Writing an Edge Tracker in ML: A Case Study RM 96/13

J. Sérot
LASMEA, Univ. B. Pascal de Clermont-Ferrand
Campus des Cezeaux, 63177 Aubiere cedex, France
E-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr

November 27, 1996

1 Introduction

Due to the high level of abstraction they offer, functional programming languages (FLs) have been advocated as very efficient tools for fast software prototyping [1]. The combination of pattern matching, higher-order functions and recursion enables the construction of concise, readable and easily modifiable programs. Yet, application of FLs to image processing raises interesting specific problems, especially if efficiency concerns are taken into account [2]. This memo will try to illustrate some of these points using an example found to be representative while designing a larger piece of vision software in ML: a basic edge tracking algorithm.

In a first section, we will concentrate on a sequential implementation of this algorithm, showing how very simple program transformation techniques can lead to very succint expressions of solutions, without sacrifying to efficiency. In the second section, we will introduce the use of the so-called functional *skeletons* as very effective parallelizing tool, starting from the previously defined sequential version of the edge tracker.

It must be said that our goal, when writing this memo was not merely to give an account of a particular implementation work on a classical algorithm but rather to demonstrate how functional languages can effectively support and aid the programmer when reasoning both at semantic and operational level. Hence the deliberately incremental and tutorial style of this paper.

2 Sequential implementation

2.1 First attempt

Basically, the goal of an edge-tracking algorithm is to transform a matrix representation of the image contours into a list of linked edges.

It starts from:

- a binary images of the edge points, obtained from any thin edge extraction process (such as the Canny-Deriche detector)
- a list of end-point coordinates, which will act as seeds for the tracking process itself (NB: for the sake of brievety, we do not detail here how these endpoints are identified and extracted from the binary image. The corresponding function may be found in appendix A).

It returns a list of linked edges, that we will represent here simply as a list of coordinates (although the actual implementation is better hidden within an appropriate abstract datatype (ADT):

```
type coord = int * int
type edge = coord list
```

Note: the image type may remain abstract here, provided that we suppose the existence of the adequate access functions (`create`, `get_pixel`, ..)

So, let's start with the given signature:

```
val track_edges : coord list -> img -> edge list
```

The basic sequential approach is to start tracking from every endpoint trying to extend it to an edge, and appending the result to a list of edges tracked so far. So, the highest functional level is a fold (left or right, this does not matter here) over the endpoint list:

```
let track_edges endpoints im = List.foldl (track im) [] endpoints
```

The empty list ([]) argument acts as initial list of tracked edges. The `track` operator is supposed to take the initial image, a point (its coordinates), a list of edges tracked so far and return a new list of edges, possibly augmented with the edge tracked from the current point:

```
val track : img -> edge list -> coord -> edge_list
```

For this, it will use an auxilliary function `track_from`, which will actually implement the recursive tracking process starting from a given "seed" point and building an edge by accumulating neighbooring points. Informally:

```
track_from image current_point =
    if found a neighboor p' to current_point in image
    then return current_point :: track_from image p'
    else [current_point]
```

So, the `track` folding operator may now be written:

```
let track im edges endpoint =
    let new_edge = track_from im endpoint
    in new_edge::edges
```

The hardest work to do now is to write the `track_from` definition. Using a few list manipulating higher order functions, this appears to be relatively straightforward:

```
let rec track_from im ((y,x) as p) =
    let ps = [(y+1,x); (y,x+1); (y,x-1); (y-1,x);
              (y+1,x+1); (y+1,x-1); (y-1,x+1); (y-1,x-1)]
    and select (y,x) = get_pixel im y x
    in match ( List.find select ps ) with
        | Some p' -> p :: track_from p'
        | None    -> [p]'
```

We first build a list of neighboors coordinates `ps`. Note that list ordering actually reflects the scanning order used by the algorithm. In this case:

```
8 4 7
3 . 2
6 1 5
```

Note: it is assumed here that `get_pixel im y x` returns the value of the pixel at coordinate `(y,x)` in image `im`. For the sake of brevety, it is also assumed that this function simply returns a false value for any out-of-bounds coordinates.

The next tracking point is picked up from this list using the `List.find` higher order function, as the first non-false neighbour [1].

This first formulation of the `track_from` function suffers from a serious bug: since it does not keep track of points that are aggregated to edges, there's a potentiality for these points to be aggregated more than once. This must be seen as an incorrect behaviour since, strictly speaking, a point must belong to at most one edge. Moreover, the whole process may (and in fact will !) loop going forth and back from one endpoint of one edge to another. The obvious solution to this is to explicitely keep track of all the points that have already been added to constructed edges to be sure not to reconsider them later on. From a first naive implementation point of view, this can easily be done using an accumulating list which will be passed along with the accumulated edges (and updated accordingly) along the track invocations:

```
let track im (edges,visited) endpoint =
    if not member endpoint visited then
        let new_edge, visited' = track_from im endpoint visited
        in (new_edge::edges, visited')
    else
        (edges, visited)
```

The `track_from` function may now be rewritten:

```
let rec track_from im ((y,x) as p) v =
    let ps = [(y+1,x); (y,x+1); (y,x-1); (y-1,x);
              (y+1,x+1); (y+1,x-1); (y-1,x+1); (y-1,x-1)]
    and select (y,x) = get_pixel im y x & not (member (y,x) v)
    in match ( List.find select ps ) with
        | Some p' -> let e', v' = track_from im p' p::v
                     in p::e' v'
        | None    -> [p], p::v
```

Of course, the visited list must be initialized at the fold level, which now becomes, assuming that we dont care about the final visited resulting list:

[1] The definition of auxilliary functions, like `List.find` may be found in appendix D

```
let track_edges im endpoints =
    let final_edges, _ = List.foldl (track im) ([],[]) endpoints
    in final_edges
```

2.2 Efficiency issues - 2nd attempt

The previous schema, while functionnaly correct, is however highly inneffi-cient, because of the linear O(n) search of the `visited` list that we need to perform when considering each neighbour of each successive tracked point. Of course, we could use a more sophisticated data-structure for the `visited` information to speed up the search (maybe at the cost of hopefully small slowdown at insertion time): coordinate indexed hash-tables, binary search trees, or ... But wait, what's the best structure for 2D-coordinate based indexing ? That's an ... *image*. So we might in fact define `visited` as a 2D-image of booleans, and take advantage of this to rewrite `track_from` in the following manner:

```
let rec track_from im ((y,x) as p) v =
    let ps = ...
    and select (y,x) = get_pixel im y x & not (get_pixel v y x)
    and v' = put_pixel v y x true in
    match ( List.find select ps ) with
        | Some p' -> let e', v'' = track_from im p' v'
                     in p::e', v''
        | None    -> [p], v'
```

The fold process will take care of providing a initialized `visited` image to subsequent functions, using something like:

```
let init im = create_img (height im) (width im) false in
let final_edges, _ = List.foldl (track im) ([],init im) endpoints
```

As this would be rather evident to every functional programmer ever confronted with the distinction between "pure" and "impure" functional languages, the latest formulation of the `track_from` function raises a well-known major question regarding the interpretation of the `put_pixel` function: from a purely functional point of view, this function should take a image `i`, a pair of coordinates `(y0,x0)`, a pixel value `v` and return a new image `i'` defined as follows: `i'.(y0,x0) = v` and `i'.(y,x) = i.(y,x)` elsewhere [2].

This interpretation is clearly unacceptable from an efficiency point of view, since it will involve the creation of at least as many intermediate images as there are tracked points. Although these intermediate images will probably reclaimed "on the fly" by the garbage collector (since their scope is limited to one invocation of the `track_from` function), the associated overhead will in fact make the whole process practically unusable.

The only work-around seems to give up a little bit of purity and rely on a imperative-flavoured interpretation of `put_pixel`, operating by side-effect on its argument. To distinguish it from its "pure" companion, let's call the corresponding function `set_pixel`. It can be defined within a language supporting updatable references, for instance using the Caml array assignment statement:

```
val set_pixel : img -> int -> int -> pixel -> unit
let set_pixel im y x v = im.(y).(x) <- v
```

The `track_from` function may then be slightly modified:

```
let rec track_from im ((y,x) as p) v =
    let ps = ...
    and select (y,x) = get_pixel im y x & not (get_pixel v y x)
    and _ = set_pixel v y x true in
    in match ( List.find select ps ) with
        | Some p' -> let e', v' = track_from im p' v
                     in p::e', v'
        | None    -> [p], v'
```

to reflect that its `visited` argument is now updated in-place throughout the whole tracking process.

2.3 Well-soudness of the last transformation

This kind of transformation, although resulting in a dramatic efficiency gain, has a significative counterpart on the semantic side: since we have lost the

[2] we adopt here the following infix notation for images: `im.(y,x)` designates pixel at coordinate `(y,x)` in image `im`

referential transparency which is the "hallmark" of pure FLs, we now have to prove the well-soudness of the new algorithm by explicitely ensuring that the `visited` value is used in a single-threaded manner throughout the whole program. Hopefully, all we need here is to effectively enforce a purely sequential operational interpretation for the main `fold` function. This happens to be inherently enforced by most ML compilers. In this case, we can safely replace `put_pixel` by `set_pixel` to go from O(n) to O(1) execution times for the `track` function. Incidently, by doing that, we do not even loose referential transparency at the level of the `track_edges` function because the mutable data `visited` is local to this function.

This kind of transformation, relying on the detection of single-threaded use of values, may be performed in most of the case at compile-time, so that the programmer should in fact continue to reason with pure functions like `put_pixel` at the functional level and let the compiler translate them to their imperative counterparts whenever possible. This approach has been successfully illustrated with at least two real size implementation of FLs: Sisal [3] and Clean [4]. The Sisal compiler uses a data-flow technique analysis to safely replace each applicative array operation by an imperative update, while the Clean compiler relies on a "uniqueness" type information which can be provided by the programmer or infered by the type checker.

Appendices A and B give the complete source code of the edge-tracking algorithm, in Caml (a widely used ML dialect), along with measured execution times.

3 Deriving a parallel implementation

3.1 Going for farming ?

At the very first sight, the `track_edges` function might seem suited to farm-like parallelism:

- it produces a list of results (edges) from a list of tasks (each endpoint may be viewed as a tracking-task descriptor)
- the tasks are intrinsically not load-balanced (some edges may be short, others long)
- no particular ordering is required for the results

So we might think of a `server` process dispatching endpoint coordinates to a fixed set of `workers`, each worker being responsible of tracking an edge from its given endpoint by running an instance of the `track_from` function. The gathering of the tracked edges would be handled by the server (in a less strict order than of the one involved by the `fold` in the sequential version, though).

This approach however raises a fundamental problem: the distributed `track_from` "instances" all require global access to the source image (`im`) and the `visited` image.

For the source image, since it is not modified by the `track_from` function, it is always possible to broadcast it to each worker before they actually start to track edges.

The problem is much more serious for the `visited` image, since it must be *updated* concurrently by workers, as illustrated by the following example:

```
A +
  ++++       * B
     +      **
     +  C ***
     +++**
        *
        *
        *
        D
```

Suppose Worker1 starts tracking from A, and Worker 2 from B. Without a global `visited` information - letting Worker 1 know, for example, that Worker 2 has tracked points from C to D - we will end up with a wrong set of edges, containing non-disjoint elements (`[A..D]` and `[B..D]` here).

Moreover, each edge might eventually tracked concurrently from its two endpoints, and therefore counted twice !

It is clear, then, that the tracking algorithm cannot be farmed safely without providing a kind of read/write access from each worker to a global piece of data. From a functional point of view, that means that the fold argument is no longer *associative*. From a operational point of view - and assuming that no direct communication can take place between workers - the only solution is to keep the "global" `visited` data on the server, letting it handle read/written requests from the workers. This solution, while theoritically feasible, is clearly highly innefficient because it effectively creates a major communication bottleneck (each worker makes at least 8 read request and 1 write request per tracked point !). Incidently, it may also lead to non-deterministic results, since the order in which the requests will be handled indirectly determines the final segmentation of multi-branch edges (see example).

Finally, it should be noticed that this problem does remain with `visited` implemented as a list, showing that it has nothing to see with the particular applicative or imperative formulation of the algorithm.

3.2 Split it up !

As part of our research on *skeletal parallelism* we found edge tracking as a strongly illustrative example of the so-called SCM (Split, Compute and Merge) skeleton. Let us recall that a *skeleton* is a higher-order function which goal is to encapsulate a common parallel programming paradigm.

In our case, the parallel paradigm may be stated as follows:

1. split the source image in n (say horizontal) strips
2. run the sequential algorithm on each strip, obtaining locally tracked edges
3. merge "broken" edges which have been artificially split because they were crossing strip boundaries

This is precisely what is conveyed by the SCM skeleton, with signature:

```
val SCM:
    ('a -> 'b list) -> ('b -> 'c) -> ('c list -> 'd) -> 'a -> 'd
```

and which may defined *functionnaly* [3] as:

[3] Each skeleton actually has two equivalent definitions:

- a *sequential*, *applicative* one, like the one given here, establishing its denotational semantics, and used for cheking algorithm semantics.
- a *parallel*, *operational* one, making use of the actual parallel facilities of the target platform.

```
let SCM split f merge x = merge (map f (split x))
```

The track_edges function may now be reformulated as follows:

```
let par_track_edges im =
    SCM (row_block n) seq_track_edges gather_edges im
```

where:

- `val row_block : img -> int -> img list`

  returns an list of n images made from spliting its first argument into equally-sized horizontal strips

- `val seq_track_edge: img -> edge list`

  is the sequential edge_tracking function we have been described in first section, except that computation of endpoints is now carried on by the function itself.

- `val gather_edges : (edge list) list -> edge list`

  will be responsible for gathering the edges accumulated separately in each image strip, taking care in particular of merging the broken ones.

As can be easily guessed, the hardest work relies now on the writing of the gather_edges function. First, having convinced ourselves that it corresponds to an an associative operation, we can decompose it using a tree-like folding higher-order function, thus enjoying a O(log n) merging time:

```
let gather_edges = foldt merge_edges
```

Where foldt is a binary folding HOF:

```
let rec foldt f = function
    | [] -> failwith "foldt"
    | [x] -> x
    | xs -> let l,r = split_at (List.length xs/2) xs
            in f (foldt f l) (foldt f r)
```

Now, we are left with a binary function merge_edges aimed at merging edges coming from two neighbouring strips:

```
val merge_edges : edge list -> edge list -> edge list
```

This function ultimately relies on a combinatorial search of matching edge pairs at the common strip boundary. With a little abstraction, it appears that this search may be defined as an instance of the following higher-order function:

```
val split_from_match :
    ('a -> 'b -> bool) ->
    'a list -> 'b list -> ('a * 'b) list * 'a list * 'b list
```

Given two lists $xs = [x_1; ...; x_n]$ and $ys = [y_1; ...; y_m]$, `split_from_match p xs ys` returns three lists:

- the list of all pairs (x_i, y_j) such that px_iy_j holds
- the list of x_i such that there's no y_j such that px_iy_j holds
- the list of y_i such that there's no x_i such that px_iy_j holds

In our case, the predicate p will be:

```
val  compat : edge -> edge -> bool
let compat e1 e2 =
    (* e1,e2 : two edges from upper and lower strip with height h1,h2 *)
    let (y11,x11),(y12,x12) = hd e1, lst e1
    and (y21,x21),(y22,x22) = hd e2, lst e2
    and h y1 x1 y2 x2 = y1 mod h1 = h1-1 & y2 mod h2 = 0 & abs(x1-x2) < 2 in
        h y11 x11 x21 y21
     or h y11 x11 y22 x22
     or h y12 x12 y21 x21
     or h y12 x12 y22 x22
```

Finally we will assume the existence of a function handling the merging of two edges : [4]

```
val merge : edge * edge -> edge
```

[4]this function does a little more than just appending the two lists of coordinates, since it has to appropriatly adjust the local coordinates of each edge

The merge_edges function now boils down the the following:

```
let merge_edges (es1,h1) (es2,h2) =
    (* es1,es2 : edges from upper and lower strip with height h1,h2 *)
    let compat e1 e2 = ...
    let mps, es1', es2' = split_from_match compat es1 es2 in
    let mes = unzip_with merge mps in
    (mes @ es1' @ es2', h1+h2)
```

The complete implementation of the edge tracker using ths SCM skeleton is given in appendix C.

4 Conclusion

It is hoped that the approach related here will have provided fruitful insights into the process of designing, prototyping and handcrafting concise but efficient image processing algorithms in a strict functional language like ML.

As far as parallel programming techniques are concerned, the skeleton-based methodology briefly sketched here has been successfully applied to the implementation of a complete image segmentation process involving both regular data-decomposition (scm) and farming.

5 References

[1] John Hughes **Why Functional Programming Matters**, *The Computer Journal*, 32(2), February 1989
[2] N.R.Scaife, G.J.Michaelson, A.M.Wallace **A method for developing parallel vision algorithms with an example of edge tracking**, *Research Memo RM/96/1*, January 1996
[3] Information available online from *http://unix.hensa.ac.uk/parallel/languages/sisal*
[4] Information available online from *http://www.cs.kun.nl:80/ clean/*

6 Acknowledgments

This work is part of research conducted during a two-months visit within the Computer Vision Group of Heriot-Watt University (Department of Computing and Electrical Engineering), visit funded by the TRACS (Training and Research on Advanced Computer Systems) program supervised by the Edinburgh Parallel Comcputing Center (EPCC).

The author also wants to thank G. Michaelson for his careful reading of the manuscript.

A Caml source code for the sequential edge tracker

```
(* Edge tracking algorithm - sequential version *)
(* Nov 1996 *)
(* J. Serot (Jocelyn.Serot@lasmea.univ-bpclermont.fr) *)

open FImg (* Default image type: pixel = float *)
open Edge (* ADT for edges. Basically a list with cached exts *)

let is_end_point ps =
match (ps.(4) <> 0.0,
   ps.(0) <> 0.0, ps.(1) <> 0.0, ps.(2) <> 0.0,
   ps.(3) <> 0.0,                ps.(5) <> 0.0,
   ps.(6) <> 0.0, ps.(7) <> 0.0, ps.(8) <> 0.0)
with
      | (false, _,_, _,_,_,_,_,_) -> false
      | (true,true,_,false,false,false,false,false,false) -> true
      | (true,false,true,_,false,false,false,false,false) -> true
      | (true,false,false,true,false,_,false,false,false) -> true
      | (true,false,false,false,false,true,false,false,_) -> true
      | (true,false,false,false,false,false,false,_,true) -> true
      | (true,false,false,false,false,false,_,true,false) -> true
      | (true,false,false,false,_,false,true,false,false) -> true
      | (true,_,false,false,true,false,false,false,false) -> true
      | _                                                 -> false

let rec track_from ((y,x) as c) v im =
  let cs = [(y+1,x);(y,x+1);(y,x-1);        (* scanning order:  8 4 7 *)
            (y-1,x);(y+1,x+1);(y+1,x-1);    (*                  3 . 2 *)
            (y-1,x+1);(y-1,x-1)] in         (*                  6 1 5 *)
  let select ((y,x) as c) =                 (* best with lazy & !!! *)
   in_bounds im y x & get_pixel im y x <> 0.0 & not (BImg.get_pixel v y x)  in
```

```
  match (List2.find select cs) with
    | Some c' ->
        let _ = BImg.set_pixel v y x true in
        let (e',v') = track_from c' v im in Edge.add (im@@c,c) e', v'
    | None ->
       let _ = BImg.set_pixel v y x true in
       Edge.create (im@@c,c), v

let track_edges im =
  let nr = nb_rows im and nc = nb_cols im in
  let endpoints = filter_local nbh33_8 is_end_point zero im in
  let track (edges,visited) ((y,x) as endpoint) =
    if BImg.get_pixel visited y x then
      (edges,visited)
    else
      let new_edge,visited' = track_from endpoint visited im
      in (new_edge::edges,visited') in
  let init = BImg.create (nb_rows im) (nb_cols im) (Some false) in
  let (es,vs) = List.fold_left track ([],init) endpoints in
  es
```

B Execution times for the sequential edge tracker

Execution times have been reported on a Sun 4-51 workstation, running version 1.02 of the Objective Caml compiler (available from Inria, at http://pauillac.inria.fr/caml) both in bytecode and native mode. Source image was "lena.pgm", 256x256. 1123 edges were tracked

	Bytecode	Native
APPLICATIVE	51.52	21.26
IMPERATIVE	0.53	0.09

C Caml source code for the parallel edge tracker

```
(* Edge tracking algorithm - parallel version *)
(* Nov 1996 *)
(* J. Serot (Jocelyn.Serot@lasmea.univ-bpclermont.fr) *)

open FImg
open Edge

let is_end_point ps = ... (* same as sequential case *)

let rec track_from ((y,x) as c) v im = ... (* same as sequential case *)

let translate (h,w) e = Edge.map (function (v,(y,x)) -> (v,(y+h,x+w))) e

let track_edges im =
  let nr = nb_rows im and nc = nb_cols im and id = get_tag im in
  let endpoints = filter_local nbh33_8 is_end_point zero im in
  let track (edges,visited) ((y,x) as endpoint) =
    if BImg.get_pixel visited y x then
      (edges,visited)
    else
      let new_edge,visited' = track_from endpoint visited im
      in (new_edge::edges,visited')
  and init = BImg.create (nb_rows im) (nb_cols im) (Some false) in
  let (es,vs) = List.fold_left track ([],init) endpoints in
  let es' = List.map (translate (nr*id,0)) es in
  (es', nr)

let merge_edges (es1,h1) (es2,h2) =
  (* es1,es2 : edges from upper and lower strip with height h1,h2 *)
  let compat e1 e2 =
    let (_,(y11,x11)),(_,(y12,x12)) = Edge.ext1 e1, Edge.ext2 e1
    and (_,(y21,x21)),(_,(y22,x22)) = Edge.ext1 e2, Edge.ext2 e2
    and p y1 x1 y2 x2 = y1 mod h1 = h1-1 & y2 mod h2 = 0 & abs(x1-x2) < 2 in
       p y11 x11 x21 y21
    or p y11 x11 y22 x22
    or p y12 x12 y21 x21
    or p y12 x12 y22 x22 in
  let mps, es1', es2' = split_from_match compat es1 es2 in
```

```
  let mes = List2.unzip_with Edge.merge mps in
  (mes @ es1' @ es2', h1+h2)

let par_track_edges im =
  let es, _ = scm (row_block np) track_edges (foldt merge_edges) im in
    (* np is assumed to be bound to the number of available processes *)
  es
```

D A few useful HOFs

```
val find :  ('a -> bool) -> 'a list -> 'a option
let rec find p = function
  | [] -> None
| x::xs -> if p x then Some x else find p xs

val split_from_match :
  ('a -> 'b -> bool) ->
   'a list -> 'b list -> ('a * 'b) list * 'a list * 'b list
let split_from_match p xs ys =
  let rec h ms nmxs ys = function
    | [] -> (ms, nmxs, ys)
    | x::xs -> match (find_and_remove (p x) ys) with
       | (Some y, ys') -> h ((x,y)::ms) nmxs ys' xs
       | (None, _)     -> h ms (x::nmxs) ys xs
  in
    h [] [] ys xs

val find_and_remove : ('a -> bool) -> 'a list -> 'a option * 'a list
let rec find_and_remove p  = function
  | [] -> (None, [])
  | x::xs -> if p x then (Some x, remove x xs)
             else let r,xs' = find_and_remove p xs in (r, x::xs');;

val unzip_with : ('a -> 'b -> 'c) -> ('a * 'b) list -> 'c list
let rec unzip_with f = function
  | [] -> []
  | (x,y)::xys -> f x y :: unzip_with f xys

val split_at : int -> 'a list -> 'a list * 'a list
```

```
let rec split_at n l =
  if n <= 0 then ([],l) else
  match l with
    | [] -> ([],[])
    | x::xs -> let (xs',xs'') = split_at (n-1) xs in (x::xs',xs'')
```