

Functional Array Fusion

Manuel M. T. Chakravarty
University of New South Wales
School of Computer Science & Engineering
Sydney, Australia
chak@cse.unsw.edu.au

Gabriele Keller^{*}
University of New South Wales
School of Computer Science & Engineering
Sydney, Australia
keller@cse.unsw.edu.au

ABSTRACT

This paper introduces a new approach to optimising array algorithms in functional languages. We are specifically aiming at an efficient implementation of irregular array algorithms that are hard to implement in conventional array languages such as Fortran. We optimise the storage layout of arrays containing complex data structures and reduce the running time of functions operating on these arrays by means of equational program transformations. In particular, this paper discusses a novel form of combinator loop fusion, which by removing intermediate structures optimises the use of the memory hierarchy.

We identify a combinator named *loopP* that provides a general scheme for iterating over an array and that in conjunction with an array constructor *replicateP* is sufficient to express a wide range of array algorithms. On this basis, we define equational transformation rules that combine traversals of *loopP* and *replicateP* as well as sequences of applications of *loopP* into a single *loopP* traversal.

Our approach naturally generalises to a parallel implementation and includes facilities for optimising load balancing and communication. A prototype implementation based on the rewrite rule pragma of the Glasgow Haskell Compiler is significantly faster than standard Haskell arrays and approaches the speed of hand coded C for simple examples.

1. INTRODUCTION

Functional programming languages typically focus on lists rather than arrays due to the more elegant algebraic properties of the former. Notable exceptions are special purpose languages like Sisal [9], SAC [32], and FISh [18], which target applications from computational science and engineering that are usually implemented in array-centred languages, such as Fortran. Consequently, the design of these languages tends to be influenced by their imperative predecessors; in

^{*}The second author has been working at the University of Technology, Sydney, while performing the work reported in this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'01, September 3-5, 2001, Florence, Italy.

Copyright 2001 ACM 1-58113-415-0/01/0009 ...\$10.00.

fact, SAC is an abbreviation for Single Assignment C and FISh makes heavy use of imperative features. In line with this approach, Sisal, SAC, and FISh as well as their implementations also focus on what the high-performance computing community calls *regular* algorithms. These operate over data structures that can conveniently be stored in rectangular arrays and they iterate over regularly shaped index spaces. For many applications this is not sufficient.

Irregular algorithms that operate on sparse data structures (such as sparse matrices) or arrays of trees (such as adaptive or hierarchical numeric algorithms) require richer data structures and index spaces. Sophisticated data structures are certainly a key strength of modern functional languages; so, it stands to reason that if we want to make an impact on array programming with functional languages, we should try to beat imperative languages in applications where irregular algorithms are central. Having realised this, it is startling that most work on arrays in functional languages appears to have focused on regular code [2, 12, 14, 23]. May be this is because of the traditional focus on regular problems in array languages.

Another reason is probably that irregular algorithms are, independent of the language, much harder to implement efficiently, especially on parallel architectures. However, Blelloch & Sabot [8] observed that every irregular algorithm can be transformed into a *semi-regular* one that operates on a flattened representation of the original nested structures and uses explicit structural information to represent the irregularity of the original data. We have previously shown how this technique, known under the name *flattening transformation*, can be generalised to apply to typed, general-purpose functional languages like Haskell and Standard ML [11].

However, flattening is only half the story. Flattened array algorithms are combinator-based and generate many intermediate structures. In fact, they are rather far away from what previous work identified as being efficient formulations of array algorithms in general-purpose functional languages: They should be based on *unboxed* and *updatable* arrays [16, 33]. An unboxed array stores the binary representation of an element (e.g., an integer number) directly, whereas a boxed array would store a pointer to a heap-allocated cell containing the integer value. Unboxed arrays not only save on memory, but allow a compiler to better predict the memory layout, which can lead to large performance improvements due to better cache utilisation. Updatable, or single-threaded arrays avoid superfluous copying. There are also approaches that aim at similar efficiency using persistent arrays [23], but they often do not support unboxed values well.

Serrarens [33] clearly demonstrates the performance difference between a straight-forward implementation of an array algorithm using lazy arrays and an optimised version based on unboxed and updatable arrays. He performs the transformation of the straight-forward into the optimised implementation manually. This surely is not what we want!

So, the goal of the research presented here is simply stated: Given a purely functional, combinator-based implementation of an array algorithm, automatically produce an efficient implementation that (1) avoids intermediate structures, (2) uses unboxed arrays, and (3) is based on destructive updates. We achieve this by a novel form of loop fusion—deforestation [37] of arrays if you like—in combination with standard optimisations like inlining and specialisation operating on overloaded unboxed arrays. While we are specifically interested in array code generated by the flattening transformation and present an implementation of our approach in Haskell, the techniques presented in this paper are neither restricted to flattening-generated code nor to Haskell, but are generally applicable. In fact, recent work in the context of Standard ML [4] is exploring an array model that is very similar to the one that we are discussing and we expect that our implementation technique is useful in the Standard ML scenario, too. In summary, the contributions of this paper are the following:

- Two array combinators (an array constructor and a loop abstraction) that are sufficient to express a wide range of array computations (Section 4)
- A novel form of loop fusion based on equational rewrite rules that amalgamate consecutive loops over arrays (Section 4)
- Fusion over both arguments of an array variant of *zip*—deforestation techniques for lists often stumble over this case (Section 5.1)
- A method for applying array fusion across function boundaries of recursive functions (Section 5.2)
- Figures of the running times (Section 6) for a first implementation (Section 3 & 4.4)

The implementation is based on a number of advanced features of the Glasgow Haskell Compiler [35], namely unboxed types [29], mutable arrays [26], and optimising rewrite rules [25]. However, these features merely affect the internals of our array library—an ordinary user is still provided with a clean, purely functional interface. In fact, as we shall discuss in Section 2, we do not even expect the application programmer to directly use our array combinators; instead, we provide a convenient interface based on array comprehensions and well-known combinators like those for lists in Haskell’s prelude. The runtime figures indicate that for some algorithms based on sparse and irregular structures, our approach to arrays is more efficient than both Haskell’s standard arrays as well as the corresponding list algorithms. Moreover, we get very close to the speed of hand-coded C for simple examples.

Loop fusion is not a new technique. It has been extensively studied (see, e.g., [1, 21, 22, 31, 39]) and it is used in varying stages of sophistication in probably all high-performance compilers. It is often used to increase locality of reference, to avoid communication, and to increase the granularity of

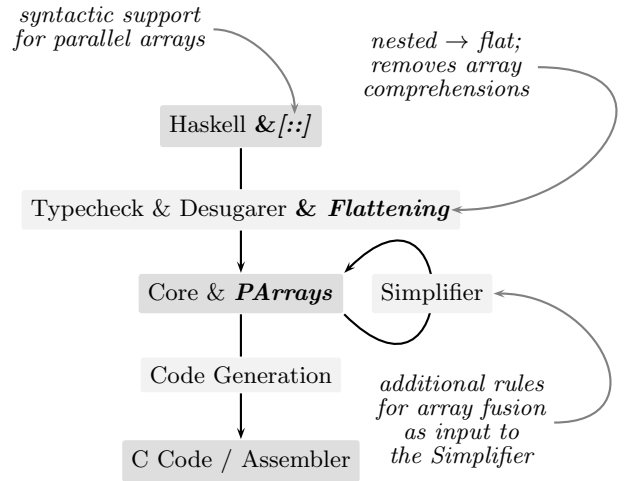


Figure 1: GHC with extra array support

parallelism. The unusual features of our approach are the explicit handling of segmented loops, which are crucial for irregular codes, and the use of equational rewrite rules to express loop fusion. The latter simplifies the implementation and is more accessible to formal analysis.

The remainder of the paper is organised as follows: Section 2 provides an overview of the various transformations that we apply. Section 3 outlines the interface and implementation of the array library. Section 4 presents the two array combinators over which the fusion rules are defined and introduces the rules needed to fuse functions consuming a single array. Section 5 covers the more elaborate case of functions consuming multiple arrays and of fusion across function boundaries. Section 6 presents benchmarks. Section 7 discusses related work and draws conclusions.

2. THE BIG PICTURE

Let us start with an outline of a compilation framework for functional arrays, which also clarifies the role that loop fusion has in this framework. Figure 1 displays the main phases of the Glasgow Haskell Compiler (GHC) [35], which we enrich with support for arrays (set in *bold italic*). The frontend reads Haskell modules, which include syntactic sugar for arrays (in the figure, indicated by the array notation $[\cdot\cdot:]$). After type checking, the desugarer converts the input into *Core*, which is GHC’s central intermediate representation. We extend the desugarer with a flattening transformation that maps nested array operations to array combinators that operate only on flat arrays containing elements of basic type; these combinators are provided by our array library *PArrays*. We have previously presented a formalisation of this flattening transformation [11]. GHC’s simplifier optimises the Core language using a wide range of source-to-source transformations [28, 30, 27]. We extend these transformations by equational loop fusion using GHC’s support for rewrite rules that are specified as pragmas in the source language [25]. The code generation itself is not affected.

2.1 Parallel Arrays: An Example

As an example, consider a function that multiplies a sparse matrix by a vector, where the sparse matrix is stored in the so-called *compressed sparse row* format [13]. This format represents a sparse row by an array of column-index/value pairs, where each pair represents a non-zero element of the sparse matrix. An array of these sparse rows implements a sparse matrix. To improve readability, we will take the freedom to use the special syntax `[:·:]` for the array constructor in the following Haskell code. Please note that, although we are using Haskell here, the presented concepts apply to other typed functional languages such as Standard ML, too.

```
type SparseRow    = [:(Int, Float):] -- index, value
type SparseMatrix = [:(SparseRow):]
```

Now consider the multiplication of a sparse matrix with a dense vector, resulting in another dense vector. Using a little more special syntax, namely array comprehensions,¹ we can implement sparse-matrix/vector multiplication as follows:

```
smvm      :: SparseMatrix → [:(Float):] → [:(Float):]
smvm sm vec =
  [:(sumP [:(x * (vec !: col) | (col, x) ← row:] | row ← sm:])
   products of one row
```

The function `sumP` adds up the elements of an array and the infix operator `(!:)` represents indexing.

This small algorithm already shows why conventional lists are often suboptimal for algorithms from scientific computing. These algorithms tend to use indexing in inner loops; an operation that is of constant cost for arrays and of cost proportional to the length of the indexed list for lists. In `smvm`, the indexing in the inner array comprehension is frequently executed and, as the index values depend on the input data, there is little scope for static optimisation.

Despite the bias towards arrays, the processed data structure as well as the computational behaviour of `smvm` are highly irregular, as the number of non-zero elements in different rows may vary significantly. Consequently, the principal data structure of conventional array languages like Fortran—namely regular arrays, where all sub-structures must be of the same size—is also not appropriate. Overall, nested arrays that allow sub-structures of varying size seem to be the most adequate structure. Other important algorithms, such as, adaptive iterative algorithms, have similar properties.

Blelloch [6] has demonstrated that irregular, nested arrays are well suited for expressing the parallelism in algorithms like `smvm` and others. In fact, the implementation technique presented in this paper was originally aimed at optimising node programs in parallel implementations of nested array languages. Consequently, our approach is suitable for a range of sequential and parallel architectures [10].

The reason for calling these arrays *parallel arrays* is, however, more fundamental. Whenever any element of a parallel array is demanded, all elements are evaluated.² In other words, we assume a parallel evaluation semantics for arrays. We distinguish combinators over parallel arrays from those over lists by the suffix “P”, as in `sumP`.

¹There is no conceptual reason for the special syntax. It is just a matter of presentation.

²In a lazy language like Haskell, this means that they are evaluated to WHNF.

2.2 Changing the Data Representation

The first program transformation that we apply to a definition like `smvm` is called *flattening* and its original formulation is due to Blelloch & Sabot [8]. It converts data structures containing arrays of non-basic type into structures containing arrays of basic type only and rewrites nested array iterations into a single, flat iteration. The benefit is twofold: (1) Traversals over arrays of basic types use the processor cache more efficiently, as the elements can be stored unboxed in a contiguous memory region; and (2) it improves load balancing and data distribution in a parallel implementation.

We recently extended flattening to be suitable for typed functional languages such as Haskell and Standard ML [11]. In the following, we summarise the main properties of the code resulting from flattening, as flattened code is the input to the transformations discussed in this paper.

To distinguish the type and function names of the original from the flattened variants, we underline the latter. Central to understanding flattening is to understand how it affects types. Most importantly, flattened data structures only contain arrays of basic type (`Int`, `Float`, and so on); to distinguish them from the nested arrays of the source language, we denote them with the type constructor `PArray`. Flattening represents arrays of arrays—e.g., `[::(Float)::]`—by two arrays: a *data vector* containing all the elements and a *segment descriptor* storing the structure, i.e., the lengths of all sub-arrays. For example, `[::(Float)::]` becomes `(Segd, PArray Float)` where

```
type Segd = PArray Int -- segment descriptor
```

So, the value `[::(1, 2, 4), [:(3, 5):]]` is now represented by the data vector `[1, 2, 4, 3, 5]` and the segment descriptor `[3, 0, 2]`. Moreover, arrays of pairs are represented by a pair of arrays. Considering these two rules, the type for `SparseMatrix` should become clear.

```
type SparseRow    = (PArray Int, PArray Float)
type SparseMatrix = (Segd, SparseRow)
```

Having presented the types, let us have a look at the code transformation. The flattened code `smvm` makes use of the three new functions `zipWithP`, `backpermuteP`, and `sumSP` operating on flat arrays.

```
smvm :: SparseMatrix → PArray Float → PArray Float
smvm (segd, (ind, val)) vec =
  sumSP segd (mulV (backpermuteP vec ind) val)
where
  mulV = zipWithP (*)
```

A capital “S” in a function name indicates a segmented function, i.e., a function that takes the segmentation of an array into account. For example, `sumSP segd arr` individually sums up the sub-arrays of `arr` as determined by the segment descriptor `segd`, resulting in an array of sums. Segmented functions are known to be useful for the high-performance implementation of array algorithms [5, 7]. Moreover, the function `backpermuteP` is a permutation operation where the permutation vector gives the source rather than the destination index of each value. The flattened `smvm` correlates to the original `smvm` as follows: The `backpermuteP` performs all the indexing operations `vec !: col` of the original code in one collective operation. Then, `mulV` computes all products between vector and matrix elements; and finally,

sumSP sums up all products that belong to the same row of the matrix as indicated by the segment descriptor *segd*.

Flattened array code uses a whole range of array combinators, such as *sumSP* and *backpermuteP*, the most common of which are defined in Appendix A. The efficient implementation of programs constructed from these combinators is the aim of the present paper—whether these programs were generated by flattening or are hand coded is secondary.

2.3 Fusion of Array Operations

From now on, let us assume that, as in *smvm*, all array computations are based on the combinators from Appendix A and operate on flat arrays only—with one exception. For reasons explained in Section 5.1, we can extend the scope of fusion by also permitting arrays of pairs.

In *smvm*, we have three array combinators applied in sequence, all of whose index space ranges over the non-zero elements of the sparse matrix. Considering cache performance, it is inefficient to implement the combinators in three distinct iterations unless all data fits into the cache. This is where loop fusion comes into play. It amalgamates all three combinators of *smvm* into one, which can be implemented by a single iteration over the non-zero elements of the sparse matrix. This leads us to the question of which combinator can express the result of fusion, in *smvm* and in general.

It turns out that, as in the case of list fusion based on *foldr/build* [15], we need two elementary array combinators that can express all the others as well as all possible outcomes of fusion. The first of these two combinators, the array constructor *replicateP n v*, simply creates an array of length *n* with all elements having the value *v*. The second, the array consumer *loopP*, is more involved, as it can express various forms of mapping, reduction, scan, and permutation operations as well as combinations of them. Its type is

```
loopP :: (e → a → (Maybe e', a)) -- m: mutator
      → (a → a)                  -- h: segment
      → (a → Bool)                -- p: collect
      → a                          -- accumulator
      → SPArray e
      → (SPArray e', PArray a, a)
```

where *SPArray* is a *PArray* paired with its segment descriptor. An expression of the form *loopP m h p a pa* traverses the segmented input array *pa* once from left to right. The most interesting argument is the mutator *m*, which can be regarded as a combination of the functions that are passed to the list combinators *map*, *filter*, and *foldl*. It is applied to each array element *e*, which it either maps to *Just e'*; or, if its filtering component requires so, to *Nothing*.³ In addition, the mutator can use and update the accumulator *a*, which enables the implementation of fold and scan functionality.

The remaining two arguments, *h* and *p*, determine the behaviour of the loop at segment boundaries. The function *h* updates the accumulator at each segment boundary; for example, in the case of a segmented sum, it would reset the accumulator. Finally, *p* is used at the end of each segment to determine whether the current accumulator value is included into the accumulator array *PArray a* of the result. Overall, the loop produces three results: (1) an array containing all *e'* returned by the mutator *m*, (2) an array of accumulator values sampled in dependence on *p*, and (3)

³In Haskell, *Maybe a* is a value that is either *Nothing* or *Just v*, where *v* must be of type *a*.

the final accumulator value. Section 4.1 defines *loopP* more precisely.

Given *loopP*, we can now present the fully fused variant of *smvm*, which is the result of applying the technique presented in this paper to *smvm* (where *zipP* is the array version of Haskell’s list function *zip*):

```
smvm :: SparseMatrix → PArray Float → PArray Float
smvm (segd, (ind, val)) vec = projAccs
  (loopP combine (const 0) (const True) 0 segdIndVal)
  where
    segdIndVal      = (segd, (zipP ind val))
    combine (i, v) acc = (Nothing,
                          (vec !: i) * v + acc)
```

The code essentially traverses the segmented array of index/value pairs *segdIndVal* and in each iteration extracts the vector element *vec !: i* corresponding to the column index of a matrix value *v*. The product of the two values is added to the running sum maintained in the accumulator. The running sum is stored in the accumulator array (this is what *const True* specifies) and reset to zero at each segment boundary (this is what *const 0* specifies).

How we get from *smvm* to the fully fused code of *smvm* is described in the rest of this paper. Figure 4, in Section 6, shows the improved running time of the fused code over *smvm* as well as over the same algorithm implemented with standard Haskell arrays or with lists.

3. BASIC ARRAYS

Before diving into the details of loop fusion, let us look more closely at the array type. *PArray e* is the type of arrays containing elements of basic type *e*, where *e* is guaranteed to be stored unboxed (cf. [29] regarding unboxed values in Haskell). The restriction on element types is enforced by a type class *PAE* (meaning “Parallel Array Element”):

```
class PAE e where
  (!:) :: PArray e → Int → e
  ...
```

There are instances of *PAE* for all basic Haskell types, such as *Int*, *Float*, and so on. The use of overloading has the advantage that we can store elements unboxed in these arrays, which in turn improves performance significantly (cf. [33] for a benchmark). In fact, this overloading-based approach to unboxed arrays was inspired by the Haskell libraries of GHC [36], which in turn follow the Clean array library [16].

The above class definition is not complete. In fact, most of the class methods are impure and operate on a mutable variant of *PArray*. These state-based, monadic functions include operations for allocating uninitialised arrays, and reading and writing to these arrays. The impure functions are not visible to a user of the library and as the details are not relevant to this paper, we spare them. More about destructive array functions in Haskell can be found in [26].

The impure methods of *PAE* are used to implement the purely functional, basic array combinators *replicateP* and *loopP* on which array fusion is based. Finally, the user-visible interface of the *PArrays* library is defined in terms of the basic combinators *replicateP* and *loopP*. It includes array versions of the common list processing functions, for example, *mapP*, *zipWithP*, *filterP*, *enumFromToP*, *foldP*⁴,

⁴The function *foldP* leaves the reduction order unspecified,

$sumP$ as well as segmented versions where this makes sense. Moreover, array-specific functions, such as permutations (e.g., $backpermuteP$), are included.

As mentioned earlier, we assume a parallel evaluation semantics for arrays; i.e., whenever a single element of an array is demanded, the whole array is evaluated. Such arrays are also called *strict*. They are less flexible than lazy arrays [2], but necessary as we would like to guarantee a clear parallel interpretation of our array combinators [11].

Furthermore, we use the following two abbreviations for segment descriptors and segmented arrays:

```
type Segd      = PArray Int
type SArray e = (Segd, PArray e)
```

As an example of a simple combinator-based array program, consider the following definition:

```
sumSq :: Int → Int
sumSq n = sumP (mapP square (enumFromToP 1 n))
where
  square x = x * x
```

First, $sumSq$ generates an array containing the values 1 up to n over which, then, the scalar function $square$ is mapped. Finally, the result of applying $mapP$ is summed up. The two intermediate arrays (produced by $enumFromToP$ and $mapP$, respectively) can both be eliminated by fusion.

4. FUSION: THE SIMPLE CASE

Section 2.3 already mentioned that previous work indicates that successful fusion techniques are based on a small number of combinators over which equational fusion rules are defined and which are sufficiently expressive to denote all computations that we want to consider for fusion [15, 24, 34]. The most successful approach—in terms of actual usage—to equational fusion, namely $foldr/build$ [15], is restricted to handling structure traversals that are expressed in terms of a combinator library—such as, e.g., the Haskell Prelude. We impose the same restriction. However, as functional arrays are generally processed by combinators implementing bulk operations and not by recursive traversals, this restriction is of little practical consequence in the case of arrays.

In the remainder of this section, we will first define the meaning of the $loopP$ combinator in more detail, and then, explore the basic fusion rules. We defer the more complicated cases to the following section.

4.1 The $loopP$ Combinator

Now, we will finally provide a precise definition of the semantics of the combinator $loopP$, whose type and intuition we discussed in Section 2.3. For the purpose of keeping the definition concise, we represent arrays as lists and segmented arrays as list of lists. In other words, we define the denotational semantics of $loopP$, but ignore the operational properties of arrays for the moment.

```
type PArray a = [a] -- Just for the purpose...
type SArray a = [[a]] -- ...of this definition
```

```
loopP m h p a0 [] = ([], [], a0)
loopP m h p a0 (seg : segs) = (seg' : segs', as', a')
```

and thus, requires an associative reduction function—this is useful for the parallel implementation.

```
where
  (seg', a) = loop1 m (h a0) seg
  (segs', as, a') = loopP m h p a segs
  as' = if p a then a : as else as
```

```
loop1 :: (e → a → (Maybe e', a))
       → a → PArray e → (PArray e', a)
```

```
loop1 m a'0 [] = ([], a'0)
loop1 m a'0 (e : es) = (es'', a'')
```

```
where
  (me, a') = m e a'0
  (es', a'') = loop1 m a' es
  es'' = case me of
    Nothing → es'
    Just e' → e' : es'
```

The auxiliary function $loop1$ handles the traversal of a single segment of the segmented array. For each segment, it returns the corresponding segment of the result array (which will be shorter if m ever returns *Nothing*) as well as the value of the accumulator after the whole segment has been processed. The function $loopP$ invokes $loop1$ once for each segment and combines the resulting segments into the first component of its result. Moreover, it applies h to the accumulator value before processing a segment and uses p to determine the accumulator values that are collected into the second component of the overall result.

The arguments h and p are only needed to implement array traversals which take the segmentation structure of an array into account. As some functions, most notably $mapP$, are entirely oblivious to an array's segmentation, we make their definitions more readable by introducing the abbreviation $loopA$:

```
loopA m a pa = loopP m id (const False) a pa
```

In fact, we will define some fusion rules in terms of $loopA$ instead of $loopP$ where this increases readability without omitting important information.

For example, we can now define $mapP$ and $sumP$ as follows:

```
mapP f = let m e _ = (Just (f e), ()) in loopA m ()
sumP = let m e a = (Nothing, e + a) in loopA a 0
```

Given the definition of $loopP$, it is not difficult to see that it can only produce arrays of a size smaller or equal to that of its input array, which raises the question as to how we generate new arrays; for example, to implement a combinator, such as $enumFromToP$, which we used in $sumSq$ in Section 3.

4.2 Array Generation with $replicateP$

Fortunately, it turns out that a simple combinator, which corresponds to the list function $replicate$ in Haskell, is all that is needed to generate arrays:

```
replicateP :: PAE e ⇒ Int → e → PArray e
```

It generates arrays of arbitrary size, where all elements are initialised to the same value.

With the help of $replicateP$, we can indeed implement the function $enumFromToP$ mentioned in the previous section:

```
enumFromToP :: (PAE e, Num e)
             ⇒ e → e → PArray e
```

```

enumFromToP from to = projMap $
  loopA mut from (replicateP len 1)
  where
    len      = (to - from + 1) 'max' 0
    mut e acc = (Just acc, acc + e)

```

where $projMap (pa, -, _) = pa$ projects the first component of result of a loop. Unfortunately, this is a rather wasteful definition. First, $replicateP$ constructs an array consisting of 1s only; and then, the actual result is computed from the array of 1s by $loopA$. Obviously, it would be more efficient to generate the resulting array in a single iteration, but neither $loopP$ nor $replicateP$ are able to do the job on their own.

An interesting property of parallel arrays of unit type $()$ can help us out of this dilemma: The only informational content of an array of type $PArray ()$ is its length. This property depends on the parallel evaluation semantics, as it guarantees that the whole array is undefined if only a single element is undefined (which is not the case for lazy arrays). As a consequence, there is no need to ever construct an array of type $PArray ()$; simply storing its length is sufficient. We can implement this special representation easily using the overloading provided by the type class PAE :

```

instance PAE () where
  _ !: i = ()
  ...

```

As a result, the evaluation of $replicateP n ()$ is very cheap. So, it is the ideal candidate for providing an array over which $loopP$ can iterate to produce a new array. This indeed leads us to our first fusion rule:⁵

$$\langle \text{loop/replicate fusion} \rangle \forall m a n e .$$

$$loopA m a (replicateP n e) \mapsto$$

$$loopA (\lambda _ \rightarrow m e) a (replicate n ())$$

It states that any occurrence of $replicateP$ followed by $loopA$ should be replaced by generation of a unit array followed by a modified call to $loopA$, where the mutator m contains the constant e inline. Due to the optimised representation of unit arrays, this rule removes the explicit construction of the intermediate array. It should be obvious how the rule $\langle \text{loop/replicate fusion} \rangle$ optimises the implementation of $enumFromToP$ stated earlier.

Generally, we denote rewrite rules as follows:

$$\langle \text{rule name} \rangle \forall v_1 \dots v_n. exp_1 \mapsto exp_2$$

where the v_i are the free variables in the rules. These rules should be read as *replace every occurrence of exp_1 by exp_2* .

4.3 Fusion Rules for Loops

More sophisticated is the fusion of two consecutive loops:

$$\langle \text{loop/loop fusion} \rangle \forall m_1 a_1 m_2 a_2 pa.$$

$$loopA m_2 a_2 (projMap (loopA m_1 a_1 pa)) \mapsto$$

⁵The same rule for $loopP$ does not add anything new, so we leave it with the simpler $loopA$.

```

let
  m3 e (acc1, acc2) =
    case m1 e acc1 of
      (Nothing, acc'1) -> (Nothing, (acc'1, acc2))
      (Just e' , acc'1) ->
        case m2 e' acc2 of
          (Nothing, acc'2) -> (Nothing, (acc'1, acc'2))
          (Just e2 , acc'2) -> (Just e2 , (acc'1, acc'2))
in
  drop1stAcc (loopA m3 (a1, a2) pa)

```

The most interesting aspect is the way in which the mutators m_1 and m_2 of the two loops are combined into one, called m_3 . The new mutator first applies m_1 and, only if that returns a result, with $Just e'$, that would have been stored in the intermediate array, it calls m_2 . Moreover, the accumulators of the two loops are combined into a single accumulator of pair type. As a consequence, the second and third component of the result of the fused loop have to be adjusted to obtain the same result that is produced by the loops before fusion. This adjustment is performed by

$$drop1stAcc :: (PAE e_1, PAE e_2) \Rightarrow$$

$$(x, PArray (e_1, e_2), (e_1, e_2)) \rightarrow (x, PArray e_2, e_2)$$

$$drop1stAcc (pa, as, (a_1, a_2)) = (pa, mapP snd as, a_2)$$

To cater for the case where more than two consecutive loops are to be fused, we have to ensure that any $drop1stAcc$ followed by a $projMap$ is eliminated. We achieve this by a simple rewrite rule:

$$\langle \text{projMap/drop1stAcc elimination} \rangle \forall v.$$

$$projMap (drop1stAcc v) \mapsto projMap v$$

4.4 Rewrite Rules in GHC

GHC supports the specification of equational rewrite rules that are used by the Simplifier (see Figure 1) to apply domain-specific optimisations [25]. These rules are included as pragmas into source files. In our case, they are part of the definition of the $PArrays$ library.

For example, the elimination rule for $drop1stAcc$ stated at the end of the previous subsection is denoted as

```

{-# RULES
"projMap/drop1stAcc"
forall v. projMap (drop1stAcc v) = projMap v
#-}

```

This rule will make the optimiser spot occurrences of the pattern $projMap(drop1stAcc v)$, where v can be an arbitrary expression, and replace it by the right-hand side of the rule.

The rewrite mechanism offered by GHC facilitates the implementation of array support considerably—it would not have been possible to provide a prototype implementation in the given time frame otherwise. However, the implementation of such rules in practise can be challenging, due to the interaction with other optimisations such as inlining. On the other hand, the optimisations provided by the Simplifier by default are crucial to simplify and efficiently implement the loop bodies generated by our library.

4.5 How it all works together

Let us now go back to the $sumSq$ example (from Section 3) to see how the different techniques work together. We start from the definition of $sumSq$ after unfolding the definitions

for $mapP$ and $sumP$ as well as that for $enumFromToP$ after application of (loop/replicate fusion):

```
sumSq n =
  projAcc $ loopA f1 0 $
  projMap$ loopA f2 () $
  projMap$ loopA f3 n pa
  where
    f1 e a = (Nothing, e + a) -- sumP
    f2 e _ = (Just (square e), ()) -- mapP
    f3 _ a = (Just a, a + 1) -- enumFromToP
    size = (m - n + 1) 'max' 0
    pa = replicateP size ()
```

As pa and $size$ do not change, we omit them in the next step of the transformation. The rule (loop/loop fusion) is immediately applicable, and fuses the loops of $enumFromToP$ and $mapP$ into a single loop. For f_2 as well as f_3 , the first component of the result pair is a *Just* value, independent of the actual values of their arguments. Therefore, the case-distinction introduced by the fusion rule can be eliminated by GHC’s Simplifier. Moreover, (projMap/drop1stAcc elimination) fires once. Overall, we get the definition

```
sumSq n =
  projAcc $ loopA f1 0 $
  projMap$ loopA f4 (n, ()) pa
  where
    f4 _ (a, _) = (Just (square a), (a + 1))
    f1 e a = (Nothing, e + a)
```

The next application of (loop/loop fusion), then, fuses the remaining two loops, resulting in a definition with a single loop, which does not create any array at all:

```
sumSq n =
  projAcc $ loopA f5 ((n, ()), 0)
  where
    f5 _ ((a1, _), a2) = (Nothing,
      ((a1 + 1, ()), square a1 + a2))
    size = (m - n + 1) 'max' 0
    pa = replicateP size ()
```

5. ADVANCED FUSION

In the presentation of fusion, so far, we have ignored two additional complications. Firstly, some functions, like $zipWithP$, consume two or more arrays in lockstep. We might want to fuse such functions with the producers of both arrays. This is tricky, and important list-based fusion techniques are not able to fuse two consumers. Secondly, for a fusion rule to fire, the fused combinators have to be adjacent in the program code, which we usually achieve by combining inlining with simplification. There are, however, situations, such as the function boundaries of recursive functions, where inlining is not applicable. In the rest of this section, we shall illustrate our solutions to these two problems.

5.1 Traversing Two Arrays Simultaneously

The flattened version of the sparse-matrix vector multiplication code $smvm$ from Section 2.2 contained the equation

$$mulV = zipWithP (*)$$

which implements vectorised multiplication—i.e., multiplication lifted pointwise. In $smvm$, $mulV$ was applied to the result of a $backpermuteP$ and one of the arguments to

$smvm$. In Appendix A, $backpermuteP$ is defined in terms of $loopP$. So, we would obviously like to fuse $zipWithP$ and $backpermuteP$. Furthermore, in some applications of $smvm$ —after the definition of $smvm$ has been inlined—the second argument to $zipWithP$ may also be the result of an application of $loopP$ or $replicateP$. In this case, we want to fuse $zipWithP$ with both of its arguments. Otherwise, if only one of the arguments is in the “right” form (i.e., the result of $replicateP$ or $loopP$), we want fuse this argument into the loop. The list fusion method `foldr/build` implemented in GHC is not able to perform fusion in all these cases [15]—instead, fusion is only possible for a single argument that has to be fixed when implementing zip . Other approaches [34, 24] claim to fuse zip fully, but the details are not entirely clear and no working system including these techniques is available.

Our approach to this problem consists of two components: (1) an unboxed representation for arrays of pairs and (2) a set of specialised fusion rules.

5.1.1 Arrays of Pairs are Pairs of Arrays

Let us now consider functions that consume two or more arrays in lockstep. As with generators, we can reduce the problem to a single function, in this case $zipP$. For example, we can implement $zipWithP$ by $zipP$ followed by a loop:

```
zipWithP :: (PAE a, PAE b, PAE c)
  => (a -> b -> c)
  -> PArray a -> PArray b -> PArray c
zipWithP f pa1 pa2 = loopA applyF () (zipP pa1 pa2)
  where
    applyF (e1, e2) _ = (Just (f e1 e2), ())
```

In Section 2.2, we mentioned that flattening represents an array of pairs as a pair of arrays—as witnessed in the type $SparseRow$. This actually means that $zipP$ does not have to traverse the argument arrays at all, nor does it produce a new array. Instead, it produces a pair containing references to the two arrays. Having only one function, namely $zipP$, which handles the lockstep traversal of multiple arrays, simplifies the corresponding fusion rules significantly.

Nevertheless, we need to represent values of type $PArray (e1, e2)$ and define array operations on them, while retaining the property that arrays store unboxed basic data only. Or in other words, how can we implement an instance of PAE for pairs? We do so by performing a runtime dispatch on the element type of arrays. In fact, as all elements of $PArray$ must be instances of the type class PAE , it suffices to define an instance for pairs that goes as follows:

```
instance (PAE e1, PAE e2) => PAE (e1, e2) where
  (PAPair pa1 pa2) !: i = (pa1 !: i :: e1, pa2 !: i :: e2)
```

Here $PAPair$ is the pair constructor that we use to represent arrays of pairs.

It is interesting to note the relationship between our use of overloading and Harper & Morrisett’s [17] intensional type analysis. As Weirich [38] has also observed, type classes can be used to implement some forms of intensional type analysis. In essence, the method dispatch via dictionaries that implements type classes is used to realise Harper & Morrisett’s **typecase**. This relationship provides a route to implementing our approach in a compiler that supports **typecase**, but not type classes.

5.1.2 Fusion Rules for *zipP*

We now have to consider two cases for which the fusion rules should fire: The argument array of *zipP* can either be a result of an application of *replicateP* or *loopP*. For the former, we can drag the generation into the succeeding loop, thereby eliminating *zipP* altogether. Again, we specify the rules for *loopA* only, to keep the presentation clearer:

$$\begin{aligned} \langle \text{zip/replicate fusion} \rangle \forall m a n e_1 es_2. \\ \text{loopA } m a (\text{zipP } (\text{replicateP } n e_1) es_2) \mapsto \\ \text{loopA } (\lambda e \rightarrow m (e_1, e)) a es_2 \end{aligned}$$

We omit the symmetric rule for fusing the second argument.

We handle fusion of a *loopP* that occurs as an argument to *zipP* by propagating the *loopP* through *zipP*. The side condition is that the mutator of the loop never drops elements, which implies that the loop preserves the length of the array. We can express this constraint by using an auxiliary function restricting the loop function:

$$\begin{aligned} \text{mapSFL} :: (e \rightarrow a \rightarrow (e', a)) \\ \rightarrow (e \rightarrow a \rightarrow (\text{Maybe } e', a)) \\ \text{mapSFL } (e, a) = (\text{Just } e, a) \end{aligned}$$

Now the fusion rule is

$$\begin{aligned} \langle \text{zip/loop propagation} \rangle \forall f a es_1 es_2. \\ \text{zipP } (\text{loopA } (\text{mapSFL } f) a es_1) es_2 \mapsto \\ \text{loopA } (\text{mapSFL } f') a (\text{zipP } es_1 es_2) \\ \text{where} \\ f' (e_1, e_2) a = \text{let } (e'_1, a') = f e_1 a \text{ in } ((e'_1, e_2), a') \end{aligned}$$

Again, we omit the corresponding rule for the second argument. The use of *mapSFL* allows us to express the side condition without leaving the framework of applicative rewrite rules, which is what GHC supports. Having to use these special functions may seem too restrictive, but consider that this is not visible at the interface of the array library, which consists of combinators such as *mapP* and *foldP*. We use *mapSFL* only in the library-internal implementation of the combinators.

5.2 Fusion over Function Boundaries

Equational fusion has a serious weakness: For the fusion rules to fire, the combinators have to appear adjacently in the program code. Given specialised functions such as *mapP* and *foldP*, which are defined in terms of *loopP*, this is virtually never the case in the source code. So, the whole technique relies on other optimisation techniques—in particular, inlining—to convert the code such that fusion rules can fire. However, there are situations, where inlining alone does not help. Consider the following (slightly artificial) function definition:⁶

$$\begin{aligned} \text{foo} &:: P\text{Array } Int \rightarrow Int \\ \text{foo } xs \mid \text{nullP } xs &= 0 \\ &\mid \text{otherwise} = \text{let} \\ &\quad n = \text{lengthP } xs \\ &\quad v = \text{sumP } xs \\ &\quad \text{in} \\ &\quad \text{foo } (\text{replicateP } (n - 1) v) \end{aligned}$$

The function reduces its argument array to a scalar from which it produces another array, which is passed to the next recursion. In other words, the array created by the subexpression *replicateP (n - 1) v* is immediately consumed and

⁶This situation also occurs frequently in realistic functions.

discarded in the next recursive step. It is clearly a waste of time and memory to build it in the first place.

As discussed earlier, *sumP* is implemented in terms of *loopP*, so after inlining *sumP* the array constructed by the application of *replicateP* is immediately consumed by a *loopP* in the following recursion. A perfect opportunity for fusion, but the use of *replicateP* and *loopP* are separated by a function boundary. Thus, the fusion rule cannot fire!

This is very similar to the situation, where a primitive value is boxed for a (recursive) function call, only to be immediately unboxed by the callee. Recognising that *replicateP* is an array constructor and *loopP* a destructor, the situation is very similar indeed. With this insight, it is not surprising that the idea behind the constructor specialisation technique of [25] provides the seed for a solution of our problem.

The essential idea is to generate a specialised version of *foo* for the case, where it is called with an argument constructed by *replicateP*. In this case, we can pull the use of *replicateP* into *foo*'s body and obtain the following variant:

$$\begin{aligned} \text{fooR} &:: Int \rightarrow Int \rightarrow Int \\ \text{fooR } n v \mid \text{nullP } xs &= 0 \\ &\mid \text{otherwise} = \text{let} \\ &\quad n = \text{lengthP } xs \\ &\quad v = \text{sumP } xs \\ &\quad \text{in} \\ &\quad \text{foo } (\text{replicateP } (n - 1) v) \\ \text{where} \\ &\quad xs = \text{replicateP } n v \end{aligned}$$

Now *sumP* is immediately applied to the result of *replicateP* and, after some inlining, (loop/replicate fusion) can fire. All that is left to be done, is to replace every occurrence *foo* with an argument applying *replicateP* by an appropriate call to *fooR*. Again, GHC's rewrite rules come to the rescue. For each specialised version of a function, we generate a simple rule like this:

$$\begin{aligned} \langle \text{foo/replicateP specialisation} \rangle \forall n v. \\ \text{foo } (\text{replicateP } n v) \mapsto \text{fooR } n v \end{aligned}$$

When applied in the body of *fooR* itself, *fooR* becomes recursive. In combination with (loop/replicate fusion), we get

$$\begin{aligned} \text{fooR} &:: Int \rightarrow Int \rightarrow Int \\ \text{fooR } n v \mid n == 0 &= 0 \\ &\mid \text{otherwise} = \text{fooR } (n - 1) v \\ \text{where} \\ &\quad v = \text{projAcc } (\text{loopA } \text{mut } 0 (\text{replicateP } n ())) \\ &\quad \text{mut } _ (i, a) = (\text{Nothing}, (i + 1, i + a)) \end{aligned}$$

Voila! We have successfully eliminated all arrays. As mentioned, *replicateP n ()* does not actually construct an array, so *loopA* will compile to a simple loop adding up 1 to *n*.

6. PERFORMANCE

The figures presented in this section were obtained with a first experimental implementation of equational loop fusion on the basis of GHC's rewrite rules. We used the current development version of GHC (version 5.01) with the following optimisation options `-O2 -fliberate-case-threshold100 -funfolding-use-threshold10 -fno-method-sharing`. In addition, we patched the compiler to use a maximum worker-wrapper argument count of 20 (rather than the default of 6). All C code was compiled with `gcc 2.96` using `-O2`. All

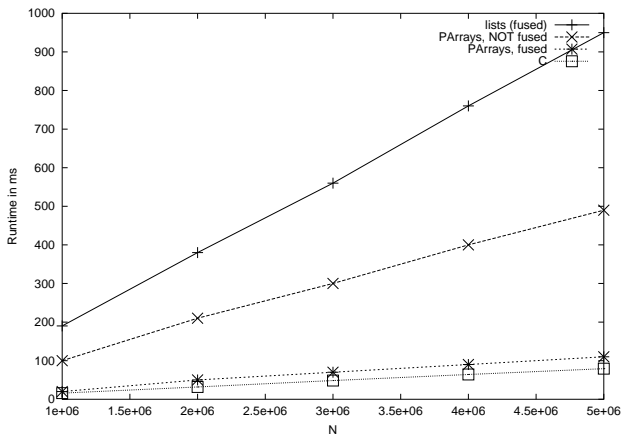


Figure 2: Performance of *sumSq*

tests were executed on an unloaded 333MHz PII with 256kB second level cache, running GNU/Linux.

6.1 Basic Loop Kernels

6.1.1 Sum-Square

The function *sumSq* from Section 3 is an extreme example that demonstrates the possible impact that loop fusion can have, as all intermediate structures can be eliminated. Figure 2 displays the performance of applying *sumSq* to values between 1,000,000 and 5,000,000. There are four versions: (1) “fused”, fully fused and optimised; (2) “not fused,” but still optimised; (3) “lists (fused)”, the Haskell program *sum* (*map square* [1..*n*]), which GHC fuses using *foldr/build*; and (4) “C”, the following C code:

```
result = 0;
for (i = 0; i < size; i++)
    result += i * i;
```

The fully optimised and fused array code is only 26% to 39% slower than the C version. Moreover, loop fusion improves the running time of this function by a factor of 4.5 to 5 over the flattened code as it can remove all arrays from this code. The list-based program is slower, as GHC manages to eliminate only one out of the two intermediate structures with its current Prelude definitions.

6.1.2 Sieve of Eratosthenes

The second benchmark is a simple version of the Sieve of Eratosthenes to compute the prime numbers up to a given bound. Using standard Haskell arrays, the algorithm is as follows:

```
primes :: Int -> [Int]
primes n | n <= 2 = []
         | otherwise =
```

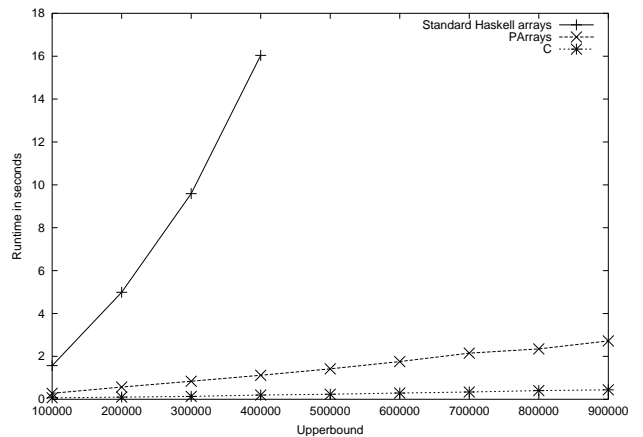


Figure 3: Performance of the prime sieve

let

```
sqrPrms = primes $ ceiling (sqrt (fromIntegral n))
sieves  = concat
          [[2 * p, 3 * p .. n - 1] | p <- sqrPrms]
range   = (2 :: Int, n - 1)
flags   = accumArray (&&) True range
          (zip sieves (repeat False))
```

in

```
[n | (n, f) <- assoc flags, f]
```

Figure 3 displays the execution times for (1) standard Haskell arrays, (2) an equivalent program using *PArrays*, and (2) a corresponding C program based on inplace updates. This benchmark clearly shows that *PArrays* perform at least an order of magnitude better than standard Haskell arrays in GHC. However, the hand-written C code is still by a factor of 4 to 5.5 faster than our array library. The main reason for this performance gap is that the code requires a so-called “default back permute”, a permutation function very similar to standard Haskell’s *accumArray*. This function cannot be expressed with *loopP* in its current form, which means that it cannot fuse with adjacent loops.

6.1.3 Sparse Matrix Vector Multiplication

Figure 4 displays the running times for the sparse matrix vector multiplication *smvm* applied to a set of matrices with 160,000 non-zero elements, but varying density (from dense to 0.1% non-zero elements). The figure contains curves for (1) standard Haskell arrays, (2) *PArrays* optimised, but not fused, (3) *PArrays* fully fused, and (4) hand-written C code. The version of the code using standard Haskell arrays is fused by GHC using *foldr/build* (Haskell arrays are constructed and reduced via lists). Nevertheless, they are not able to compete with the code based on *PArray*. Comparing the execution times for the fused and not fused, but flattened *PArray* code, we see that loop fusion improves the performance of the code by a factor of 4 to 6. Nevertheless, the hand-coded C program is still nearly a factor of 2 faster than the fused *PArrays* code. As there is still one unboxing operation performed per segment in the flattened matrix representation, we hope to be able to close that gap further by improving unboxing.

We have also tested a purely list-based version of *smvm*,

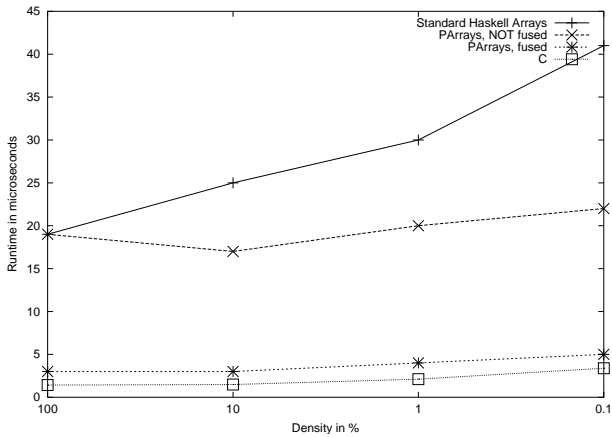


Figure 4: Timing of *smvm* (160k non-zero elements)

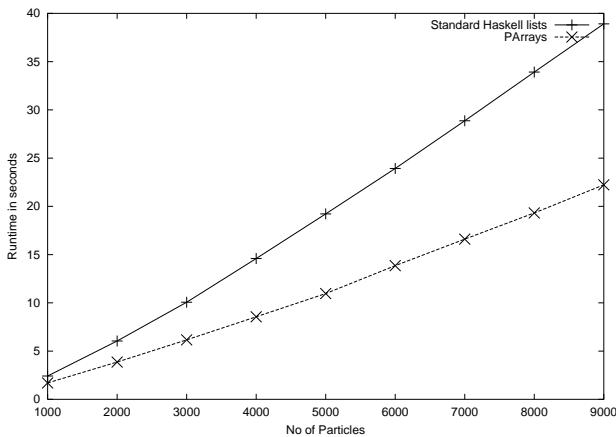


Figure 5: Performance of the Barnes-Hut code

but despite `foldr/build`, the code was too slow to be displayed in the graph.

6.2 Barnes-Hut N -Body Code

The largest example that we have tested to date is an implementation of the Barnes-Hut N -body algorithm [3], which computes the gravitational forces between a set of particles. This code is highly irregular and centred around a tree-shaped spatial decomposition. We have compared a standard Haskell implementation of the code with a version using a combination of arrays and trees as discussed in [19]. For the benchmark, we used a standard “Plummer” distribution of 1000 to 9000 particles. The array-based code clearly outperforms the standard Haskell code; although, the absolute performance still leaves significant room for improvement.

7. CONCLUSIONS

The two main techniques that we use in the implementation of arrays, flattening and fusion, happen in two different stages of the compilation, yet they do interact. The initial motivation to use fusion was to ease shortcomings of the flattening transformation by combining sequences of array

traversals into fewer, but more complex operations that exhibit better locality of reference. More surprisingly, however, flattening, or more precisely the flattened representation of the data types, also simplifies fusion: By expressing all array generators in terms of *replicateP* of unit type, we could simplify the framework. Similarly, the *zip* problem is simplified by the fact that *zipP* itself does not actually produce a new array, since arrays of pairs are represented by a pair of arrays.

7.1 Related Work

Loop fusion for imperative languages is well researched [1, 39, 21, 22, 31]. However, the challenges and techniques of loop fusion in imperative and functional contexts differ significantly. The extensive use of index calculations and side effects in imperative array algorithms often requires sophisticated analysis techniques before loops can be manipulated. In functional and, in particular, combinator-based approaches the data flow is more explicit, which provides more scope for transformations. In a functional context, it is especially important to remove intermediate structures and superfluous copying and, instead, use update-in-place.

Anderson & Hudak [2] argue for monolithic, lazy arrays defined by Haskell array comprehensions and adapt subscript analysis, such that it can be used to implement some algorithms more efficiently. They focus on regular code.

Ellmenreich, Lengauer & Griebel [14] also handle Haskell array comprehensions and adapt an analysis that was originally introduced for imperative programs to the functional case. They also focus on regular code.

Chuang [12] introduced combinator-based arrays for ML. He stays quite close to typical list combinators, but also considers update-in-place. He mentions loop fusion briefly, but only in the form of typical list fusion rules, such as fusion of $map f \circ map g$. The emphasis is, again, on regular code.

O’Neill & Burton [23] introduce a method for fast persistent arrays. They, as well as related approaches, essentially aim at a fast, purely functional update operation for single elements without copying the whole array. This provides some of the efficiency gains of update-in-place, but completely ignores the issue of unboxing.

7.2 Future Work

So far, we only appeal to intuition to reason that the rewrite rules are indeed optimisations. A more systematic treatment requires a cost model which takes into account the cost of memory access, and ideally, even the memory hierarchy to guide transformations. Moreover, we are currently investigating how we can extend the scope of the transformation to include tupling; i.e., combining two independent loops over structures of the same size. Such a transformation cannot be expressed as a simple rewrite rule, as it requires that certain side conditions hold. Dependent types or constraint-based analysis may provide a solution here.

Moreover, we plan to integrate the approach presented here with our previous work on integrating fusion with a distributed implementation of arrays [20]. The distributed implementation will use the parallel semantics of *PArrays* to make use of multiple processing nodes.

7.3 Acknowledgements

We are greatly indebted to Simon Peyton Jones who helped us in many ways. He not only explained to us the subtleties

of GHC's rewrite rules and other optimisations, but also extended GHC with a number of optimisations that were crucial for us to make progress, and he pointed us to the constructor specialisation transformation that came in so helpful in Section 5.2. Furthermore, he and Kai Engelhardt gave us valuable feedback on early versions of this paper. Moreover, we are grateful to Kevin Glynn and Bernard Pope as well as the anonymous referees for their detailed comments and helpful suggestions.

8. REFERENCES

- [1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois, 1979.
- [2] S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 137–149, 1990.
- [3] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.
- [4] G. Blelloch, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming*, 2001. <http://www.cs.cmu.edu/~rwh/papers/triangulations/jfp.ps>. To appear.
- [5] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.
- [6] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
- [7] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug. 1993.
- [8] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.
- [9] D. Cann. Retire fortran? A debate rekindled. *Communications of the ACM*, 35(8):81, Aug. 1992.
- [10] M. M. T. Chakravarty and G. Keller. How portable is nested data parallelism? In *Proc. of 6th Annual Australasian Conf. on Parallel And Real-Time Systems*, pages 284–299. Springer-Verlag, 1999.
- [11] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In P. Wadler, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.
- [12] T. Chuang. A functional perspective of array primitives. In *2nd Fuji Int. Workshop on Functional and Logic Programming*, pages 71–90, 1996.
- [13] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.
- [14] N. Ellmenreich, C. Lengauer, and M. Griehl. Application of the polytope model to functional programs. In J. Ferrante, editor, *Proc. 12th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC'99)*. Computer Science and Engineering Department, UC San Diego, 1999.
- [15] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232. ACM, 1993.
- [16] J. H. v. Groningen. The implementation and efficiency of arrays in Clean 1.1. In W. Kluge, editor, *Proceedings of Implementation of Functional Languages, 8th International Workshop, IFL '96, Selected Papers*, number 1268 in LNCS, pages 105–124. Springer-Verlag, 1997.
- [17] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd Annual Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.
- [18] C. B. Jay. Partial evaluation of shaped programs: experience with FISH. In O. Danvey, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, 1999.
- [19] G. Keller and M. M. T. Chakravarty. Flattening trees. In D. Pritchard and J. Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470 in Lecture Notes in Computer Science, pages 709–719, Berlin, 1998. Springer-Verlag.
- [20] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In J. Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.
- [21] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320. Springer Verlag, 1993.
- [22] N. Manjikian. Combining loop fusion with prefetching on shared-memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP '97)*. IEEE Computer Society Press, 1997.
- [23] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, 1997.
- [24] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. The calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.
- [25] S. Peyton Jones, T. Hoare, and A. Tolmach. Playing by the rules: rewriting as a practical optimisation technique. 2001. Microsoft Research Cambridge. <http://research.microsoft.com/Users/simonpj/papers/rules.ps.gz>.
- [26] S. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
- [27] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *International*

Workshop on Implementation of Declarative Languages, 1999. <http://research.microsoft.com/Users/simonpj/papers/inline.ps.gz>.

- [28] S. L. Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Berlin, 1996. Springer-Verlag.
- [29] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the International Conference on Functional Programming and Computer Architecture*, 1991.
- [30] S. L. Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving bindings to give faster programs. In *Proceedings of the International Conference on Functional Programming*, 1996.
- [31] G. Roth and K. Kennedy. Loop fusion in High Performance Fortran. In *Conference Proceedings of the 1998 International Conference on Supercomputing*, pages 125–132. ACM Press, 1998.
- [32] S.-B. Scholz. On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proceedings of APL '98*, pages 40–45. ACM Press, 1998.
- [33] P. Serrarens. Implementing the conjugate gradient algorithm in a functional language. In W. Kluge, editor, *Proceedings of Implementation of Functional Languages, 8th International Workshop, IFL '96, Selected Papers*, number 1268 in LNCS, pages 125–140, 1997.
- [34] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 306–316. ACM Press, New York, 1995.
- [35] The GHC Team. Glasgow Haskell Compiler. <http://haskell.org/ghc/>, 2001.
- [36] The GHC Team. Haskell Libraries: Language support. <http://haskell.cs.yale.edu/ghc/docs/latest/set/sec-lang.html>, 2001.
- [37] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
- [38] S. Weirich. Type-safe cast: Functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM Press, 2000.
- [39] M. Wolf and M. Lam. An algorithmic approach to compound loop transformations. In T. G. A. Nicolau, D. Gelernter and D. Padua, editors, *Advances in Languages and Compilers for Parallel Computing*, pages 243–259. The MIT Press, 1991.

APPENDIX

A. THE ARRAY LIBRARY & RULES

Figure 6 displays part of the interface of the *PArrays* library and provides definitions for some of the combinators that can be defined in terms of *replicateP* and *loopP*.

```

data PArray e
type Segd      = PArray Int
type SPArray e = (Segd, PArray e)

idSAL :: PAE a => a -> a
idSAL acc = acc

noSAL :: PAE a => a -> ()
noSAL acc = ()

falseSPL :: PAE acc => a -> Bool
falseSPL _ = False

noAL :: ()
noAL = ()

mapEFL :: (PAE e, PAE e') =>
  (e -> e') -> (e -> a -> (Maybe e', a))
mapEFL f = \ e a -> (Just $ f e, a)

foldEFL :: (PAE e, PAE a) =>
  (e -> a -> a) -> (e -> a -> (Maybe (), a))
foldEFL f = \ e a -> (Nothing, f e a)

scanEFL :: (PAE e, PAE acc) =>
  (e -> a -> a) -> (e -> a -> (Maybe a, a))
scanEFL f = \ e a -> (Just a, f e a)

projMap :: (a, b, c) -> a
projMap (x, y, z) = x

projAccs :: (a, b, c) -> b
projAccs (x, y, z) = y

projAcc :: (a, b, c) -> c
projAcc (x, y, z) = z

mapP :: (PAE e, PAE e') =>
  (e -> e') -> SPArray e -> SPArray e'
mapP f =
  projMap $ loopP (mapEFL f) noSAL falseSPL noAL

filterP :: PAE e =>
  (e -> Bool) -> SPArray e -> SPArray e
filterP p =
  projMap $ loopP (filterEFL p) noSAL falseSPL noAL

enumFromToP ::
  Int -> Int -> SPArray Int
enumFromToP start end =
  projMap $ loopP (scanEFL (+)) idSAL falseSPL start pa
  where
    len = 0 `max` (end - start + 1)
    pa = replicateP len 1

foldP :: (PAE e, PAE e') =>
  (e -> e' -> e') -> e' -> SPArray e -> e'
foldP g n =
  projAcc . loopP (foldEFL g) idSAL falseSPL n

```

Figure 6: Common array combinators