J. Functional Programming **8** (4): 323–333, July 1998. Printed in the United Kingdom

© 1998 Cambridge University Press

Efficient graph algorithms using lazy monolithic arrays

THOMAS JOHNSSON

Department of Computer Science, Chalmers University of Technology,
S-412 96 Göteborg, Sweden
(*e-mail:* johnsson@cs.chalmers.se)

1 Introduction

Many, perhaps even most, algorithms that involve data structures are traditionally expressed by incremental updates of the data structures. In functional languages, however, incremental updates are usually both clumsy and inefficient, especially when the data structure is an array.

In functional languages, we instead prefer to express such array algorithms using *monolithic arrays* – wholesale creation of the final answer – both for succinctness of expression, efficiency (only one array created) and (sometimes) implicit parallelism. The ease with which the solution can be reformulated of course depends on the problem, and varies from trivial (e.g. matrix multiplication), to challenging (e.g. solving linear equation systems using Gauss elimination, which in fact can be done by creating only two arrays, recursively defined, of which one is the answer). Other problems have been notoriously resistant to attack; these usually involve some unpredictable processing order of the elements. One such problem is graph marking, i.e. marking the nodes reachable from a set of roots. Hitherto, no functional method has been known except emulating the traditional imperative solution (King & Launchbury, 1995; Launchbury & Peyton Jones, 1995).

The contribution of this paper is to show how this problem, and some related ones, can be solved using a novel array creation primitive, lazier than previous ones. Thus, in section 2 we specify the array creation primitive `lazyArray`. In section 3 we give the solution to the graph marking and depth-first numbering problems using our lazy arrays. In section 4 we show how a related and more general problem, that of computing the transitive closure of a binary relation, can be solved in a similar manner. Graph based unification is an essential ingredient in a time and space efficient type inferencer: in section 5 we give such an algorithm in the same style. In section 6 we show how lazy arrays can be implemented efficiently with low-level graph reduction code. In section 7 we discuss space efficiency. Finally, in section 8 we relate our approach to state monadic computations, and show how to cast lazy array programs into monadic form. Programs in this paper will be given in the non-strict purely functional language Haskell.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

2 Lazy arrays

The array creation primitive `array` in Haskell takes a list of associations, i.e. index-value pairs (i,v). Each index may occur only once in the list, and must be inside the bounds specified in the `array` call. With `accumArray (⊕) z` each index may occur more than once, and each value is ⊕'ed into the corresponding index (with `z` as the initial value). Both `array` and `accumArray` evaluate the entire association list and all indices (but not the values) before returning an array. This is too strict for our purposes.

We will be using a new array creation primitive, which we shall call `lazyArray`. The semantics of `lazyArray` could be expressed in terms of ordinary Haskell `arrays` as follows (for the sake of simplicity we assume the one-dimensional case):

```
lazyArray (1,n) xs = array (1,n) [(i ,  [ v | (j,v) <- xs, i==j ])
                                 | i <- [1..n]
                                 ]
```

That is, for each index i of the resulting array a list of the elements with the appropriate index are filtered out (elements out of range are silently ignored). This is akin to `accumArray (:) [] bounds xs` in Haskell, except that `lazyArray` is lazier: the key point is that the array is created before the the list `xs` is demanded, and so the value of this list is allowed to depend on the contents of the array just being created.

The above definition of `lazyArray` has a bad time complexity. Since each element of the resulting array is an independent filtering of the association list `xs`, the complexity of `lazyArray` becomes $\mathcal{O}(n|\texttt{xs}|)$ if all array elements are eventually demanded in their entirety. However, it is possible to implement `lazyArray` in $\mathcal{O}(|\texttt{xs}|)$ using low-level graph reduction code – see section 6.

3 Graph marking and depth-first numbering

Many graph algorithms work by visiting nodes while also filling in information in the nodes. A very basic algorithm of this kind, for which no previous purely functional monolithic array solution has been found, is *depth-first numbering*: starting from a root or set of roots, visit nodes in depth first order, and if a node is previously unvisited assign an order number and visit its successor nodes. (The related algorithm for obtaining a depth-first spanning forest by King and Launchbury (1995) uses incrementally updateable arrays in a state monad (Wadler, 1992; Peyton Jones & Wadler, 1993; Launchbury & Peyton Jones, 1995).)

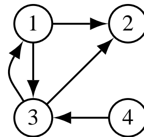
We now present an algorithm to do depth first numbering of a graph, which constructs a single monolithic lazy array. We represent nodes by indices into an array, and the graph itself is an array of lists of successor nodes. Consider the following function, `dfn`, which visits nodes in a graph.[1]

[1] In Haskell, `!` is the array indexing operator, and `bounds` of an array returns a pair with the lower and upper bound.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

```
dfn graph roots = a
   where
      a = lazyArray (bounds graph) (visit 1 roots)
      visit u [] = []
      visit u (i:l) = (i,u) : if head(a!i) == u
                              then visit (u+1) (graph!i ++ l)
                              else visit u l
```

By visiting a node i, we put a number u into the list of the ith element of the array. If this is the first time node i is visited, this number u will be the first element of the list in the i'th element and thus `head(a!i) == u` yields true. In that case the depth-first traversal is continued by first considering the successor nodes `graph!i`. So for example with the graph



taking 1 as the single root node, we get the following call to `dfn`:

```
dfn (array (1,4) [(1, [2,3]), (2, []), (3, [2,1]), (4, [3])]) [1]
```

which returns

```
array (1,4) [(1, [1,4]), (2, [2,4]), (3, [3]), (4, [])] .
```

The nodes reachable from a set of roots are those whose element lists are non-empty:

```
reachable graph roots = amap (not . null) (dfn graph roots)
```

where `amap` maps a function over an array. To get a proper depth-first numbering of all the nodes in the graph, (corresponding to the order in which nodes are visited in a depth-first traversal of the depth-first spanning forest), just consider the head elements of the result from `dfn` (we must take all nodes as potential roots.[2]):

```
depthFirstNumbering graph = amap head (dfn graph (indices graph))
```

from which the depth-first spanning forest can be obtained easily.

4 Transitive closure

In the previous section, each list in the array has the rôle of a mark bit or a 'sticky' value, in that only the head element of the list is of interest. In a more general use of this technique, one would make use of the entire list up to the most recent value put there.

A graph can be thought of as defining a binary relation; thus, graph marking is only a special case of a more general problem: computing the transitive closure. As an interesting generalisation of the program in the last section, we want to compute

[2] In Haskell the function `indices` returns a list of all valid indices of an array.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

the set of all node names (indices) that can reach a certain node (index) by following one or more arcs in the graph. We want the result to be an array of lists. Computing the closure proceeds as follows: The function `propagate` keeps a list of pairs (i, x), where x is to be propagated to i. If x propagates to i, it also propagates to the nodes `graph!`i. However, we need to be able to stop the potentially infinite process of just adding the same elements to a node over and over again, so we accompany the x's by unique numbers. The solution is given below.

```
closure graph = amap (nub . map snd) a
   where
      a = lazyArray (bounds graph)
                    (propagate 1 [(j,i)|i <- indices graph,
                                        j <- graph!i])
      propagate u [] = []
      propagate u ((i,x):l) =
            (i, (u,x)) :
            if head[ u' | (u',v')<-a!i, v'==x] == u
            then propagate (u+1) ( [ (j,x) | j <- graph!i] ++ l)
            else propagate u l
```

The array `a` is now an array of lists of pairs of a unique number and an actual value propagated there. The expression `head[ u' | (u',v')<-a!i, v'==x] == u` returns true if the element `x` just added to the list `a!i` has not been added and propagated previously, by comparing the unique numbers. The function `nub` removes duplicates in a list. Taking the same graph as in section 3 as an example of its use,

```
closure (array (1,4) [(1, [2,3]), (2, []), (3, [2,1]), (4, [3])])
```

returns

```
array (1,4) [(1, [1,3,4]), (2, [1,3,4]), (3, [1,3,4]), (4, [])]  .
```

The algorithm shown here can be characterised as a 'work-list' algorithm, in that the argument of `propagate` is a list of remaining work to do. For comparison, with iterative techniques which repeatedly construct arrays until the values stabilise, it is entirely possible that sets stay the same iteration after iteration, and thus sets will have been recomputed unnecessarily. With the work-list approach, only one array is created, and individual set elements are propagated to nodes in the graph only where necessary.

Very many other 'monotonic set equation' problems can be solved in a similar manner: data flow (e.g. live variable) analysis in code generation, computing LR set of items in LR parser generation, etc.

5 Graph-based unification

Polymorphic type inference in compilers for languages like ML, Haskell, etc. can be a very time-consuming process. Milner (1978) gives two algorithms, one in a functional style, and one in a more imperative style using one global substitution.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

The traditional efficient (and imperative!) solution, as described by Cardelli (1987), uses graph-based unification.

Type variables are represented by nodes, which when originally uninstantiated, has the value of e.g. `Tvar i`. When instantiated, it is updated with the corresponding type constructor, e.g. `Tcon c ts`, where `c` is the type constructor and `ts` is a list of pointers to its type subexpressions.

We now give an algorithm for graph based unification using lazy arrays. Nodes are represented by elements of a lazy array, which are now (lists of) pairs of a unique number and a type expression.

Functions `unifyA` and `unify` take lists of pairs of type expressions to unify. The key case is unifying a type variable with any other type expression; i.e.

```
unify u ((Tvar v1, t2) : eqs) .
```

The crux is that we don't know whether type variable `v1` has been instantiated previously in the unification process, and we can't just test it without risk of entering a black hole! So we first assume that the variable is uninstantiated, and 'emit' `(v1, (u, t2))` for the array `a`. We then look at the head element of `a!v1` to see if indeed we have just provided the first 'update' for `v1`, by looking at the unique number. The complete algorithm is given below.

```
data Texpr = Tvar Int | Tcon String [Texpr]
unifyA n equations = a
  where
    a = lazyArray (1,n) (unify 1 equations)
    unify u []  = []
    unify u ((Tcon c1 as1, Tcon c2 as2) : eqs) =
            if c1==c2 then unify u (zip as1 as2 ++ eqs)
                      else error"Type error"
    unify u ((Tvar v1, Tvar v2) : eqs) | v1==v2 = unify u eqs
    unify u ((Tvar v1, t2) : eqs) =
            (v1, (u, t2)) :
            case head(a!v1) of
            (u',t') -> if u==u' then unify (u+1) eqs
                                else unify u ((t',t2) : eqs)
    unify u ((t1, Tvar v2) : eqs) = unify u ((Tvar v2, t1) : eqs)
```

To take a concrete example of using this algorithm, consider the *twice* function

$$\lambda^1 f^2.\lambda^3 x^4.f^2\$^5(f^2\$^6x^4)$$

where superscripts denote type variables for (sub) expressions, and \$ denotes infix function application. So type variable 1 denotes the type of the entire lambda expression, 2 denotes the type for f, and 5 denotes the type of that function application. We obtain a call of `unifyA` with the following type equations:

```
unifyA 6 [(Tvar 2, Tcon"->"[Tvar 4,Tvar 6]),
          (Tvar 2, Tcon"->"[Tvar 6,Tvar 5]),
          (Tvar 3, Tcon"->"[Tvar 4,Tvar 5]),
          (Tvar 1, Tcon"->"[Tvar 2,Tvar 3])]
```

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

which yields the following answer:

```
array (1, 6) [
 (1, [(5, Tcon"->"[Tvar 2, Tvar 3])]),
 (2, [(1, Tcon"->"[Tvar 4, Tvar 6]),(2, Tcon"->"[Tvar 6, Tvar 5])]),
 (3, [(4, Tcon"->"[Tvar 4, Tvar 5])]),
 (4, [(2, Tvar 6)]),
 (5, []),
 (6, [(3, Tvar 5)])]
```

Note that the algorithm has (attempted to) instantiate variable 2 twice, and 5 not at all. The resulting types have one free type variable represented by variable 5.

6 A linear implementation of `lazyArray`

As pointed out in section 2, the straightforward formulation in Haskell of `lazyArray` has an unnecessary factor of n in the time complexity; it is possible, however, to implement it in $\mathcal{O}(|\texttt{xs}|)$, by letting the graph reduction 'processes' communicate a little behind the scenes. As a starting point, let us reformulate `lazyArray`:

```
lazyArray (1,n) xs = array (1,n) [ (i, fi i xs | i <- [1..n]) ]
    where fi i [] = []
          fi i ((j,x):ys) | i==j = x : fi i ys
          fi i ((j,x):ys) | i!=j = fi i ys
```

The expression `fi i xs` filters from `xs` those entries pertinent to index `i`. In the third case of `fi`, rather than throwing away `x`, `fi i` should add an entry under index j on behalf of `fi j` ! To be able to find its 'buddy' closures, it is useful to have an auxiliary array pointing to them. As a further simplification, it is useful to store the list in one place, namely in conjunction with the auxiliary array, and have the closures point to the auxiliary array rather than the association list. Figure 1 shows the resulting implementation of `fi`, written in C (declarations of node structs etc omitted). Variables `xs` etc correspond to the same variables in the functional version. `EVAL_fi` is assumed to be called from `EVAL` with the arguments of the closure, i.e. the index `i` and a pointer to the auxiliary array `auxp`, plus a pointer `fi_p` to the same closure. In the `NIL` case an alternative action is to set *all* `fi` closures to `NIL`, rather than just the one evaluated. Figure 2 shows the the graph constructed on the heap by `lazyArray`, and figure 3 shows the state of this subgraph after evaluating the fifth element of the array, which is a `fi` closure, with `[(3,`a`), (4,`b`), (3,`c`), (5,`d`), …]` as the association list argument.

7 A note on space efficiency

The depth first numbering algorithm in section 3 has the same time complexity as the imperative algorithm, but worse space complexity: `lazyArray` is a potential source of space leaks. The reason is that all further attempts to number a node when

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

```
EVAL_fi(int i, Pointer auxp, Pointer fi_p){
   Pointer xs = auxp->assoclist;
   while(1){ /*  exited with return. */
      EVAL(xs);
      switch(xs->tag){
      case CONS:{ Pointer jp, x; int j;
                  Pointer jx = xs->head; Pointer ys = xs->tail;
                  EVAL(jx); jp = jx->fst; x = jx->snd;
                  EVAL(jp); j = jp->theint;
                  if(j is within the bounds of the array){
                     Pointer buddyp = auxp->arr[j];
                     buddyp->tag = CONS; buddyp->head = x;
                     buddyp->tail = allocnode(TAG_fi, j, auxp);
                     if(i==j){ auxp->assoclist = ys; return; }
                  }
                }break;
      case NIL: { fi_p->tag = NIL; return; }
}  }  }
```

Fig. 1. Low-level implementation in C of the function `fi`.

it is already numbered result in additional elements in the lists of the lazy array.[3] This can be remedied by using a slightly more general version of `lazyArray`, which maps a function over the elements:

```
lazyMapArray f b xs = amap f (lazyArray b xs)
```

The application of `head` can then be moved from the test to the application of `lazyMapArray`:

```
depthFirstNumbering graph = a
   where a = lazyMapArray head (bounds graph) ( ... )
         visit u (i:l) = (i, u) : if a!i == u
                                  then  ... else ...
```

The space leak vanishes when all elements of `a` have been evaluated, or if the garbage collector succeeds in reducing the `head` applications (Wadler, 1987). The graph unification algorithm in section 5 has the same deficiency, which can be fixed in the same manner.

Another way of implementing `lazyMapArray` is to build it directly, in the way described in section 6, by also building the application of the mapped function while building the rest. This is the way we implemented it in Lazy ML (where it is called `array` and is the *only* array constructor). An even more efficient implementation for the important case `lazyMapArray head` is possible, which avoids using an auxiliary array. Again taking a Haskell formulation as a starting point:

```
lazyArrayHead (1,n) xs = array (1,n) [ (i, fi i xs) | i <- [1..n] ]
                         where fi i ((j,x):ys) | i==j = x
                               fi i ((j,x):ys) | i!=j = fi i ys
```

[3] I'm grateful to John Launchbury for this observation.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

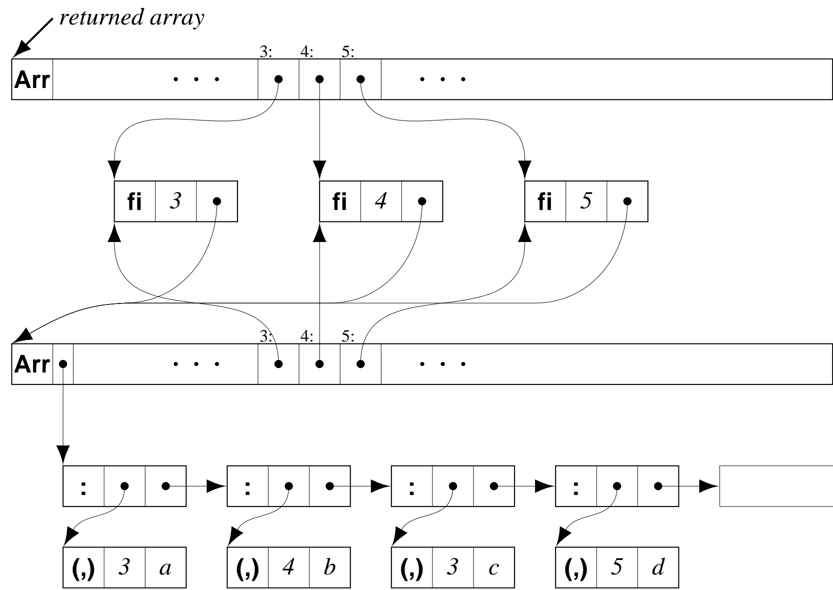


Fig. 2. Graph constructed by `lazyArray` *bnds* `[(3,`*a*`), (4,`*b*`), (3,`*c*`), (5,`*d*`), …]`.

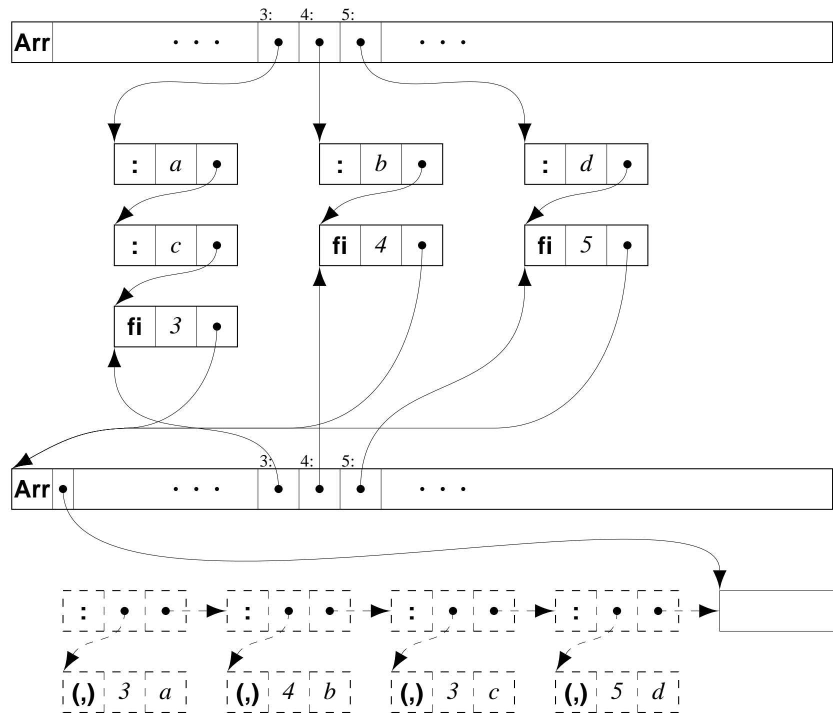


Fig. 3. Graph reduction state after evaluating the fifth element of the array.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

In the second case, `fi` would find its buddy closure via the single (returned) array, and should update it if it exists, i.e. has not been reduced previously.

8 Abstracting it with monads

Coding algorithms in the manner shown in this paper, with a recursively defined array and with careful threading of its association list, is quite tricky and it is all too easy to make a mistake which results in a ‘black hole’ error. So in order to make it safe, it is desirable to code up the essentials of the technique into higher order functions. We will show how to do that in the form of a state monad, and we will follow the recipe from Wadler (1992).

We first define the type of lazy array monads `LA`, with type parameters `i` which is the type of indices, `e` which is the type of the elements, and `a` which is the type of the value returned. The association list for the array plays the rôle of the state.

```
type LA i e a = [(i, (Int,e))] -> Int -> Array i [(Int,e)]
              -> (a, [(i, (Int,e))], Int)
```

The obligatory unit and bind operations in the monad could be defined in the manner shown below. The association list `s` is threaded backwards, while the count `u` is threaded forwards, and the array `a` is passed in but never changes. (These correspond respectively to backward state, forward state, and read-only state in (Wadler, 1995).)

```
unitLA :: a -> LA i e a
unitLA x = \s u a -> (x, s, u)

bindLA :: LA i e a -> (a -> LA i e b) -> LA i e b
f ‘bindLA‘ g = \s u a -> let (x1, s1, u1) = f s2 u a
                             (x2, s2, u2) = g x1 s u1 a
                         in  (x2, s1, u2)
```

Design of a comprehensive lazy array monad library that covers all possible situations is beyond the scope of this paper. However, for the problems discussed in this paper we can distinguish between two main types of operations: setting the value at an index, and adding an element to a set at an index.

In sections 3 and 5 the values of the arrays are ‘sticky’ single elements. The following operation, `setLA`, should cover most such situations.

```
setLA :: (Ix i, Eq e) => i -> e -> LA i e (Bool,e)
setLA i x = \s u a -> let (u’,x’) = head(a!i)
                      in ((u’==u, x’), (i,(u,x)) : s, u+1)
```

It takes an index and a value, and returns a pair: a boolean, true if this was the first time a value was set for this index, and the actual first value, i.e. the one that ‘stuck’.

In section 4, in the `closure` function, the array elements are regarded as sets. For this particular problem, we need an operation in the monad for adding an element `x` to the set at index `i`. As a value `addLA` returns a boolean value, true if this is the first time a particular value is being added to the list at index `i`.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

```
addLA :: (Ix i, Eq e) => i -> e -> LA i e Bool
addLA i x = \s u a -> (head[w|(w,y)<-a!i,y==x]==u, (i,(u,x)):s, u+1)
```

To execute an LA monad value, we provide the `runLA` function:

```
runLA :: (Ix i) => (i,i) -> LA i e a -> (Array i [e], a)
runLA b f = (amap (map snd) a, x)
            where a = lazyArray b s'
                  (x, s', u') = f [] 0 a
```

Given these primitives, the `closure` program from section 4 can be reformulated as follows.

```
closure graph = amap nub (fst(runLA (bounds graph)
                                    (propagate [(j,i)
                                               |i <- indices graph,
                                                j <- graph!i ])))
propagate [] = unitLA()
propagate ((i,x):l) = addLA i x `bindLA` \firsttime ->
                      if firsttime
                      then propagate ([ (j,x) | j <- graph!i] ++ l)
                      else propagate l
```

This code looks strikingly similar to code for doing it in the monad of mutable states (King & Launchbury, 1995; Launchbury & Peyton Jones, 1995)!

9 Concluding remarks

The purpose of this paper has been to show the use and implementation of a form of monolithic arrays, lazier than previously in, e.g. Haskell, and to show some important example algorithms using them. No doubt, very many other algorithms in the same style exist.

The implementation mechanism described in section 6 is quite similar to that of Sparud (1993), but his was developed independently and for a different reason: plugging space leaks caused by lazy pattern matching in `let` and `where` expressions. In his case the structure is a tuple (rather than an array), and selection is done with `fst`, `snd` etc (rather than indexing). In both cases, reduction of other closures than the demanded one is performed 'free of charge', on the 'buddies'; in his case the buddies are the closures `fst e`, `snd e`, etc. With the suggested implementation of `lazyMapArrayHead` the similarity is striking.

The behaviour of `lazyArray` and its suggested cousins is quite reminiscent of I-structures in Id (Arvind *et al.*, 1989), in that values arrive at the array in a 'data flow fashion', in an order different from that prescribed by lazy evaluation.

Fully general updateable arrays can be emulated using `lazyArray`, at a cost. The elements of the lists would then be (apart from the unique numbers) `Write x` and `Read`. Indexing is done by first putting a value `Read` in the list of the index selected, and then returning the last `x` in `Write x` before this `Read`. Of course, indexing gets

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press

more and more expensive for each new read and write of the same index. But for applications with a small number of reads and writes this is definitely a viable alternative. Also, if we only ever wanted the most recent value, an implementation could throw away the old values of the list and only keep the most recent one.

Although state monads with mutable arrays (Launchbury & Peyton Jones, 1995) provide a more general language construct (many mutable arrays, arbitrary updating) and thus any imperative algorithm can be coded in it, it is interesting that with the 'dressing up' of lazy arrays exemplified in section 8 we have arrived at basically the the same language construct for updateable arrays section 6, but from a completely different direction!

Acknowledgements

This work benefitted greatly from discussions with Shail Aditya, John Launchbury, and many people in 'multi group', especially Lennart Augustsson and John Hughes. This presentation also benefitted greatly from many constructive comments from Phil Wadler. The lazy array primitives were implemented and added to Lazy ML while the author was visiting Glasgow on an SERC visiting fellowship 1989-90.

References

Arvind, Nikhil, R. S. and Pingali, K. K. (1989) I-structures: Data structures for parallel computing. *ACM Trans. Programming Languages and Systems*, **11**(4), 598–632.

Cardelli, L. (1987) Basic polymorphic typechecking. *Science of Computer Programming*, **8**, 147–172.

King, D. and Launchbury, J. (1995) Structuring depth-first search algorithms in Haskell. *Proceedings of the 22nd Symposium on Principles of Programming Languages.*

Launchbury, J. and Peyton Jones, S. (1995) State in Haskell. *Lisp and Symbolic Computation*, **8**(4), 293–341.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Computer and Systems Sciences*, **17**, 348–375.

Peyton Jones, S. L. and Wadler, P. (1993) Imperative functional programming. *Proc. 1993 Symposium Principles of Programming Languages.*

Sparud, J. (1993) Fixing some space leaks without a garbage collector. *Proc. 6th Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, 117–122. ACM Press.

Wadler, P. (1987) Fixing some space leaks with a garbage collector. *Software–Practice and Experience.*

Wadler, P. (1992) The essence of functional programming. *Proc. 1992 Symposium on Principles of Programming Languages*, 1–14.

Wadler, P. (1995) How to declare an imperative. *Proc. International Logic Programming Symposium*, Portland, OR. MIT Press.

https://doi.org/10.1017/S0956796898003062 Published online by Cambridge University Press