# Internship report : Tail recursion modulo constructor

Émile Trotignon

2022

## Abstract

During my 4.5 month intership at OCamlPro under the guidance of Pierre Chambart and Vincent Laviron, I studied the TRMC transformation, that is an extension of the famous tail-recursion optimisation. I implemented regular TRMC, and then two extended versions, and explored the limits of these extensions.

The software produced during my internship can be found here : `https://github.com/EmileTrotignon/mlambda`.

## 1 Tail-recness

Lets look at the following function :

```
let rec map f li =
  match li with
  | [] -> []
  | ele :: li ->
    let ele' = f ele in
    let li' = map f li in
    ele' :: li'
```

Evaluating a call to `map f li` will allocate memory proportionnal to the size of `li`, in addition to the new list, because when you execute the recursive call, you need to remember :

- The code pointer to `ele' :: li'` to be able to resume the execution after the return.
- The value of `ele'` to be able to evaluate

`ele' :: li`.
`List.map` can be made tail-recursive :

```
let rec map_aux f li acc =
  match li with
  | [] -> acc
  | ele :: li ->
```

```
    let ele' = f ele in
    map_aux f li (ele' :: acc)

let map f li =
  let li = map_aux f li in
  List.rev li
```

Here, in a call to `map_aux`, once `map_aux` is called there is nothing to do except to write the returned value in the return adress/register of the current call and jump to the caller of the current call. Therefore we can use no memory, just give the current return information to the new call of `map_aux`. Doing this is called the tail-recursive (tail-rec) optimisation, because the recursive call is in tail position : it is directly returned, nothing is done with its result. The tail-rec optimisation has been well known for a long time, and is especially critical, not only for efficiency reasons but also for correctness.

Most systems separate the memory allocated to a process between "stack" and "heap", and in most languages, function calls consume stack memory, which is often in short supply compared to heap memory that is used to store, for instance, lists.

This means that, if you use the first version of the `map` function on a large list, it may use up all stack memory and the whole programm will fail, even though there was ample space for the return value and the input list on the heap.

On a usual linux system with default configuration, using Ocaml 4.xx, you may run into this issue.

As shown, since the second version of map is tail-rec, it will not have this issue, and will work fine on large inputs. However it is also slower because you need to reverse the list at the end, that is not ideal.

# 2 Tail rec modulo constructor

Lets look again at the first version of `map` :

```
let rec map f li =
  match li with
  | [] -> []
  | ele :: li ->
    let ele' = f ele in
    let li' = map f li in
    ele' :: li'
```

It is not tail-rec because instead of returning the recursive call, we bind to `li'` and use it later. However, you can see that the uses of `li'` are very light : we only use it to initialize a constructor. An option to make this function tail-rec would therefore be to allocate the constructor to `ele' :: x` where `x` is a dummy

value (0 for instance), and then give the adress of the hole in the constructor to the recursive call.

This would also mean that we have to write our result to an adress instead of returning it, since our caller would have made the same assumptions. We call this 'destination passing style', and it would look like the following for map :

```
let rec map_dps dst f li =
  match li with
  | [] ->
    dst <- []
  | ele :: li ->
    let ele' = f ele in
    let block = ele' :: _ in
    map_dps (block + 2) f li ;
    dst <- block
```

We pass `block + 2` to the rec call because a block of a list cell is of size 3 : Index 0 contains the representation of tag `::`, index 1 contains `ele'` and index 2 is uninitialised (represented by `_`).

A mild issue with the program is that the OCaml runtime does not allow for interior pointers such as `block + 2` so we have to pass a pointer to the start of the block and an index :

```
let rec map_dps dst i f li =
  match li with
  | [] ->
    dst.i <- []
  | ele :: li ->
    let ele' = f ele in
    let block = ele' :: _ in
    map_dps block 2 f li ;
    dst.i <- block
```

This is also not tailrec, but it does not change the semantic of the function to swap the last two lines :

```
let rec map_dps dst i f li =
  match li with
  | [] ->
    dst.i <- []
  | ele :: li ->
    let ele' = f ele in
    let block = ele' :: _ in
    dst.i <- block ;
    map_dps block 2 f li
```

Here, `dst.i` is written with an incomplete value, but it does not mater because the value will become completed by the time that the function return :

- During the execution, the number of uninitialized fields is always one, when you create a new one with `let block = ele' :: _ in`, an other one is filled with `dst.i <- block`.
- Eventually, `dst.i <- []` is executed, and the last uninitialized field is filled.

We also need a wrapper to be able to use it as if it was a normal function :

```
let map f li =
  let block = alloc 1 in
  map_dps block 0 f li ;
  block.0
```

This costs an extra allocation, we can avoid it this way :

```
let map f li =
  match li with
  | [] -> []
  | x :: li ->
    let y = f x in
    let block = y :: _ in
    map_dps block 2 f li ;
    block
```

There still is an issue with `map_dps` : In OCaml, the `list` type is immutable, you just can't write to a list block, nor do you have a default dummy value to put for _ that would work with any type. You could maybe use unsafe features of OCaml such as the `Obj` module, but thats generally a bad idea. However, we can mutate values and ignore typing this way in some intermediate languages of the OCaml compiler. That means that if we have a way to automate the transformation, it can be included as an optimisation in the OCaml compiler. In fact, this is already done starting with OCaml 4.14 : `https://github.com/ocaml/ocaml/pull/9760`.

The researchers behind this PR also published their findings : [BCS21]. My work is a follow-up to this paper and explores ways to expand this optimisation.

# 3 Framework

For easier experimenting, I did not work with the OCaml compiler, as it is far too complex a codebase for my uses. I made up a small langage called MLambda.

Here is the type of its AST :

```
type ident = string

type cons = string
```

```
type primitive = PrString of string | PrInt of int | PrBool of bool

and pattern =
  | PAny
  | PPrim of primitive
  | PVar of ident
  | PCons of {cons: cons option; payload: pattern list}

and expr =
  | EVar of ident
  | EFunc of {args: ident list; body: expr}
  | EApply of {func: expr; args: expr list}
  | EPrim of primitive
  | EUnit
  | ECons of {cons: cons option; payload: expr list}
  | EMatch of {arg: expr; branches: (pattern * expr) list}
  | EPrimFunc of string * (value list -> value)

and value =
  | VInt of int
  | VBool of bool
  | VUnit
  | VString of string
  | VCons of cons
  | VArray of value array
  | VFunc of {mutable env: value Env.t; args: ident list; body: expr}
  | VPrFunc of string * (value list -> value)

type struct_item =
  | Binding of {name: ident; is_rec: bool; body: expr}
  | MutualRecBindings of (ident * expr) list

type program = struct_item list
```

You may notice that the `expr` node is very simple ; there are very few constructs.

There is for instance no `let _ = _ in`, `if _ then _ else` or `_ ; _` construct. In fact, these three construct are all implemented as special cases of the `match _ with _` node : The parser is able to parse these constructs, and generate the correct `match` expression, and I also provide smart constructors to generate them. The printer prints every `match` that look like an `if` or a `let` as an `if` or a `let`, even if it was constructed as a regular `match`.

As for the semantics of the langage, the most important type is `value`. You may notice that the only ways to agregate multiple values the `VArray` and the `VFunc` node. The `env` field of `VFunc` is not intended to be manipulated in any other way than using them to evaluate the body of the function, it is mutable solely to be

able to evaluate recursive functions.

However, the choice of a `VArray` node is because we want every value that would be allocated on the heap in OCaml to be mutable. To make arrays usable we provide three built-in function : `alloc`, `write` and `proj`. The parser interprets `arr.i` as `proj arr i` and `arr.i <- v` as `write arr i v`.

A few other built-in functions are provided, such polymorphic equality and basic arithmetic.

Such built-in function are provided through the `EPrimFunc` node of the `expr` type.

In all other aspects, the semantics are analogous to OCaml's. For more detail consult the `eval.ml` file.

There is also a test-suite for testing programs and transformations. It is built in such a way that you can write an MLambda program and the expected result once, and test that different transformations of that program produce the correct result.

# 4 Regular TRMC

The algorithm for simple TRMC optimisation is quite simple.

Once you are in the context of a recursive function `f`, you go through its body. You assume that you have an argument `dst` and `i`, and that there is a function `f_dps` (that we are currently building, but thats how recursion works).

The destination and the index constitute a state of the transformation, and they start as `dst` and `i`.

For most expr cases, you simply fail. The only cases that do not trigger a failure are :

- Constructors
- Recursive calls
- Matches

## 4.1 Constructors

When you find a constructor you have to explore its branches, to find out which branch has the recursive call. For instance :

```
MyCons (e1, e2, e3)
```

Where `e2` contains a recursive call in TMC position is gonna be compiled as :

```
let block_n = MyCons (e1, (), e3)
dst.i <- block_n ;
e2'
```

Where `e2'` is the transformation of `e2`, knowing that the current destination is `block_n` and the current index is `2`.

## 4.2 Recursive calls

When you have a recursive call :

```
f a1 a2 a3
```

You simply compile it into :

```
f_dps dst i a2 a2 a3
```

## 4.3 Matches

When you have a match :

```
match e with
| p1 -> e1
| p2 -> e2
...
| pn -> en
```

If the number of branches is 1, it may be a `let`, or an `if` if there are two branches.

To transform this, you first transform the expressions `e1, e2, _ , en` into `e1', e2', _ , en'`. Then, you check if you had at least one success. If you did not, the whole transformation fails. If you did, then the transformation will succeed, but you need to handle the branches that failed. If the branch `k` failed, then `ek'` is `dst.index <- ek`. If it didnt, you already have `ek'`.

Then the transformed expression is as follows :

```
match e with
| p1 -> e1'
| p2 -> e2'
...
| pn -> en'
```

You may notice that `e` is not inspected at all. If we are in the case of a `let` :

```
match e with
| p1 -> e1
```

This is equivalent to :

```
let p1 = e in e1
```

Then the procedure described above is just trying to transform `e1` into `e1'`, generating `let p1 = e in e1'`, or failing if the tranformation fails.

All of this allows the following transformation :

```
let rec map_double f li =
  match li with
  | [] -> []
  | ele :: li ->
    f ele :: f ele :: map_double f li
```

into :

```
let rec map_double f li =
    let dst = alloc 1 in
    print "entry dst =" ;
    print dst ;
    map_double_dps dst 0 f li ;
    dst.0

and map_double_dps dst i f li =
    print "dst =" ;
    print dst ;
    match li with
    | [] -> dst.i <- []
    | ele :: li ->
      let block_2 = f ele :: () in
      dst.i <- block_2 ;
      let block_3 = f ele :: () in
      block_2.2 <- block_3 ;
      map_double_dps block_3 2 f li
```

### 4.4 Correctness

There is a straightfoward proof of the correctness the above.

You do it by recursion on the size of the expressions. The recursion hypothesis is that for any transformable expression `e` of size `n` and destination `dst.i`, then the transformation gives an expression `e'` such that evaluating `e'` writes the result of evaluating `e` in `dst.i`.

The proof is straightfoward. In some cases the transformation fails, but these cases are outside the scope of our claims.

Effects are a concern. However, you can notice that the transformation leaves destructed expressions intact, and they are the only explicit effect-ordering construction in the langage. The constructor fields are interpreted in a certain order, but this is not something that would be part of the specification of the language, and it is not specified in OCaml.

## 5 Small extensions

The above will not allow to transform the below :

```
let rec map_double_let f li =
  match li with
```

```
  | [] -> []
  | ele :: li ->
    let li = map_double f li in
    f ele :: f ele :: li
```

because the recursive call is not in the right position and we will never detect it. To be able to detect it, we could modify the transformation to track every binding of a recursive call, and check that it is only used once. This approach was used at first.

But it is way simpler to just inline every binding that is used only once in a first pass, before the TRMC transformation. To do this in practice, we would need to try and be careful to not change the order of effects, this means not inlining lets with side effects past other side effects. Here, a side effect would a call to function that is not known for its lack of side effects.

Anyway, after inlining, `map_double_let` becomes `map_double` and we are able to transform it.

# 6 TRMC with multiple return values

Lets look at the following function :

```
let rec partition_map f li =
  match li with
  | [] -> ([], [])
  | x :: li -> (
    let e = f x in
    let (left, right) = partition_map f li in
    match e with
    | Left (v) -> (v :: left, right)
    | Right (v) -> (left, v :: right) )
```

At a first glance, it is neither tailrec not TRMC : the recursive call is destructed by a pattern, which means that we actually have to get the value returned by the recursive call to access its fields, before building the value we return. However, this function can also be seen as a function returning two value, and each of this value is only used to initalize a constructor field.

This means that on principle, we do not need to know the values of `left` and `right` before executing the end of the function.

We can in fact transform it in DPS form by using two destinations :

```
let rec partition_map f li =
  let block = alloc 2 in
  partition_map_dps block 0 block 1 ;
  (block.0; block.1)
and partition_map_dps dst0 i0 dst1 i1 f li =
  match li with
  | [] ->
```

```
    dst0.i0 <- [] ;
    dst1.i1 <- []
  | ::(x, li) ->
    let e = f x in
    match e with
    | Left v ->
      let block_1 = v :: () in
      dst0.i0 <- block_1 ;
      partition_map_dps block_1 2 dst1 i1 f li
    | Right v ->
      let block_2 = v :: () in
      dst1.i1 <- block_2 ;
      partition_map_dps dst0 i0 block_2 2 f li
```

Doing this automatically will require weirder flow of information than the simpler version above. The main reason is that to discover that this is indeed a function that returns two values, we cannot rely on typing (there is none in flambda, but that's not the issue here), but we need to be sure that each recursive call is destroyed into a pair.

When we have a branch that has a recursive call and then return something, we need to do two things :

- Write the ingoing destinations with the correct (but incomplete) blocks
- Perform the recursive call with the correct destinations

To be able to write the correct destination, we should be able to match fields of the output constructor to destinations.

To be able to perform the recursive call with the correct destinations we should be able to remember adresses corresponding to the holes. These holes should be where variables bound by the destruction of the original recursive call were. We call such variables "recursive variables".

The choice we made to satisfy these conditions is to perform the transformation in four phases :

- "Before destructor" where we look for the recursive call and its destruction pattern.
- "Before constructor" where we look for a constructor.
- "During constructor" where we match the constructor to the destructor, and look for the expressions corresponding to each field.
- "After constructor" where we know to which destination the current destination should write its result, but we want to find which recursive variable is used.

This transformation is called TRMC-MUR.

## 6.1 After constructor

This step is performed by the function `expr_dps_after_construct`. It is the step that look the closest to the regular TRMC transformation.

It takes as input :

- The block `b` and the index `i` we are supposed to write to. These are both expressions.
- The set `rec_vars` of recursive variables.
- The expression `e` we are transforming

It returns :

- If unsuccessful : `None`
- If successful : `Some(v, e')`

Where `e'` writes the result of `e` in `b.i`, except for a hole where the recursive variable `v` was mentionned, and then return the pair constituting the destination to that hole.

It does this by matching on `e`. Three cases are possible successes:

- A variable that is a member of `rec_vars`.
- A match.
- A constructor.

### 6.1.1 Variables

When encountering a match on a variable `v` that is a member of `rec_vars` we return the pair `(v, (b, i))` (`(b, i)` being an MLambda expression, not an OCaml one).

### 6.1.2 Matches

When encountering a match :

```
match em with
| p0 -> e0
...
|pn -> en
```

First we make sure that `em` does not mention a variable from `rec_vars`.

Then, for each branch `| pn -> en`, we call `after_constructor` on `eb`, while removing the variable from `pn` from `rec_vars`. We get `(vn, en')` this way. Every branch must be successful. Then if forall `n n' , vn = vn'`, we return a pair of containing `v0` and the following code :

```
match em with
| p0 -> e0'
...
| pn -> en'
```

### 6.1.3 Constructors

When encountering a constructor `Cons (e1, e2, ..., en)` we do the following :

We pick the name `bl` of the block we will allocate.

We try to find an expression `ek` in `e1, e2, ..., en` by, for each expression `ej` :

- Checking that all the other expression do not mention a variable from `rec_vars`.
- performing a recursive call with `b = bl`, `i = j`, `rec_vars` staying the same, and `e=ej`

We get the pair `v, ej'` from the first successful recursive call.

Then we return the variable `v` along with the following expression :

```
let bl = Cons (e1, ..., e(j-1), _, e(j+1), ..., en) in
b.i <- bl ;
ej'
```

## 6.2 During constructor

This step is performed by the function `expr_dps_construct`. It is fairly simple, and would not exist if we did not allow patterns with nested constructors.

It takes as arguments :

- The set `rec_vars` of variable introduced by destroying the recursive call.
- The pattern `p` that destroyed the recursive call.
- The expression `e` we are transforming, that should be a constructor.

It returns a list of pairs of variable and expression. The expressions :

- Write the correct constructor in the correct destination.
- Return a pair of a block and an indice that point to a hole that was a reference to a recursive variable in the old expression.

These expressions are paired with the recursive variable that used to be in place of their holes.

It works by matching on the pair `(p, e)`. If `p` is not a variable pattern or a constructor pattern, then the function fail.

If `p` is a variable pattern, then `e` can be anything, and we call `expr_dps_after_construct` and return the singleton list containing its return. If the call to `expr_dps_after_construct`, then we fail as well.

If `p` is a constructor pattern, then `e` must be a constructor expression of same length, on pain of failure. Then we perform a recursive call on each pair of subpattern and subexpressions. If any of them fail, we fail as well, if they all succeed, we concatenate their result and return.

## 6.3 Before constructor

This function takes as input :

- The information of the recursive call : name of the function and list of arguments. This allows to generate a recursive call to the DPS version.
- The pattern with which the recursive call was destroyed, that we will call the "recursive pattern" from now on.
- The expression we are tranforming.

It returns :

- If unsuccessful : `None`
- If successful : `Some e` where `e` is the transformed expression.

First we transform the pattern into a map of variable to indices. This is deterministic, and gives us the number of the destination and index of each recursive variable. following expression :

```
let (dst1, i1) = e1
and (dst2, i2) = e2
and (dst3, i3) = e3 in
f_dps dst1 i1 dst2 i2 dst3 i3 args
```

## 6.4 Before destructor

This step needs as input :

- The information needed to identify a recursive call, that is the name of the current function and the number of arguments it expects.
- The expression `e` we want to transform in dps form

It returns :

- If unsuccesful : `None`
- If successful : `Some (p, e')` where `p` is the pattern by which the recursive call is destroyed and `e'` is the DPS version of `e`.

This function works by matching on the input expression. It is very simple, it has only two cases, the first one is when `e` is a recursive call, the second works with any `match`. All other constructs result in a failure.

In case of a recursive call, we call the `before_constructor` function, and return its result (provided it is successful, it returns an `expr option`) along with the pattern we found.

In case of a match, we perform a recursive call on each branch. We do not expect a success on every branch, we can see in `partition_map` that some branches do not perform a recursive call (otherwise, recursive functions would never finish).

However we need a success on at least one branch, because we need to return a pattern. If we do not, we perform a soft failure by returning None. If we had at least one success, we need to check that the patterns from each successfully transformed branch are compatible, and we return any of them, that we will call `p` in the following.

Now we need to return an expression. We leave the expression that is being matched on intact, as well as the patterns from each branch. However we need to replace the expression. For the branches that were successfully transformed, we just use the transformed expression. If `e` was not successufully transformed, we transform it into `let p = e in ...` where `...` is an expression that take every variable from `p` and writes it into its destination.

For instance, in `partition_map`, when transforming the top-level match, we will find `p = (left, right)` from the second branch.

Then we can tranform the second branch : `[] -> ([], [])` becomes :

```
[] ->
  let (left, right) = ([], []) in
  dst0.i0 <- left ;
  dst1.i1 <- right
```

In this case, it would look better to do as presented higher, and directly write `[]` to each destination. However, this method is far more robust, it would still work if we had defined a `empty_list_pair` variable above, and a later inlining pass can always transform it into the nicer shape if possible.

## 6.5 Correctness

Here, proving correctness would be much more complex, because there is more state and different function. However we believe that it is mostly a matter of expressing the recursion invariants for each step.

## 6.6 Limitations

This transformation works well, although it requires the function to have a specific shape. Outside of the obvious requirement of being a recursive function, it needs to have the following shape :

```
let f a b c =
  ...
  let Cons ( p ) = f a' b' c' in
  ...
  Cons e
```

We also need `e` to have a certain shape :

It needs to be a list a of expression such that each variable introduced by `p` is mentionned once and only once.

You can notice that `map_double` does not have this shape, while `map_double_let` does. With the regular TRMC, it was the opposite : we needed to perform inlining on `map_double_let` to make it transformable.

Here we need to do the opposite of inlining. We need to be sure that a recursive call will be bound to a pattern (destructed sounds excessive because it may be a variable pattern).

The A-normal form transformation ensure this. If a trivial expression is either a variable or a constant, an expression is in A-normal form when only trivial expressions are allowed in function call arguments and constructor fields.

```
let rec map_double f li =
  match li with
  | [] -> []
  | ele :: li ->
    f ele :: f ele :: map_double f li
```

We be tranformed into :

```
let rec map_double f li =
    match li with
    | [] -> []
    | ::(ele, li) ->
      let fv_3 = map_double f li in
      let fv_2 = f ele in
      let fv_1 = f ele in
      let fv_0 = ::(fv_2, fv_3) in
      ::(fv_1, fv_0)
```

This is does not satisfies the constraints, as the visible constructor `::` does not mention the recursive variable `fv_3`. We could fix this by keeping count of let bindings in some way, but it is simpler to just undo the ANF transformation by inlining once the recursive call is found. This means transforming the following code :

```
let rec map_double f li =
    match li with
    | [] -> []
    | ::(ele, li) ->
      let fv_3 = map_double f li in
      f ele :: f ele :: fv_3
```

Our algorithm works perfectly on this, the limitations regarding the visibility of the recursive call are not "hard" limitations, and transforming the code in obviously equivalent ways is able to fix them.

# 7 TRMC with multiple return values and multiple use of recursive variables

However there is a limitation that is way harder to patch, and that is the fact that there needs to be one and only mention of each recursive variable. This is a hard limitation because in this framework, you always need to give a single destination per recursive variable to the recursive call, and each recursive always writes in each destination it got as input.

For instance, this function is not transformable with TRMC-mur :

```
let rec complete_tree n =
  if (n = 0) then
    Leaf
  else (
    let sub = complete_tree (sub n 1) in
    Node (n, sub, sub)
  )
```

A way to slightly relax this limitation would be to provide a "blank destination" that could either a `NULL` pointer (which would require checking for `NULL` before writing) or jut allocating a cell without ever reading it. This would only allow for one or zero mention of each recursive variable, but not for two like in `complete_tree`.

An more flexible framework is to replace destination pointers with lists of destination pointers. Thne instead of writing to a destinations pointers, we write to all the pointers in the list. This allows the previous function to have multiple holes corresponding to the same recursive variable.

In order to build the destination list, we chose to a have a reference to an empty list for each recursive variable. We push destinations to the lists when needed.

This gives the following code for `complete_tree` :

```
let rec write_all li v =
  match li with
  | [] -> ()
  | (dst, i) :: li ->
    dst.i <- v ;
    write_all li v

let rec complete_tree n =
    let dst = alloc 1 in
    complete_tree_dps ((dst, 0) :: []) n ;
    dst.0

and complete_tree_dps dst_in_0 n =
    if equals n 0 then
      let sub = Leaf in
      write_all_dst dst_in_0 sub
    else
      let dst_out_0 = ref [] in
      let block15 = Node(n, (), ()) in
      write_all_dst dst_in_0 block15 ;
      dst_out_0.0 <- (block15, 2) :: dst_out_0.0 ;
      dst_out_0.0 <- (block15, 3) :: dst_out_0.0 ;
      complete_tree_dps dst_out_0.0 (n - 1)
```

The algorithm to obtain this is substantially the same as the one for TMRC-mur, except with fewer checks and different boilerplate code being created. The code of the transformation is available for more detail.

## 7.1 Overhead

This transformation has the obvious disadvantagethat the generated code uses extra lists, which has an overhead. It would be nice to remove this overhead while still keeping MURMUS capabilities.

Lets first notice something : if we are not using a loop, recursive variable can only be used in a syntacly-bounded number of constructors. Considering that we could give up on such function, it could be possible to do a first pass to count the maximum possible occurences of each recursive variable, and compile the function accordingly. Instead of a list of destination, each recursive variable could have multiple destinations.

We can use the `null` trick to have only the destination used in the executed branch be written to.

If we also compute the minimum number of uses in the first pass, we can also have some destinations skip such checks. This would means that in the case of a function where the number of uses is deterministic we would have no overhead at all.

## 7.2 Limitations

There still are some limitations with this transformation. The most notable is the following :

```
let rec map_double_cond = fun f p li ->
  match li with
  | [] -> []
  | :: (ele, li) -> (
    let cond = p ele in
    let ele' = f ele in
    :: (f ele,
        if cond then
          :: (f ele, map_double_cond f p li)
        else map_double_cond f p li) )
```

This function is not transformable neither with TMRC-MURMUS nor TRMC-MUR, although it was with regular TRMC.

The reason is that the ANF form of this function is the following :

```
let rec map_double_cond =
  fun f p li ->
    match li with
    | [] -> []
    | ::(ele, li) ->
      let cond = p ele in
      let ele' = f ele in
      let fv_5 =
        if cond then
          let fv_4 = map_double_cond f p li in
          let fv_3 = f ele in
          ::(fv_3, fv_4)
        else
          map_double_cond f p li
      in
      let fv_1 = f ele in
      ::(fv_1, fv_5)
```

Here, the recursive call is not visible, the transformation will not be able to see it. However, there is no good reason for this to not be transformable, it really should be, but transforming it would require a more sophisticated toolbox than the ANF and inlining transformations I am using.

# 8 Conclusion

TRMC can be extended pretty far, although syntactical limits that do not feel like they should exist are a big issue.

My contribution include the 2 two algorithms, which also imply a classification of transormable functions. I also programmed

a framework to do experiment on future transformation, more easily than by directly modifying the OCaml source code.

Further work could be either implementing the described algorithms in the OCaml compiler or formalizing the transformation and proove their correctness.

# References

[BCS21] Frédéric Bour, Basile Clément, and Gabriel Scherer. Tail modulo cons. *CoRR*, abs/2102.09823, 2021.