Breaking the Complexity Barrier
of Pure Functional Programs
with Impure Data Structures

Pieter Wuille and Tom Schrijvers[*]

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName`@cs.kuleuven.be`

Abstract. Pure functional programming language offer many advantages over impure languages. Unfortunately, the absence of destructive update, imposes a complexity barrier. In imperative languages, there are algorithms and data structures with better complexity.
We present our project for combining existing program transformation techniques to transform inefficient pure data structures into impure ones with better complexity. As a consequence, the programmer is not exposed to the impurity and retains the advantages of purity.

1 Introduction

Pure functional programming offers many advantages over normal-style imperative programming. It leads to very elegant and straightforward code. Referential transparency, the absence of side effects, greatly simplifies the analysis and optimization process of pure functional programs.

However, the lack of side effects, and destructive update in particular, imposes a barrier on the best achievable time and space complexity. Arrays, the most efficient data structure known for representing a modifiable mapping from integers to elements of a fixed size, can be implemented with constant-time modification and constant-time retrieval when destructive update is available.

Algebraic data types (ADTs) are the building block for data structures in pure functional languages. Though many ADT-based purely functional implementations of arrays exist, they all require at least a logarithmic complexity for access or modification, and often logarithmic memory usage per modification.

In our project we aim to achieve the efficiency of imperative code in functional programs, by transforming the data structures used to other, possibly impure ones. This paper consists of a preliminary study of our work in progress. Our contributions are as follows:

- A systematic approach using a sequence of transformations to improve the efficiency of code that uses purely functional data structures. (Section 3),
- A number of case studies in which this approach is applied to real examples (Section 4), and
- Experiments that show their effect on performance (Section 5).

2 Preliminaries

We introduce a simple first-order functional language to demonstrate code examples. Our *target* language is an impure extension of the pure *source* language. The syntax is based on Haskell [8], but the operational semantics is strict in order to accommodate the target language's impurity.

2.1 The Pure Source Language

A program consists of a number of first-order function definitions and data type definitions. A first-order function definition is of the form:

$$f \; :: \; \tau_1 \; * \; \tau_2 \; * \; \ldots \; * \; \tau_m \; \texttt{->} \; \tau$$
$$f \; v_1 \; v_2 \; \ldots \; v_m \; \texttt{=} \; e$$

where f is the function name, the v_i are the formal parameters and e is the body, an expression. The first line is the function signature, which contains the parameter types τ_i and the return type τ.

The table below lists the possible expressions:

v	variable
$C\,e_1 \ldots e_n$	constructor application
$f\,e_1 \ldots e_m$	function application
`if` e_c `then` e_t `else` e_f	if-then-else
`case` e_p `of` $\overline{p_i \rightarrow e_i}$	pattern match

where a pattern p is of the form $C\,v_1 \ldots v_n$. Note that all constructor and function applications must be saturated: it is a first-order language!

We support the usual primitive data types like `Int`, `Bool` and `String`. Other, polymorphic algebraic, data types can be defined with the familiar Haskell syntax.

Some further syntactic conveniences from Haskell are also used, such as the list notation with `[]` and `:`, record field names, and the traditional infix operators like `+`.

2.2 The Impure Target Language Extension

In order to improve the complexity of the source language, we generate impure code in the target language. Hence, we extend our first-order language with two effectful features, not allowed in source language programs.

- We allow a field of an algebraic type to be declared mutable, and provide a *fieldname*_set :: *adt* * *fieldtype* -> () function. For example, a mutable integer could be declared as `data MutInt = {mutable val :: Int}`, and modified using `val_set`.
- Our target language provides dynamically growing arrays that can be destructively overwritten. The interface of the array type `Array a` is:

`empty :: Array a`	create an empty array
`set :: Array a * Int * a -> ()`	update a position
`get :: Array a * Int -> a`	get element at given position

Note that a function only run for its side-effect has `()` as return type.

Impure features must be used inside a `do ... end` block. This block is to be understood as equivalent to Haskell's `unsafePerformIO` operation which is passed an IO action. The last expression before the `end` is the return value of the construct, as if it was preceded by a `return` in the equivalent Haskell code.

Using impure operations is solely at the discretion of the programmer, who must take responsibility that it causes no observable side effects.

3   Four-Step Approach to Improved Complexity

In this section, we explain our general approach for improving data structures and changing program complexity. The approach consists of four steps:

1. *Abstracting interfaces*: hide the concrete pure data structure representation behind an interface with abstract operations.
2. *Interface implementations*: choose a better (possibly impure) data structure representation from a library of implementations to implement the interface.
3. *Extending the abstraction*: extend the interface with additional abstract operations (optional).
4. *Specialization*: remove the the abstraction layer, inline the chosen implementation and optimizing further (optional).

We illustrate each of the above transformation steps in the following, with a migration from a list to an array representation.

3.1   Step 1: Abstracting Interfaces

The first step is to abstract data structure interfaces. An interface for a data structure consists of an abstract type with all required operations for the data structure, but independent of its internal representation.

For instance, a program that operates on a list data structure typically uses a concrete representation of those lists directly, using pattern matches and constructor applications. This however makes it very hard to change the representation of the list without breaking existing code.

The abstract AList type Lists are traditionally implemented using the algebraic data type:

```
data List a = Nil | Cons a (List a)
```

Code operating on such a List, uses constructors and pattern matches. These are, however, completely inflexible. Whenever we want to change the list's representation, all constructors and pattern matches throughout the entire program must change too. To simplify this task we introduce an abstract type `AList a`, that does not suffer from this problem. It features the following operations:

`nil   :: AList a`	create an empty list
`cons  :: a * AList a -> AList a`	add an element to a list.
`isnil :: AList a -> Bool`	check whether the given list is empty.
`head  :: AList a -> a`	return the first element.
`tail  :: AList a -> AList a`	return the tail.

Transformation Existing code using the concrete List representation, is easily transformed to use the abstract interface instead, by the transformation $\alpha()$:

$$
\begin{aligned}
\alpha(\texttt{[]}) &= \texttt{nil} \\
\alpha(e_1\texttt{:}e_2) &= \texttt{cons } \alpha(e_1) \ \alpha(e_2) \\
\alpha \begin{pmatrix} \texttt{case } e \texttt{ of} \\ \texttt{[] -> } e_1 \\ v_1\texttt{:}v_2 \texttt{ -> } e_2 \end{pmatrix} &= \begin{aligned} &\texttt{if isnil } \alpha(e) \\ &\quad \texttt{then } \alpha(e_1) \\ &\quad \texttt{else } \alpha(e_2)[v_1 \mapsto \texttt{head } \alpha(e), v_2 \mapsto \texttt{tail } \alpha(e)] \end{aligned}
\end{aligned}
$$

Note that this transformation is very straightforward, because there is almost a one-to-one mapping between the operations on the representation and the operations on the abstract type. We require that all operations on the representation have a corresponding (sequence of) abstract operations. If this is not the case, the abstract interface must be extended first (see section 3.3).

Merely applying the transformation does not result in any performance gain. On the contrary, we are introducing an abstraction layer, which adds some overhead. Furthermore, the basic operations on concrete data structures (native operators, pattern matching, constructors) are usually constant-time with very small constant factors. If we want to achieve any improvement to complexity, we will need to do more.

3.2   Step 2: Interface Implementations

Once a (suitable) abstract form has been created, we need an implementation for it. Which implementation is best often depends on what operations are used on it, their frequency, and even what kinds and amounts of data are stored in it. The decision what implementation to choose is a very complex one, and is beyond the scope of this document. A naive but still useful way is trying different implementations from a library for each identified data structure and measuring the performance through a benchmark.

We will describe three implementations for the AList interface. The first one is the native one, mimicking the original List representation we started with, the second is one based on arrays with exactly the same semantics, and a third based on so-called *VLists*.

The List implementation for ALists To recreate the original implementation, we represent the AList as the native algebraic data type List, and implement all five primitive operations using constructors and pattern matches on them.

The Array implementation for ALists For some applications using a different, impure representation can be much faster. Arrays are the most popular impure representation of lists.

We define our array representation of an AList not as an impure array alone, but rather as the tuple of an Array and the list length. The Array stores the elements in order, and the length indicates the position of the last element in the array. By keeping track of the length, the actual array can be larger than the number of elements. This is useful for two reasons. Firstly, as long as there is space left, we can extend the list without copying the Array. Note that the Array stores the elements in reverse order: the last (oldest) element in the list is at index 0 in the Array.

Fig. 1. Using Arrays to represent ALists

Secondly, we can share the same Array between two ALists, one of which is a suffix of the other. This by itself however, would lead to problems when a given AList is extended in two different ways. Since the extensions would share the same Array to store their elements, they would be overwriting each other's data. The solution for this is keeping track of one more piece of information: the length of the longest AList using the Array. When modifying the Array through an AList whose length is less than the longest-used length of the Array, the Array is copied first (using the function `copy`). Hence, all operations but `cons` take always $\mathcal{O}(1)$ time, while `cons` takes either $\mathcal{O}(1)$ or $\mathcal{O}(n)$ time.

In summary, we end up with the following implementation:

```
data AListBack a = LB {array :: Array a, mutable size :: Int}
data AList a = L {back :: AListBack a, size :: Int}
nil = L {back = LB {array = empty, size = 0 }, size = 0 }
unshare l = if (size l) > (size (back l))
            then L { back =
              LB {array = copy (array (back l)), size = size l},
              size = size l }
            else l
```

```
modify l e = do
                Array.set (array (back l)) (size l) e
                size_set (back l) ((size l)+1)
                L {back = back l, size = (size l)+1}
            end
cons l e = modify (unshare l) e
head l = Array.get (array (back l)) ((size l)-1)
tail l = L {back = back l, size = (size l)-1}
isnil l = (size l)==0
```

The VList implementation for ALists When no `copy` operations are required for `cons`, Arrays are the best we can achieve. Otherwise, a more involved data structure is more appropriate, VLists [1], for which `cons` always takes $\mathcal{O}(1)$ time.

The idea of the VList is to present an AList as a linked list of *blocks* of exponentially increasing size. Each block consists of an Array (of pre-determined size) to store successive list elements, a mutable field to count the number of used positions in the Array, and a reference to the preceding block. A particular
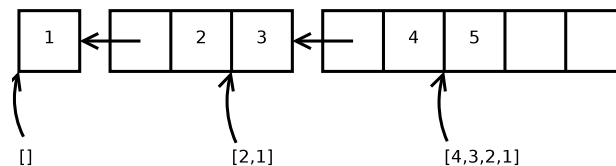
Fig. 2. Using VLists to represent ALists

list is represented, in the same way as with plain Arrays, by a tuple that refers to a given position in a given block.

When adding a new element to a VList, we check whether the next position in the last block is available and, if so, simply put it there. Otherwise, we create a new block with the original list reference as its predecessor and containing the new element. In the worst case, the VList degenerates to a linked-list.

3.3 Step 3: Extending the abstraction

One way of improving the efficiency is by extending the abstract interface itself. Assume we add an operation to our abstract AList interface:

| `nth :: int * AList a -> a` | return the nth element of the given list |

For expressivity purposes, it is not necessary to add this operation, as it can be implemented in terms of the abstract operations `isnil`, `head` and `tail`:

```
nth n l = if isnil l
   then error "index of out bounds"
   else if n=0 then head l else seek (n-1) (tail l)
```

The above can act as a default implementation, which works for any represen-
tation that provides `isnil`, `head` and `tail`, but it is not efficient. It calls `tail`
n times in a loop, and thus has at least linear time complexity.

For some implementations however, it is possible to provide a more efficient
version of the `nth` operation, exploiting properties of the concrete representation.

Arrays For example, in the earlier Array implementation for ALists, it is possible
to implement it as:

```
nth n l = Array.get (array (back l)) ((size l)-1-n)
```

Note that this implementation has $\mathcal{O}(1)$ rather than $\mathcal{O}(n)$ time complexity.

VLists For VLists, the `nth` operation takes constant time iff each block is suffi-
ciently full. Assume the optimal case, where n blocks have been filled completely.
Hence, the ith block contains 2^{i-1} elements, and $n-i+1$ blocks must be traversed
to reach them. This means at most an amortized[1] 2 steps per element:

$$\lim_{n \to \infty} \frac{\sum_{i=1}^{n} (n - i + 1)2^{i-1}}{\sum_{i=1}^{n} 2^{i-1}} = \lim_{n \to \infty} \frac{2^{n+1} - n - 2}{2^n - 1} = 2$$

Limitation While the abstract interface provides the `nth` operation, this is of
no consequence if the programmer does not use it. She may directly implement
such an operation herself rather than using the interface function. Consider the
programmer-written code:

```
seek n l = case l of
   [] -> error "index out of bounds"
   a:b -> if n==0 then a else seek (n-1) b
```

This would be abstracted to this loop of `head`s:

```
seek n l = if isnil l
   then error "index out of bounds"
   else if n==0 then head l else seek (n-1) (tail l)
```

instead of to this probably more efficient version:

```
seek n l = nth n l
```

---

[1] Yet, the first block takes n steps, i.e. logarithmic in the number of elements.

To overcome this problem in general, algorithm-recognition techniques need to be investigated. However, in cases where the result of an inefficient operation is not only semantically, but also representationally equivalent to the result of the optimized version, Section 3.4 presents an alternative.

3.4 Step 4: Specialization

Rather than manually adding additional operations to the abstract data structure interface and implementations, we can leave hem implemented on top of the original primitives, and rely on program transformation techniques to optimize them.

We illustrate this process on the seek function below, that uses the abstract AList interface described earlier:

```
seek n l = if isnil l
              then error "index out of bounds"
              else if n == 0
                      then head l
                      else seek (n-1) (tail l)
```

Implementation Inlining Once the implementation of the abstract interface is chosen, it can be be *inlined* [11] (unfolded) at the use sites.

This removes the overhead of the abstraction layer introduced earlier, and generating code that again directly operates on the data structure representation. If we had chosen the same representation before and after abstraction the interface (Step 1), then we would end up with the original code after inlining.

However, the idea is that we choose a different data structure than the one the original code happens was written for, say an Array:

```
seek n l = if size l == 0
              then error "index out of bounds"
              else if n == 0
                      then get (array (back l)) ((size l)-1)
                      else seek (n-1)
                              (L {back = back l, size = size l - 1})
```

Field Selector Inlining and Outward Floating of Case Expressions As many field selectors are used on the same variable l, we inline them as case expressions and float them outward (see [7]) so they can be shared:

```
seek n l = case l of
  L lb ls ->
   if ls == 0
     then error "index out of bounds"
     else if n == 0
             then get (array lb) (ls-1)
             else seek (n-1) (L {back = lb, size = ls - 1})
```

Constructor Specialization If we specialize the seek function with respect to the list argument [9], we get a helper function with two separate arguments (`ls` and `lb`):

```
seek n l = case l of
  L lb ls -> seek_L n lb ls
seek_L n lb ls =
  if ls == 0
    then error "index out of bounds"
    else if n == 0
      then get (array lb) (ls-1)
      else seek_L (n-1) lb (ls-1)
```

Strength Reduction After realizing the recursive call decreases the first and second arguments equally, until the second one is zero, we can use strength reduction (eg. [15]) to reach:

```
seek n l = case l of
  L lb ls -> seek_L n lb ls
seek_L n lb ls =
  if n >= ls
    then error "index out of bounds"
    else get (array lb) (ls-1-n)
```

Inlining Again After inlining the now non-recursive seek_L into seek, we end up with the now constant-time seek function:

```
seek n l = case l of
  L lb ls -> if n >= ls
    then error "index out of bounds"
    else get (array lb) (ls-1-n)
```

Note that strength-reduction is the crucial step in going from linear to constant time complexity. However, the other steps are essential for exposing the opportunity.

4   Case Study

The techniques described in the previous section have been applied (manually) to some examples, to show their effectiveness. We will now describe these in more detail.

The examples we consider are:

– Mergesort, a comparison sorting algorithm on lists
– Union-find, an algorithm for maintaining partitions in a set
– Perfect shuffle, an algorithm to randomly permute a list of elements

4.1 Mergesort

Mergesort is an elegant $\mathcal{O}(n \log n)$ comparison sorting algorithm. It can be implemented in a purely functional way, on Lists, or imperatively on destructably overwritable Arrays. Although constant factors differ, their complexity does not.

We consider the implementation in the Ocaml [12] standard library; it is a pure version of mergesort that acts as a stable list sorting algorithm.

Step 1: Abstraction Since this algorithm is implemented directly on Lists, it was abstracted to only use the earlier described AList interface with `nil`, `cons`, `isnil`, `head` and `tail`.

Step 2: Implementation Three different implementations are considered: the naive List implementation, the Array version, and a version based on VLists.

Step 3: Extending The algorithm uses an $\mathcal{O}(n)$ operation, called `chop`, which corresponds with the Haskell function `drop`:

```
drop k l =
   if k==0 then l
           else case l of
                  [] -> error "index out of bounds"
                  a:b -> drop (k-1) b
```

This drop operation is actually a repeated `tail`, as it can also be written as (after abstraction):

```
drop k l =
   if k==0 then l
           else if (isnil l) then error "index out of bounds"
                             else drop (k-1) (tail l)
```

If we add this operation to the AList interface used, a more efficient implementation can be provided eg. for the Array version:

```
drop k l =
   if k > (size l) then error "index out of bounds"
                   else L {back = back l, size = (size l)-k }
```

Step 4: Specialization The optimized implementation of `drop` for the Array version, could have been obtained through inlining the Array implementation code, and optimizing it. Since the transformation was done by hand, inlining was not done however, making specialization impossible.

4.2 Union Find

Union-find is the well-known algorithm for maintaining disjoint partitions in a given set. The underlying data structure is called a disjoint-set data structure. It provides only three operations:

- `makeset`: create a singleton in the disjoint-set structure containing a given element
- `find`: find the representative element of the given set. Checking whether two elements are in the same set is done by comparing their representatives.
- `union`: merge two sets (given by an element that is contained in them) into one. By doing this, all elements of both sets will get the same representative.

The algorithm given in [13] models the items in the set as nodes in a forest, where each node refers to one other node to which it is equal. By adding the optimizations *path compression* (where each node is made to point to the root upon traversal) and *union by rank* (the shallowest tree is made to point to the tallest one), it attains an amortized time complexity of $\mathcal{O}(n\alpha^{-1}(n))$, i.e. near-linear time.[2]

A variant of union-find, called persistent union-find [3], can be implemented in a purely functional way. Instead of storing the points-to relation inside the nodes, the nodes only contain an index into a shared *persistent array* that contains the indices of the node pointed to.

A persistent array is a data structure, comparable to the normal Array type, yet it creates a new version upon every modification instead of doing an in-place update.

Its interface is defined as:

`type PArray a`	the persistent array type
`init :: List a -> PArray a`	create an array with given contents
`get  :: PArray a * Int -> a`	get element at given position
`set  :: PArray a * Int -> PArray a`	set element at given position

Step 1: Abstraction The code given in [3] already uses an abstracted version of the persistent array, so adding an abstraction layer was not necessary.

Step 2: Implementation Again, multiple implementations of this data type are possible, with different performance characteristics.

A naive implementation with lists is possible, representing the array elements as list elements. This enables all earlier AList implementations to be used for the persistent array. However, a structure using an AList as backend would need to copy on average half of the elements on each modification.

A better alternative is using balanced trees to implement a map from indices to elements. Balanced trees have a $\mathcal{O}(\log n)$ look-up and modification time, which

---

[2] α^{-1} is the near-constant inverse Ackermann function.

would be preferable over ALists' $\mathcal{O}(1)$ look-up time (if Arrays or VLists are used) but $\mathcal{O}(n)$ modification time.

Even better performance can be obtained by using an observably persistent data structure proposed in [6], as shown in [3], which does use destructive update. The key idea is representing the latest version used of the persistent array as a real Array, and other versions as a tree of modifications each pointing to a predecessor. It is however possible to swap the Array (the top) and a modification to it (a child) in constant time, to make an older version the latest used.

Further optimizations are not necessary, so steps 3 and 4 can be skipped.

4.3 Perfect Shuffle

In general, a *perfect shuffle* algorithm is one that takes a list of n elements and a random number generator, and returns a random permutation of this list, with each permutation equally likely.

In the pure setting, we replace the random number generation with a list $[a_0, a_1, \ldots, a_{n-1}]$, where $0 \leq a_i < n - i$. Note that there are $n!$ possible a_i lists for a given n. Our purely functional algorithm then starts with the input list of elements to be permuted, and an empty output list. Take the a_0th element from the input list, and move it to the output list. Continue by taking the a_ith element from the input list and moving it to the output list. Repeat until the input list is empty.

The critical data structure here is a list from which an element with a given index can be removed as efficiently as possible. This is the interface for such a structure, which we will call an *erase list* or *EList*:

`type EList a`	erase list type
`create  :: List a -> EList a`	create a new one with given contents
`get     :: EList a * Int -> a`	get element at given position
`extract :: EList a * Int -> EList a`	remove element at given position

The order must be consistent between `get` and `extract`, passing a same index to those must retrieve/remove the same element. However, `extract` need not preserve the ordering of elements. This is a crucial property to achieve a good complexity.

Step 1: Abstraction Since we write this algorithm ourselves, we start with an abstracted version already.

Step 2: Implementation The most naive implementation is an ordinary List. Comparable to using Lists for the persistent arrays from the previous section, each modification would require on average half of the elements to be copied. The complete shuffling algorithm would end up having $\mathcal{O}(n^2)$ time complexity.

An alternative is using complete binary trees, with all nodes having a field listing the amount of subnodes they contain, as shown in [14]. Since in each

operation the path to the root needs to be duplicated, this results in a pure $\mathcal{O}(n \log n)$ shuffling algorithm.

If we drop the pureness restriction however, and reuse the persistent array from the previous section, we can achieve $\mathcal{O}(n)$. We represent an *EList* by the head of a persistent array. Thus an EList becomes a combination of a reference to a PArray and an integer (giving the length of the head that is used):

```
data EList a = EL {ar :: PArray a, len :: Int}
```

Creating a new EList is done by initializing a PArray with all elements, and creating an EList referring to the whole:

```
create l = EL {ar=PArray.init l, len=List.length l}
```

Retrieving an element is trivial, but when removing one, we decrease the length-value, yet copy the element that would be discarded by this over the element that is requested to be dropped:

```
extract l i = EL {
  ar=PArray.set (ar l) i (Parray.get (ar l) (len l)-1),
  len=(len l)-1}
```

Again, there is no need to optimize further.

5   Experimental Evaluation

We have implemented all three of the above case studies in Ocaml to experimentally evaluate our approach. All programs were compiled with Ocaml 3.09.2 to native executables. Their average CPU usage time was measured on an Intel Core2 Duo 6400 system for many problem sizes, and is shown in graphs.

Mergesort The mergesort algorithm was benchmarked for three different implementations of the abstract `AList` type: using concrete Lists, Ocaml arrays and VLists. The Array and VList variant are faster than the concrete Lists through a constant-time `drop` operation. The VList improves upon the Array by decreasing the amount of data that needs to be copied, as can be seen in Figure 3.

However, mergesort has the optimal time complexity for a comparison sort algorithm: $\mathcal{O}(n \log n)$, even with the $\mathcal{O}(n)$ `drop` operation. Therefore, optimizations that improve this operation can at best only affect a constant factor of the complete algorithm's run time. In the graphs can be seen that for large problem sizes Array-based approaches are indeed slightly more efficient.

Union-find For the union-find algorithm, the benchmark (see Figure 4) was done with the persistent array implemented using Lists, Ocaml trees, and Baker's data structure. Trees provide a large speedup over the naive List implementation, but Baker's structure wins, since only access to the latest version is required, for which that data structure provides constant-time operations.
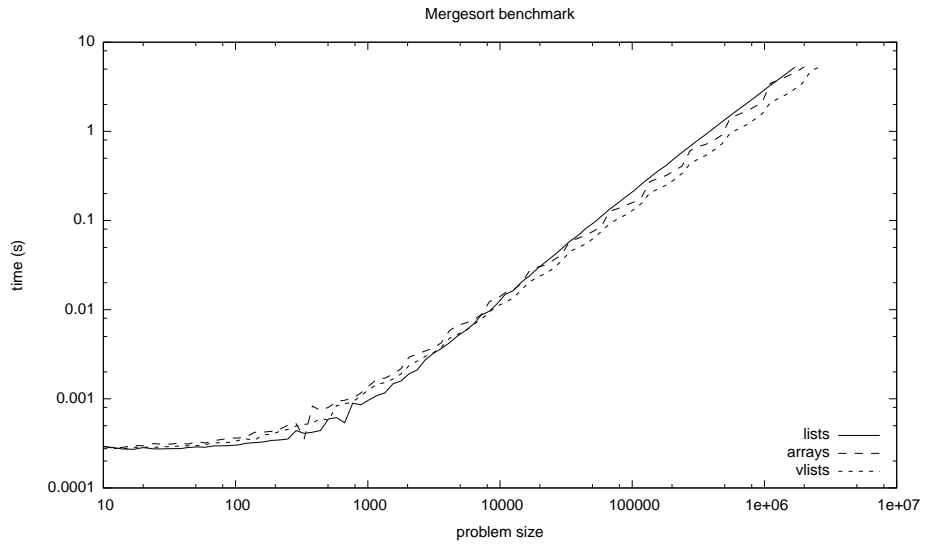
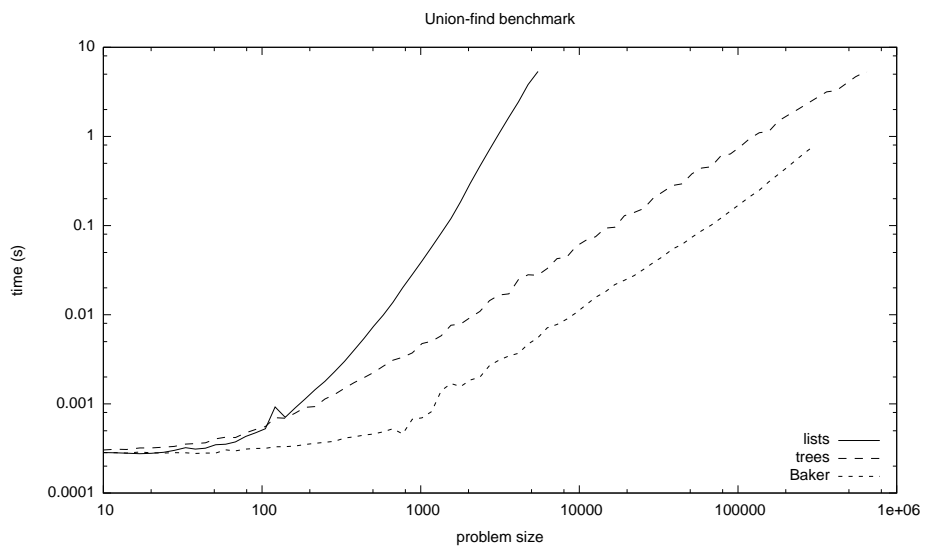**Fig. 3.** Mergesort benchmark

Fig. 4. Union-find benchmark

Shuffle The shuffle algorithm was benchmarked with five different combinations to implement the EList: one using Ocaml native Lists, one using a tree-based structure as described in [14], and an implementation that uses the earlier persistent arrays.

The result can be found in Figure 5. The implementations that use lists are clearly slower than the others. For large problem sizes the difference between the tree-based implementations and the one using Baker's structure becomes more distinguished. Also notice the overhead the double abstraction causes, as the persistent-array using Lists is slower than just Lists, and persistent-array using trees is slower than just trees.
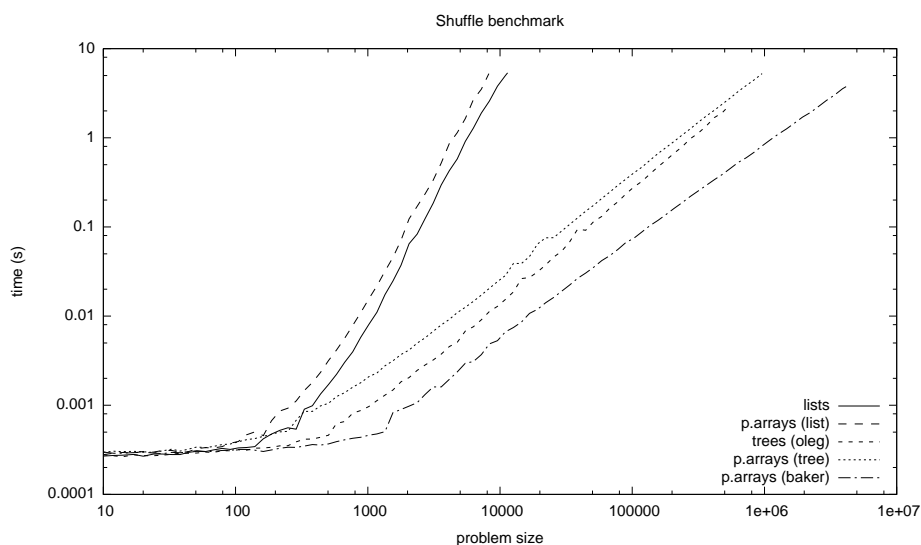

Fig. 5. Shuffle benchmark

6 Related work

Much effort has gone towards the development of efficient purely functional data structures, and program transformations for program optimization. However, the many efficient pure functional data structures developed, e.g. those in the Edison library [10], do not overcome the complexity barrier.

Also, of the many works conducted on program transformation and optimization, the majority, e.g. deforestation [4], focusses on the improvement of constant factors, and not complexity. Only a small minority can improve complexity, e.g. the worker-wrapper transformation [5] or tupling [2], but are still limited by the complexity barrier. The reason is that these transformations all restrict themselves to pure target languages.

7 Conclusions

In this paper we have described a systematic way to improve the efficiency of purely functional code, by transforming data structures and the code operating on it. The strategy has been applied to several real-world examples by hand. Our benchmarks show that changing the data structure used has a significant impact on performance.

Future work includes implementing and automating our approach, and studying its effect. In particular we would like to 1) implement a framework for executing the first-order language presented, by compiling it to an existing strict impure functional language, and 2) implement the transformations given in this paper, initially for a few selected data structures, but with sufficient generality. Further case studies and experiments should show the strategy's effectiveness.

References

1. P. Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. In *In Implementation of Functional Languages, 14th International Workshop*, page 34.
2. W.-N. Chin, S.-C. Khoo, and N. Jones. Redundant call elimination via tupling. *Fundam. Inf.*, 69(1-2):1–37, 2006.
3. S. Conchon and J.-C. Filliâtre. A persistent union-find data structure. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46, New York, NY, USA, 2007. ACM.
4. A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, The University of Glasgow, January 1996.
5. A. Gill and G. Hutton. The worker wrapper transformation. Submitted to the Journal of Functional Programming, 2008.
6. J. Henry G. Baker. Shallow binding in lisp 1.5. *Commun. ACM*, 21(7):565–569, 1978.
7. S. L. P. Jones and A. L. M. Santos. A transformation-based optimiser for haskell. In *Science of Computer Programming*, pages 32–1, 1998.
8. S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.
9. S. P. Jones. Call-pattern specialisation for haskell programs. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 327–337, New York, NY, USA, 2007. ACM.
10. C. Okasaki. An overview of edison. In *Electronic Notes in Theoretical Computer Science*, pages 34–54. Elsevier, 2000.
11. R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, 1977.
12. J. B. Smith. *Practical OCaml*. Apress, October 2006.
13. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
14. `oleg@pobox.com`. Provably perfect shuffle algorithms, 2001. `http://okmij.org/ftp/Haskell/perfect-shuffle.txt` [Online; accessed 7-August-2008].
15. M. Wolfe. Beyond induction variables. In *ACM*, pages 162–174, 1992.