Strict and Unboxed Lists using Type Constructor Classes in a Lazy Functional Language

John van Groningen and Rinus Plasmeijer

University of Nijmegen, Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, the Netherlands {johnvg,rinus}@cs.kun.nl

Abstract. We discuss how strict and unboxed lists are defined and implemented in the lazy functional programming language Clean 2.0. Multiparameter type constructor classes are used to overload both pattern matching and construction of lists. The overhead of overloading is eliminated by creating specialized versions of overloaded functions during compilation. Our measurements of the execution time of some programs show that some programs execute several times faster using strict or unboxed lists instead of lazy lists.

1 Introduction

Lists are very convenient data structures. Of all data structures one can define in a function language, lists are probably the ones most frequently used. Functional languages therefore generally offer lists as a predefined data structure and offer special syntax for handling these predefined lists. For instance, in a lazy functional language such as Miranda, Haskell and Clean there is special syntax for pattern matching on lists and for the creation of lists, like e.g. dot-dot notations and list comprehensions. A disadvantage of the special syntax is that it only works for the predefined list type, in a lazy language this will be the lazy list. All functional languages offer the possibility to define user-defined data structures. In a language like Clean the programmer cannot only define a tailor made lazy list, it is also possible to define eager lists and unboxed lists. All these different types of lists (lazy, eager and unboxed) have different time and space properties. Sometimes it is better to use a lazy list, sometimes it is better to use a strict list or an unboxed one. Although all these lists are similar, they are actually considered of different type. They also may be differently implemented (e.g. boxed or unboxed). It is not possible to define overloaded functions that work on any list type. Also the nice syntax for the predefined lists type cannot be used for the other lists variants. For all these reasons it is in practice not so easy to switch from one list type to another. In this paper we show how the overloading mechanism is used in Clean 2.0 to solve the problems mentioned above. Special about the solution is the use of overloading in the pattern match and that unboxed lists are possible. The unboxing methods in [8], [6] and [7] cannot unbox polymorphic elements of recursive algebraic types, such as the elements of a list.

1.1 Overview of the paper

In this paper we will first explain the differences between the predefined list type in Clean (a lazy lists) and the different kinds of lists (lazy, strict and unboxed) the programmer can define by using an algebraic data structure. Then we will explain the syntactical support we offer in Clean 2.0 for handling lazy, strict and unboxed lists in a uniform way. Overloaded functions can be defined that work on any kind of list. We will explain how the implementation is done and show some measurements.

2 Lists in Clean 1.3

In this section we will explain what kind of support is offered in Clean 1.3 for the predefined lazy list. Next we explain how the programmer can define his own lazy, strict and unboxed list variants.

2.1 Lazy lists in Clean 1.3

Since Clean is a lazy functional language, the predefined type list is a lazy list. There is a lot of special syntax for lazy lists, which differs slightly from the syntax used in Haskell (like in Haskell [] is used for Nil, but [x:xs] is used for a non-empty list with head x and tail xs). For instance, the function map can be defined in Clean as follows:

```
map :: (a -> b) [a] -> [b]
map f [ ]      = [ ]
map f [x : xs] = [f x : map f xs]
```

By using a list comprehension one could have defined map equally well as:

```
map :: (a -> b) [a] -> [b]
map f list     = [f x \\ x <- list]
```

In the standard library of Clean (StdEnv) all standard functions on lists, such as map, are predefined. However, all these predefined functions are defined on lazy lists only and they cannot be used on other types of lists.

2.2 User defined lists in Clean 1.3

In Clean arbitrary data structures can be defined using an algebraic data type definition. It is also possible to define a lazy list in this way as follows:

```
:: LazyList a             = Cons a (LazyList a)
                          | Nil
```

Even though this lazy list has exactly the same property as the default predefined lazy list, it introduces a new type. As a consequence the nice syntax such as dot-dot expressions and list comprehensions available for predefined lists cannot be used for this user defined lazy list. A function like map, say map_l, can be defined as follows on this user defined lazy list:

```
map_l :: (a -> b) (List a) -> (List b)
map_l f Nil         = Nil
map_l f (Cons x xs) = Cons (f x) (map_l f xs)
```

Given the syntactical convenience offered for predefined lists it does not make sense to have a user defined lazy list in addition. However, with an algebraic data type definition one cannot only define lazy data structures in Clean, one can also define (partially) eager data structures and unboxed variants of them.

```
:: LazyList a            = Cons  a (LazyList a)
                         | Nil
:: HeadStrictList a      = HSCons !a (HeadStrictList a)
                         | HSNil
:: TailStrictList a      = TSCons a !(TailStrictList a)
                         | TSNil
:: StrictList a          = SCons !a !(StrictList a)
                         | SNil
:: HeadUnboxedIntList    = HUCons !Int HeadUnboxedIntList
                         | HUNil
:: UnboxedIntList        = UCons !Int !UnboxedIntList
                         | UNil
```

Above different algebraic data type definitions of a list are given as they can be defined in Clean. All these lists are of different type and have different properties. The first definition (LazyList) is the ordinary lazy list we have seen before. The lists will not be evaluated until its evaluation is demanded. Even then, it will first be evaluated only to head normal form such that a distinction between a Nil and a Cons can be made. The head and tail are not evaluated unless their values are needed.

With an exclamation mark one can specify in an algebraic type definition that part of a data structure should be evaluated strictly instead of lazily. Whenever such a data structure appears in a strict context, which forces its evaluation to head normal form, the marked part of the data structure will be evaluated as well. So the head element of the HeadStrictList will always be evaluated whenever the list itself is evaluated. The whole spine of the TailStrictList will be evaluated whenever the list is evaluated. The whole StrictList (all elements and the whole spine) will be evaluated whenever the list is evaluated.

Besides changing the strictness property the programmer can also influence the representation of a list. By default everything in Clean is boxed. For instance, a list of integers ([Int]) is represented by a linked list of Cons-nodes ended by the empty list (Nil). Each Cons node will contain a pointer to a node which after evaluation (!) will contain an integer value. For a lazy lists it is clearly the best to have a boxed representation: even though we have a list of integers, the list elements may be unevaluated and contain a closure. If we have a head strict list we know that the element will always be evaluated and we choose a more compact representation. Instead of a pointer to the evaluated integer we can more compactly store the integer itself into the list. Such a list is called unboxed. In Clean an unboxed representation will be chosen for strict marked elements of basic type (Int, Real, Bool, Char), of record type, tuple type and of array type. In all other cases boxed representations will be made. For instance, in the definitions above HeadUnboxedIntList is a head strict unboxed list of integers and UnboxedIntList is an unboxed lists of which all the elements as well as the spine is evaluated.

All lists defined above are similar, but they all have different time and space properties. For instance, lazy lists are very convenient (nothing is evaluated unless it is really needed for the computation, one can deal with infinite data structures), but they can be inefficient as well if actually always all lists elements are evaluated sooner or later. Strict list are often more efficient, but one has to be certain not to trigger a not used infinite computation. Spine strict lists can be more efficient as well, but one cannot handle infinite lists in this way. Unboxing saves space, but not if Cons nodes are copied often: the lists elements are copied as well while otherwise the contents could remain shared using the element pointer instead. In terms of efficiency it can make quite a difference which kind of list is actually used. This is also illustrated by the measurements at the end of the paper. It is in general not decidable which kind of list is best to use. This depends on how a list is used, the algorithms that are used in the program. A wrong choice might turn a program from useful to useless in terms of time and space consumption, from terminating to non- terminating.

All lists above are lists, yet they are all of different type. This means that handy functions defined on one kind of list cannot simply be used for another kind of list. It would be very convenient if the programmer can switch very easily from one type of list to another. It would also be nice if functions can be reused and that the programmer can easily switch from representation without having to do any recoding.

3 Clean 2.0 and lazy, strict, unboxed lists

In Clean 2.0 we have solved the problems in the following way. First of all we have introduced special syntax for strict and unboxed lists, similar as we have for lazy lists. This is of course trivial, but not unimported for the programmer. More important is that we also allow the definition of overloaded functions that work on any list. Unboxing and overloading of lists in Clean uses a similar syntax and implementation as unboxing and overloading of arrays in Clean [3].

3.1 Syntax for lazy, strict, unboxed lists

For lazy lists we use the same notation as in Clean 1.3. [ ] is the empty list and [h : t] is a list with head h and tail t. The type of lists of type a is written as [a]. So for example, the function map on lazy lists can be defined as shown before:

```
map_l :: (a -> b) [a] -> [b] // lazy list
map_l f [ ]      = [ ]
map_l f [x : xs] = [f x : map_l f xs]
```

For head strict lists, we can now use a similar syntax as for lazy lists, except that we add a ! after the [, for example:

```
map_hs :: (a -> b) [!a] -> [!b] // head strict list
map_hs f [! ]      = [! ]
map_hs f [!x : xs] = [!f x : map_hs f xs]
```

For tail strict lists, we can use a similar syntax as for lazy lists, except with a ! before the], for example:

```
map_ts :: (a -> b) [a!] -> [b!] // tail strict list
map_ts f [ !]      = [ !]
map_ts f [x : xs!] = [f x : map_ts f xs!]
```

And for head a tail strict lists, we add both !s, so one after the [and one before the], for example:

```
map_s :: (a -> b) [!a!] -> [!b!] // head and tail strict list
map_s f [!!]       = [!!]
map_s f [!x : xs!] = [!f x : map_s f xs!]
```

3.2 Syntax for unboxed lists

To make the head of lists unboxed (and therefore strict) we use a `#` instead of a `!` after the `[`, so for example:

```
map_ui :: (a -> b) [#Int] -> [#Int] // head strict, unboxed list
map_ui f [# ]      = [# ]           //                      of Int
map_ui f [#x : xs] = [#f x : map_ui f xs]
```

And for unboxed lists which are also tail strict, we use the syntax of unboxed lists, with a ! before the], so for example:

```
map_utsi :: (a -> b) [#Int!] -> [#Int!] //head unboxed+tail strict
map_utsi f [# !]      = [# !]           //           list of Int
map_utsi f [#x : xs!] = [#f x : map_utsi f xs!]
```

Unboxing is not allowed for all types of the lists, only for lists of basic types (integers, characters, reals, boolean, etc), records and arrays. In the case of arrays, using an unboxed list does not really unbox the array in the cons node, but just removes an extra node between the cons node and the array.

The nice syntax for lazy lists is also available for strict / unboxed versions, by adding `!`s or `#`. Not just for denotations and types but for list comprehensions and dot-dot expressions as well.

All these six kind of lists have a different type, so conversion functions are needed to convert one type into another. Reusing of code has now become a bit easier than without special syntax for strict or unboxed lists: just change the `!` or `#` near the brackets.

3.3 Overloading for unboxed lists

For unboxed lists we need to use different representations for cons nodes depending on the type, because the elements of the list can have different sizes. Therefore the unboxed lists are overloaded, using the type constructor class [5] UList. For example:

```
map_u :: (a -> b) [#a] -> [#b] | UList a & Ulist b
map_u f [# ]      = [# ]
map_u f [#x : xs] = [#f x : map_u f xs]
```

And for unboxed tail strict lists, using the type constructor class UTSList.

```
map_uts :: (a -> b) [#a!] -> [#b!]  | UTSList a & UTSList b
map_uts f [# !]      = [# !]
map_uts f [#x : xs!] = [#f x : map_uts f xs!]
```

These classes are instantiated for basic types, records and arrays types. Note that we do not just have overloading of function calls (nil and cons), but also have overloading in the pattern matches.

3.4 Overloading for lists

To allow the definition of functions that can be used for all previously defined types of list, we add an overloaded list using the multiparameter type constructor class [9] List. In patterns and expressions we add a | after the [to indicate that it is an overloaded list. For example:

```
map :: (a -> b) (m a) -> (n b)  | List m a & List n b
map f [|]       = [|]
map f [|x : xs] = [|f x : map f xs]
```

defines a function that can be used for all representations:

```
[ ], [! ], [ !], [!!], [# ], [#!]
```

The implementation of the overloading mechanism of the Clean 2.0 compiler takes care of creating specialized versions so that programs will run at the same speed as without overloading in most cases. This happens automatically within the same module. If an overloaded function is exported from a module, the programmer can specify which specialized versions of the function should be exported as well. These specialized versions will then be used automatically by the compiler, so that in these cases we can use overloading without loss of efficiency as well. Again we do not just have overloading of function calls, but also have overloading in the pattern matches.

4 Implementation

4.1 Implementation of [ ], [!], [!], [!!]

The unboxed lists ([ ], [!], [!] and [!!]) can be implemented just like all other (partially) strict algebraic types in the following way. For each of these types we add a predefined algebraic type definition to the compiler with a Cons (with appropriate strictness information) and a Nil constructor. These are the types LazyList, HeadStrictList, TailStrictList and StrictList from section 1.2. The compiler translates the special list syntax to these algebraic types, for example for head strict lists, the type [!a] is translated to (HeadStrictList a), [!] to HSNil and [! 1: l] to (HSCons 1 l).

Pattern matching and construction of these types can be done in the same way as for user defined algebraic types as follows.

Pattern matching is implemented by translating patterns to case expressions. Case expressions load the descriptor of the node, and compare this descriptor with the required descriptor (for example HSNil or HSCons). In case of a cons node, the head and tail are loaded from the cons node.

A nil is constructed by allocating a node in the heap with just a Nil, HSNil, TSNil or SNil descriptor (actually, a new node is not allocated each time, but a preallocated node is used in most cases).

How a cons is constructed depends on whether the cons is in a strict context or in lazy context, and in a lazy context, whether the strictly annotated elements of the cons (the head or tail) have already been evaluated. In a strict context, or in a lazy context where the compiler can infer that the strictly annotated elements of the cons have already been evaluated, a cons is constructed by allocating a node in the heap with a Cons, HSCons, TSCons or SCons descriptor and pointers to the head and the tail nodes.

Otherwise, we cannot construct the cons node immediately because we have to evaluate some elements of the cons node first. We use three auxiliary functions (hscons, tscons and scons) to implement this, one for each (partially) strict cons, that evaluates the strictly annotated elements of the cons before constructing the cons node. For example for head strict lists this function is defined as:

```
hscons h t
    = let! evaluated_h=h // evaluate the head
      in   HSCons evaluated_h t
```

Note that we do not need such a function for lazy lists. So for a cons in a lazy context where the compiler cannot infer that the strictly annotated elements of the cons have already been evaluated, a closure is allocated with a descriptor of one of these functions and pointers to the head and tail.When such a closure is evaluated, the strictly annotated elements are evaluated by the code for the function, and then the node is overwritten with a cons node.

The descriptors in Clean are used to describe the size and contents of nodes in Clean for the garbage collector, pattern matching and for printing. Because the sizes and contents of all nil and all non unboxed cons nodes are the same, and we would like to print all lists in the same way, and Clean is a typed language, we can use the same Nil descriptor for all lists types, and the same Cons descriptor for all non unboxed lists types at runtime. So at runtime all these four lists have the same representation, except that different evaluation functions (hscons, tscons and scons) are used. Of course these evaluation functions, now always create a node with a Cons descriptor.

By using the same descriptors we can provide efficient conversions from one kind of list to another. The following conversion are now possible without traversing the list:

```
[! ], [ !]    -> [ ]
[!!]          -> [ ], [! ],[ !]
```

or with traversing, but without copying:

```
[ ], [! ], [ !] ->  [!!]
[ ]             ->  [ !]
```

4.2 Implementation of (overloaded) unboxed lists

Because the representation of a cons of an unboxed list depends on the type of the element, a different Cons descriptor and node is needed for each type of unboxed list. For the basic types and array types these are predefined, but this is not possible for record types. Therefore the compiler generates a new Cons descriptor (and code for the conversion function) in each module where a cons of an unboxed list of records is used (at most once per module).

We use the same representation for Nil as for non unboxed lists for all types of unboxed lists. This makes it possible to decide whether a node is a nil or cons node during pattern matching without knowing the type of the list element(s). Instead of recognizing a cons node by testing that the descriptor in the node is the appropriate Cons descriptor, we detect a cons node when the descriptor is not the Nil descriptor. This is necessary for implementing unboxed list of records, because there can be more than one Cons descriptor for such a list if it is used in more than one module. But more importantly, it makes it easier to implement overloaded pattern matching of lists.

Unboxed lists and unboxed tail strict lists can use the same representation for cons nodes at runtime, so it possible to provide an efficient conversion function without traversing from `[# !] -> [# ]`.

We do not have special syntax for each unboxed list type in Clean 2.0, but only an overloaded notation for unboxed lists (see section 3.3). We use the following predefined type constructor class to implement unboxed head strict lists:

```
class UList e
where
        cons_u   :: !e [#e] -> [#e]
        decons_u :: ![#e] -> (!e,[#e])
```

and the following overloaded constant function:

```
nil_u :: [#e] | UList e
```

Denotations of unboxed lists are translated to `cons_u` and `nil_u` function calls. For example `[#1]` is converted to `(cons_u 1 nil_u)`. `decons_u` is used during pattern matching to load the head (first element of yielded tuple) and tail (second element of yielded tuple) of a cons node.

For example:

```
map_u :: (a -> b) [#a] -> [#b] | UList a & Ulist b
map_u f [# ]      = [# ]
map_u f [#x : xs] = [#f x : map_u f xs]
```

Will be translated to:

```
map_u :: (a -> b) [#a] -> [#b] | UList a & Ulist b
map_u f list =
    case list of
        Nil     = nil_u
        not Nil = let! (x,xs) = decons_u list in
                        cons_u (f x) (map_u f xs)
```

The implementation of the `nil_u` function yields a Nil node, for all unboxed lists, because we use the same representation for nil. For `cons_u` en `decons_u` we provide instances for all unboxed list types. Each of these `cons_u` instances, will evaluate and unboxed its first argument (the head) and yield the appropriate cons node with an unboxed head and a pointer to the tail node. Each `decons_u` instance will load the elements of the cons node passed as argument, box the head and yield a tuple with the boxed head as first element and the tail as second element.

For example for `[#Int]`:

```
instance UList Int
where
    cons_u h t
        // unboxed_h=evaluate and unbox h
        = Cons_Int unboxed_h t
    decons_u (Cons_Int h t)
        // boxed_h = box h
        = (boxed_h,t)
```

These instances are predefined for the basic types and array types, but are generated by the compiler for unboxed lists of records when they are required. To generate efficient code, the code generation pass of the compiler recognizes calls of `cons_u` and `decons_u` instances, and uses the type of the instance to build a cons node or fetch the arguments of a cons node directly, without using the definitions of the instances of `cons_u` and `decons_u`. The compiler also recognizes the `nil_u` function to generate efficient code.

At runtime, we decide whether a node is a cons or nil node in the same way as for non unboxed lists, so to detect a cons node we check whether the descriptor is not the Nil descriptor.

4.3 Implementation of overloading for unboxed tail strict lists

The implementation of unboxed tail strict lists is similar to the implementation for unboxed lists, except of course that the tail is evaluated before building a cons node.

For unboxed head and tail strict lists the following class is predefined:

```
class UTSList e
where
    cons_uts  :: !e ![#e!] -> [#e!]
    decons_uts  :: ![#e!] -> (!e,![#e!])
```

and the following overloaded constant function:

```
nil_uts :: [#e!] | UTSList e
```

`nil_uts` yields a Nil node, just like `nil_u` for unboxed lists. And the instance for `cons_uts` and `decons_uts` are similar to the instances of `cons_u` and `decons_u`, except that `cons_uts` also evaluates the tail.

So for example:

```
map_uts :: (a -> b) [#a!] -> [#b!] | UTSList a & UTSlist b
map_uts f [#!]       = [#!]
map_uts f [#x : xs!] = [#f x : map_uts f xs!]
```

Will be translated to:

```
map_uts :: (a -> b) [#a!] -> [#b!]  | UTSList a & UTSlist b
map_uts f list =
    case list of
        Nil     = nil_uts
        not Nil = let! (x,xs) = decons_uts list in
                        cons_uts (f x) (map_uts f xs)
```

The code generator pass of the compiler recognizes `cons_uts`, `decons_uts` and `nil_uts` calls to generate efficient code, in the same way as for `cons_u`, `decons_u` and `nil_u`.

4.4 Implementation of overloading for lists

To implement overloaded lists (lists with `[| ]` notation) the following multiparameter type constructor class is predefined:

```
class List l e
where
    cons :: e (l e) -> (l e)
    decons :: !(l e) -> (e,(l e))
```

And the following overloaded constant function:

```
nil :: (l e) | List l e
```

This overloaded list can be instantiated with all previously described lists: `[ ], [! ], [ !], [!!], [#] and [#!]`.

The implementation of the nil function yields a Nil node, for all lists, because we use the same representation for nil.

To allow the use of lazy and strict variants of lists we define:

```
instance List [] a where
    cons a b = [a:b]
    decons [a:b] = (a,b)

instance List [!] a where
    cons a b = [!a:b]
    decons [!a:b] = (a,b)

instance List [ !] a where
    cons a b = [a:b!]
    decons [a:b!] = (a,b)

instance List [!!] a where
    cons a b = [!a:b!]
    decons [!a:b!] = (a,b)
```

and for the unboxed variants we define:

```
instance List [#] a | UList a where
    cons a b = cons_u a b
    decons a = decons_u a

instance List [#!] a | UTSList a where
    cons a b = cons_uts a b
    decons a = decons_uts a
```

For example:

```
map :: (a -> b) (l a) -> (m b)  | List l a & List m b
map f [|]       = [|]
map f [|x : xs] = [|f x : map f xs]
```

Will be translated to:

```
map :: (a -> b) (l a) -> (m b)  | List l a & List m b
map f list =
    case list of
        Nil     = nil
        not Nil = let! (x,xs) = decons list in
                        cons (f x) (map f xs)
```

The compiler recognizes overloaded cons and decons calls for UList and UTSList classes, and replaces these calls by `cons_u` / `cons_uts` or `decons_u` / `decons_uts` calls while resolving overloading. In the same pass, the compiler recognizes the following contexts: List [] a, List [!] a, List [!] a and List [!!] a and tries to resolve overloading for its members (cons and decons) even when the type of a is not known. For other contexts, the compiler would only try to resolve overloading when a concrete type has been substituted for a. We can safely do this, because the implementations of cons and decons are the same for all element types of non unboxed lists. Finally, just like for the instances of UList and UTSList, the compiler recognizes calls to cons and decons during the code generation pass, and uses the type of the instance to generate efficient code.

5 Measurements

In this section we show the execution times of some programs that use lists a lot for different kind of lists, to show the improvements in efficiency that are possible. Of course, for most real programs the improvements will be much smaller, because they usually do not spend so much time manipulating lists. These programs where executed on a 450 Mhz PowerPC G4 processor. Times are shown in seconds.

The inc_sum program first generates a list of integers, then generates a new list with each element incremented by one, and finally computes the sum of this list, this is repeated many times. The results are shown in table 1. In this case, the head strict versions are faster than the lazy list versions, but the unboxed head strict lists are even faster. The tail strict versions are slower than the non tail strict versions because of higher garbage collection costs, except for the unboxed tail strict version.

Table 1. inc_sum

list type	execution time	garbage collection time	total time
[]	22.83	0.21	23.06
[!]	15.63	0.13	15.78
[#]	13.81	0.13	13.96
[!]	21.56	4.03	25.61
[!!]	13.31	3.46	16.80
[#!]	8.10	2.06	10.18

The quicksort program sorts a list of integers many times. The results are show in table 2. In this case tail strict lists are much faster than non tail strict lists. Making the elements (head) strict or unboxed is faster than using a lazy lists, but not much faster.

Table 3 shows the results for a fast fourier transform using lists of complex numbers, represented as records with two unboxed reals. We see that the tail

Table 2. quicksort

list type	execution time	garbage collection time	total time
[]	22.41	8.31	30.75
[!]	21.93	8.23	30.18
[#]	18.83	8.01	26.88
[!]	7.51	1.61	9.15
[!!]	7.06	1.65	8.76
[#!]	5.50	1.63	7.15

strict version uses a lot more memory than the lazy list version. We had to increase the size of the heap to nearly 160 megabyte to run the program. This happens because large lists of unevaluated elements are build, each consisting of several unevaluated nodes. By making these lists head strict or unboxed, these elements are evaluated when the list is created, and the memory use decreases. The head and tail strict version is the fastest. The unboxed (and tail strict) version is slower in this case than the boxed version, because the list is copied several times by split and append functions, and this is more expensive for unboxed lists of records, because the cons nodes are larger.

Table 3. fft (256*1024 complex double precision)

list type	execution time	garbage collection time	total time	heap size
[]	9.38	14.41	23.83	64m
[!]	8.45	11.68	20.15	64m
[#]	8.16	11.25	19.41	64m
[!]	7.83	13.83	21.68	160m
[!!]	4.08	3.61	7.71	32m
[#!]	4.71	3.60	8.31	32m

Finally, the results for Wang's algorithm for solving a system of linear equations based on a large tri-diagonal matrix are shown in table 3. This is the program from [4], but using unboxed records of four reals, instead of tuples of reals. The [#!]/[!!] version uses unboxed lists of basic types, but strict (boxed) lists of records. We see that the tail strict list is slower than the lazy lists. Making the elements strict or unboxed instead of lazy makes the program faster, but unboxing the records makes the program slower than with boxed (strict) records.

6 Conclusions

We have introduced one similar notation for all kinds of lists with different representations (boxed or unboxed) and strictness properties, which can also be used for list comprehensions and dot-dot expressions. We have shown how

Table 4. wang (12m heap)

list type	execution time	garbage collection time	total time
[]	7.93	3.83	11.78
[!]	5.60	3.53	9.15
[!]	8.35	4.13	12.50
[!!]	4.83	0.81	5.66
[#!]/[!!]	4.78	0.75	5.55
[#!]	5.46	1.05	6.53

overloaded functions that work for any type of these lists can be defined, how this is implemented and that switching from one type of list to another is easy and efficient. Usually there is no efficiency penalty when overloading is used, because the overhead of overloading is eliminated by creating specialized versions of overloaded functions during compilation. Our measurements have shown that large gains in efficiency are possible using strict or unboxed lists instead of lazy lists.

In the future we hope to determine strictness properties of lists automatically using strictness analysis. Another possible approach is using flow analysis [1] [2].

It is possible to generalize this system to arbitrary algebraic data types, so that overloaded function definitions for strict and unboxed versions of any data type are possible.

References

1. Boquist, Urban : Code Optimisation Techniques for Lazy Functional Languages. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, March 1999.
2. Faxen, Karl-Filip : Analysing, Transforming and Compiling Lazy Functional Programs. PhD thesis, Dept. of Teleinformatics, Royal Institute of Technology, Kista, 1997.
3. Groningen, John H.G. van : The Implementation and Efficiency of Arrays in Clean 1.1. In Proc. of Implementation of Functional Languages, 8th International Workshop, IFL '96, Selected Papers, Bad Godesberg, Germany, Kluge Ed., Springer Verlag, LNCS 1268, 1997, pp. 105-124.
4. Hartel, P. H. and Langendoen, K. G. : Benchmarking implementations of lazy functional languages. In 6th Int. Conf. Functional programming languages and computer architecture, pages 341-349, Copenhagen, Denmark, Jun 1993. ACM, New York.
5. Jones, Mark P. : A system of constructor classes: overloading and implicit higher-order polymorphism. In FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993. (Appears, in extended form, in the Journal of Functional Programming, 5, 1, Cambridge University Press, January 1995.)
6. Leroy, Xavier : Unboxed objects and polymorphic typing. Proc. 19th symp. Principles of Programming Languages, ACM press , 1992, pp 177-188.

7. Nöcker, E.G.J.M.H. and Smetsers, J.E.W. : Partially strict non-recursive data types. In Journal of Functional Programming 3:2, 1993, pp. 191-215.
8. Peyton Jones, SL. and Launchbury, J. : Unboxed values as first class citizens. Functional Programming Languages and Computer Architecture (FPCA'91), Boston, LNCS 523, Springer Verlag, Sept 1991, pp 636-666.
9. Peyton Jones, Simon, Jones, Mark, Meijer, Erik : Type classes: exploring the design space. Haskell Workshop 1997.