The Implementation and Efficiency of Arrays in Clean 1.1

John H. G. van Groningen
Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
e-mail: johnvg@cs.kun.nl

Abstract

We present a new approach to implementing arrays in a pure lazy functional programming language. The arrays can be updated destructively by using uniqueness typing, and the elements can be unboxed. We describe the implementation of these arrays in the functional programming language Clean 1.1. The performance of two sorting algorithms and a fast fourier transformation written in Clean using arrays is compared with similar programs written in C. The current implementation of Clean is on average about 25 percent slower than C for these programs.

1 Introduction

Until recently, most implementations of pure lazy functional programming languages had limited or no support for arrays. The main reason for this lack was that an implementation could not update arrays in place, but had to copy the array first before an element could be changed. This copying is necessary because there may be other references to the array in the program, and changing the array will cause a side effect, which of course is not allowed in a pure language.

This update problem also made implementation of efficient input/output very difficult. For example, writing to a file or to the screen can only be done without a side effect if there are no more references in the program to the old file (the file before the write) or the old screen.

A way to solve such problems is by using uniqueness typing [BaSm93]. Uniqueness typing tries to determine at compile time which objects are unique using a combination of reference counting, type checking and program analysis. An object passed as an argument to a function is *unique* if there is only one reference to it when the function is applied. An array update function on a unique array can now be implemented efficiently. When the update function is called at runtime, there is only one reference to the array, which can therefore be updated in place.

The remainder of this paper is organised as follows. In section 2 we describe lazy, strict and unboxed arrays in Clean. Section 3 combines these 3 kinds of arrays using a type constructor class. Section 4 discusses arrays of unique elements. In the next section the performance of unboxed arrays is compared with arrays in C. We give a few concluding remarks in section 6.

2 Lazy, Strict and Unboxed Arrays

The functional programming language Clean [PlaEe95] uses uniqueness typing, among other things, to implement destructively updateable arrays. There are three kinds of arrays in Clean: lazy, strict and unboxed. Lazy arrays are the most general ones, but also the most inefficient ones. Strict arrays are more efficient because the array elements are always evaluated (in root normal form). Unboxed arrays occupy less memory and are even more efficient. Lazy, strict and unboxed arrays are considered to be of different types. The overloading mechanism is used to handle arrays in a uniform manner. This is discussed is section 3.

2.1 Lazy Arrays

To manipulate (lazy) arrays the following functions have been predefined in the standard library of Clean 1.1:

```
createArray :: !Int e          -> .{e}
update      :: !*{.e} !Int .e -> .{.e}
select      :: !.{.e} !Int     -> .e
uselect     :: !u:{e} !Int     -> (e, !u:{e})
size        :: !.{.e}          -> Int
usize       :: !u:{.e}         -> (!Int, !u:{.e})
```

An array with elements of type e is denoted as `{e}`. Special symbols appear in the specification of a Clean type. For instance, `!` is a strictness annotation. The other strange symbols have to do with uniqueness typing. A `'*'` before a type indicates that the type is unique. A `u:` before a type indicates that the type has a uniqueness attribute variable `u`. A type annotated with a uniqueness attribute variable can be unified with either a unique or a non-unique type. Usually, such a uniqueness attribute variable occurs several times in a function type. In an instance of a function type all types annotated with the same variable are either all instantiated with a unique type or all with a non-unique type. For example, the two `u:` before the array type in the `uselect` function indicate that if `uselect` is used on a unique array, the array returned in the tuple is also unique. Furthermore, if `uselect` is used on a non-unique array, the returned array will be non-unique as well.

Dots are used as an abbreviation to reduce the number of uniqueness attribute variables. A dot before a type variable `v` indicates that all type variables `v` annotated with a dot have the same anonymous uniqueness attribute variable. Prefixing a type, that is not a type variable, with a dot indicates that the type has an anonymous uniqueness attribute variable that does not occur anywhere else in the function type. A more detailed explanation of uniqueness attributes can be found in the Clean reference manual [PlaEe95].

`createArray n e` returns a one-dimensional unique array of `n` elements, all elements are initialised with `e`. The indices of the array are the integers from `0` to `n-1`.

`update a i e` returns an array which is identical to `a`, except that the element indexed by `i` of the array is updated with `e`. This function can only be used if `a` is a unique array.

`select a i` returns element `i` of array `a`.

`uselect a i` returns a tuple with the `i`th element of the array `a` and the array itself. This function is usually used to select elements from unique arrays, because we usually cannot use the `select` function to select an element from a unique array and then later use this array as a unique array as well. The compiler does not accept this because in such a case there would be more than one reference to the array. Because `uselect` also returns a 'new' array, which is identical to the array before the selection, we can select an element from a unique array and still use the (new) unique array.

`size a` returns the number of elements of array `a`.

`usize a` returns a tuple with the number of elements of array `a` and the array. Like `uselect`, this function is usually used to obtain the size of a unique array.

Clean 1.1 also includes the following syntactic sugar to manipulate arrays. Instead of `update a i n`, one may write `{a & [i]=n}`. Several updates at once are also possible, for example: `{a & [i]=n,[j]=m}`. `select a i` can also be written as: `a.[i]`. `uselect`'s can be done in patterns, for example:

```
swap a=:([i]=ai,[j]=aj}   =   {a & [i]=aj,[j]=ai}
```

is equivalent to:

```
swap __a = let! s=uselect _a j
           in   {a &[i]=aj,[j]=ai}
         where
           (ai,_a) = uselect __a i
           (aj,a)  = s
```

Selections in patterns are evaluated immediately to prevent space leaks (see section 2.2).

Arrays are stored in the heap. In a lazy context [PlaEe95] an array is represented by a node consisting of 2 machine words (a word consists of 32 bits in current implementations). The first word is an `ARRAY` descriptor, the second word is a pointer to another node in the heap. This other node consists of 3+n words, where `n` is the number of elements. The first word of these contains a descriptor and the second word the array size. The third word is 0 for lazy arrays. The remaining `n` words contain pointers to the elements of the array (nodes in the heap).

In a strict context the 2 word array indirection node is not used. In this case an array is represented only by the node of size 3+n. The compiler will create a new array indirection node if the array is later also used in a lazy context.

The advantage of this scheme is that we save an extra pointer indirection when we want to load or store information in the array. The disadvantage is that we sometimes have to create new array indirection nodes.

In the best case the code generated by the Clean compiler for a CISC processor for the `update`, `select` and `size` functions consists of only one machine instruction. This happens if the function is in a strict context, the compiler can determine that all arguments are evaluated, and the values of the arguments are already in registers. However, the compiler has to generate 3 instructions on most RISCs to do a selection or update instead of just one.

2.2 Inefficiency of Lazy Arrays

Although the array functions (except `createArray`) described above can be implemented in O(1), efficiency is still not as good as in (strict) imperative programming languages like C. This has the following reasons:

- More memory is required to store lazy arrays. For example, in an imperative language like C an array of n integers can be stored in n machine words. In Clean we need 3+3*n machine words. The elements of the array have to be pointers to nodes in the heap, because an element of an array can be unevaluated, and unevaluated expressions are stored in the heap as closures. So, even if an array contains evaluated integers, we still have to store them in the heap. To be able to recognise that an element is an evaluated integer we have to mark it. Therefore we store a descriptor just before the value of the integer. This descriptor is used by the garbage collector. So, for every evaluated element we have to store a pointer, a descriptor and a value. This means we have to store 3 words per element.
- Access to values in array elements is more expensive. For example, to select an integer, we first have to load a pointer, then load the descriptor, then examine the descriptor to find out if the integer has already been evaluated, and if it is evaluated, load the value. If the integer has not yet been evaluated, we have to call the evaluation code and later load the value. Only one load instruction is required to select an integer from an array in a language like C on a CISC processor.

We noticed that many programs using these lazy arrays use much more memory than expected. Even if all the extra costs of lazy arrays are taken into account. The most important reasons for this high memory use are:

- The `update` function does not evaluate the new array element. So, in most cases the `update` function will store a closure in the array. In many programs an array is updated many times before an element of the array is used. In such cases many

elements of the array become closures, which are usually larger in size than an evaluated array element.

- A lazy array selector contains a reference to the whole array, not just to one element. The compiler creates closures for array selectors in a lazy context. Such a closure contains a pointer to the array and the index. The garbage collector generally cannot determine which element is selected from the array, because the index could still be unevaluated. So, even if there is just one reference to a selection closure of an array, the memory referenced by the whole array cannot be deallocated. For example, if a function creates a new vector by adding two vectors, the memory used by these two vectors cannot be deallocated as long as there are references to the new vector and only one element of this new vector is unevaluated.

2.3 Strict Arrays

To be able to write more efficient programs, Clean 1.1 also has strict arrays. A strict array with elements of type e is denoted with `{!e}`. The implementation stores strict arrays in the same way as lazy arrays, and the same kind of functions can be used for strict arrays as for lazy arrays. Different from the previous approach is that the `update`, `createArray` and `uselect` functions evaluate the array element to root normal form. The types of the functions now become:

```
createArray :: !Int !e          -> .{!e}
update      :: !*{!.e} !Int !.e -> .{!.e}
select      :: !.{!.e} !Int     -> .e
uselect     :: !u:{!e} !Int     -> (!e, !u:{!e})
size        :: !.{!.e}          -> Int
usize       :: !u:{!.e}         -> (!Int, !u:{!.e})
```

As a consequence, the `select` function for strict arrays can be implemented more efficiently than for lazy arrays, because it does not have to evaluate the element selected from the array. Also strict arrays cannot contain closures, so the memory use problem caused by the laziness of the update function does not occur with strict arrays. This high memory use caused by array selectors in a lazy context is still a problem, but happens less often. For example the vector add function will no longer build selection closures, because the sum of the vector elements is computed before it is stored in the new vector.

Unfortunately, the amount of memory occupied by a strict array is still as bad as for a lazy array with evaluated elements. For example, we still have to represent an array of integers by an array of pointers to integer nodes in the heap. However, the problem is now only caused by the fact that the predefined array functions are of polymorphic type. This means that the `select` function can be used to select an element from any type of array it is applied to. Therefore, array elements have to be represented in a uniform way as well.

This polymorphism also has its advantages. For example, one only has to write one function, like `swap`, `reverse` or `copy`, and one can use it for all types of arrays. In an imperative languages like C a function that swaps two integers in an arrays cannot be used to swap two reals. A swap function that can be used for all arrays can be implemented in C, if we pass the size of an element of the array to the function, and use it to calculate the addresses of the elements. But such a function is slower, and more difficult to use.

2.4 Unboxed Arrays

We now choose the most efficient representation for every type of array. So, an array of n integers is stored in n+3 machine words. The elements of such arrays are unboxed and strict. We denote such an unboxed array type with `{#e}`, where e is the type of the elements.

The Clean compiler stores the unboxed arrays in a strict context as follows: (we assume the sizes of integers and pointers are 4 bytes, and n is the number of elements of the array)

- `{#Char}`: 8+n bytes, a string descriptor, the size and the characters.

The first 12 bytes of all other unboxed array nodes are a descriptor, the size, and a descriptor that describes the elements, for example an integer descriptor for an `{#Int}`.

- `{#Int}`, `{#Real}` and `{#Bool}`: 12+es*n bytes, where es is the size of an element: 4, 8 or 1 bytes.
- Arrays of records: 12+es*n bytes, where es is the sum of the sizes of the elements of the record.
- Arrays of arrays: 12+4n bytes, for this array only. 'Unboxed' arrays of arrays are represented as arrays of pointers to arrays, without the extra array indirection node for the elements.

At most 3 align bytes follow a `{#Char}` or a `{#Bool}` to align the next node in the heap at a word aligned address (multiple of 4).

The problems caused by the different representations for different types of arrays can be solved using Clean's overloading mechanism. For every basic type, for example `Int`, we define:

```
createArray_int :: !Int !Int              -> .{#Int}
update_int      :: !*{#v:Int} !Int !v:Int -> .{#v:Int}
select_int      :: !.{#v:Int} !Int        -> v:Int
uselect_int     :: !u:{#Int} !Int         -> (!Int, !u:{#Int})
size_int        :: !.{#.Int}              -> Int
usize_int       :: !u:{#v:Int}            -> (!Int, !u:{#v:Int})
```

Of course, we don't want to write `select_int` when writing a program using integer arrays, so we define a class for every array function. For example for `select`:

```
class select e :: ! {#.e} !Int -> .e
```

and we define instances for all element types, for example:

```
instance select Int
where
   select a i = select_int a I

instance select Real
where
   select a i = select_real a i
```

Now we can still write `select` and the type checker will try to determine which instance of `select` should be used. If a programmer defines a function that uses overloaded array functions, the compiler will automatically generate specialised versions of this function, if the overloading of the array functions cannot be resolved while typing the function. In this way the compiler can remove nearly all overloading within a module. So the program will usually be just as efficient as it would have been without using overloading.

When such an overloaded function is exported, by specifying the function type in a definition module, and is called by a function in another module, the compiler cannot generate specialised versions. If this happens, the function will be very inefficient. To prevent this, the programmer can tell the compiler to generate specialised versions by adding an `export` statement in the definition module.

Using Cleans unboxed arrays as described above in combination with strictness annotations, we expect for most programs to be able to generate code which is about as efficient as code generated by imperative languages like C. We will compare the efficiency of some small Clean programs written in this way with similar programs written in C, this is done in section 5.1.

3 Combining Lazy, Strict and Unboxed Arrays

We have now defined three kinds of arrays: lazy arrays, strict arrays and unboxed arrays. So far we have used the same names for functions that manipulate each kind of array. We prefer to do this in this way, instead of having 3 different names for every function. We can then write a program using for example lazy arrays, and then later decide to change it to unboxed arrays without having to rewrite the whole program. All we have to do is change a few types. If we use a type synonym we may even have to change only one type.

We can achieve this by using Cleans type constructor classes, which are similar to Gofers type constructor classes [Jone95].

We define a type constructor class `Array` with instances lazy, strict and unboxed array with:

```
instance Array { }, {!}, {#}
```

For this class instances are defined for the predefined array functions for lazy, strict and unboxed arrays. The instances for unboxed arrays are:

```
class Array a
where
   createArray :: !Int !e         -> .(a e)         | createArray_u e
   update      :: !*(a .e) !Int .e-> .(a .e)        | update_u e
   select      :: !.(a .e) !Int   -> .e             | select_u e
   uselect     :: !u:(a e) !Int   -> (!e, !u:(a e)) | uselect_u e
   size        :: !.(a.e)         -> Int            | size_u e
   usize       :: !u:(a .e)       ->(!Int,!u:(a .e))| usize_u e
```

The classes `createArray_u`, `update_u`, etc. are the same as the classes defined for unboxed arrays in section 2.4. So for example, class `select_u` is the same as class `select` in section 2.4.

So, the instance of `select` for unboxed arrays has type:

```
select :: !{# .e } !Int -> .e | select_u e
```

If the compiler cannot resolve the overloading in the type of the element, but can determine that it is an unboxed array, it will call this `select` with three parameters: the two normal parameters (the array and the index) and a `select_u` function added by the overloading mechanism. So, all the implementation of this function has to do is call the `select_u` function with the array and the index, and then return the result of this function application.

The implementation of the instances of the `select` function for lazy and strict arrays does not need the `select_u` function, because the implementation of these selects does not depend on the type of the elements of an array. So the instances for lazy and strict arrays for `select` are:

```
select :: !{ .e } !Int -> .e | select_u e
select a i = select_lazy_array a i

select :: !{! .e } !Int -> .e | select_u e
select a i = select_strict_array a i
```

where `select_lazy_array` is the `select` function for lazy arrays and `select_strict_array` is the `select` function for strict arrays.

We define the instances of the other predefined functions in the same way.

The instances of `createArray` and `update` are strict in the argument that passes the element for strict and unboxed arrays, but not for lazy arrays. The same goes for the element returned by `uselect`. The types defined in class `Array` for these instances are lazy in these arguments/results. To prevent loss of strictness information the compiler recognises these functions, and adds the strictness information for strict and unboxed arrays after typechecking.

We can now use the predefined array functions for all types of arrays, and the typechecker will determine the right instance. There are, however, also some disadvantages:

- When the compiler cannot determine whether an array is lazy, strict or unboxed or what the type of the array element is, and it cannot generate or use a specialised version, these array functions will be very inefficient.
- If the compiler cannot determine whether an array used by a function is lazy, strict or unboxed, the overloaded function type looks rather complicated.

4 Arrays of Unique Elements

With the array functions defined above we cannot create a unique array of which the elements are unique as well. It is also not possible to select a unique element from an array with the `uselect` function. Selection can be done with the `select` function, but this can be done only once: after the selection we no longer have a unique reference to the array.

The `createArray` function cannot create a unique array because it initialises all elements with the same value. If the array has more than 1 element, there will be more references to this initial value, therefore the elements cannot be unique.

So to create an array with unique elements, we have to compute a new value for every array element. We can do this by using an array comprehension [Wadl86,AnHu90] in Clean, which computes the value for every array element.

To compile such an array comprehension, the Clean compiler uses a function that creates an array of undefined elements, which may be unique, (`_createArray`). Since the compiler knows that all array elements will be updated immediately after the array has been created, it does not have to initialise the array, so this function is also faster. The current implementation does however initialise pointers in arrays with `Nil`, because the garbage collector cannot deal with such uninitialised structures.

Selecting a unique element from an array (without losing the array) is not possible, because after the selection we would have two references to the element: the element is returned by `uselect` and the returned array contains it as well.

The only way to make a selection of a unique element possible, seems to be by removing the element from the array. We can do this by replacing the selected array element by a new value with the function:

```
replace :: !*(a .e) !Int .e -> (.e, !*(a .e)) | Array a & replace_u e
```

which returns a tuple with the selected element and an array in which the selected element has been replaced by the third argument of `replace`.

With the `replace` and `update` functions we can compute a new unique element of an array that uses the old unique value in the computation in the following way. We use `replace` to select the element, then we compute the new value, and put it back in the array with `update`. This is of course slower than a modification using `select` and `update`, but an even more serious problem is that we temporarily need to create a unique element, which is inefficient or almost impossible for some types.

In a future version of the compiler we hope to make it easier to modify unique array elements in this way, may be by adding a function or special syntax.

4.1 Multidimensional Arrays

So far we have only looked at one-dimensional arrays. Multidimensional arrays are implemented in Clean as arrays of arrays.

Using arrays of arrays has certain advantages over implementing multi-dimensional arrays as a separate array type implemented by mapping two or more indices to one index in a one-dimensional array. For example, an array of strings (`{#Char}`) can be used as a one-dimensional array, but also as a two-dimensional array. A row of a matrix can be used as a vector without having to copy it, and the row may even be unevaluated. Without loop optimisations, updating an element of an array of arrays can usually be done faster than updating an element of a flat two-dimensional array, because the multiplication that is required for the index calculation is usually more expensive than loading the address of the row. There are also disadvantages: arrays of arrays occupy more memory. Multidimensional arrays can be accessed faster when certain loop optimisations are performed.

To be able to update such an array of arrays, all one-dimensional arrays have to be unique. For instance, to be able to update a lazy two-dimensional array of integers, we have to use an array of type `*{*{Int}}`.

To select an element from such an array, we would start by using `uselect` with the first index, which yields an array of type `*{{Int}}` and an array element of type `{Int}`. The elements of the returned array (of type `*{{Int}`) are no longer unique,

since the selection added a reference to an array element. Therefore we can no longer update this array after this selection.

To update a multidimensional array, we would start with a `uselect` with the first index. The elements of the returned array are no longer unique. The returned element is also not unique, so we cannot update it.

It is not possible to select or update a multidimensional array more than once with the `select`, `uselect` and `update` functions. So we have to invent a trick. Instead of just selecting an element, we select the element and also temporarily replace the selected element in the array by a dummy using the `replace` function. To select from a two-dimensional array of type `*{*{e}}` we need to create a temporary array, use `replace`, `uselect` and then use `update`. To update an element from an array of type `*{*{e}}` we need to create a temporary array, use `replace`, and then use `update` twice.

Clearly we don't want to write selections from and updates of multidimensional arrays in this complicated and inefficient way. We decided to extend the language with syntax for multidimensional selections and updates. The compiler transforms these selections and updates to several one-dimensional selections and updates.

The compiler uses the following functions to efficiently implement multidimensional selections: (they cannot be used by the programmer)

```
_uselect :: !u:(a .e) !Int -> (.e,!u:(a .e)) |Array a & _uselect_u e
_uselectn:: !(!.(a .e),!.m) !.Int ->(.e,!.m) |Array a & _uselectn_u e
_uselectl:: !(!.(a e ),!.m) !.Int -> (e,!.m) |Array a & _uselectl_u e
_update :: !(!*(a .e),!*m) !Int .e -> *m     |Array a & _update_u e
```

The `_uselect` function is similar to the `uselect` function, but returns a unique array element. Of course this element is not unique, but the compiler uses the function in such a way that this doesn't cause any problems.

The `_uselectl` function has two arguments. The first one is a tuple with an array and a value, this value is returned in a tuple by this function. The second argument is an index. The result of the function is the array element selected by the index and the value passed to the function as the second element of the tuple.

The `_uselectn` function is similar to the `_uselectl` function, but returns a unique array element. As for `_uselect`, this element is not unique, but the compiler uses the function in such a way that this doesn't cause any problems.

The first argument of the `_update` function is a tuple consisting of an array and a value that is returned by the function. The second argument is an index into the array. The last argument is the new array element. The function updates the element selected by the index with the new array element and returns the second element of the tuple.

A two dimensional selection `a.[i,j]` in a pattern is transformed by the compiler into:
```
_uselectl (_uselect a i) j
```

A two dimensional update `{a & [i,j]=n}` is transformed into:
```
_update (_uselect a i) j n
```

`a.[i,j,k]` is transformed into:
```
_uselectl (_uselectn (_uselect a i) j) k
```

`{a & [i,j,k]=n}` is transformed into:
```
_update (_uselectn (_uselect a i) j) k n
```

If there are more than 3 dimensions, for every extra dimension a `_uselectn` is added.

We can only compute the size of the first dimension of a multidimensional unique array with `usize`. So we also have to implement such a transformation to compute the size of the other dimensions.

The argument `m` passed to and returned by the `_uselectn`, `_uselectl` and `_update` functions is always an array. This array would normally be passed and returned with an extra array indirection node. The compiler recognises these functions and prevents this inefficiency.

5 Performance

5.1 Performance Results

To compare the performance of arrays in Clean 1.1 with C we wrote variants of the well-known quicksort, heapsort and fast fourier algorithms in Clean 1.1 and C. We used unboxed arrays in Clean and added strictness annotations to types of functions when necessary to make the functions strict in all arguments and tuple results. We then rewrote the programs in C using the same data structures. Tail recursive calls in C were manually optimised to while loops, because the C compilers did not perform this optimisation.

We used a PowerMacintosh 7100/80 with a 80 MHz PowerPC 601 cpu, 32 Mb of memory and 256 Kb secondary cache running system 7.5.3 without using virtual memory. We used the most recent version of the Clean 1.1 compiler. The compiler option to check whether array indices are in-range was turned off.

We compared the performance with two C compilers: Metrowerks CodeWarrior C 9, compiler version v1.5, and Apple's MrC version 1.0f4e1. We used maximum optimisation for both compilers. Global optimisation level 4, optimise for speed,

peephole optimisation, instruction scheduling for 601, use FMADD & FMSUB for CodeWarrior and -opt speed for MrC.
`quicksort` and `heapsort` repeated the following 10 times: create a list of 200,000 integers, sort the list and check if the list was correctly sorted. The source code of these sorting programs can be found in the appendix. The `fft` and `fftc` programs create an array of 65536 complex numbers and then do a fast fourier transform followed by a reverse fast fourier transform on its result.
These are the results:

	Clean	CwC	MrC	Clean/CwC	Clean/MrC
`quicksort`	6.45	5.63	7.62	1.15	0.85
`heapsort`	12.13	8.55	7.60	1.42	1.60
`fft`	5.51	4.67	4.47	1.18	1.23
`fftc`	2.38	2.20	1.60	1.08	1.49

The first 3 columns contain execution times of PowerPC code, in seconds, of the Clean program, the C program compiled with Codewarrior C, and the C program compiled with Mr C. The last two columns are the execution time of the Clean program divided by the execution time of the C program for each C compiler.

There are two versions for the fast fourier program (`fft` and fftc). We first wrote `fft`. It was much slower than we expected. This was caused by a larger number of cache misses. We rewrote the program to use a recursive fft in the beginning that divided the array in two parts for every pass, did the other passes of the fft for the first part, then for the second part and then merged the two parts. When the parts were small enough to fit in the primary cache, we used the same algorithm as before. This reduced the number of cache misses and it was more than twice as fast.

5.2 Possible Improvements

On the PowerPC double precision floating point values have to be stored at double word aligned (multiple of 8) addresses for maximum performance. A misaligned floating point load from the cache usually takes 2 clock cycles instead of 1 to execute on a PowerPC 601, and a misaligned floating point store in the cache usually spends 5 clocks in the execution stage instead of 1. The current implementation of Clean does not store floating point values at double word aligned addresses, but only at word aligned addresses. To determine the cost of this inefficiency we changed the `fft` and `fftc` C programs to place the arrays at non double word aligned addresses.
These programs (`fftunalignc` and `fftcunalignc`) were about 20 percent slower:

	Clean	CW	MrC	Clean/CW	Clean/Mrc
`fft`	5.51	4.67	4.47	1.18	1.23
`fftunalignc`	5.51	5.78	5.47	0.95	1.01
`fftc`	2.38	2.20	1.60	1.08	1.49
`fftcunalignc`	2.38	2.65	1.92	0.90	1.24

The `fft` and `fftc` C programs store the arrays always at double word aligned addresses. The `malloc` function of one of the C compilers did not return double word aligned addresses.

So for better performance we would have to change the Clean runtime system to allocate doubles at double word aligned addresses. Allocating arrays of reals at these addresses can be done easily. Changing the sliding compacting garbage collector to move these arrays to double word aligned addresses is more difficult.

Aligning doubles on the stack is also more difficult. We probably have to align all stack frames at double word addresses, which would result in higher use of stack space even for programs that do not use reals. Another option is to use a separate stack for reals. For the `fft` programs aligning reals on the stack is not important, because few reals are stored on the stack.

The code generated for loops can be improved easily. The test for the end of the loop is usually done in a pattern or guard. In such a case the current compiler generates code which looks like this:

```
function_entry:   cmp          index,size
                  beq          guard_false
guard_true:       do something
                  bra          function_entry
guard_false:
```

We can reduce the number of branches in the loop by changing this to:

```
guard_true:       do something
function_entry:   cmp          index,size
                  bne          guard_true
guard_false:
```

Applying this transformation to quicksort would reduce the number of branches in the most important loops from 3 to 2. A similar transformation would remove an incorrectly predicted branch from the most important loop of heapsort. C compilers do a similar transformation to reduce the number of branches in loops.

Another optimisation that C compilers perform, and has not yet been implemented in the Clean compiler, is common subexpression elimination of array index calculations. The Clean compiler generates 2 machine instructions on a RISC to calculate the address of an element in an array of integers or reals: a shift instruction and an add instruction. In the `quicksort` and `fft` programs the same index is used twice in the same basic block: when two elements are swapped and in the function `merge`. The addresses can be calculated once, and then used by more load or store instructions, saving 2 instructions per additional load or store.

We could also try to eliminate the add instruction in the address calculation. The compiler has to generate it because the first element of an array is not stored at offset 0 from the address of the array, but at offset 12 or 8. If an array is used several times, we could calculate the address of the element once, and use the address as the base for the indexed load or store instruction. This saves one instruction for every additional load or store of the same array.

We could remove this extra offset calculation in all cases, if we change the way we store nodes in the heap. In the current implementation we cannot pass the address of the first element of an array instead of a pointer to the node, because the descriptor has to be at offset 0 from the address. This is necessary because of the garbage collector and the way closures are evaluated.
In the current implementation the address of a node is the address of the descriptor of the node. If instead we would store the address of the word following the descriptor, we could remove the extra offset calculation for arrays, but we would have to change this for all nodes. We would have to change the compiler, the garbage collectors and other parts of the runtime system. Code that doesn't use arrays would probably still be just as efficient on a RISC after this modification.

The code generated for the fast fourier transform by the Clean compiler can probably be improved with a simple instruction scheduler. Instruction scheduling is not very important for the `quicksort` and `heapsort` programs.

5.3 Related Work

In Haskell, lazy arrays can be created using comprehensions, but in-place array updates are not possible. Several analyses and transformations are required to compile these comprehensions efficiently [AnHu90]. Currently none of the available Haskell compilers are able to do this.

The Glasgow Haskell compiler (GHC) has been extended with an array implementation based on monads [PeWa92]. Unboxed arrays (called Bytearrays) of basic types can be manipulated. For each basic type, functions are available to create, read or write an array. For example `writeIntArray` and `writeDoubleArray` are available, but there is no `writeArray` function that can be used for every Bytearray. [Serr96] compares the performance of both GHC's Bytearrays and monolothic arrays with Clean's unboxed arrays for a conjugate gradient algorithm, and Clean was always at least ten times as fast.

Using analyses, program transformations and reference counting strict pure functional languages like SISAL [FiOl95,Feo92] and SAC [MuKlSc96] achieve good performance for scientific applications using arrays.

The Clean compiler [SNGP91] inserts the necessary coercions to box or unbox basic types, tuples and records as described in [NöSm93]. Leroy [Lero92] also

discusses automatic coercion of these data types. However, the transformation Leroy describes cannot unbox arrays. Unboxed (polymorphic) arrays are also not possible with unboxed types as described in [PeLa91]. This type system has as disadvantage that coercions have to be inserted by the programmer.

6 Conclusion

The efficiency of unboxed arrays in Clean 1.1 is good, and it can still be further improved with simple optimisations. Multidimensional arrays and array of unique elements are more difficult to use than one dimensional arrays. Strictness annotations, and sometimes small changes to the program, are still necessary to obtain the most efficient version of a Clean program.

References

[AnHu90] Andersen, S. and P. Hudak. 1990. Compilation of Haskell Array Comprehensions for Scientific Computing. In Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation, New York, pp. 137-149.

[BaSm93] Barendsen E., and J.E.W. Smetsers. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *proceedings 13th conference Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, December 1993, pp. 41-51, LNCS 761.

[Feo92] Feo, J. A Comparative Study of Parallel Programming Languages: The Salishan Problems. 1992. North Holland, ISBN 0-444-88135-2.

[FiOl95] Fitzgerald, S.M. and R.R. Oldehoeft. 1995. Update-in-place Analysis for True Multidimensional Arrays. *High Performance Functional Computing*, pp. 105-118.

[Jone95] Jones, M.P. 1995. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Journal of Functional Programming* 5(1) - January 1995, Cambridge University Press, pp. 1-35.

[NöSm93] Nöcker, E.G.J.M.H., and J.E.W. Smetsers. 1993. Partially strict non-recursive data types. In *Journal of Functional Programming* 3(2), pp. 191-215.

[Lero92] Leroy X. 1992. Unboxed objects and polymorphic typing. *Proc. 19th Symp. Principles of Programming Languages*, pp. 177-188.

[MuKlSc96] Mullin, L.R., W.E. Kluge and S. Scholz. 1996. On Programming Scientific Applications in SAC - a Functional Language Extended by a Subsystem for High-Level Array Operations. *Proc. of the 8th International Workshop on Implementation of Functional Languages.*

[PeLa91] Peyton Jones, S.L. and J. Launchbury. 1991. Unboxed values as first class citizens in a non-strict functional language. In *Proc. of Conference on Functional Programming Languages and Computer Architecture* (FPCA `91), Sept. 1991, Cambridge, LNCS 523.

[PeWa92] Peyton Jones, S.L. and P. Wadler. 1992. Imperative functional programming. *ACM Symposium on Principles of Programming Languages (POPL)*, pp.71-84.

[PlaEe95] Plasmeijer, M.J., and M.C.J.D. van Eekelen. 1995. Clean 1.1 Reference Manual. Technical Report. University of Nijmegen, The Netherlands.

[Serr96] Serrarens, P.R. A Clean Conjugate Gradient Algorithm. 1996. *Proc. of the 8th International Workshop on Implementation of Functional Languages.*.

[SNGP91] Smetsers, J.E.W., E.G.J.M.H. Nöcker, J.H.G. van Groningen and M.J. Plasmeijer. 1991. Generating Efficient Code for Lazy Functional Languages, In *Proc. of Conference on Functional Programming Languages and Computer Architecture* (FPCA '91) , Cambridge, MA, USA, Springer-Verlag, LNCS 523, pp. 592-617.

[Wadl86] Wadler. P. 1986. A new array operation for functional languages. In *Proc. Graph Reduction Workshop*, Santa Fe, Springer Verlag, LNCS 295.

Appendix: benchmark programs

1 Implementation of the Quicksort Algorithm in Clean

```
module quicksort;
import StdEnv;
:: SortElement:==Int;
:: SortArray:=={#SortElement};

quick_sort :: *SortArray -> .SortArray;
quick_sort a0
  = quick_sort1 0 (n_elements-1) a;
  {
    (n_elements,a) = usize a0;

    quick_sort1 b e a
      | b>=e
        = a;
    quick_sort1 b e a=:{[b]=ab,[m]=am}
      = find_large am (b+1) e {a & [m]=ab};
    { m=(b+e)>>1;

      find_large :: Int Int Int *SortArray -> .SortArray;
      find_large am l r a
        | l<=e && a.[l]<=am
          = find_large am (l+1) r a;
          = find_small_or_equal am l r a;

      find_small_or_equal :: Int Int Int *SortArray -> .SortArray;
      find_small_or_equal am l r a
        | r>b && a.[r]>am
          = find_small_or_equal am l (r-1) a;
        | l<r
          # (al,a)=uselect a l;
            (ar,a)=uselect a r;
          = find_large am (l+1) (r-1) {a & [l]=ar,[r]=al};
        | b==r
          = quick_sort2 (r-1) (r+1) {a & [b]=am};
          # (ar,a)=uselect a r;
          = quick_sort2 (r-1) (r+1) {a & [r]=am,[b]=ar};
```

```
      quick_sort2 l r a
        = if (l-b>=e-r)
          (quick_sort1 b l (quick_sort1 r e a))
          (quick_sort1 r e (quick_sort1 b l a));
    }
  }
unsorted_array n_elements
  = fill_unsorted_array 0 n_elements (createArray n_elements 0);
  { fill_unsorted_array i s a
      | i<s
        = fill_unsorted_array (i+1) s {a & [i]=(s-i-1) bitxor 0x2a};
        = a;
  }
check_sort :: !Int !Int !{#Int} -> Bool;
check_sort n n_elements a
  | n==n_elements = True;
  | a.[n]==n      = check_sort (n+1) n_elements a;
repeat_sort 0 = True;
repeat_sort n
  | check_sort 0 n_elements (quick_sort (unsorted_array n_elements))
    = repeat_sort (n-1);
  where { n_elements=200000; }
Start = repeat_sort 10;
```

2 Implementation of the Heapsort Algorithm in Clean

```
module heapsort;
import StdEnv;
:: SortElement:==Int;
:: SortArray:=={#SortElement};

heap_sort a0
  | n_elements<2
    = a
    = sort_heap max_index (make_heap (n_elements>>1) max_index a);
    { max_index=n_elements-1;

      make_heap :: Int !Int *SortArray -> *SortArray;
      make_heap (-1) max_index a = a;
      make_heap i max_index a=:{[i]=ai}
        = make_heap (dec i) max_index (add_to_heap i ((i<<1)+1) max_index ai a);

      sort_heap :: Int *SortArray -> *SortArray;
      sort_heap i a=:{[i]=ai,[0]=a0}
        | i==1
          = {a & [0]=ai,[i]=a0};
          = sort_heap deci (add_to_heap 0 1 deci ai {a & [i]=a0});
          with { deci=i-1; }

      add_to_heap :: Int Int !Int SortElement *SortArray->*SortArray;
      add_to_heap i j max_index ai a
        | j>=max_index
          = if (j>max_index)
            {a & [i] = ai}
            (if (ai<aj)
              {a` & [i]=aj,[j]=ai}
              {a` & [i]=ai});
          with { (aj,a`)=uselect a j; }
      add_to_heap i j max_index ai a=:{[j]=aj,[j1]=aj1}
        | aj<aj1
          = if (ai<aj1)
```

```
              (add_to_heap j1 ((j1<<1)+1) max_index ai {a & [i]=aj1})
              {a & [i]=ai};
          = if (ai<aj)
              (add_to_heap j ((j<<1)+1) max_index ai {a & [i]=aj})
              {a & [i]=ai};
        where { j1=j+1; }
    }
  where { (n_elements,a) = usize a0; }

repeat_sort 0 = True;
repeat_sort n
  | check_sort 0 n_elements (heap_sort (unsorted_array n_elements))
    = repeat_sort (n-1);
  where { n_elements=200000; }
Start = repeat_sort 10;
```

3 Implementation of the Quicksort and Heapsort Algorithms in C

```
#include <stdlib.h>
#include <stdio.h>
static void quick_sort1 (int b,int e,int a[])
{ int m,l,r,ab,am;
  while (b<e){
    m=(b+e)>>1;
    ab=a[b]; am=a[m]; a[m]=ab;
    l=b+1; r=e;
    for (;;){
      while (l<=e && a[l]<=am)
        ++l;
      while (r>b && a[r]>am)
        --r;
      if (l<r){
        int al,ar;
        al=a[l]; ar=a[r];
        a[l]=ar; a[r]=al;
        ++l; --r;
      } else
        break;
    }
    if (b==r) a[b]=am;
    else { int ar; ar=a[r]; a[r]=am; a[b]=ar; }
    l=r-1; ++r;
    if (l-b>=e-r){
      quick_sort1 (r,e,a);
      e=l;
    } else {
      quick_sort1 (b,l,a);
      b=r;
    }
  }
}
static void quick_sort (int a[],int n_elements)
{ quick_sort1 (0,n_elements-1,a);
}
static void add_to_heap (int i,int j,int max_index,int ai,int a[])
{
  while (j<max_index){
    int j1,aj,aj1;
    j1=j+1; aj=a[j]; aj1=a[j1];
    if (aj<aj1){
      if (ai<aj1){
```

```
        a[i]=aj1;
        i=j1; j=(j1<<1)+1;
      } else {
        a[i]=ai;
        return;
      }
    } else {
      if (ai<aj){
        a[i]=aj;
        i=j; j=(j<<1)+1;
      } else {
        a[i]=ai;
        return;
      }
    }
  }
  if (j>max_index){
    a[i]=ai;
  } else {
    int aj;
    aj=a[j];
    if (ai<aj){ a[i]=aj; a[j]=ai; } else { a[i]=ai; }
  }
}
static void heap_sort (int a[],int n_elements)
{
  int max_index,i;
  if (n_elements<2)
    return;
  max_index=n_elements-1;
  for (i=n_elements>>1; i!=-1; --i)
    add_to_heap (i,i+i+1,max_index,a[i],a);
  i=max_index;
  while (i!=1){
    int ai;
    ai=a[i]; a[i]=a[0]; --i;
    add_to_heap (0,1,i,ai,a);
  }
  { int a0,ai; a0=a[0]; ai=a[i]; a[0]=ai; a[i]=a0; }
}
int main (void)
{
  long begin_time,end_time; int n_elements,*a,i,int count;
  n_elements=200000; a=malloc (n_elements*sizeof (int));
  if (a!=NULL){
    printf ("\n"); begin_time=TickCount();
    for (count=0; count<10; ++count){
      for (i=1; i<=n_elements; ++i)
        a[i-1]=(n_elements-i) ^ 0x2a;
      /* quick_sort (a,n_elements); */
      heap_sort (a,n_elements);
      check_sort (a,n_elements);
    }
    end_time=TickCount(); printf ("%g\n",(double)(end_time-begin_time)/60.0);
  }
  return 1;
}
```