```
(* Interfaces d'iterateurs / de flux *)
module type SimpleIter =
sig
  type 'a t
  val vide : 'a t
  val cons : 'a -> 'a t -> 'a t
  val unfold : ('s -> ('a * 's) option) -> 's -> 'a t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val constant : 'a -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module type Iter =
sig
  include SimpleIter
  val uncons : 'a t -> ('a * 'a t) option
  val apply : ('a -> 'b) t -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
end

(* Le module Flux implantant l'interface de flux Iter *)
type 'a flux = Tick of ('a * 'a flux) option Lazy.t;;
module Flux : Iter with type 'a t = 'a flux =
  struct
    type 'a t = 'a flux = Tick of ('a * 'a t) option Lazy.t;;

    let vide = Tick (lazy None);;

    let cons t q = Tick (lazy (Some (t, q)));;

    let uncons (Tick flux) = Lazy.force flux;;

    let rec apply f x =
      Tick (lazy (
      match uncons f, uncons x with
      | None         , _             -> None
      | _            , None          -> None
      | Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx)));;

    let rec unfold f e =
      Tick (lazy (
      match f e with
      | None         -> None
      | Some (t, e') -> Some (t, unfold f e')));;

    let rec filter p flux =
      Tick (lazy (
      match uncons flux with
      | None        -> None
      | Some (t, q) -> if p t then Some (t, filter p q)
                       else uncons (filter p q)));;

    let rec append flux1 flux2 =
      Tick (lazy (
      match uncons flux1 with
      | None          -> uncons flux2
      | Some (t1, q1) -> Some (t1, append q1 flux2)));;

    let constant c = unfold (fun () -> Some (c, ())) ();;
    (* implantation rapide mais inefficace de map *)
    let map f i = apply (constant f) i;;

    let map2 f i1 i2 = apply (apply (constant f) i1) i2;;
end

(* Parametres globaux de la simulation     *)
(* dt : pas de temps                       *)
(* box_x : paire d'abscisses (xmin, xmax)  *)
(* box_y : paire d'ordonnees (ymin, ymax)  *)
module type Frame =
  sig
    val dt : float
    val box_x : float * float
    val box_y : float * float
  end

module Drawing (F : Frame) =
  struct
    let draw r =
      let ref_r = ref r in
      let ref_handler_alrm = ref Sys.(Signal_handle (fun _ -> ())) in
      let ref_handler_int  = ref Sys.(Signal_handle (fun _ -> ())) in
      let handler_alrm i =
        begin
          match Flux.uncons !ref_r with
          | None                          ->
             begin
               Sys.(set_signal sigalrm !ref_handler_alrm);
               Sys.(set_signal sigint  !ref_handler_int)
             end
          | Some (((x, y), (dx, dy)), r') ->
             begin
               (*Format.printf "r=(%f, %f); dr = (%f, %f)@." x y dx dy;*)
               Graphics.clear_graph ();
               Graphics.draw_circle (int_of_float x) (int_of_float y) 5;
               Graphics.synchronize ();
               (*ignore (read_line ());*)
               ref_r := r'
             end
        end in
      let handler_int i =
        begin
          ref_r := Flux.vide
        end in
      begin
        let (inf_x, sup_x) = F.box_x in
        let (inf_y, sup_y) = F.box_y in
        let size_x = int_of_float (sup_x -. inf_x) in
        let size_y = int_of_float (sup_y -. inf_y) in
        Graphics.open_graph (Format.sprintf " %dx%d" size_x size_y);
        Graphics.auto_synchronize false;
        Sys.(ref_handler_alrm := signal sigalrm (Signal_handle
handler_alrm));
        Sys.(ref_handler_int  := signal sigint  (Signal_handle handler_int));
        Unix.(setitimer ITIMER_REAL { it_interval = F.dt; it_value = F.dt })
      end
  end
```

```
(* Caracteristiques du système:  *)
(* pas de temps + etat du mobile *)
module type Params =
  sig
    val dt : float
    val masse0 : float
    val position0 : float * float
    val vitesse0  : float * float
  end

module FreeFall (F : Frame) =
  struct
    let (|+|) (x1, y1) (x2, y2) = (x1 +. x2, y1 +. y2)
    let (|*|) k (x, y) = (k *. x, k *. y)

    let integre dt flux =
      let init = (0., 0.) in
      let rec acc =
        Tick (lazy (Some (init, Flux.map2 (fun a f -> a |+| (dt |*| f)) acc flux)))
      in acc

    let g = 9.81;;
    (* r = r0 + Integ dr
       dr = dr0 + Integ ddr
       ddr = 0, -g
     *)

    let run (position0, vitesse0) =
      let acceleration = Flux.constant (0., -. g) in
      let vitesse      = Flux.(map2 ( |+| ) (constant vitesse0) (integre F.dt acceleration)) in
      let position     = Flux.(map2 ( |+| ) (constant position0) (integre F.dt vitesse)) in
      Flux.map2 (fun a b -> (a, b)) position vitesse
  end

module Bouncing (F : Frame) =
  struct
    (* version avec unfold sans récursivité directe *)
    let unless flux cond f_cond =
      Flux.unfold (fun (init, f) ->
          match Flux.uncons f with
               | None         -> None
               | Some (v, f') -> if not (init && cond v)
                                 then Some (v, (init, f'))
                                 else match Flux.uncons (f_cond v) with
                                      | None         -> None
                                      | Some (v, f') -> Some (v, (false, f'))
        ) (true, flux)

    (* version avec récursivité, donc paresse explicite *)
    let rec unless flux cond f_cond =
      Tick (lazy (
                match Flux.uncons flux with
                | None        -> None
                | Some (t, q) -> if cond t then Flux.uncons (f_cond t) else Some (t, unless q cond f_cond)
        ))

    let contact_1d (infx, supx) x dx = (x <= infx && dx < 0.) || (x >= supx && dx > 0.)

    let rebond ((x, y), (dx, dy)) =
      (x, y),
      ((if contact_1d F.box_x x dx then -. dx else dx),
       (if contact_1d F.box_y y dy then -. dy else dy))

    let contact ((x, y), (dx, dy)) = contact_1d F.box_x x dx || contact_1d F.box_y y dy

    module FF = FreeFall (F)

    let rec run etat0 =
      unless (FF.run etat0) contact (fun etat -> run (rebond etat))
  end

module Init =
  struct
    let dt = 0.01
    let box_x = (0., 800.)
    let box_y = (0., 600.)
  end

module Draw = Drawing (Init)
module Bounce = Bouncing (Init)

let _  =
  let position0 = (300., 400.) in
  let vitesse0 = (25., 15.) in
  Draw.draw (Bounce.run (position0, vitesse0));;
```