An Implementation of the Haskell Array Prelude Using Pruned Radix Tries

FLARE Group
Computer Science Division
University College of Swansea
Report No FLARE/S/7

6th Mar, 1992

Abstract

As part of a collaborative project to investigate the applicability of functional programming to a variety of application areas we are using Haskell to implement a solution to a problem in computational fluid dynamics. In this implementation our solution makes extensive use of arrays. Arrays in the Glasgow Prototype Haskell Compiler are currently implemented as lists with the obvious consequences for update and access costs. Furthermore the current implementation does not cater for sparse vectors and matrices. We found the current implementation to be unusable for our application since for large arrays the execution time was excessive and the program often ran out of heap space. This report describes an implementation of a substitute module for arrays which attempts to overcome the disadvantages outlined.

The fact that problem invites a large amount of iterative numerical calculation also causes the sort of *space-leak* that can occur with lazy evaluation. Ways which deal with the space-leak problem are also discussed in the context of the array implementation.

1 Introduction

Declarative functional programming has been shown to be a promising approach to overcome the difficulties of writing large correct software systems and of exploiting the power of parallel computing. In the UK a FLARE (Functional Languages Applied to Realistic Exemplars) project is underway with the dual aims of transferring effective functional programming technology and tools from the laboratory into the hands of serious users, and demonstrating that functional languages are a suitable vehicle for effectively programming parallel machines.

As part of this project we have begun work on using Haskell to develop a computational fluid dynamics (CFD) exemplar. We have successfully taken a sequential Taylor-Galerkin/pressure-correction finite element algorithm [1, 2], implemented in FORTRAN and used extensively by the CFD research group at Swansea, and implemented it in Haskell.

As might be expected this implementation makes extensive use of arrays. Arrays in the Glasgow Prototype Haskell Compiler are currently implemented as simple lists which result in linear access and update costs. Our exemplar involves solving large but *sparse* systems of linear equations. Using the standard Array Prelude is very demanding on space when the problems are reasonably large in size, and we soon had problems with heap or stack overflow.

In order to make progress we were forced to build a more efficient representation until a proper direct access array was provided by the Haskell development team at Glasgow. Also we wished to deal with sparse vectors and matrices. This requirement means that even when direct access arrays are provided it would still be necessary to implement sparse versions of the data structures for the present exemplar.

We have developed an alternative Array Prelude which will cater for such structures and only store essential non-zero data.

The Haskell language has been designed with a non-strict semantics and the natural lazy evaluation approach can involve the storage of large numbers of unevaluated expressions which lead to what have been termed *space-leaks* [3]. To tackle the space leakage problem, some form of force evaluation may be necessary. Runciman [4] presents an elegant, simple and systematic strategy which partially forces evaluation. Directly applying his strategy to our implementation however has a side-effect. This, along with a modified strategy, is discussed in this paper.

Laziness may also have other unfortunate affects in numerical applications, for instance, when the order of calculation is important. If we consider the problem of Choleski matrix decomposition, then column computation sequence outperforms row computation sequence in that it generally accesses less data entries in each individual decomposition stage [5] resulting in less paging for large matrices. The order of evaluation is crucial in these methods, but in a completely lazy environment, computation is driven by data requirements and so a required order of computation may not be guaranteed. Some mechanism of forcing the evaluation is again essential.

We first describe the current implementation of arrays employed which stores sparse arrays efficiently and then discuss the force evaluation problem. We have compared the performance of our implementation (both with and without forced evaluation) with the performance of the current list implementation and the results show a significant improvement in performance.

2 Choice of data structure

For convenience we have made a restriction on the type used for indexing. Unlike the standard array which allows any type in the class `Ix`, we restrict it to `Int`. This is usually good enough for solving engineering problems and may make more efficient implementations possible. Array access with generalised indexing is still possible if we use the function

```
index ::  (i,i) -> i -> Int
```

in class `Ix`, which returns the integer index of a value in a pair of bounds. Because of this restriction, the targeted array is one-dimensional. This does not impose significant difficulties in practice as higher dimensionality can always be achieved by using an array of arrays or further introducing higher level indexing. Hence we will only consider the storage of sparse *vectors*.

In the first instance we consider three data structures for storing sparse entries in a vector.

1. **Index-value pairs.**
   Conventionally, we store the index-value pairs in a list. In the context

of Haskell, `[Assoc Int a]` is a natural choice. The beauty of this approach is its simplicity but it suffers from slow data access. Unless the array size is small, efficient access cannot be guaranteed and sorting items in the list is time consuming.

2. **Radix tries.**
   A radix trie is another choice. By a radix trie we mean a tree structure which does not keep keys in its internal nodes. The binary bit-pattern of the key then indicates the location of an item in the trie. It is possible to know in advance the size of a trie needed to accommodate all entries and hence we can put all leaves at the same depth. This has the advantage of making it easier to construct the trie from a list of leaves bottom-up.

   The following are some advantages of this approach:

   - Although there is an increase in the storage overhead compared to a simple linear data structure, we are guaranteed an $O(log(n))$ access time. Such a trie can be 'sparse' as sub-tries which contain only unwanted leaves can either be pruned from time to time or completely ignored. Therefore, this approach is likely to reduce the overall storage for a sparse vector.
   - The flexibility of the trie structure (being able to easily expand and contract) can accommodate changes in the sparsity pattern (although this is unlikely to happen in our application).
   - We do not have to store keys in leaves as there is a one-to-one mapping between the index and the leaf position in the trie. For the same reason, testing for equality of two arrays can be partially done by a structure comparison of the corresponding tries (if there is a mismatch in the structure no further tests are required). This reduces item searching activities.
   - Items are guaranteed to be kept in order.

3. **Patricia tree[6].**
   The above radix trie approach has a problem in that the height of a trie has to be fixed even if only one leaf is present. Morrison's Patricia tree does not present such a problem. *Every* node in a Patricia tree stores an item. However, for each node, two extra fields for key and bit index are necessary.

The latter Patricia tree approach seems to be overly complex for our current requirements and we have decided not to use this method. A suitable option seems to be the combination of the first two ideas: using the first for small vectors and the second for larger ones. In Haskell, as in other modern languages, it is possible to devise functions which automatically select from and switch between two structures and hide the differences from the user. However, this increases the complexity of the software and the benefit gained may not be very significant. The list approach would only be used for small vectors, and in these cases the trie approach would not be an unacceptable choice so we have decided to implement only the trie approach.

3 Implementation

3.1 Definitions of data types

We have chosen the radix trie as the underlying data structure as it is relatively easy to implement. The data type is defined as

```
data Bin_Trie a = Null | Leaf a | Bin_Trie a :^: Bin_Trie a
```

This differs from the conventional definition in that a `Null` trie is introduced. This is designed to represent a trie of arbitrary size whose leaves all have the default value. We could just use the conventional definition (omitting the `Null`). But this makes the difference between a default entry and a non-default entry less easy to detect and hence is likely to a) make trie pruning more difficult; and b) introduce extra work for equality testing of two tries (the default values need to be compared only once). Notice that there is no key field in the branch construction. This can reduce the space required at the cost of extra computation.

The array type can be further constructed as

```
data S_array a =
  Mk_t_Array (Int,Int) Int (Maybe a) (Bin_Trie a)
```

where `Maybe a` is defined as

```
data Maybe a = Nothing | Just a
               deriving (Eq)
```

and `deriving (Eq)` automatically generates a *derived instance declaration* for data-type `Maybe a` in class `Eq`.

The first argument of the constructor `Mk_t_Array` is designed for holding the array bounds as a pair. The second field holds the maximum number of leaves of the trie and is purely for convenience of computation. The `Maybe` type is for the default value [7]. When the default value is defined, a `Just v` is placed there, otherwise a `Nothing` is used. This makes the checking of a read access to an undefined array entry straightforward (its implementation is given later). The last field is the place where the data is stored.

3.2 Module interface

For every function in the `PreludeArray` we have implemented a counterpart (names of functions are prefixed with `sparse_`). The signatures of function calls are kept as close to the originals as possible [8]. For example, the signature for function `array` is

```
array :: (Ix i) => (i,i) -> [Assoc i a] -> Array i a
```

where `Ix` (class) is defined as the context of the type variable `i`, and its counterpart's signature is

```
sparse_array :: (Int,Int) -> [Assoc Int a] -> S_array a
```

For some functions, the elementary data type, `a`, should also be in class `Eq` (their standard counterparts do not require this). This is because these functions check if array values are equal to the default value. This should not present any serious difficulties in practice.

To facilitate the operations concerned with sparseness, a few extra functions have also been included and they are listed below.

1. `sparse_def_array` and `sparse_def_listArray`
   They have been designed to simulate `array` and `listArray` functions,

except that users are allowed to define a default value for any unspecified entries. (A call to function `sparse_array` or `sparse_listArray` leads to an undefined default value.) The signatures of these functions are

```
sparse_def_array ::
  (Int,Int) -> a -> [Assoc Int a] -> S_array a
```

and

```
sparse_def_listArray ::
  (Int,Int) -> a -> [a] -> S_array a
```

2. `new_sparse_assocs`
   The function `assocs` (or `sparse_assocs`) expects every entry in an array to be defined, otherwise an access violation is indicated. There are situations where the entries with a default value are unimportant (defined or undefined). To avoid the violation, a function, `new_sparse_assocs`, has thus been constructed. It generates a list of index and value associations with default entries removed.

3.3 Run-time control of storage space

To make better use of storage, a control mechanism has been implemented. It includes two aspects.

1. All leaves containing a default (or undefined) value are replaced by a `Null` node.
2. Whenever a node is found to have two `Null` sub-tries, it is replaced by a `Null`.

The second requirement can usually be implemented in the following manner.

```
fork :: Bin_Trie a -> Bin_Trie a -> Bin_Trie a
fork Null Null = Null
fork br1 br2 = br1 :^: br2
```

The function, gen_trie, uses fork to construct a trie.

```
gen_trie :: Int -> [Assoc Int (Bin_Trie a)] -> Bin_Trie a
gen_trie _ [] = Null
gen_trie 1 [_:=v] = v
gen_trie 1 _ = err_multi  -- error message
gen_trie s as =
  fork (gen_trie s' as1) (gen_trie s' (map_as s' as2))
  where
  s' = s `div` 2
  (as1,as2) = partition (\(i:=_)->(i<s')) as
```

3.4 Retrieving a value

Because of the existence of Null nodes, the retrieval of values is slightly complicated. This is achieved by calling function get_v_from_trie.

```
get_v_from_trie :: (Bin_Trie a) -> (Maybe a) -> a
get_v_from_trie (Leaf v) _ = v
get_v_from_trie _ (Just v) = v
get_v_from_trie _ _ = err_undefed -- error message
```

When a Leaf or Null node is identified, it is passed to get_v_from_trie with a default value. It first checks if the node is a Leaf by pattern matching. If a Leaf is found, its value is returned otherwise the default value should be returned. If there is no definition for the default value, an error flag is raised.

3.5 Equality

Equality has been defined on the S_array and Maybe. For Maybe, (==) is automatically available as we have defined it to be a derived instance of Eq.

Nevertheless, the equality of two arrays cannot be specified in this way. According to the definition of equality for the standard array, two arrays are equal if and only if their associations are equal. We could therefore just use this definition. However this could result in repeated testing of the equality of default values. If the default values are complex (an array for example), much time could be wasted. Another point is, the standard definition tolerates differences between default values as long as they are never referenced. To achieve this, we need to define our own comparison of two tries rather than rely on an automatically derived equality function. The following is an implementation of array comparison. When two `Null` nodes which represent data entries within array bounds are encountered, their equality is decided by a comparison of two default values. This comparison is performed at most once and its result is shared.

```
instance (Eq a) => Eq (S_array a) where
  (Mk_t_Array b1 sz d1 b_trie1) == (Mk_t_Array b2 _ d2 b_trie2) =
    b1 == b2 && eq_trie leaf_no sz b_trie1 b_trie2
    where
    leaf_no = b_size b1
    eq_def = d1 == d2
    eq_trie 0 _ _ _ = True
    eq_trie l s (lb1:^:rb1) (lb2:^:rb2) =
      eq_trie (min s' l) s' lb1 lb2 &&
      eq_trie (max 0 (l-s')) s' rb1 rb2
      where
      s' = s `div` 2
    eq_trie _ _ Null Null = eq_def
    eq_trie _ _ (Leaf x) (Leaf y) = x == y
    eq_trie _ _ _ _ = False
```

4 Forcing evaluation

Runciman [4] suggests a partial forcing evaluation strategy for preventing space leaks. It involves introducing a class `Normal` over which a function `normal` is defined.

```
class Normal a where normal :: a -> Bool
```

This function matches the pattern of its parameter and always yields the value `True`. However, to obtain this truth value the argument to `normal` must be (at least partially) evaluated.

For example, the definition of `normal` on the data type `Bin_Trie a` can be

```
instance Normal (Bin_Trie a) where
  normal (_ :^: _) = True
  normal (Leaf _) = True
  normal Null = True
```

An expression evaluation can be forced in the following way.

```
exp = f x y

v | normal exp = g1 exp
```

where `normal exp` is a guard.

Under such a scheme, any construction of a trie should be performed by calling a function `fork`.

```
fork :: Bin_Trie a -> Bin_Trie a -> Bin_Trie a
fork br1 br2 | normal br1 && normal br2 = br1 :^: br2
```

This approach aims at an eager construction of a data structure and space leaks can be prevented from the outset. Notice that the function only performs pattern matching at the first level. Such a `normal` is suitable for branch update. When one branch of a fork is updated and hence the fork has to be reconstructed, the amount of work to check the other branch is kept minimum. However, this approach may not be quite appropriate for a Haskell array because the laziness in array evaluation will be lost. The following recurrence example given in the Haskell report (p. 62, [8]) is no longer possible because it requires the access itself before it is constructed:

```
a = array (1,100) ((1 := 1) : [i := i * a!(i-1) | i <- [2..100]])
```

To keep the array laziness, a modification of the above approach, which ask the function `normal` to match not only the first data structure level but also the lower levels, can be used. For example, the above definition can be declared as

```
instance Normal (Bin_Trie a) where
  normal (br1 :^: br2) = normal br1 && normal br2
  normal (Leaf v) = normal v
  normal Null = True
```

In this approach, the construction of a data structure need not be eager, *i.e.*,

```
fork br1 br2 = br1 :^: br2
```

can be used. The forced evaluation is done when a value is used.

Unfortunately the same expression could be forced several times, if the following code is used.

```
exp = f x y

v1 | normal exp = g1 exp

v2 | normal exp = g2 exp
```

The way in which the function `normal` is defined implies that the whole structure of the expression will be traversed twice, resulting in unnecessary computation.

One possible way round the problem is to pack an expression with a flag indicating whether it has been forced. This flag is tested before any forced evaluation is performed. An implementation of this approach is given below.

```
-- data type of a package
data (Normal a) => Norm_able a = Norm_pack Bool a deriving ()

-- pack an object with a flag (the result of the normal function)
pack_obj obj = Norm_pack (normal obj) obj

-- get the value of a flag (forced evaluation is automatic
-- if the value is not yet set)
normalise_pack (Norm_pack b _) = b
```

```
-- retrieve an object from the package
retrieve_obj (Norm_pack _ obj) = obj
```

Hence the forced evaluation can be carried out in the following manner.

```
exp = pack_obj (f x y)

v1 | normalise_obj exp = g1 (retrieve_obj exp)

v2 | normalise_obj exp = g2 (retrieve_obj exp)
```

No extra forced evaluation results.

However, we still have to be careful about this strategy. It might be too expensive to force evaluate an array when only a few entry update operations have been performed on it. Indeed, this approach does not prevent space-leaks. It is merely a mechanism which allows space-leaks to be removed from time to time.

The `normal` function can also be used for determining the order of a numerical calculation in the same way as described above. All that is necessary is to define `normal` on numerical data types. For example, on the `Floating` data type, it can be defined as

```
instance Normal Floating where
  normal 0 = True
  normal _ = True
```

We have discussed two force evaluation strategies. Both have pros and cons. The choice has to be application dependent. We have experimentally implemented both.

5 Some Results

In order to evaluate the efficiency of our implementation a number of timing tests with various sized arrays were performed. These are summarised in the following table. The table includes four sets of results where each represents the time spent on executing the algorithm under the stated constraints:

1. construction (only) of new arrays, eager, *i.e.*, force-evaluated using Runciman's strategy [4];
2. construction plus one access of all entries of new arrays, eager;
3. construction plus one access of all entries of new arrays, lazy, *i.e.*, no forced evaluation for the tree construction;
4. construction plus one access of all entries of standard arrays (lazy);

The standard array is implemented as a list, therefore the expected complexity of accessing all elements is $O(n^2)$.

size	set 1	set 2	set 3	set 4
512	1.5	2.3	2.8	22.5
1024	3.1	5.1	5.2	88.2
2048	6.7	11.0	11.1	348.9
4096	13.9	23.4	24.3	1389.8
8192	29.6	51.0	56.7	-
16384	69.2	130.8	167.1	-
32767	264.2	>785*	>524.6*	-

$*$: out of heap space after that time

These results indicate that the new implementation of arrays agree with the theory that the construction and access of all entries have a complexity of $O(n \log(n))$. The new eager array implementation runs slightly faster and appears to be less space demanding than the lazy form. Since it takes longer to run out of heap space we assume that rather more of the array was constructed before it failed. Whether the increase in speed and decrease in space requirement is worth the loss of expressive power compared to the lazy array is probably an application dependent decision. In general though we feel justified in claiming that our new implementation of the array prelude is a significant improvement on the current list-based implementation. It should be noted that the above results have been obtained with non-sparse arrays and that if sparse arrays were used then we would expect the improvements in performance to be even more marked.

Acknowledgements

Special thanks to Colin Runciman and Sandra Foubister at York for their advice on preventing space leaks and its implementation.

This research has been supported by grants from the UK Science and Engineering Research Council and the Department of Trade and Industry (SERC GR/F 99076 C2117).

References

[1] P.Townsend and M.F. Webster. An algorithm for the three-dimensional transient simulation of non-Newtonian fluid flows. In G.N. Pande and J. Middleton, editors, *Proc. Int. Conf. on Numerical Methods in Engineering: Theory and Applications*, volume II, Swansea, 1987. NUMETA, Nijhoff, Dordrecht.

[2] D.M. Hawken, H.R. Tamaddon-Jahromi, P. Townsend, and M.F. Webster. A Taylor-Galerkin-based algorithm for viscous incompressible flow. *Int Journal of Numerical Methods in Fluids*, 10:327–351, 1990.

[3] W. Stoye. The implementation of functional languages using custom hardware. *PhD thesis, University of Cambridge Computer Laboratory*, Dec. 1985.

[4] C. Runciman. TIP in Haskell – Another Exercise in Functional Programming, (FLARE/Y/3). Technical report, University of York, 1991.

[5] I.S. Duff, A.M. Erisman, and J.K. Reid and. *Direct Methods for Sparse Matrices*. Clarendon Press. Oxford, 1986.

[6] D.R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *ACM J.*, 15(4):514–534, 1968.

[7] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.

[8] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Technical report, University of Glasgow and Yale, Aug 1991.

Appendix: Some new test results

These results were obtained by timing inside Haskell (Calling C precedures). The following table corresponds to that in the last Section except the set 2. The set 2 here excludes array construction times. The test conditions were equavilent but not exactly the same. Larger heap spaces were used for these experiments if array sizes were ≤ 8192. Also, the new force evaluation function were more expensive.

size	set 1	set 2	set 3	set 4
512	1.5	0.94	2.1	15.3
1024	3.5	1.9	4.9	60.8
2048	7.1	4.4	10.1	239.7
4096	14.3	9.7	22.0	952.5
8192	30.8	21.4	49.5	3822.5
16384	63.8	42.9	108.0	-
32767	141.7	110.2	278.3	-