The evolution of an OCaml programmer

Christophe Deleuze

May 23, 2008

Abstract

A few ways to describe the computation of (should I just say "program"?) the factorial function in OCaml. We begin with the classical *for* loop and recursion and end with Church naturals in the lambda calculus. On the way, we meet tail, structural, and open recursion, streams and point-free style, Peano, functors, continuation passing style (CPS), meta-programming, `call/cc`, trampolined style, reactive programming, and linear logic (not necessarily in this order, though). Inspiration came from [12], itself inspired by [2].

We start with one of the most straightforward programs for the factorial function: as a recursive procedure, generating a linear recursive process [1].

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
```

But then we realize that OCaml also has imperative features, so we can eliminate these costly recursive calls! Here's an imperative procedure generating a linear iterative process.

```
let fact n =
  let a = ref 1 in
  for i = 1 to n do a := !a * i done;
  !a
```

Hey, this imperative stuff is ugly, didn't you know that tail recursion can be as efficient as a loop? Here's a recursive procedure generating a linear iterative process.

```
let fact n =
  let rec f i a = if i = n then a else f (i+1) ((i+1)*a)
  in
  f 0 1
```

The next, rather stupid, implementation could arguably be described as a linear recursive process generated by an iterative procedure. Rant: OCaml arrays have their first element at position 0, which is often inconvenient (remember Pascal arrays?)

```
let fact n =
  let a = Array.make (n+1) 1
  in
  for i = 2 to n do
    a.(i) <- a.(i-1) * i
  done;
  a.(n-1)
```

We've been using syntactic sugar for function declarations, let's avoid this and make the lambda explicit:

```
let fact =
  let rec f i n a = if i = n then a else f (i+1) n ((i+1)*a)
  in
  function n -> f 0 n 1
```

or alternatively:

```
let fact = function n ->
  let rec f i a = if i = n then a else f (i+1) ((i+1)*a)
  in
  f 0 1
```

Feeling lispy? Turn these infix operators into prefix functions (requires even more parentheses than in Lisp!):

```
let fact = function n ->
  let rec f i a = if i = n then a else f ((+) i 1) ( ( * ) ((+) i 1) a)
  in
  f 0 1
```

This “if then else” expression is too Pascalish, it doesn’t look mathematical enough; let’s being more declarative by using a transformational style:

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1)
```

Oops! We forgot to use tail recursion!

```
let fact n =
  let rec f n a = match n with
  | 0 -> a
  | n -> f (n-1) (a*n)
  in
  f n 1
```

General recursion is not the only way to go. Recursion operators can be fun(ctional) too. Did you know you could do so many things with fold? [8]

```
let rec list_from_to n m =
  if n > m then [] else n :: (list_from_to (n+1) m)

let fact = fun n -> List.fold_right (fun n acc -> acc*n) (list_from_to 1 n) 1
```

But again, neither `list_from_to` nor `fold_right` are tail-recursive.

```
let list_from_to n m =
  let rec mkl l m =
    if n > m then l else mkl (m::l) (m-1)
  in
  mkl [] m

let fact = fun n -> List.fold_left (fun acc n -> acc*n) 1 (list_from_to 1 n)
```

That’s good but hey, how useful is tail recursion if this `list_from_to` stuff uses linear space? Lazy lists (aka streams) allow us to do the same thing in constant space with the very same formulation. Look:

```
let stream_from_to n m =
  let f i = if (n+i) <= m then Some (n+i) else None
  in
  Stream.from f

let stream_fold f i s =
  let rec help acc =
```

```
    try
      let n = Stream.next s in
      help (f acc n)
    with Stream.Failure -> acc
  in
  help i

let fact = fun n -> stream_fold (fun acc n -> acc*n) 1 (stream_from_to 1 n)
```

If you read Backus' paper about FP [3], you'd probably prefer to write it in point-free style[1]:

```
let insert unit op seq =
  stream_fold
    (fun acc n -> op acc n)
    unit
    seq

let iota = stream_from_to 1

let fact =
  let (/) = insert
  and ( @ ) f g x = f(g x)  (* sadly, we can't use 'o' as infix symbol *)
  in
  ((/) 1 ( * )) @ iota
```

We could define the stream insert with recursion rather than using the fold operator:

```
let insert unit op seq =
  let rec help acc =
    try
      let n = Stream.next seq in
      help (op acc n)
    with Stream.Failure -> acc
  in
  help unit
```

If we don't like OCaml built-in Stream module, it's easy to define streams with state-full functions. Rather than raising an exception at stream end, we'll use an option type for its value.

```
let stream_from_to n =
  let current = ref (n-1)
  in
  fun m () -> if !current < m then begin
    incr current;
    Some !current
  end else None

let stream_fold f i st =
  let rec help acc =
    match st () with
      Some n -> help (f acc n)
    | None -> acc
  in
  help i

let fact n = stream_fold ( * ) 1 (stream_from_to 1 n)
```

Open recursion allows many nice things like tracing calls or extending types [11, 7].

[1] FP's insert operator is denoted '/' and the unit parameter is given by the `unit` functional form (takes a function as argument and returns its unit if there is one.)

```
let ofact f n =
  if n = 0 then 1
  else n * f (n-1)

let rec fact n = ofact fact n
```

Or alternatively

```
let rec fix f x = f (fix f) x
let fact = fix ofact
```

Repeat: “I should use tail recursion”...

```
let ofact f acc n =
  if n = 0 then acc
  else f (n*acc) (n-1)

let fact n =
  let rec help a n = ofact help a n
  in
  help 1 n
```

All these functions make use of the quite limited CPU based int type. OCaml provides exact integer arithmetic through the `Big_int` module:

```
open Big_int

let rec fact = function
    0 -> zero_big_int
  | n -> mult_big_int (big_int_of_int n) (fact (n-1))
```

Another way to get rid of this int data type is to build natural numbers from scratch, as defined, for example, by Peano. In the process we switch from general to structural recursion (thus syntactically ensuring termination and making our function total) :

```
type nat = Zero | Succ of nat

let rec (+) n m =
  match n with
  | Zero    -> m
  | Succ(p) -> p + (Succ m)

let rec ( * ) n m =
  match n with
  | Zero       -> Zero
  | Succ(Zero) -> m
  | Succ(p)    -> m + (p * m)

let rec fact n =
  match n with
  | Zero    -> Succ(Zero)
  | Succ(p) -> n * (fact p)

(* convenience functions *)

let rec int_of_peano = function
    Zero    -> 0
  | Succ(p) -> succ (int_of_peano p)

let rec peano_of_int = function
    0 -> Zero
  | n -> Succ(peano_of_int (n-1))
```

Tail-recursive versions are left as (easy) exercises to the reader!

What about using functors? Whatever representation we use for naturals, they can be defined as a module with the NATSIG signature defined below. Then, a functor taking a module of NATSIG signature can be defined to provide a factorial function for any NATSIG datatype.

```
module type NATSIG =
  sig
    type t
    val zero:t
    val unit:t
    val mul:t->t->t
    val pred:t->t
  end

module FactFunct(Nat:NATSIG) =
  struct
    let rec fact n:Nat.t =
      if n = Nat.zero then Nat.unit else Nat.mul n (fact (Nat.pred n))
  end
```

Here are examples of uses with native integers, Peano naturals and OCaml `big_ints`:

```
module NativeIntNats =
  struct
    type t = int
    let zero = 0
    let unit = 1
    let mul = ( * )
    let pred n = n-1
  end

module PeanoNats =
  struct
    type t = Zero | Succ of t
    let zero = Zero
    let unit = Succ(Zero)

    let rec add n m =
      match n with
      | Zero    -> m
      | Succ(p) -> add p (Succ m)

    let rec mul n m =
      match n with
      | Zero       -> Zero
      | Succ(Zero) -> m
      | Succ(p)    -> add m (mul p m)

    let pred = function Succ(n) -> n | Zero -> Zero
  end


module BigIntNats =
  struct
    type t = Big_int.big_int
    let zero = Big_int.zero_big_int
    let unit = Big_int.unit_big_int
    let pred = Big_int.pred_big_int
    let mul =  Big_int.mult_big_int
  end
```

```
module NativeFact = FactFunct(NativeIntNats)
module PeanoFact = FactFunct(PeanoNats)
module BigIntFact = FactFunct(BigIntNats)
```

In the previous examples, to make the function tail-recursive we used a second *accumulator* parameter, which in some sense remembered the past computations (actually just their result).

Another way to turn a recursive function into a tail-recursive one is to use *continuation passing style* (aka CPS)[2]. The series of recursive calls actually *build* as a function the sequence of computations to be done, that will all be performed on the final base call (for n=0).

This could be considered cheating since, although the function is tail-recursive, the size of the "accumulator" grows linearly with the number of recursive calls...

```
let fact =
  let rec f n k =
    if n=0 then k 1
    else f (n-1) (fun r -> k (r*n))
  in
  function n -> f n (fun x -> x)
```

This is not so different from using meta-programming. Metaocaml [15] is a nice meta-programming extension of OCaml. Using that, you can write a function that takes an integer n as parameter and produces the code that, when run, will compute the factorial of n... (just for fun)

```
let rec gfact = function
  | 0 -> .<1>.
  | n -> .< n * .~(gfact (n-1)) >.

let fact n = .! gfact n
```

Of course, this can be made tail-recursive too! Then it looks very similar to the previous example using continuations:

```
let gfact n =
  let rec f n acc =
    if n=0 then acc
    else f (n-1) .< n * .~acc>.
  in f n .<1>.
```

Using first class continuations, we can run our function step by step as a synchronous process possibly among other processes [5].[3]

```
open Callcc

type 'a process = Proc of ('a -> 'a) | Cont of 'a cont

(* Primitives for the process queue.  Exit continuation will always be first *)

let enqueue, dequeue =
  let q = ref [] in
  (fun e -> q := !q @ [e]),
  (fun () -> match !q with
    e::f::l -> q := e::l; f     (* keep exit cont at head of queue *)
  | [e]     -> q := []; e)      (* no more process to run *)

let run p =
```

[2]Could someone provide me with a good reference for this?

[3]Straight OCaml does not feature first class continuations but a toy extension for them exists [9]. A more robust library also supports delimited continuations [13].

```
  match p with
    Proc proc -> proc ()
  | Cont k -> throw k ()

(* Queue back a suspending process, dequeue and run *)

let swap_run p =
  enqueue p;
  run (dequeue ())

(* Two functions to be used in processes.  halt to terminate,
   pause_handler to suspend *)

let halt () = run (dequeue ())

let pause_handler () =
  callcc (fun k -> swap_run (Cont k))

let create_process th =
  Proc (fun () -> th (); halt ())

(* Dispatcher puts exit continuation in process queue, adds processes
   through init_q and runs first process. *)

let dispatcher init_q =
  callcc
    (fun exitk ->
      enqueue (Cont exitk);
      init_q ();
      halt ())

(* Factorial as a process *)

let fact_maker n =
  create_process
    (fun () ->
      let rec fact n =
        if n=0 then 1
        else begin
          pause_handler();
          n * fact (n-1)
        end
      in Printf.printf "fact=%i\n" (fact n))

let fact n = dispatcher (fun () -> enqueue (fact_maker n));;
```

Trampolined style [6] is another way to run our function step by step concurrently with others, but without the need for first class continuations. Instead, we'll use CPS. That implies that we must use a tail recursive version of factorial in this case.

```
type 'a thread = Done of 'a | Doing of (unit -> 'a thread)

let return v = Done v
let bounce f = Doing f

(* factorial function *)

let rec fact_tramp i acc =
  if i=0 then
    return acc
  else
```

```
    bounce (fun () -> fact_tramp (i-1) (acc*i))

(* one thread scheduler *)

let rec pogostick f =
  match f () with
  | Done v -> v
  | Doing f -> pogostick f

(* give our fact trampoline function to the scheduler *)

let fact n =
  pogostick (fun () -> fact_tramp n 1)
```

Functional reactive programming is still another way to describe concurrent, synchronous, computations. In reactiveML [10], a factorial computing process could be written as:

```
let rec process fact n =
  pause;
  if n<=1 then 1
  else
    let v = run (fact (n-1)) in
    n*v
```

In a *linear logic* based language [4], each bound name is required to be referenced exactly once. This may seem odd but ensures nice properties, e.g. avoiding the need for garbage collection. Pretending we have some "linear OCaml" compiler, we could write the factorial this way:

```
let dup x = (x,x)   (* dup and kill would be    *)
let kill x = ()     (* provided by linear OCaml *)

let rec fact n =
  let n,n' = dup n
  in
  if n=0 then (kill n'; 1)
  else
    let n,n' = dup n' in
    n' * fact (n-1)
```

Computing with Peano numbers like we did previously is fun, but let's be more functional and use Church naturals instead! Sadly, although they can be defined in OCaml, the type system rejects the pred operator. Never mind, let's quickly build[4] a lambda calculus interpreter![5]

There may be a subtle bug in the way alpha conversion is performed. Can you find it?

```
(* the type of lambda terms *)

type lambda = Var of string | Abs of string * lambda | App of lambda * lambda

(* free variables in t *)

let rec fv t =
  match t with
  | Var x -> [ x ]
  | Abs(x,l) -> List.filter (fun e -> e<>x) (fv l)
  | App(t1,t2) -> fv t1 @ fv t2
```

[4] I must admit it took me (much) more time than I first expected...

[5] Code for `cbn` and `nor` is from [14].

```
(* in t, rename olds present in seen -- seen under lambdas *)

let rec alpha olds seen nb t =
  match t with
  | Var s -> if List.mem s seen then Var (s^nb) else t
  | App(l1,l2) -> App(alpha olds seen nb l1, alpha olds seen nb l2)
  | Abs(s,l)   ->
      if List.mem s olds then Abs(s^nb, alpha olds (s::seen) nb l)
      else                    Abs(s, alpha olds seen nb l)

(* body[arg/s], alpha conversion already done *)

let rec ssubst body s arg =
  match body with
  | Var s'     -> if s=s' then arg else Var s'
  | App(l1,l2) -> App(ssubst l1 s arg, ssubst l2 s arg)
  | Abs(o,l)   -> if o=s then body else Abs(o, ssubst l s arg)

let gen_nb =
  let nb = ref 0 in function () -> incr nb; !nb

(* body[arg/s], avoiding captures *)

let subst body s arg =
  let fvs = fv arg in
  ssubst (alpha fvs [] (string_of_int (gen_nb())) body) s arg

(* call by name evaluation *)

let rec cbn t =
  match t with
  | Var _ -> t
  | Abs _ -> t
  | App(e1,e2) -> match cbn e1 with
    | Abs(x,e) -> cbn (subst e x e2)
    | e1'      -> App(e1', e2)

(* normal order evaluation *)

let rec nor t =
  match t with
  | Var _ -> t
  | Abs(x,e) -> Abs(x,nor e)
  | App(e1,e2) -> match cbn e1 with
    | Abs(x,e) -> nor (subst e x e2)
    | e1' -> let e1'' = nor e1' in
      App(e1'',nor e2)

(* some useful basic constructs *)

let succ = Abs("n", Abs("f", Abs("x",
               App(Var"f", App(App(Var"n",Var"f"),Var"x")))))

let pred = Abs("n",Abs("f",Abs("x",
             App(App(App(Var"n",
                     (Abs("g",Abs("h",App(Var"h",App(Var"g",Var"f")))))),
                 (Abs("u",Var"x"))),
             (Abs("u",Var"u"))))))

let mult = Abs("n",Abs("m",Abs("f",Abs("x",
             App(App(Var"n",(App(Var"m",Var"f"))),Var"x")))))
```

```
let zero = Abs("f",Abs("x",Var"x"))

let t = Abs("x",Abs("y",Var"x"))
let f = Abs("x",Abs("y",Var"y"))
let iszero = Abs("n",App(App(Var"n",Abs("x",f)), t))

let y = Abs("g", App(Abs("x", App(Var"g",App(Var"x",Var"x"))),
                     Abs("x", App(Var"g",App(Var"x",Var"x")))))

(* now let's build the factorial function *)

let fact =
  let ofact = Abs("f",Abs("n",
                          App(App(App(iszero,Var"n"),
                                  App(succ,zero)),
                              App(App(mult,Var "n"),
                                  (App(Var"f",App(pred,Var"n")))))))
  in App(ofact,App(y,ofact))

(* convenience functions *)

let church_of_int n =
  let rec coi n = if n=0 then Var"x" else App(Var"f", coi (n-1))
  in
  Abs("f",Abs("x", coi n))

exception NotaNat

let int_of_church n =
  let rec ioc n f x =
    match n with
      Var x' -> if x=x' then 0 else raise NotaNat
    | App(Var f',r) -> if f=f' then (ioc r f x) + 1 else raise NotaNat
    | _ -> raise NotaNat
  in
  match n with
    Abs(f,Abs(x,r)) -> ioc r f x
  | _ -> raise NotaNat

let fact n = int_of_church (nor (App(fact,church_of_int n)))
```

All right, that's enough for today. Next time, we may talk about objects, syntax extensions, MVars and a few other nice things.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984. http://mitpress.mit.edu/sicp/

[2] Unkown author. The evolution of a programmer. Article in newsgroup, circa 1990. http://www.pvv.ntnu.no/~steinl/vitser/evolution.html

[3] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. http://www.stanford.edu/class/cs242/readings/backus.pdf

[4] Henry G. Baker. Linear logic and permutation stacks–the forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[5] D. P. Friedman. Applications of continuations. Invited Tutorial, Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 1988. http://www.cs.indiana.edu/~dfried/appcont.pdf

[6] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999. http://citeseer.ist.psu.edu/ganz99trampolined.html

[7] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

[8] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. http://www.cs.nott.ac.uk/~gmh/bib.html#fold

[9] Xavier Leroy. OCaml-callcc: call/cc for OCaml. OCaml library. http://pauillac.inria.fr/~xleroy/software.html

[10] Louis Mandel and Marc Pouzet. Reactive ML. http://www.reactiveml.org

[11] Bruce McAdam. Y in practical programs (extended abstract), 2001. http://citeseer.ist.psu.edu/mcadams01practical.html

[12] Fritz Ruehr. The evolution of a haskell programmer. Web page, 2001. http://www.willamette.edu/~fruehr/haskell/evolution.html

[13] Amr Sabry, Chung chieh Shan, and Oleg Kiselyov. Native delimited continuations in (byte-code) ocaml. OCaml library. http://okmij.org/ftp/Computation/Continuations.html#caml-shift

[14] Peter Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation: complexity, analysis, transformation*, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[15] Walid Taha et al. MetaOCaml: A compiled, type-safe, multi-stage programming language. Web page. http://www.metaocaml.org