Optimising Recursive Functions Yielding Multiple Results in Tuples in a Lazy Functional Language

John H.G. van Groningen*

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

Abstract. We discuss a new optimisation for recursive functions yielding multiple results in tuples for lazy functional languages, like Clean and Haskell. This optimisation improves the execution time of such functions and also reduces the amount of memory allocated in the heap by these functions, which reduces garbage collection costs. In some cases execution time is improved by more than a factor of two and allocation costs by a factor of four. Furthermore, the space leak that is caused by selector nodes is removed.
This is achieved by reusing nodes allocated in the previous iteration of the recursion to create the nodes for the next iteration, by updating these nodes. Only the parts of the nodes that have changed are updated. Because of these updates, the code that is used to select an element of a tuple is not executed anymore for many selections, because the selector node was overwritten with a new selector node or the result before it is evaluated.

1 Introduction

In lazy functional programming languages functions yielding multiple results often yield these results in a tuple. Unfortunately, current compilers do not generate very efficient code for such functions. Compilers usually generate code that evaluates the result of a function to root normal form. So if the result of a function is a tuple denotation, the elements of the tuple are not evaluated, because of laziness. Therefore often thunks have to be created for the elements of the tuple. Furthermore, because the elements of the tuple are in lazy contexts, function arguments can usually be evaluated only by pattern matching and guards. Therefore few arguments of such a function are strict, so strictness analysis does not help much.

Of course, for other lazy data structures we have these problems as well, but there is another reason for inefficient compilation of tuples: lazy pattern matching. Because matching of tuples always succeeds, this is often done in a let or where binding, which uses lazy pattern matching, instead of in the pattern of a function definition or case expression, which uses strict pattern matching.

* Supported by STW and Appligraph

To compile a lazy pattern match of a function call yielding a tuple, the compiler has to create a thunk for the function call, and for each element of the tuple (if it is used) a thunk that selects the appropriate element of the tuple. Creating all these thunks can be avoided if the compiler can determine that at least one of the elements of the tuple, or the tuple itself, needs to be evaluated to compute the result (reduced to root normal form) of the function. This could happen when a guard uses some elements of the tuple. But often this optimisation is not possible, and all these thunks have to be created in the heap.

For example, a function that splits a list in a list with smaller elements and a list with larger (or equal) elements (in Clean):

```
split v [e:l]
    | e < v
        = ([e:small],large)
    | otherwise
        = (small,[e:large])
    where
        (small,large) = split v l
split v [] = ([],[])
```

To compile `(small,large) = split v l` the compiler generates code that allocates three thunks in the heap: a `split v l` thunk, a thunk to select the first element of the tuple, and a thunk that select the second element of the tuple (see lower part of figure 1).

To show how inefficient this is, we use an example. The following function sorts a list using quicksort (and appends the list in the second argument):

```
quick_sort [e:l] t
    = quick_sort small [e: quick_sort large t]
    where
        small = [v \\ v<-l | v<e]
        large = [v \\ v<-l | v>=e]
quick_sort [] l = l
```

The list comprehensions that compute `small` and `large` both traverse the list `l`. We would like to traverse this list once, and compute both the smaller and larger elements during this traversal to improve performance. To do this we uses the `split` function defined above:

```
quick_sort [e:l] t
    = quick_sort small [e: quick_sort large t]
    where
        (small,large) = split e l
quick_sort [] l = l
```

However, this 'optimised' version is not faster but slower. Sorting a list of 10000 integers 50 times now takes 8.05 seconds instead of 4.53 seconds.

The disappointing performance is caused by the laziness resulting from the (lazy) tuple result and the lazy tuple pattern match of `split e l`'s result.

In Clean, a programmer can prevent this problem by making the tuple elements yielded by `split` strict using annotations in the type of `split`, but this changes the semantics of this function, and could in general result in using more memory or even in a non terminating or slower program.

There is yet another problem with the way we have compiled the lazy pattern match on tuples: we have introduced a space leak [2, 8, 4]. If during execution of the program, a function is evaluated that yields a tuple for which there are selector nodes, the selector nodes are not immediately evaluated. Usually the function will be evaluated because one of the selector nodes is evaluated, this selector node is updated with the value of the element, but all the other selector nodes are not evaluated. So all the other selector nodes still contain a reference to the whole tuple, including the element for which the selector node was already updated. So if the updated selector node no longer has references, we have a (temporary) space leak, because there are still references to the element from the other selector nodes.

To prevent such space leaks, the garbage collector could be modified to recognise nodes that select an element of a tuple node [10, 9], and move the references of the selector nodes to the element, update the selector node with the selected element, or update the node with an indirection to the element. This is implemented in the garbage collector of the Clean compiler.

Another way to prevent this space leak, is by inserting functions that update all selector nodes with the selected element or with an indirection to that element when one of the selector nodes is evaluated [7]. This unfortunately increases the memory use by the program, because extra nodes are allocated for these update functions, and is probably slower. The Chalmers Haskell-B compiler can generate these update functions, but does not do this by default.

2 New Optimisation

In this section we will present an optimisation for many recursive functions yielding tuples that improves the execution time of such functions and also fixes the space leak discussed above. This optimisation will be explained using the function `split` as example.

Consider that the function `split` (see Sect. 1) is called and the guard succeeds (because `e < v`), then the code for `split` will create four new nodes in the heap, and update the root node of `split` with a tuple node. This graph is shown in Fig. 1.

Assume that later the just created `split` node is evaluated. This will happen if one of the selector nodes (`SEL 1` or `SEL 2`) that select from `split` are evaluated. We now assume that the guard fails (because not (`e < v`)). Then again four new nodes are created in the heap, and the root node (the `SPLIT` node) is updated with a `TUPLE` node. The resulting graph is shown in Fig. 2. Because the evaluation of the `SPLIT` node was started by one of the selector nodes, this selector node

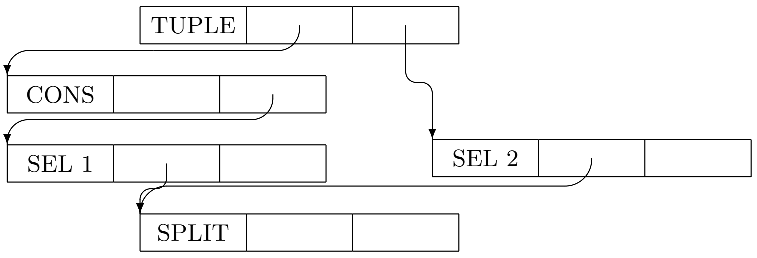

Fig. 1. Graph after calling split

would normally be updated with the selected tuple element (after evaluating this element). In this case this would be with a `SEL 1` node (for the `SEL 1` node) or a `CONS` node (for the `SEL 2` node). But we will not show this. The selector nodes have an unused second argument so that they can be updated with a `CONS` node with two arguments.

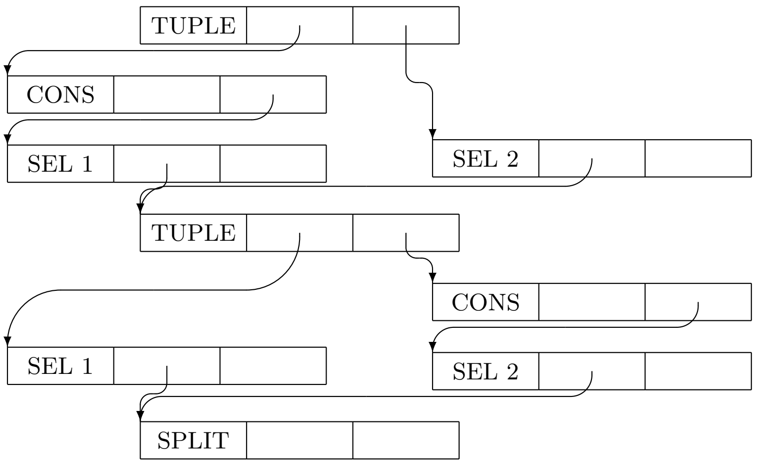

Fig. 2. Graph after calling split twice

If we again assume that the `SPLIT` node will be evaluated and the guard succeeds (because `e < v`), we obtain Fig. 3.

2.1 First Improvement

Note that each time that we evaluated a `SPLIT` node, a new node was created for both the first and second element of the tuple. If the guard succeeds the

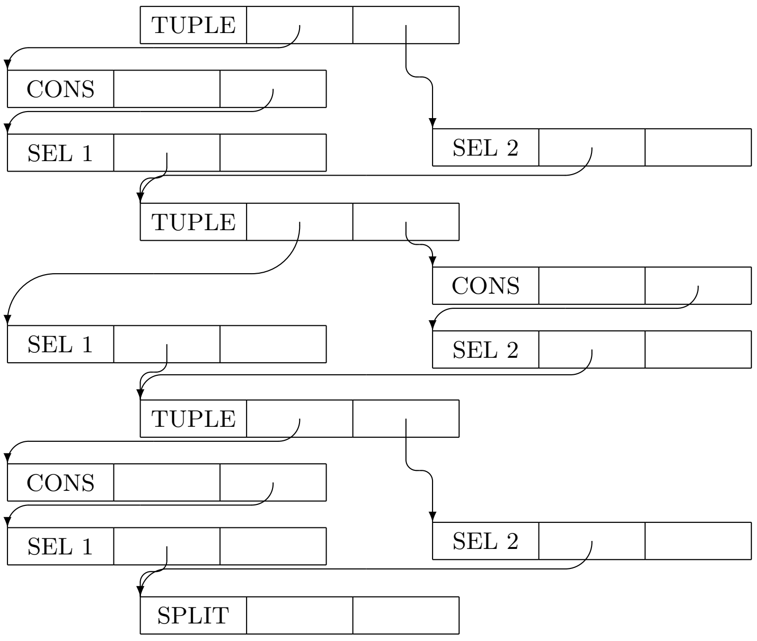

Fig. 3. Graph after calling split 3 times

first element is new `CONS` node and the second element is a new `SEL 2` node. Otherwise, when the guard fails, the first element is a new `SEL 1` node and the second element a new `CONS` node.

Furthermore, we see that all the tuple nodes (except the first one) are always referred to by a `SEL 1` node and a `SEL 2` node only. So for each element of such a `TUPLE` node, there is exactly one corresponding selector node. This has happened, because these `TUPLE` nodes were first created as `SPLIT` nodes, with a selector node for each tuple element pointing to this node, and later these `SPLIT` nodes were updated and became `TUPLE` nodes.

If a selector node is evaluated, it evaluates its argument (the tuple), then selects the appropriate element of the tuple, evaluates this element, and updates the root (selector) node with this result. Note that we can update all the selector nodes of `TUPLE` nodes in Fig. 3, because the tuple is already evaluated, and the element is either an (evaluated) `CONS` node, or another selector node.

Consequently, after evaluating a `SPLIT` node, we can always immediately update all the selector nodes of this `SPLIT` node. A function that implements this, will update the `SEL 1` node with the first element of the tuple, and does not have to create new node for this first element. The same optimisation can be used for the other elements, the `SEL 2` node is updated with the second element of the tuple, and no new node has to be created for the second element of the tuple.

We will call this function `split2`. This function has the same arguments as `split` plus one extra argument for each selector node that has to be updated.

Sparud [7] also creates functions with extra arguments for selectors that have to be updated, but he does not combine this update function with the function that calculates the tuple, and therefore his transformation does not make programs faster or allocate less memory. Nöcker [3] did combine those functions, but he used annotations to make the elements of the tuple and the recursive call of the function strict.

We will now show what happens to our example when we apply this first optimisation. Initially we again start with the result of `split` (if `e < v`). This graph is shown in Fig. 4. Now the `split` function does not create a new `split` thunk (as in Fig. 1), but a `split2` node with two extra arguments, one for each selector node. So the first time `split` is called, it is slower and uses more memory, because a larger node has to be created.

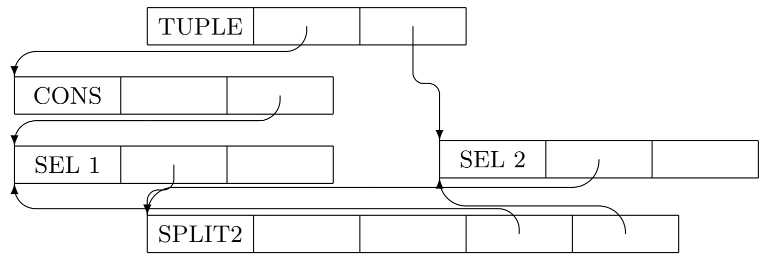

Fig. 4. Graph after calling split

Now assume that the `SPLIT2` node is evaluated (and not (`e < v`)), because one of the selector nodes of this node is evaluated. No new node has to be created for the first element of the tuple (the `SEL 1` node of the new `SPLIT2` node), because we can update the `SEL 1` node of the old `SPLIT2` node (this node has now become a `TUPLE` node) with this `SEL 1` node. Similarly, no new node has to be created for the second element (the `CONS` node), because we can update the `SEL 2` node of the old `SPLIT2` node. So after evaluating `split2` we obtain the graph in Fig. 5.

If we compare the situation before this optimisation (see Fig. 2), we see that we now have to create two new nodes (of which one is larger), instead of four new nodes, because we no longer have to allocate new nodes for the elements of the tuple.

Now again assume that the `SPLIT2` node is evaluated (and `e < v`). Again no new node has to be created for the first element of the tuple (the `CONS` node), because we can update the `SEL 1` node of the old `SPLIT2` (now `TUPLE`) node. Similarly, no new node has to be created for the second element (the `SEL 2`

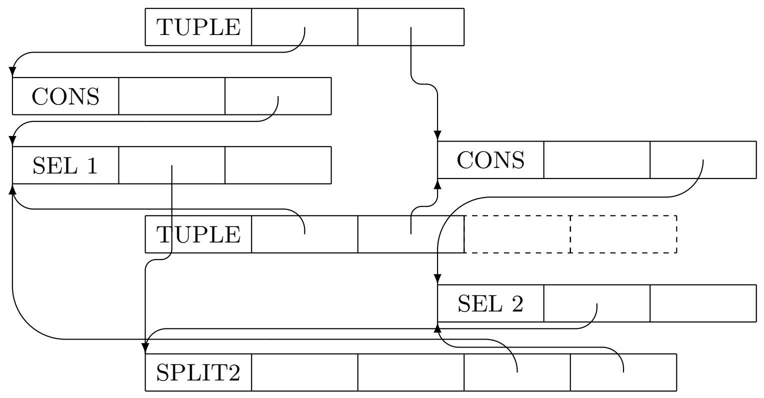

Fig. 5. Graph after calling split and split2

node), because we can update the `SEL 2` node of the old `SPLIT2` (now `TUPLE`) node. So we obtain the graph in Fig. 6.

If we compare the situation after three iterations of `split` before (Fig. 3) and after the optimisation (Fig. 6), we see that we have created four nodes consisting of three words less, but have twice allocated two words that are not use anymore (the dashed boxes) and one node is larger, a `SPLIT2` node instead of a `SPLIT` node.

2.2 Second Improvement

When we look at Fig. 5 and Fig. 6 we see that the `TUPLE` nodes (except the first one) are not used anymore. These nodes were first created as `SPLIT2` nodes which were only used by the selector nodes, but these selector nodes have been updated, and therefore these references have disappeared. Consequently, the `SPLIT2` does not have to be updated with a `TUPLE` node, and becomes garbage after evaluating `split2`. But when evaluating `split2` we also have to create a new `SPLIT2` node for the next iteration of the recursion.

Therefore, we can make the following improvement. Instead of creating a new `SPLIT2` node, we update the old (root) `SPLIT2` node, that would otherwise become garbage, with the new `SPLIT2` node.

We now examine what happens to our example reduction of `split` after applying this optimisation. We start with the same graph (Fig. 4) as after the previous optimisation.

If the `SPLIT2` node is evaluated (and not `(e < v)`), we can updated the `SEL1` and `SEL2` nodes with the first (`SEL 1` node) and second (`CONS` node) element of the tuple, in the same way as after the first optimisation. But now we can also update the `SPLIT2` node with the new `SPLIT2` node, instead of creating a new `SPLIT2` node and updating the `SPLIT2` with a `TUPLE` node (as in Fig. 5). So we have to create only one new node, the `SEL 2` node (Fig. 7).

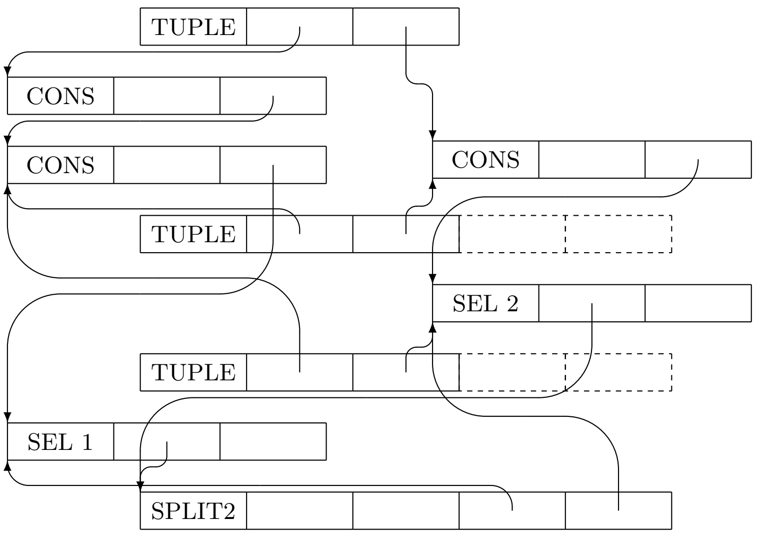

Fig. 6. Graph after calling split and split2 twice

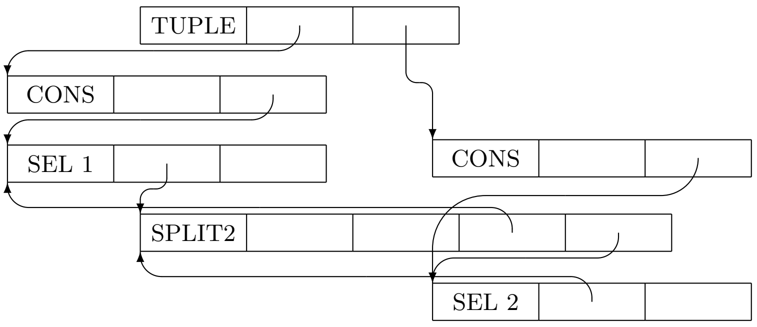

Fig. 7. Graph after calling split and split2

If we compare the situation without optimisations (Fig. 2) with the new graph (Fig. 7), we see that we have allocated only one new node, instead of four new nodes, during the last evaluation of `split`/`split2`. In total we now have 6 nodes instead of 9, and only one node is larger.

If we evaluate the `SPLIT` node again (and `e < v`), we can again update the selector nodes with the first and second elements of the tuple. And again, we can also update the `SPLIT2` node with the new `SPLIT2` node, instead of allocating a new one. So again we have to create only one new node, the `SEL 1` node (Fig. 8).

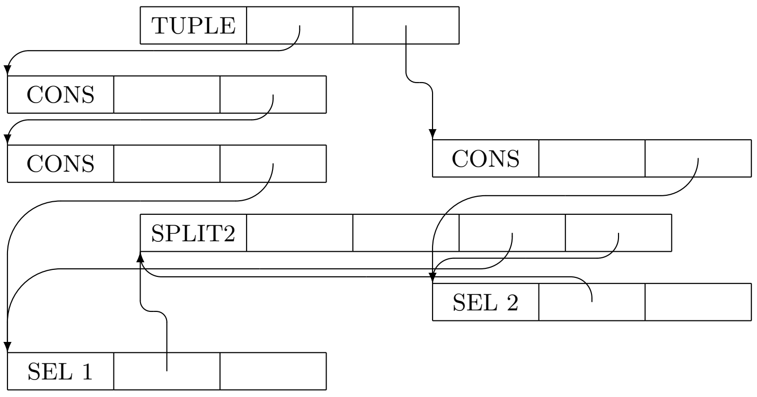

Fig. 8. Graph after calling split and split2 twice

If we again compare the situation without optimisations (Fig. 3) with the new graph (Fig. 8), we see that during each evaluation of `split`/`split2` we allocate only one new node instead of 4. And in total we now have 7 nodes instead of 13, and only one node is larger.

Also we no longer create selector nodes of `TUPLE` nodes, because these are updated, and so we have also fixed the space leak.

2.3 Final Optimised Function

The `split2` function now updates several nodes. Instead of updating all the words of these nodes, we can omit some of these updates, because in some cases the old node already contains the right value.

When a selector node is updated with a new selector node, both the first word that contains the descriptor (e.g. `SEL 1`) and the first argument already contain the right value, and therefore do not have to be updated. For example, if the guard of `split2` succeeds, the second element of the tuple is a `SEL 2` node with the `SPLIT2` node as argument, but the node that is updated is already a `SEL 2` node with the `SPLIT2` node as argument, so nothing has to be done to create this new node. Otherwise, if the guard of split2 fails, the first element of the tuple is a SEL 1 node with the SPLIT2 node as argument, but again the node that is updated is already a SEL 1 node with the SPLIT2 node as argument, so nothing has to be done to create this new node.

Furthermore, the first word of the SPLIT2 node always already contains a split2 descriptor. And the arguments of such a node that contain the pointers to the selector nodes of the node, often do not have to be updated as well. For example, if the guard of split2 succeeds, the third argument already contains the right pointer to the SEL 1 node, and when the guard fails, the fourth argument already contains the right pointer to the SEL 2 node.

It is also possible that the other arguments of SPLIT2 already contain the right values, but our compiler currently does not optimise this case.

Finally, we have to use different selector functions for selecting tuple elements from functions that update the selector like split2. Normally the code for a selector evaluates the first argument of the selector node, then selects the tuple element from the tuple, evaluates this element and updates the selector node with the result. But the selector code of optimised functions like split2, also starts with evaluating the first argument, but now this evaluation will cause this selector node to be overwritten with an element of the tuple. So we continue by checking if the selector node is already evaluated, if not we jump to the evaluation code of this thunk, otherwise we return to the caller of the selector code.

Note that this code does not depend on the element of the tuple that is to be selected, so when can use this same code for all selectors. And thus all selector nodes (SEL 1 and SEL 2) of SPLIT2 can be the same. In the code below we call these nodes select nodes.

So, after applying all the optimisations we obtain the following optimised version of split (in Clean like pseudo-code with explicit UPDATE_ARGUMENTs and UPDATE_NODEs):

```
split v [e:l]
    | e < v
        = let   t = split2 v l small large
                small = select t
                large = select t
          in    ([e:small],large)
    | otherwise
        = let   t = split2 v l small large
                small = select t
                large = select t
          in    (small,[e:large])
split v [] = ([],[])

r=:split2 v [e:l] sel1 sel2
    | e < v
        = let   small = select r
          in    UPDATE_NODE sel1 [e:small]
```

```
                UPDATE_ARGUMENTS r (ARG1 = v) (ARG2 = l) (ARG3 = small)
    | otherwise
        = let   large = select r
          in    UPDATE_NODE sel2 [e:large]
                UPDATE_ARGUMENTS r (ARG1 = v) (ARG2 = l) (ARG4 = large)
r=:split2 v [] sel1 sel2
    = UPDATE_NODE sel1 []
      UPDATE_NODE sel2 []
```

The recursive call of `split`, that was defined using a where clause in the original `split` function, is moved to both guarded alternatives (and renamed to `split2`) by the compiler in both functions. This enables the compiler to generated better code for `split2`.

2.4 Updating with Black Holes

There is still one problem to be solved that we have not yet mentioned. When the evaluation of a thunk begins, the descriptor of the thunk is immediately overwritten with a black hole descriptor. This is necessary to be able to detect cycles in the spine of the reduction (i.e. a function tries to use its own result to compute the result). If such an error occurs, a black hole node will be evaluated and an error message is printed. Furthermore, this update prevents a space leak, because it removes the references to the arguments of the function from this node.

While explaining our optimisation we have assumed that a thunk is not changed when it is evaluated, so that it can be reused to build new nodes. Black holing would cause some problems, so our current implementation does not update the `select` nodes and recursive function thunks introduced by the optimisation with black hole nodes.

To prevent the problems mentioned above, we intend to change the compiler so that it updates the descriptor of the function thunks with a special black hole descriptor when the evaluation of the thunk begins. This special black hole will also cause an error message to be printed when it is evaluated. To prevent the space leak, the garbage collector will treat the arguments that contain the pointers to the selector nodes as normal pointers, but it will treat the other arguments as non-pointers (e.g. integers).

In our implementation thunks may contain both pointers and non-pointers (e.g. unboxed integers). Pointers are stored at the beginning of thunks, non pointers at the end. So, if we would store the normal arguments of the function thunks at the end and the arguments that contain the pointers to the selector nodes at the beginning of the thunk, the garbage collector will do the right thing if it thinks that the black hole node is a thunk with number of selector nodes pointers, and number of normal function arguments non-pointers.

We do not have to update the `select` nodes with a black hole node. These nodes only contain a pointer to the function thunk, which will immediately be updated with a special black hole node, so black holing this `select` node is not necessary to prevent a space leak. It is also not required in order to detect cycle in spine errors, because if the select node is on a cycle, so will the function thunk that it selects the element from, and we can already detect cycle in spine errors for these function thunks. We will just detect the error a little bit later.

So to implement this, we just have to generate a few extra instructions, to update the descriptor of the thunk of the recursive function (that is generated by the optimisation) when its evaluation starts with a special black hole node, and to restore the original descriptor when the thunk is updated for the next iteration of the recursion. This will make programs only slightly slower.

2.5 Detecting Functions that can be Optimised

We can apply this optimisation for a function f if:

1. f contains a recursive call of f in a lazy context, and
2. The result of this call of f is a tuple t, and
3. All references to tuple t are by selectors in f, and
4. For all elements of the tuple t there is at least one selector, and
5. For each alternative of f:
    (a) The result is a tuple denotation, or
    (b) The result is a call of f.

We implemented this in the compiler as follows. For each recursive call to the current function f in a lazy context, we use a counter that is initialised with zero and incremented when we encounter a selector in a lazy context of this call of f. If such a counter becomes equal to the total number of references to the call of f, we examine whether all the possible results of f are tuple denotations or calls of f. If this is the case we have found a call and function that can be optimised, therefore the function f and the variable, to which the result of the call of f is assigned, are marked. This information is then used during code generation. Whether a new selector node needs to be allocated for the next recursive call or not, is determined just before code generation by examining the results of f that are tuple denotations. If an element of such a tuple is a selection of a call of f that is optimised, and it selects the right element, no new selector node has to be allocated, otherwise it is necessary to create a new node. This information is stored in a bit vector with one bit for each element of the tuple, and used to determine how to generate code for the optimised recursive calls and its selections, and for the elements of tuples yielded by f.

This optimisation can be used in more situations, for example when a record or a single constructor algebraic data type is used instead of a tuple, and for mutually recursive functions with the same number of arguments. The calls from other functions to an optimised function can in some cases be optimised, by directly calling the optimised version of the function instead. Then sometimes the non-optimised version of the function does not even have to be generated.

2.6 Advantages and Disadvantages

The advantages of this optimisation are:

1. Faster execution, because:
   (a) Fewer instructions are executed (except for the first call of the function).
   (b) The code for many selectors is never executed, because these selector nodes are updated before the code for this selector is executed.
   (c) If a tuple element of the result of the function is returned in the tuple yielded by the recursive call at the same position in the tuple, no code has to be generated for this selector (except for the first call).
   (d) Fewer cache misses, because less memory is allocated for all calls of f after the first one, usually (1 + number of function arguments) + 3 * tuple size words less.
   (e) Lower garbage collection costs, because less memory is allocated.
2. No space leak because all selector nodes are updated.

The disadvantages of this optimisation are:

1. More code is generated, because two versions are generated of each optimised function.
2. Higher memory use because the thunk for the recursive function call is larger, because it also contains pointers to the selector nodes. The number of extra words is the number of elements of the tuple.
3. Allocates more memory if the recursive function is not called recursively, but is called just once from another function, because of the larger thunk for the recursive call (that is not evaluated).

3 Measurements

We have implemented the optimisation by extending our compiler for the lazy functional language Clean version 1.3 [6, 1]. In this section we measure the efficiency of this new optimisation by comparing the execution speed with and without this optimisation of some programs that use functions that can be optimised. We used two different computers, a PowerPC G3 with a 266 MHz PPC 750 processor and a PC with a 350 MHz AMD K6-2 processor. The two smallest programs were also ported to Haskell and run on the PC using the Glasgow Haskell compiler, to show that our compiler generates state of the art code for these programs even without this new optimisation.

3.1 Quicksort

The quicksort program we used creates a list of 10000 random integers, sorts this list using quicksort and computes the sum. This is repeated 50 times. We measured quicksort using the `split` function and quicksort using two separate filter functions using list comprehensions. The source code of both these functions can be found in Sect. 1.

Table 1 contains the results, the following versions were run: "Split" is a quicksort using the `split` function without the new tuple optimisation, "Split optimised" is the same program with this optimisation, "Split *" is the same program, except that it sorts a unique list and uses an optimisation that uses this uniqueness type information, but without the new tuple optimisation, "Split * optimised" again sorts a unique list and uses both optimisations, finally "Comprehensions" is the quicksort that uses two list comprehensions instead of a `split` function.

Column 3 lists the execution time not including garbage collection (GC) time in seconds, column 4 the time in seconds spend collecting garbage, column 5 the total execution time in seconds, and column 6 the speedup of the tuple optimisation, calculated by dividing the time for "Split" by the time for "Split optimised" and the same for "Split *" and "Split * optimised".

All the Clean programs were run using 2 megabytes of heap and a next heap size factor of 20, except the fast Fourier programs, these were run with a heap of 6 megabytes.

Table 1. Quicksort in Clean

	Processor	Time w/o GC(s)	GC time(s)	Time(s)	Speedup
Split	266 MHz PPC 750	5.73	2.30	8.05	
	350 MHz K6-2	5.86	2.77	8.64	
Split optimised	266 MHz PPC 750	3.16	0.36	3.53	2.28
	350 MHz K6-2	3.11	0.50	3.61	2.39
Split *	266 MHz PPC 750	5.36	1.76	7.13	
	350 MHz K6-2	5.40	2.43	7.83	
Split * optimised	266 MHz PPC 750	2.55	0.05	2.60	2.74
	350 MHz K6-2	2.57	0.27	2.85	2.75
Comprehensions	266 MHz PPC 750	3.31	1.21	4.53	
	350 MHz K6-2	3.18	1.60	4.79	

Table 2 lists the execution times for the same program in Haskell compiled with the ghc compiler with -O2.

Table 2. Quicksort in Haskell on a PC with a 350MHz AMD K6-2

	Compiler	OS	Time w/o GC(s)	GC time(s)	Time(s)
Split	Ghc4.03 -O2	Windows 98			23.29
Split	Ghc4.04 -O2	Linux	9.93	4.39	14.33
Comprehensions	Ghc4.03 -O2	Windows 98			20.21
Comprehensions	Ghc4.04 -O2	Linux	12.46	4.03	16.49

3.2 Takedrop

The takedrop program creates a list of the integers from 1 to 2000, and then repeats the following 2000 times: use the `takedrop` function to split the list in the first 2000 elements and the rest of the list, and then concatenate these two lists again using an append function.

The `takedrop` function (often also called splitAt) can be optimised by our transformation.

```
takedrop :: Int *[.a] -> ([.a],[.a])
takedrop 0 xs = ([],xs)
takedrop _ [] = ([],[])
takedrop n [x:xs]
    #! n1=n-1
    # (xs‘,xs‘‘) = takedrop n1 xs
    = ([x:xs‘],xs‘‘)
```

(`#` in Clean has the same semantics as let .. in, but the defined values have a different scope)

The results are in Table 3, just like for quicksort with `split`, we measured versions without tuple and uniqueness optimisations, with tuple optimisation only ("Optimised"), with uniqueness optimisation only using a unique list ("*"), and with both optimisations ("* optimised").

Table 3. Takedrop in Clean

	Processor	Time w/o GC(s)	GC time(s)	Time(s)	Speedup
	266 MHz PPC 750	4.98	3.11	8.10	
	350 MHz K6-2	7.49	3.06	10.55	
Optimised	266 MHz PPC 750	4.11	0.40	4.51	1.80
	350 MHz K6-2	6.69	0.50	7.19	1.47
*	266 MHz PPC 750	5.53	2.26	7.81	
	350 MHz K6-2	8.83	2.34	11.18	
* optimised	266 MHz PPC 750	2.81	0.11	2.95	2.65
	350 MHz K6-2	4.68	0.10	4.79	2.33

Table 4 lists the execution times for this program in Haskell.

Table 4. Takedrop in Haskell on a PC with a 350MHz AMD K6-2

Compiler	OS	Time w/o GC(s)	GC time(s)	Time(s)
Ghc 4.03 -O2	Windows 98			29.38
Ghc 4.04 -O2	Linux	10.64	8.12	18.76

3.3 LZW compression

The LZW program compresses a 77k text file 10 times uses LZW compression. This program is similar to the Haskell program in [5]. There are two functions in this program that can be optimised with the tuple optimisation, the `code_string_` function that computes the next code, and the `file_to_list` function that makes a list of the characters in the file. No strictness annotations were used in this program.

```
code_string_ (Pt k v t l r) next_code c input2 input old_code
  | c<k
    # (input_l,nl,l‘) = code_string_ l next_code c input2 input old_code
    = (input_l,nl,Pt k v t l‘ r)
  | c>k
    # (input_r,nr,r‘) = code_string_ r next_code c input2 input old_code
    = (input_r,nr,Pt k v t l r‘)
  | c==k
    # (input‘,n,t‘) = code_string input2 t next_code v
    = (input‘,n,Pt k v t‘ l r)
code_string_ PtNil next_code c input2 input old_code
  | next_code>=max_entries
     = (input, old_code, PtNil)
  | otherwise
     = (input, old_code, Pt c next_code PtNil PtNil PtNil)

file_to_list :: *File -> ([Char],*File);
file_to_list input_file
  # (s,c,input_file) = freadc input_file
  | s          # (l,input_file) = file_to_list input_file
               = ([c : l],input_file)
  | otherwise  = ([],input_file)
```

The results are in Table 5. The versions using uniqueness optimisation use a unique tree with `PT` and `PtNil` constructors.

Table 5. LZW compression in Clean

	Processor	Time w/o GC(s)	GC time(s)	Time(s)	Speedup
	266 MHz PPC 750	6.06	2.30	8.38	
	350 MHz K6-2	5.95	2.76	8.71	
Optimised	266 MHz PPC 750	3.96	1.03	5.01	1.67
	350 MHz K6-2	3.88	1.16	5.04	1.73
*	266 MHz PPC 750	5.43	2.03	7.48	
	350 MHz K6-2	5.56	2.28	7.84	
* optimised	266 MHz PPC 750	3.95	0.90	4.86	1.54
	350 MHz K6-2	3.70	1.09	4.79	1.64

3.4 Fast Fourier Transform

The Fast Fourier Program does 10 FFT's of a list with 16384 complex numbers. A record with two strict Real's was used to store the complex numbers.

Two functions in this program can be optimised using our transformation, the `merge` function that does most of the computations, and a `split` function that splits the list in two lists with the elements at even and odd positions.

```
merge :: *[Complex] *[Complex] Int Int -> (.[Complex],.[Complex])
merge [] [] i length = ([],[])
merge [e:re] [o:ro] i length
    = let! ui = e+prod ; umi= e-prod
      in   ([ui : urest],[umi : umrest]);
    where
        (urest,umrest)  = merge re ro (inc i) length
        prod = {re=cos z,im=sin z} * o
        z = toReal i*pi_2 / toReal length

split :: *[Complex] -> (.[Complex],.[Complex])
split [a,b : rest]
    # (even, odd) = split rest
    = ([a : even],[b : odd])
split [] = ([],[])
```

The results are in Table 6. The versions using uniqueness optimisation uses unique lists.

Table 6. Fast Fourier Transform in Clean

	Processor	Time w/o GC(s)	GC time(s)	Time(s)	Speedup
	266 Mhz PPC 750	5.50	1.68	7.18	
	350 Mhz K6-2	5.05	1.96	7.02	
Optimised	266 Mhz PPC 750	4.55	1.00	5.56	1.29
	350 Mhz K6-2	4.02	1.15	5.18	1.36
*	266 Mhz PPC 750	5.01	1.16	6.20	
	350 Mhz K6-2	4.42	1.34	5.76	
* optimised	266 Mhz PPC 750	4.13	0.50	4.63	1.34
	350 Mhz K6-2	3.68	0.71	4.39	1.31

4 Conclusion

This new transformation can be applied for many recursive functions yielding multiple results in a tuple in lazy functional languages. It can make programs that frequently use such functions much faster, sometimes more than twice as fast. Because it also reduces the amount of memory that is allocated in the heap, the garbage collector has to be run far less often, and so garbage collection costs are reduced considerably. For some of our test programs by more than a factor of six. Furthermore, it also fixes the space leak that is caused when selector functions are used to select the elements of tuples. But of course, only for the selector functions used in the recursive functions that are optimised.

References

1. Groningen, J.H.G. van, Nöcker, E.G.J.M.H., Smetsers, J.E.W.: Efficient Heap Management in the Concrete ABC Machine. Proceedings of the Third International Workshop on the Implementation of Functional Languages on Parallel Architectures. Technical Report Series CSTR 91-07. University of Southampton. U.K. 1991.
2. Hughes, J.: The design and implementation of programming languages. PhD thesis. Oxford University. July 1983. Programming Research Group, technical monograph PRG-40.
3. Nöcker, E.G.J.M.H.: Efficient Parallel Functional Programming - Some Case Studies -. Proceedings of the Fifth International Workshop on Implementation of Functional Languages. Nijmegen. The Netherlands. September 1993. Technical Report 93-21. 51-68.
4. Peyton Jones, S.L.: The implementation of Functional Programming Languages. Prentice-Hall 1987.
5. Sanders, P., Runciman, C.: LZW Text Compression in Haskell. Glasgow Workshop on Functional Programming 1992. Ayr. Scotland. 215-226.
6. Smetsers, J.E.W., Nöcker, E.G.J.M.H., Groningen, J.H.G. van, Plasmeijer, M.J.: Generating Efficient Code for Lazy Functional Languages. FPCA'91. Cambridge. MA. USA. Springer Verlag. LNCS 523. 1991. 592-617.
7. Sparud, J.: Fixing Some Space Leaks without a Garbage Collector. FPCA'93. June 1993. Copenhagen,Denmark. ACM press. 117-122.
8. Stoye, W.: The implementation of functional languages using custom hardware. PhD thesis. Cambridge University. December 1985. Computing Laboratory, technical report 81.
9. Turner, D.: A proposal concerning the dragging problem. October 1985. Burroughs ARC internal report.
10. Wadler, P.: Fixing some space leaks with a garbage collector. Software Practice and Experience. 18(9):595-608. September 1987.