (new (update (fill (stream } xs)) ps)))) qs) \\
= & \text{\{apply } \mathbf{recycling}\}} \\
& \text{new (update (update (fill (stream } xs)) ps) qs)
\end{aligned}$$

But this is not all! By integrating stream fusion and recycling, the new system is also able to handle the second use case from the introduction, as the following transformation sequence shows:

$$\begin{aligned}
& \text{map (+1) } xs \ // \ ps \\
= & \text{\{inline\}} \\
& \text{new (update (fill (stream (new (fill (mapS (+1) (stream } xs)))))) ps) \\
= & \text{\{apply } \mathbf{fusion} \text{ or } \mathbf{recycling}\}} \\
& \text{new (update (fill (mapS (+1) (stream } xs))) ps)
\end{aligned}$$

In a sense, the last rewriting step performs both fusion and recycling which explains why either of the two rewrite rules can be applied here. It can be seen either as executing the update in-place or as fusing the *map* with the subsequent stream-based copying. In any case, the nondeterminism does not lead to problems since the two rewrite rules are confluent.

4 Recycling for transformers

The last unsolved problem are computations in which transformers such as *map* cannot be fused with preceding operations but can be executed in-place, as in our third use case. Even with the framework developed in the previous section, *map (+1) (xs // ps)* allocates two vectors where one would be sufficient. Here, the elements are incremented *after* updating the array. This can be done in-place but so far, we have not introduced any mechanisms for handling such cases.

Before explaining our solution, it is important to point out that this form of in-place execution is only possible for a restricted set of array operations which meet the following conditions:

- they do not change the type of the elements,
- they process the array sequentially and
- the result fits into the original array.

Fortunately, many important operations such as filtering and scanning fall into this category. Calls to *map* can be optimised in this way (as in our example) as long as they do not violate the first requirement.

4.1 Monadic streams

In the context of our fusion framework, we can observe that certain stream transformers of type $Stream\ a \rightarrow Stream\ a$ can be executed in-place, thus recycling mutable vectors. Since such in-place operations are necessarily monadic, we must generalise streams to support monadic computations. Fortunately, this generalisation is straightforward:

```
data MStream m a =  $\exists s. MStream\ (s \rightarrow m\ (Step\ s\ a))\ s\ Int$ 
```

Monadic streams are parametrised by a monad m and the stepping function is executed in that monad. $MStream$ is strictly more general than $Stream$ as the latter can be obtained by instantiating the former at the identity monad Id :

```
type Stream a = MStream Id a
```

Most stream operations can be trivially reimplemented to work on monadic streams. Again, we use $mapS$ as an example:

```
mapS :: Monad m => (a → b) → MStream m a → MStream m b
mapS (MStream next s n) = MStream next' s n
where
  next' s = do r ← next s
           case r of
             Yield x s' → return (Yield (f x) s')
             Skip s'    → return (Skip s')
             Done       → return Done
```

It is easy to verify that the semantics of $mapS$ remains unchanged for $Stream$ with the new definitions. In the rest of the paper, we will assume that streams are defined as described above and that all stream operations have been suitably generalised to monadic streams.

The main advantage of monadic streams is their ability to model mutable arrays, whereas pure streams are only restricted to immutable ones. To make use of this functionality, we must provide conversions from $MutableVector$ to $MStream$ and back. The first direction is straightforward:

```
streamM :: MutableVector s a → MStream (ST s) a
streamM v = MStream next 0 n
where
  n                = lengthMVector v
  next i | i < n = do x ← readMV v i
                   return (Yield x (i + 1))
  | otherwise = return Done
```

Since our goal is to execute stream transformers in-place, the inverse operation should overwrite an existing mutable vector rather than allocating a new one. Unsurprisingly, its implementation is quite similar to $unstream$ from Section 2.2:

$$\begin{aligned}
\text{unstreamM} &:: \text{MutableVector } s \ a \rightarrow \text{MStream } (ST \ s) \ a \\
&\hspace{15em} \rightarrow ST \ s \ (\text{MutableVector } s \ a) \\
\text{unstreamM } v \ (\text{MStream } \text{next } s \ _) &= \mathbf{do} \ n \leftarrow \text{loop } s \ 0 \\
&\hspace{15em} \text{return } (\text{sliceMV } v \ 0 \ n)
\end{aligned}$$

where

$$\begin{aligned}
\text{loop } s \ i &= \mathbf{do} \ r \leftarrow \text{next } s \\
&\mathbf{case} \ r \ \mathbf{of} \\
&\quad \text{Yield } x \ s' \rightarrow \mathbf{do} \ \text{writeMV } v \ i \ x \\
&\hspace{10em} \text{loop } s' \ (i + 1) \\
&\quad \text{Skip } s' \rightarrow \text{loop } s' \ i \\
&\quad \text{Done} \rightarrow \text{return } i
\end{aligned}$$

Note that *unstreamM* assumes that the vector is large enough to hold all elements of the stream and correctly adjusts its length if there are fewer elements, as required for in-place filtering.

The two conversions are sufficient to implement stream-based in-place transformers for mutable vectors. This operation is provided as a function on *New* since it will be later used in rewrite rules:

$$\begin{aligned}
\text{transform} &:: (\forall m. \text{Monad } m \Rightarrow \text{MStream } m \ a \rightarrow \text{MStream } m \ a) \\
&\hspace{15em} \rightarrow \text{New } a \rightarrow \text{New } a \\
\text{transform } f \ (\text{New } \text{init}) &= \text{New } \$ \mathbf{do} \ v \leftarrow \text{init} \\
&\hspace{15em} \text{unstreamM } v \ (f \ (\text{streamM } v))
\end{aligned}$$

Note that *f* must be polymorphic in the monad as the rewrite system introduced below will instantiate it at the identity monad in addition to *ST*.

4.2 In-place stream transformers

For the newly gained ability to execute stream transformers in-place to be useful, pure stream operations must be replaced by their monadic counterparts whenever possible. Our fusion framework cannot identify such opportunities automatically. Instead, we introduce a special combinator which allows us to “mark” stream transformers which can benefit from recycling:

$$\begin{aligned}
\text{inplace} &:: (\forall m. \text{Monad } m \Rightarrow \text{MStream } m \ a \rightarrow \text{MStream } m \ a) \\
&\hspace{15em} \rightarrow \text{Stream } a \rightarrow \text{Stream } a \\
\text{inplace } f &= f
\end{aligned}$$

Semantically, *inplace* simply restricts its polymorphic argument to the identity monad. To the fusion system, however, it identifies the stream transformer as a candidate for in-place execution. This information is used in the following rewrite rule which ties together the mechanisms developed in this section:

$$\langle \mathbf{inplace} \rangle \ \forall f \ p. \text{fill } (\text{inplace } f \ (\text{stream } (\text{new } p))) \mapsto \text{transform } f \ p$$

The rule eliminates an unnecessary array allocation (*fill*) by executing the stream transformer *f* in-place, thus recycling the vector created by *p*. It highlights the

role of *inplace* since this transformation is only valid for some *f*. Again, it is our responsibility to identify and mark such transformers.

To handle the last use case, *map* must be marked as *inplace* but only if it does not change the type of the elements. In fact, we can define a special version of *map* which is always a candidate for in-place execution:

$$\begin{aligned} \text{inplace_map} &:: (a \rightarrow a) \rightarrow \text{Vector } a \rightarrow \text{Vector } a \\ \text{inplace_map } f &= \text{unstream} . \text{inplace } (\text{mapS } f) . \text{stream} \end{aligned}$$

Note that *inplace_map* has a more restrictive type than *map* but is semantically equivalent otherwise. All we need to do now is replace *map* by *inplace_map* if and only if the types allow it. Fortunately, rewrite rules yet again provide a solution here:

$$\langle \mathbf{inplace_map} \rangle \quad \text{map} \mapsto \text{inplace_map}$$

The type of *inplace_map* constraints the applicability of the rule which is precisely what we want. In fact, this technique, known as specialisation, is so useful that GHC's support for it actually predates the rewrite rule mechanism.

With this piece of the puzzle in place, our framework is finally capable of properly optimising the third use case:

$$\begin{aligned} &\text{map } (+1) (xs // ps) \\ = &\{\text{specialise with } \mathbf{inplace_map}\} \\ &\text{inplace_map } (+1) (xs // ps) \\ = &\{\text{inline}\} \\ &\text{new } (\text{fill } (\text{inplace } (\text{mapS } (+1)) (\text{stream } (\text{new } (\text{update } (\text{clone } xs) ps)))))) \\ = &\{\text{apply } \mathbf{inplace}\} \\ &\text{new } (\text{transform } (\text{mapS } (+1)) (\text{update } (\text{clone } xs) ps)) \end{aligned}$$

Expanding the remaining combinators and verifying that the array elements are indeed incremented in-place is left as an exercise to the reader.

4.3 Monadic stream fusion

Interestingly, the last example would also be correctly optimised by rewriting *inplace f (stream (new p))* to *stream (new (transform f p))* and subsequently applying either **fusion** or **recycling**. Although arguably simpler, such a rule would interfere with stream fusion in some slightly more complex cases:

$$\begin{aligned} &\text{map } (> 5) (\text{map } (+1) (xs // ps)) \\ = &\{\text{specialise, inline and apply } \mathbf{fusion}\} \\ &\text{new } (\text{fill } (\text{mapS } (> 5) \\ &\quad (\text{inplace } (\text{mapS } (+1)) (\text{stream } (\text{new } (\text{update } (\text{clone } xs) ps)))))) \\ = &\{\text{rewrite as described above}\} \\ &\text{new } (\text{fill } (\text{mapS } (> 5) \\ &\quad (\text{stream } (\text{new } (\text{transform } (\text{mapS } (+1)) (\text{update } (\text{clone } xs) ps)))))) \end{aligned}$$

Since $mapS (> 5)$ cannot be executed in-place, $mapS (+1)$ should not be, either! Doing so effectively “unfuses” the two $mapS$, resulting in three loops instead of two. In contrast, **inplace** avoids this pitfall by requiring that the output of the stream transformer is immediately converted to a vector and not passed on to another stream consumer.

Unfortunately, this does not completely solve the problem as demonstrated by the following transformation sequence:

$$\begin{aligned}
& map (> 5) (map (+1) (xs // ps)) \\
= & \{inline\} \\
& new (fill (mapS (> 5) (stream (new (fill \\
& \quad (inplace (mapS (+1)) (stream (new (update (clone xs) ps)))))))))) \\
= & \{apply \mathbf{inplace}\} \\
& new (fill (mapS (> 5) (stream (new (transform \\
& \quad (mapS (+1)) (update (clone xs) ps))))))
\end{aligned}$$

Here, **inplace** was applied *before fusion*, thus preventing the two $mapS$ from being fused. There are two ways to avoid this. Firstly, GHC provides a staging mechanism for rewrite rules which would allow us to give precedence to fusion over recycling. A better solution, however, is to undo the effects of **inplace** if the vector is immediately converted back to a stream. The following rule accomplishes this:

$$\langle \mathbf{uninplace} \rangle \quad \forall f p. \\
\quad stream (new (transform f p)) \mapsto inplace f (stream (new p))$$

It is easy to verify that **uninplace** is equivalent to the inverse of **inplace** immediately followed by **fusion**. In the problematic example, applying **uninplace** after the last step restores the desired behaviour.

To ensure that the rewrite system is confluent two additional, fairly obvious, rules are required:

$$\begin{aligned}
\langle \mathbf{inplace2} \rangle \quad & \forall f g s. \quad inplace f (inplace g s) \mapsto inplace (f \circ g) s \\
\langle \mathbf{mfusion} \rangle \quad & \forall f g p. \quad transform f (transform g p) \mapsto transform (f \circ g) p
\end{aligned}$$

To see why they are necessary, consider all possible rewriting steps for the term $map (+1) (map (+1) (xs // ps))$ where both $maps$ can be executed in-place.

5 Benchmarks

To test our approach we have implemented the Rootfix algorithm [17] which, given a tree labelled by numbers, computes the sum of labels on the path from the root for each node. Thus, by labelling all nodes with 1 the algorithm can be used to determine the depth of each node. It operates on a special array-based encoding of a tree derived from its parenthetical representation. For instance, the complete binary tree of depth 3 can be written as " $((()())(())())$ ", i.e.,

each node is represented by a pair of parentheses enclosing the parenthetical representations of its children. The array encoding is obtained by storing the indices of the left and right parentheses of all nodes in two separate arrays, indexed by the preorder number of the nodes. For the complete binary tree, this results in the two arrays $\langle 0, 1, 2, 4, 7, 8, 10 \rangle$ and $\langle 13, 6, 3, 5, 12, 9, 11 \rangle$.

Rootfix is a data parallel algorithm which is of particular importance to us since we intend to employ the framework developed in this paper in the Data Parallel Haskell project [6]. Its implementation is quite simple:

```

rootfix :: Num a => Vector a -> Vector Int -> Vector Int -> Vector a
rootfix xs ls rs = let zs    = replicate (length xs * 2) 0
                    vs     = zs /// zip ls xs /// zip rs (map negate xs)
                    sums = prescanl' (+) 0 vs
                in
                    map (index sums) ls

```

Here, (*///*) is similar to (*//*) but takes a vector of value/index pairs instead of a list and *prescanl'* computes the prefix sum of an array with a strict accumulator. The numerous fusion and recycling opportunities are easy to spot and GHC does a good job here, applying **fusion** three times, **recycling** twice and **inplace** once.

The performance improvements are encouraging. Recycling reduces the number of array allocations from 5 to 2 compared to stream fusion alone. For a perfect binary tree of depth 23, the algorithm runs in 3517ms on a 2.6GHz Intel Core 2 Duo, as opposed to 5040ms with only stream fusion, a speedup of roughly 1.4. The results are similar for other tree sizes (the shape of the tree does not affect the performance). It remains to be seen how useful recycling will be for larger algorithms but we expect it to provide significant benefits in many cases.

6 Conclusion

We have described an optimisation framework for array programs which extends stream fusion with advanced recycling mechanisms for situation in which loop fusion is not possible. The new system is able to optimise more programs than stream fusion alone while remaining manageable with only 6 core rewrite rules. Thanks to GHC's excellent optimisation capabilities, it can be implemented as a library and does not require changes to the compiler. The initial performance gains are encouraging although the framework is yet to be tested in large real-world programs.

While we have restricted ourselves to arrays and stream fusion, we believe that the concepts developed in this paper are easily transferrable to other data structures and fusion systems. Moreover, it would be interesting to see if our approach can be extended to work across function boundaries and in particular to recursive functions with single-threaded uses of arrays. Here, we envision a system similar to constructor specialisation [14]. In the near future, we intend to integrate recycling into the Data Parallel Haskell project which provided the original motivation for this work.

References

1. Wadler, P.: Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, (Special issue of selected papers from 2nd European Symposium on Programming) **73**(2) (1990) 231–248
2. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: *Conference on Functional Programming Languages and Computer Architecture*. (1993) 223–232
3. Johann, P.: Short cut fusion: Proved and improved. In: *SAIG 2001: Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation*, Springer-Verlag (2001) 47–71
4. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Functional programming*, ACM Press (2002) 124–132
5. Chakravarty, M.M.T., Keller, G.: Functional array fusion. In Leroy, X., ed.: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ACM Press (2001) 205–216
6. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, ACM (2007) 10–18
7. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: from lists to streams to nothing at all. In: *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional programming*, ACM (2007) 315–326
8. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting Haskell strings. In: *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, Springer-Verlag (2007) 50–64
9. Bloss, A.: Update analysis and the efficient implementation of functional aggregates. In: *Proceedings of the 4th international conference on Functional programming languages and computer architecture*, ACM (1989) 26–38
10. Odersky, M.: How to make destructive updates less destructive. In: *In Proc. 18th ACM Symp. on Principles of Programming Languages*, ACM Press (1991) 25–36
11. Sastry, A.V.S., Clinger, W., Ariola, Z.: Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In: *Conference on Functional Programming Languages and Computer Architecture*, ACM Press (1993) 266–275
12. Wadler, P.: Linear types can change the world. In: *Programming Concepts and Methods*, North (1990) 347–359
13. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: *SIGPLAN Conference on Programming Language Design and Implementation*. (1994) 24–35
14. Peyton Jones, S.: Call-pattern specialisation for Haskell programs. In: *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional programming*, ACM (2007) 327–337
15. Peyton Jones, S., Santos, A.L.M.: A transformation-based optimiser for Haskell. *Sci. Comput. Program.* **32**(1-3) (1998) 3–47
16. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC. In Hinze, R., ed.: *2001 Haskell Workshop*, ACM (2001)
17. Leiserson, C.E., Maggs, B.M.: Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica* **3** (1988) 53–77