Functional Programming

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Lecture 10: FRP

Zippers. Functional Reactive Programming. GUIs.

"*Zipper*" in *Haskell Wikibook* and "*The Zipper*" by Gerard Huet
"*How* froc *works*" by Jacob Donham
"*The Haskell School of Expression*" by Paul Hudak
"*Deprecating the Observer Pattern with* Scala.React" by Ingo Maier, Martin Odersky

If you see any error on the slides, let me know!

Zippers

- We would like to keep track of a position in a data structure: easily access and modify it at that location, easily move the location around.

- Recall how we have defined *context types* for datatypes: types that represent a data structure with one of elements stored in it missing.

```
type btree = Tip | Node of int * btree * btree
```

$$\begin{aligned} T &= 1 + xT^2 \\ \frac{\partial T}{\partial x} &= 0 + T^2 + 2\,x\,T\,\frac{\partial T}{\partial x} = TT + 2\,x\,T\,\frac{\partial T}{\partial x} \end{aligned}$$

```
type btree_dir = LeftBranch | RightBranch
type btree_deriv =
  | Here of btree * btree
  | Below of btree_dir * int * btree * btree_deriv
```

- **Location = context + subtree**! But there's a problem above.

- But we cannot easily move the location if `Here` is at the bottom.

  The part closest to the location should be on top.

- Revisiting equations for trees and lists:

$$
\begin{aligned}
T &= 1 + x\,T^2 \\
\frac{\partial T}{\partial x} &= 0 + T^2 + 2\,x\,T\,\frac{\partial T}{\partial x} \\
\frac{\partial T}{\partial x} &= \frac{T^2}{1 - 2\,x\,T} \\
L(y) &= 1 + y\,L(y) \\
L(y) &= \frac{1}{1 - y} \\
\frac{\partial T}{\partial x} &= T^2\,L(2\,x\,T)
\end{aligned}
$$

  I.e. the context can be stored as a list with the root as the last node.

  - Of course it doesn't matter whether we use built-in lists, or a type with `Above` and `Root` variants.

- Contexts of subtrees are more useful than of single elements.

```
type 'a tree = Tip | Node of 'a tree * 'a * 'a tree
type tree_dir = Left_br | Right_br
type 'a context = (tree_dir * 'a * 'a tree) list
type 'a location = {sub: 'a tree; ctx: 'a context}
let access {sub} = sub
let change {ctx} sub = {sub; ctx}
let modify f {sub; ctx} = {sub=f sub; ctx}
```

- We can imagine a location as a rooted tree, which is hanging pinned at one of its nodes. Let's look at pictures in http://en.wikibooks.org/wiki/Haskell/Zippers

- Moving around:

```
let ascend loc =
  match loc.ctx with
  | [] -> loc                                              Or raise exception.
  | (Left_br, n, l) :: up_ctx ->
    {sub=Node (l, n, loc.sub); ctx=up_ctx}
  | (Right_br, n, r) :: up_ctx ->
    {sub=Node (loc.sub, n, r); ctx=up_ctx}
let desc_left loc =
  match loc.sub with
  | Tip -> loc                                             Or raise exception.
  | Node (l, n, r) ->
    {sub=l; ctx=(Right_br, n, r)::loc.ctx}
let desc_right loc =
  match loc.sub with
  | Tip -> loc                                             Or raise exception.
  | Node (l, n, r) ->
    {sub=r; ctx=(Left_br, n, l)::loc.ctx}
```

- Following *The Zipper*, let's look at a tree with arbitrary number of branches.

```
type doc = Text of string | Line | Group of doc list
type context = (doc list * doc list) list
type location = {sub: doc; ctx: context}

let go_up loc =
  match loc.ctx with
  | [] -> invalid_arg "go_up: at top"
  | (left, right) :: up_ctx ->        Previous subdocument and its siblings.
    {sub=Group (List.rev left @ loc.sub::right); ctx=up_ctx}
let go_left loc =
  match loc.ctx with
  | [] -> invalid_arg "go_left: at top"
  | (l::left, right) :: up_ctx ->      Left sibling of previous subdocument.
    {sub=l; ctx=(left, loc.sub::right) :: up_ctx}
  | ([], _) :: _ -> invalid_arg "go_left: at first"
```

```
let go_right loc =
  match loc.ctx with
  | [] -> invalid_arg "go_right: at top"
  | (left, r::right) :: up_ctx ->
    {sub=r; ctx=(loc.sub::left, right) :: up_ctx}
  | (_, []) :: _ -> invalid_arg "go_right: at last"
```

Go to the first (i.e. leftmost) subdocument.

```
let go_down loc =
  match loc.sub with
  | Text _ -> invalid_arg "go_down: at text"
  | Line -> invalid_arg "go_down: at line"
  | Group [] -> invalid_arg "go_down: at empty"
  | Group (doc::docs) -> {sub=doc; ctx=([], docs)::loc.ctx}
```

Example: Context rewriting

- Our friend working on the string theory asked us for help with simplifying his equations.
- The task is to pull out particular subexpressions as far to the left as we can, but changing the whole expression as little as possible.
- We can illustrate our algorithm using mathematical notation. Let:
  - x be the thing we pull out
  - $C[e]$ and $D[e]$ be big expressions with subexpression e
  - operator $\circ$ stand for one of: $*, +$

$$
\begin{aligned}
D[(C[x] \circ e_1) \circ e_2] &\Rightarrow D[C[x] \circ (e_1 \circ e_2)] \\
D[e_2 \circ (C[x] \circ e_1)] &\Rightarrow D[C[x] \circ (e_1 \circ e_2)] \\
D[(C[x] + e_1)\, e_2] &\Rightarrow D[C[x]\, e_2 + e_1\, e_2] \\
D[e_2\, (C[x] + e_1)] &\Rightarrow D[C[x]\, e_2 + e_1\, e_2] \\
D[e \circ C[x]] &\Rightarrow D[C[x] \circ e]
\end{aligned}
$$

- First the groundwork:

```
type op = Add | Mul
type expr = Val of int | Var of string | App of expr*op*expr
type expr_dir = Left_arg | Right_arg
type context = (expr_dir * op * expr) list
type location = {sub: expr; ctx: context}
```

- Locate the subexpression described by p.

```
let rec find_aux p e =
  if p e then Some (e, [])
  else match e with
  | Val _ | Var _ -> None
  | App (l, op, r) ->
    match find_aux p l with
    | Some (sub, up_ctx) ->
      Some (sub, (Right_arg, op, r)::up_ctx)
    | None ->
      match find_aux p r with
      | Some (sub, up_ctx) ->
        Some (sub, (Left_arg, op, l)::up_ctx)
      | None -> None

let find p e =
  match find_aux p e with
  | None -> None
  | Some (sub, ctx) -> Some {sub; ctx=List.rev ctx}
```

- Pull-out the located subexpression.

```
let rec pull_out loc =
  match loc.ctx with
  | [] -> loc                                                Done.
  | (Left_arg, op, l) :: up_ctx ->                           $D[e \circ C[x]] \Rightarrow D[C[x] \circ e]$
    pull_out {loc with ctx=(Right_arg, op, l) :: up_ctx}
  | (Right_arg, op1, e1) :: (_, op2, e2) :: up_ctx
      when op1 = op2 ->     $D[(C[x] \circ e_1) \circ e_2] / D[e_2 \circ (C[x] \circ e_1)] \Rightarrow D[C[x] \circ (e_1 \circ e_2)]$
    pull_out {loc with ctx=(Right_arg, op1, App(e1,op1,e2)) :: up_ctx}
  | (Right_arg, Add, e1) :: (_, Mul, e2) :: up_ctx ->
    pull_out {loc with ctx=   $D[(C[x] + e_1) e_2] / D[e_2 (C[x] + e_1)] \Rightarrow D[C[x] e_2 + e_1 e_2]$
        (Right_arg, Mul, e2) ::
          (Right_arg, Add, App(e1,Mul,e2)) :: up_ctx}
  | (Right_arg, op, r)::up_ctx ->                            Move up the context.
    pull_out {sub=App(loc.sub, op, r); ctx=up_ctx}
```

- Since operators are commutative, we ignore the direction for the second piece of context above.

- Test:

```
let (+) a b = App (a, Add, b)
let ( * ) a b = App (a, Mul, b)
let (!) a = Val a
let x = Var "x"
let y = Var "y"
let ex = !5 + y * (!7 + x) * (!3 + y)
let loc = find (fun e->e=x) ex
let sol =
  match loc with
  | None -> raise Not_found
  | Some loc -> pull_out loc
# let _ = expr2str sol;;
- : string = "(((x*y)*(3+y))+(((7*y)*(3+y))+5))"
```

- For best results we can iterate the `pull_out` function until fixpoint.

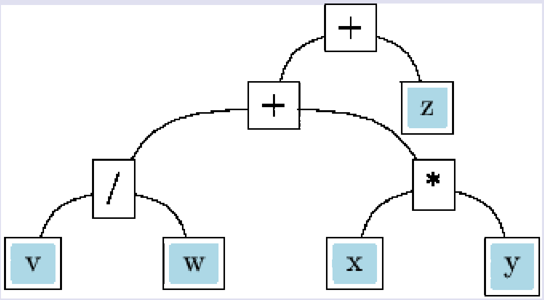
Adaptive Programming aka.Incremental Computing

- Zippers are somewhat unnatural.
- Once we change the data-structure, it is difficult to propagate the changes – need to rewrite all algorithms to work on context changes.
- In *Adaptive Programming*, aka. *incremental computation*, aka. *self-adjusting computation*, we write programs in straightforward functional manner, but can later modify any data causing only minimal amount of work required to update results.
- The functional description of computation is within a monad.
- We can change monadic values – e.g. parts of input – from outside and propagate the changes.
  - In the *Froc* library, the monadic *changeables* are `'a Froc_sa.t`, and the ability to modify them is exposed by type `'a Froc_sa.u` – the *writeables*.

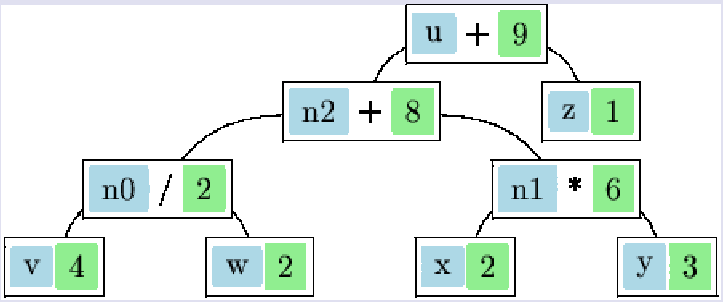
Dependency Graphs (explained by Jake Dunham)

- The monadic value `'a changeable` will be the *dependency graph* of the computation of the represented value `'a`.
- Let's look at the example in *"How froc works"*, representing computation

```
let u = v / w + x * y + z
```

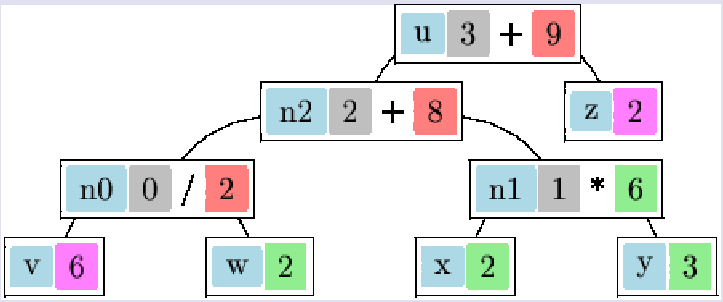


- and its state with partial results memoized



where `n0, n1, n2` are interior nodes of computation.

- Modify inputs v and z simultaneously



- We need to update n2 before u.

- We use the gray numbers – the order of computation – for the order of update of n0, n2 and u.

- Similarly to parallel in the concurrency monad, we provide bind2, bind3, ... – and corresponding lift2, lift3, ... – to introduce nodes with several children.

```
let n0 = bind2 v w (fun v w -> return (v / w))
let n1 = bind2 x y (fun x y -> return (x * y))
let n2 = bind2 n0 n1 (fun n0 n1 -> return (n0 + n1))
let u = bind2 n2 z (fun n2 z -> return (n2 + z))
```

- Do-notation is not necessary to have readable expressions.

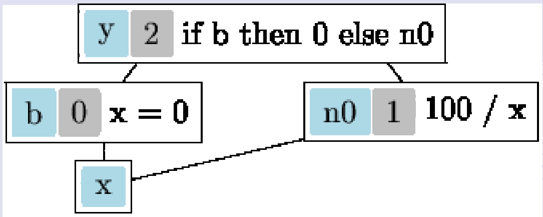
```
let (/) = lift2 (/)
let ( * ) = lift2 ( * )
let (+) = lift2 (+)
let u = v / w + x * y + z
```

- As in other monads, we can decrease overhead by using bigger chunks.

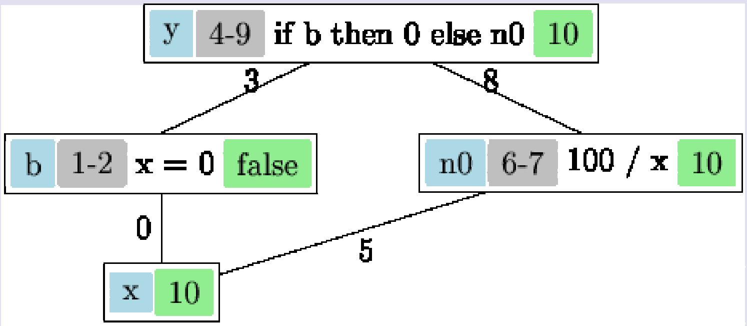
```
let n0 = blift2 v w (fun v w -> v / w)
let n2 = blift3 n0 x y (fun n0 x y -> n0 + x * y)
let u = blift2 n2 z (fun n2 z -> n2 + z)
```

- We have a problem if we recompute all nodes by order of computation.

```
let b = x >>= fun x -> return (x = 0)
let n0 = x >>= fun x -> return (100 / x)
let y = bind2 b n0 (fun b n0->if b then return 0 else n0)
```



- Rather than a signle "time" stamp, we store intervals: begin and end of computation



- When updating the y node, we first detach nodes in range 4-9 from the graph.
  - Computing the expression will re-attach the nodes as needed.
- When value of b does not change, then we skip updating y and proceed with updating n0.
  - I.e. no children of y with time stamp smaller than y change.
  - The value of y is a link to the value of n0 so it will change anyway.
- We need memoization to re-attach the same nodes in case they don't need updating.
  - Are they up-to-date? Run updating past the node's timestamp range.

Example using *Froc*

- Download *Froc* from https://github.com/jaked/froc/downloads
- Install for example with

  ```
  cd froc-0.2a; ./configure; make all; sudo make install
  ```

- `Froc_sa` (for *self-adjusting*) exports the monadic type `t` for changeable computation, and a handle type `u` for updating the computation.

-
  ```
  open Froc_sa
  type tree =
  | Leaf of int * int
  | Node of int * int * tree t * tree t
  ```

  Binary tree with nodes storing their screen location.

  We will grow the tree
  by modifying subtrees.

-
  ```
  let rec display px py t =
    match t with
    | Leaf (x, y) ->
      return
        (Graphics.draw_poly_line [|px,py;x,y|];
         Graphics.draw_circle x y 3)
    | Node (x, y, l, r) ->
      return (Graphics.draw_poly_line [|px,py;x,y|])
      >>= fun _ -> l >>= display x y
      >>= fun _ -> r >>= display x y
  ```

  Displaying the tree is changeable effect: whenever the tree changes, displaying will be updated. Only new nodes will be drawn after update.

  We return a throwaway value.

-
  ```
  let grow_at (x, depth, upd) =
    let x_l = x-f2i (width*.(2.0**(~-.(i2f (depth+1))))) in
    let l, upd_l = changeable (Leaf (x_l, (depth+1)*20)) in
    let x_r = x+f2i (width*.(2.0**(~-.(i2f (depth+1))))) in
    let r, upd_r = changeable (Leaf (x_r, (depth+1)*20)) in
    write upd (Node (x, depth*20, l, r));
    propagate ();
    [x_l, depth+1, upd_l; x_r, depth+1, upd_r]
  ```

  Update the old leaf and keep handles to make future updates.

-
  ```
  let rec loop t subts steps =
    if steps <= 0 then ()
    else loop t (concat_map grow_at subts) (steps-1)
  let incremental steps () =
    Graphics.open_graph " 1024x600";
    let t, u = changeable (Leaf (512, 20)) in
    let d = t >>= display (f2i (width /. 2.)) 0 in    Display once
    loop t [512, 1, u] steps;        – new nodes will be drawn automatically.
    Graphics.close_graph ();;
  ```

- Compare with rebuilding and redrawing the whole tree. Unfortunately the overhead of incremental computation is quite large. Byte code run:

depth	12	13	14	15	16	17	18	19	20
incremental	0.66s	1s	2.2s	4.4s	9.3s	21s	50s	140s	255s
rebuilding	0.5s	0.63s	1.3s	3s	5.3s	13s	39s	190s	∞

Functional Reactive Programming

- FRP is an attempt to declaratively deal with time.
- *Behaviors* are functions of time.
  - A behavior has a specific value in each instant.
- *Events* are sets of (time, value) pairs.
  - I.e. they are organised into streams of actions.
- Two problems
  - Behaviors / events are well defined when they don't depend on future
  - Efficiency: minimize overhead
- FRP is *synchronous*: it is possible to set up for events to happen at the same time, and it is *continuous*: behaviors can have details at arbitrary time resolution.
  - Although the results are *sampled*, there's no fixed (minimal) time step for specifying behavior.
  - Asynchrony refers to various ideas so ask what people mean.

- Ideally we would define:

```
type time = float
type 'a behavior = time -> 'a                    Arbitrary function.
type 'a event = ('a, time) stream        Increasing time instants.
```

- Forcing a lazy list (stream) of events would wait till an event arrives.
- But behaviors need to react to external events:

```
type user_action =
| Key of char * bool
| Button of int * int * bool * bool
| MouseMove of int * int
| Resize of int * int
type 'a behavior = user_action event -> time -> 'a
```

- Scanning through an event list since the beginnig of time till current time, each time we evaluate a behavior – very wasteful wrt. time&space.

  Producing a stream of behaviors for the stream of time allows to forget about events already in the past.

```
type 'a behavior =
  user_action event -> time stream -> 'a stream
```

- Next optimization is to pair user actions with sampling times.

  ```
  type 'a behavior =
    (user_action option * time) stream -> 'a stream
  ```

  `None` action corresponds to sampling time when nothing happens.

- Turning behaviors and events from functions of time into input-output streams is similar to optimizing interesction of ordered lists from $O(m\,n)$ to $O(m+n)$ time.

- Now we can in turn define events in terms of behaviors:

  ```
  type 'a event = 'a option behavior
  ```

  although it betrays the discrete character of events (happening at points in time rather than varying over intervals of time).

- We've gotten very close to *stream processing* as discussed in lecture 7.
  - Recall the incremental pretty-printing example that can "react" to more input.
  - Stream combinators, *fork* from exercise 9 for lecture 7, and a corresponding *merge*, turn stream processing into *synchronous discrete reactive programming*.

- Behaviors are monadic (but see next point) – in original specification:

  ```
  type 'a behavior = time -> 'a
  val return : 'a -> 'a behavior
  let return a = fun _ -> a
  val bind :
    'a behavior -> ('a -> 'b behavior) -> 'b behavior
  let bind a f = fun t -> f (a t) t
  ```

- As we've seen with changeables, we mostly use lifting. In Haskell world we'd call behaviors *applicative*. To build our own lifters in any monad:

  ```
  val ap : ('a -> 'b) monad -> 'a monad -> 'b monad
  let ap fm am = perform
    f <-- fm;
    a <-- am;
    return (f a)
  ```

  - Note that for changeables, the naive implementation above will introduce unnecessary dependencies. Monadic libraries for *incremental computing* or FRP should provide optimized variants if needed.
    - Compare with `parallel` for concurrent computing.

- Going from events to behaviors. `until` and `switch` have type

  ```
  'a behavior -> 'a behavior event -> 'a behavior
  ```

  `step` has type

  ```
  'a -> 'a event -> 'a behavior
  ```

  - `until b es` behaves as `b` until the first event in `es`, then behaves as the behavior in that event
  - `switch b es` behaves as the behavior from the last event in `es` prior to current time, if any, otherwise as `b`
  - `step a b` starts with behavior returning `a` and then switches to returning the value of the last event in `b` (prior to current time) – a *step function*.
- We will use "*signal*" to refer to a behavior or an event. But often "signal" is used as our behavior (check terminology when looking at a new FRP library).

Reactivity by Stream Processing

- The stream processing infrastructure should be familiar.

```
type ’a stream = ’a stream_ Lazy.t
and ’a stream_ = Cons of ’a * ’a stream
let rec lmap f l = lazy (
  let Cons (x, xs) = Lazy.force l in
  Cons (f x, lmap f xs))
let rec liter (f : ’a -> unit) (l : ’a stream) : unit =
  let Cons (x, xs) = Lazy.force l in
  f x; liter f xs
let rec lmap2 f xs ys = lazy (
  let Cons (x, xs) = Lazy.force xs in
  let Cons (y, ys) = Lazy.force ys in
  Cons (f x y, lmap2 f xs ys))
let rec lmap3 f xs ys zs = lazy (
  let Cons (x, xs) = Lazy.force xs in
  let Cons (y, ys) = Lazy.force ys in
  let Cons (z, zs) = Lazy.force zs in
  Cons (f x y z, lmap3 f xs ys zs))
let rec lfold acc f (l : ’a stream) = lazy (
  let Cons (x, xs) = Lazy.force l in
  let acc = f acc x in
  Cons (acc, lfold acc f xs))
```

Fold a function over the stream producing a stream of partial results.

- Since a behavior is a function of user actions and sample times, we need to ensure that only one stream is created for the actual input stream.

```
type (’a, ’b) memo1 =
  {memo_f : ’a -> ’b; mutable memo_r : (’a * ’b) option}
let memo1 f = {memo_f = f; memo_r = None}
let memo1_app f x =
  match f.memo_r with
  | Some (y, res) when x == y -> res
  | _ ->
    let res = f.memo_f x in
    f.memo_r <- Some (x, res);
    res
let ($) = memo1_app
type ’a behavior =
  ((user_action option * time) stream, ’a stream) memo1
```

Physical equality is OK – external input is “physically” unique. While debugging, we can monitor whether `f.memo_r = None` before.

- The monadic/applicative functions to build complex behaviors.
  - ○ If you do not provide type annotations in `.ml` files, work together with an `.mli` file to catch problems early. You can later add more type annotations as needed to find out what's wrong.

```
let returnB x : 'a behavior =
  let rec xs = lazy (Cons (x, xs)) in
  memo1 (fun _ -> xs)
let ( !* ) = returnB
let liftB f fb = memo1 (fun uts -> lmap f (fb $ uts))
let liftB2 f fb1 fb2 = memo1
  (fun uts -> lmap2 f (fb1 $ uts) (fb2 $ uts))
let liftB3 f fb1 fb2 fb3 = memo1
  (fun uts -> lmap3 f (fb1 $ uts) (fb2 $ uts) (fb3 $ uts))
let liftE f (fe : 'a event) : 'b event = memo1
  (fun uts -> lmap
    (function Some e -> Some (f e) | None -> None)
    (fe $ uts))
let (=>>) fe f = liftE f fe
let (->>) e v = e =>> fun _ -> v
```

- Creating events out of behaviors.

```
let whileB (fb : bool behavior) : unit event =
  memo1 (fun uts ->
    lmap (function true -> Some () | false -> None)
      (fb $ uts))
let unique fe : ’a event =
  memo1 (fun uts ->
    let xs = fe $ uts in
    lmap2 (fun x y -> if x = y then None else y)
      (lazy (Cons (None, xs))) xs)
let whenB fb =
  memo1 (fun uts -> unique (whileB fb) $ uts)
let snapshot fe fb : (’a * ’b) event =
  memo1 (fun uts -> lmap2
    (fun x->function Some y -> Some (y,x) | None -> None)
      (fb $ uts) (fe $ uts))
```

- Creating behaviors out of events.

The step function: value of last event.

```
let step acc fe =
 memo1 (fun uts -> lfold acc
   (fun acc -> function None -> acc | Some v -> v)
   (fe $ uts))
```

Transform a value by a series of functions.

```
let step_accum acc ff =
 memo1 (fun uts ->
   lfold acc (fun acc -> function
   | None -> acc | Some f -> f acc)
    (ff $ uts))
```

- To numerically integrate a behavior, we need to access the sampling times.

```
let integral fb =
  let rec loop t0 acc uts bs =
    let Cons ((_,t1), uts) = Lazy.force uts in
    let Cons (b, bs) = Lazy.force bs in
    let acc = acc +. (t1 -. t0) *. b in   $b = \mathrm{fb}(t_1), \mathrm{acc} \approx \int_{t \leqslant t_0} f.$
    Cons (acc, lazy (loop t1 acc uts bs)) in
  memo1 (fun uts -> lazy (
    let Cons ((_,t), uts') = Lazy.force uts in
    Cons (0., lazy (loop t 0. uts' (fb $ uts)))))
```

  - In our *paddle game* example, we paradoxically express position and velocity in mutually recursive manner. The trick is the same as in chapter 7 – integration introduces one step of delay.

- User actions:

```
let lbp : unit event =
  memo1 (fun uts -> lmap
    (function Some(Button(_,_)), _ -> Some() | _ -> None)
    uts)
let mm : (int * int) event =
  memo1 (fun uts -> lmap
  (function Some(MouseMove(x,y)),_ ->Some(x,y) | _ ->None)
    uts)
let screen : (int * int) event =
  memo1 (fun uts -> lmap
    (function Some(Resize(x,y)),_ ->Some(x,y) | _ ->None)
    uts)
let mouse_x : int behavior = step 0 (liftE fst mm)
let mouse_y : int behavior = step 0 (liftE snd mm)
let width : int behavior = step 640 (liftE fst screen)
let height : int behavior = step 512 (liftE snd screen)
```

The Paddle Game example

- A *scene graph* is a data structure that represents a “world” which can be drawn on screen.

```
type scene =
| Rect of int * int * int * int                    position, width, height
| Circle of int * int * int                                position, radius
| Group of scene list
| Color of Graphics.color * scene                color of subscene objects
| Translate of float * float * scene          additional offset of origin
```

- Drawing a scene explains what we mean above.

```
let draw sc =
  let f2i = int_of_float in
  let open Graphics in
  let rec aux t_x t_y = function
  | Rect (x, y, w, h) ->
    fill_rect (f2i t_x+x) (f2i t_y+y) w h
  | Circle (x, y, r) ->
    fill_circle (f2i t_x+x) (f2i t_y+y) r
  | Group scs ->
    List.iter (aux t_x t_y) scs
  | Color (c, sc) -> set_color c; aux t_x t_y sc
  | Translate (x, y, sc) -> aux (t_x+.x) (t_y+.y) sc in
  clear_graph ();
  aux 0. 0. sc;
  synchronize ()
```

Accumulate translations.

↙ Set color for sc objects.

"Fast and clean" removing of previous picture.

Synchronize the *double buffer* – avoiding flickering.

- An animation is a scene behavior. To animate it we need to create the input stream: the user actions and sampling times stream.
  - We could abstract away drawing from time sampling in `reactimate`, asking for (i.e. passing as argument) a producer of user actions and a consumer of scene graphs (like `draw`).

```
let reactimate (anim : scene behavior) =
  let open Graphics in
  let not_b = function Some (Button (_,_)) -> false | _ -> true in
  let current old_m old_scr (old_u, t0) =
    let rec delay () =
      let t1 = Unix.gettimeofday () in
      let d = 0.01 -. (t1 -. t0) in
      try if d > 0. then Thread.delay d;
          Unix.gettimeofday ()
      with Unix.Unix_error ((* Unix.EAGAIN *)_, _, _) -> delay () in
    let t1 = delay () in
    let s = Graphics.wait_next_event [Poll] in
    let x = s.mouse_x and y = s.mouse_y
    and scr_x = Graphics.size_x () and scr_y = Graphics.size_y () in
  let ue =
    if s.keypressed then Some (Key s.key)
    else if (scr_x, scr_y) <> old_scr then Some (Resize (scr_x,scr_y))
    else if s.button && not_b old_u then Some (Button (x, y))
    else if (x, y) <> old_m then Some (MouseMove (x, y))
    else None in
  (x, y), (scr_x, scr_y), (ue, t1) in
open_graph "";                                                Open window.
display_mode false;                         Draw using double buffering.
let t0 = Unix.gettimeofday () in
let rec utstep mpos scr ut = lazy (
  let mpos, scr, ut = current mpos scr ut in
  Cons (ut, utstep mpos scr ut)) in
let scr = Graphics.size_x (), Graphics.size_y () in
let ut0 = Some (Resize (fst scr, snd scr)), t0 in
liter draw (anim $ lazy (Cons (ut0, utstep (0,0) scr ut0)));
close_graph ()                Close window – unfortunately never happens.
```

- General-purpose behavior operators.

```
let (+*) = liftB2 (+)
let (-*) = liftB2 (-)
let ( *** ) = liftB2 ( * )
let (/*) = liftB2 (/)
let (&&*) = liftB2 (&&)
let (||) = liftB2 (||)
let (<*) = liftB2 (<)
let (>*) = liftB2 (>)
```

- The walls are drawn on left, top and right borders of the window.

```
let walls =
  liftB2 (fun w h -> Color (Graphics.blue, Group
    [Rect (0, 0, 20, h-1); Rect (0, h-21, w-1, 20);
     Rect (w-21, 0, 20, h-1)]))
    width height
```

- The paddle is tied to the mouse at the bottom border of the window.

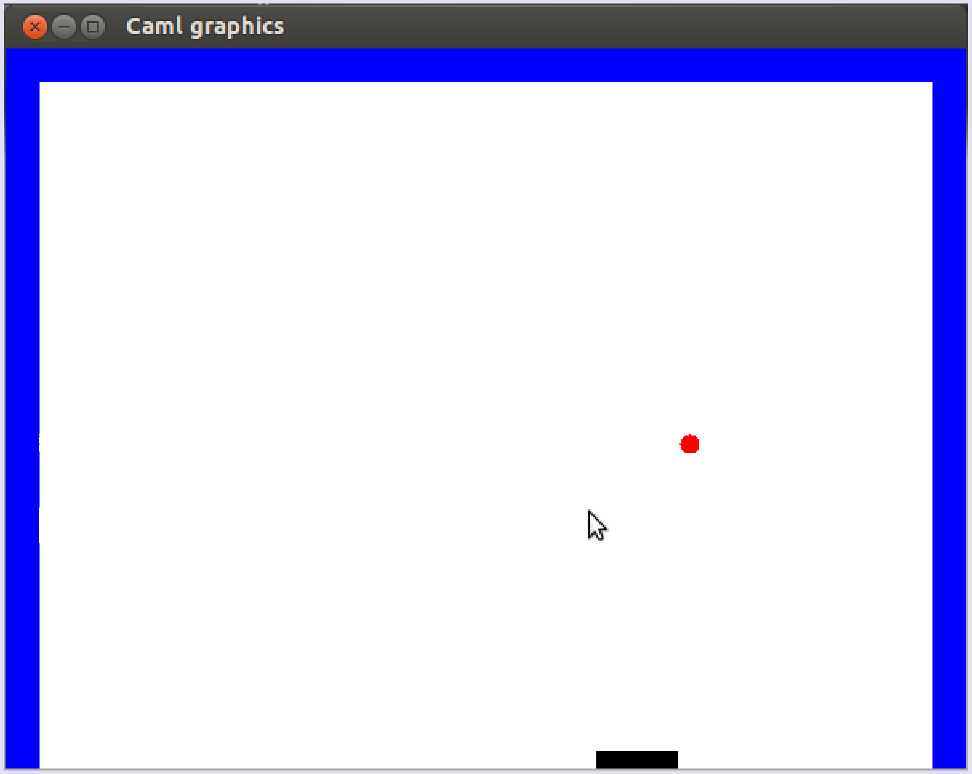
```
let paddle = liftB (fun mx ->
  Color (Graphics.black, Rect (mx, 0, 50, 10))) mouse_x
```

- The ball has a velocity in pixels per second. It bounces from the walls, which is hard-coded in terms of distance from window borders.
  - Unfortunately OCaml, being an eager language, does not let us encode recursive behaviors in elegant way. We need to unpack behaviors and events as functions of the input stream.
  - `xbounce ->> (~-.)` event is just the negation function happening at each horizontal bounce.
  - `step_accum vel (xbounce ->> (~-.))` behavior is `vel` value changing sign at each horizontal bounce.
  - `liftB int_of_float (integral xvel) +* width /* !*2` – first integrate velocity, then truncate it to integers and offset to the middle of the window.
  - `whenB ((xpos >* width -* !*27) ||* (xpos <* !*27))` – issue an event the first time the position exceeds the bounds. This ensures there are no further bouncings until the ball moves out of the walls.

```
let pbal vel =
  let rec xvel_ uts =
    step_accum vel (xbounce ->> (~-.)) $ uts
  and xvel = {memo_f = xvel_; memo_r = None}
  and xpos_ uts =
    (liftB int_of_float (integral xvel) +* width /* !*2) $ uts
  and xpos = {memo_f = xpos_; memo_r = None}
  and xbounce_ uts = whenB
    ((xpos >* width -* !*27) ||* (xpos <* !*27)) $ uts
  and xbounce = {memo_f = xbounce_; memo_r = None} in
  let rec yvel_ uts =
    (step_accum vel (ybounce ->> (~-.))) $ uts
  and yvel = {memo_f = yvel_; memo_r = None}
  and ypos_ uts =
    (liftB int_of_float (integral yvel) +* height /* !*2) $ uts
  and ypos = {memo_f = ypos_; memo_r = None}
  and ybounce_ uts = whenB (
    (ypos >* height -* !*27) ||*
      ((ypos <* !*17) &&* (ypos >* !*7) &&*
          (xpos >* mouse_x) &&* (xpos <* mouse_x +* !*50))) $ uts
  and ybounce = {memo_f = ybounce_; memo_r = None} in
  liftB2 (fun x y -> Color (Graphics.red, Circle (x, y, 6)))
    xpos ypos
```

- Invocation:

```
ocamlbuild Lec10b.native -cflags -I,+threads
  -libs graphics,unix,threads/threads --
```

- 

Reactivity by Incremental Computing

- In *Froc* behaviors and events are both implemented as changeables but only behaviors persist, events are "instantaneous".
    - Behaviors are composed out of constants and prior events, capture the "changeable" aspect.
    - Events capture the "writeable" aspect – after their values are propagated, the values are removed.

  Events and behaviors are called *signals*.
- *Froc* does not represent time, and provides the function `changes : 'a behavior -> 'a event`, which violates the continuous semantics we introduced before.
    - It breaks the illusion that behaviors vary continuously rather than at discrete points in time.
    - But it avoids the need to synchronize global time samples with events in the system. It is "less continuous but more dense".

- Sending an event – `send` – starts an *update cycle*. Signals cannot call `send`, but can `send_deferred` which will send an event in next cycle.
  - Things that happen in the same update cycle are *simultaneous*.
  - Events are removed (detached from dependency graph) after an update cycle.
- *Froc* provides the `fix_b`, `fix_e` functions to define signals recursively. Current value refers to value from previous update cycle, and defers next recursive step to next cycle, until convergence.
- Update cycles can happen "back-to-back" via `send_deferred` and `fix_b`, `fix_e`, or can be invoked from outside *Froc* by sending events at arbitrary times.
  - With a `time` behavior that holds a `clock` event value, events from "back-to-back" update cycles can be at the same clock time although not simultaneous in this sense.
  - Update cycles prevent *glitches*, where outdated signal is used e.g. to issue an event.

- Let's familiarize ourselves with *Froc* API: http://jaked.github.com/froc/doc/Froc.html
- A behavior is written in *pure style*, when its definition does not use `send`, `send_deferred`, `notify_e`, `notify_b` and `sample`:
  - `sample`, `notify_e`, `notify_b` are used from outside the behavior (from its "environment") analogously to observing result of a function,
  - `send`, `send_deferred` are used from outside analogously to providing input to a function.
- We will develop an example in a pragmatic, *impure* style, but since purity is an important aspect of functional programming, I propose to rewrite it in pure style as an exercise (ex. 5).
- When writing in impure style we need to remember to refer from somewhere to all the pieces of our behavior, otherwise the unreferred parts will be **garbage collected** breaking the behavior.
  - A value is referred to, when it has a name in the global environment or is part of a bigger value that is referred to (for example it's stored somewhere). Signals can be referred to by being part of the dependency graph, but also by any of the more general ways.

Reimplementing the Paddle Game example

- Rather than following our incremental computing example (a scene with changeable parts), we follow our FRP example: a scene behavior.
- First we introduce time:

```
open Froc
let clock, tick = make_event ()
let time = hold (Unix.gettimeofday ()) clock
```

- Next we define integration:

```
let integral fb =
  let aux (sum, t0) t1 =
    sum +. (t1 -. t0) *. sample fb, t1 in
  collect_b aux (0., sample time) clock
```

For convenience, the integral remembers the current upper limit of integration. It will be useful to get the integer part:

```
let integ_res fb =
  lift (fun (v,_) -> int_of_float v) (integral fb)
```

- We can also define integration in pure style:

  ```
  let pair fa fb = lift2 (fun x y -> x, y) fa fb
  let integral_nice fb =
    let samples = changes (pair fb time) in
    let aux (sum, t0) (fv, t1) =
      sum +. (t1 -. t0) *. fv, t1 in
    collect_b aux (0., sample time) samples
  ```

  The initial value `(0., sample time)` is not “inside” the behavior so `sample` here does not spoil the pure style.

- The `scene` datatype and how we `draw` a scene does not change.

- Signals which will be sent to behaviors:

```
let mouse_move_x, move_mouse_x = make_event ()
let mouse_move_y, move_mouse_y = make_event ()
let mouse_x = hold 0 mouse_move_x
let mouse_y = hold 0 mouse_move_x
let width_resized, resize_width = make_event ()
let height_resized, resize_height = make_event ()
let width = hold 640 width_resized
let height = hold 512 height_resized
let mbutton_pressed, press_mbutton = make_event ()
let key_pressed, press_key = make_event ()
```

- The user interface main loop, emiting signals and observing behaviors:

```
let reactimate (anim : scene behavior) =
  let open Graphics in
  let rec loop omx omy osx osy omb t0 =
    let rec delay () =
      let t1 = Unix.gettimeofday () in
      let d = 0.01 -. (t1 -. t0) in
      try if d > 0. then Thread.delay d;
          Unix.gettimeofday ()
      with Unix.Unix_error ((* Unix.EAGAIN *)_, _, _) -> delay () in
  let t1 = delay () in
  let s = Graphics.wait_next_event [Poll] in
  let x = s.mouse_x and y = s.mouse_y
  and scr_x = Graphics.size_x () and scr_y = Graphics.size_y () in
  if s.keypressed then send press_key s.key;
  if scr_x <> osx then send resize_width scr_x;
  if scr_y <> osy then send resize_height scr_y;
  if s.button && not omb then send press_mbutton ();
  if x <> omx then send move_mouse_x x;
  if y <> omy then send move_mouse_y y;
  send tick t1;
  draw (sample anim);
  loop x y scr_x scr_y s.button t1 in
open_graph "";
display_mode false;
loop 0 0 640 512 false (Unix.gettimeofday ());
close_graph ()
```

We can send signals one by one.

After all signals are updated, observe behavior.

- The simple behaviors as in `Lec10b.ml`. Pragmatic (impure) bouncing:

```
let pbal vel =
  let xbounce, bounce_x = make_event () in
  let ybounce, bounce_y = make_event () in
  let xvel = collect_b (fun v _ -> ~-.v) vel xbounce in
  let yvel = collect_b (fun v _ -> ~-.v) vel ybounce in
  let xpos = integ_res xvel +* width /* !*2 in
  let ypos = integ_res yvel +* height /* !*2 in
  let xbounce_ = when_true
    ((xpos >* width -* !*27) ||* (xpos <* !*27)) in
  notify_e xbounce_ (send bounce_x);
  let ybounce_ = when_true (
    (ypos >* height -* !*27) ||*
      ((ypos <* !*17) &&* (ypos >* !*7) &&*
          (xpos >* mouse_x) &&* (xpos <* mouse_x +* !*50))) in
  notify_e ybounce_ (send bounce_y);
  lift4 (fun x y _ _ -> Color (Graphics.red, Circle (x, y, 6)))
    xpos ypos (hold () xbounce_) (hold () ybounce_)
```

- We hold on to xbounce_ and ybounce_ above to prevent garbage collecting them. We could instead remember them in the "toplevel":

```
let pbal vel =
  ...
  xbounce_, ybounce_,
  lift2 (fun x y -> Color (Graphics.red, Circle (x, y, 6)))
    xpos ypos
let xb_, yb_, ball = pbal 100.
let game = lift3 (fun walls paddle ball ->
  Group [walls; paddle; ball]) walls paddle ball
```

- We can easily monitor signals while debugging, e.g.:

```
  notify_e xbounce (fun () -> Printf.printf "xbounce\n%!");
  notify_e ybounce (fun () -> Printf.printf "ybounce\n%!");
```

- Invocation:

```
ocamlbuild Lec10c.native -cflags -I,+froc,-I,+threads -libs
froc/froc,unix,graphics,threads/threads --
```

Direct Control

- Real-world behaviors often are *state machines*, going through several stages. We don't have declarative means for it yet.
  - Example: baking recipes. *1. Preheat the oven. 2. Put flour, sugar, eggs into a bowl. 3. Spoon the mixture.* etc.
- We want a *flow* to be able to proceed through events: when the first event arrives we remember its result and wait for the next event, disregarding any further arrivals of the first event!
  - Therefore *Froc* constructs like mapping an event: `map`, or attaching a notification to a behavior change: `bind b1 (fun v1 -> notify_b ~now:false b2 (fun v2 -> …))`, will not work.
- We also want to be able to repeat or loop a flow, but starting from the notification of the first event that happens after the notification of the last event.

- `next e` is an event propagating only the first occurrence of `e`. This will be the basis of our `await` function.

- The whole flow should be cancellable from outside at any time.

- A flow is a kind of a *lightweight thread* as in end of lecture 8, we'll make it a monad. It only "stores" a non-unit value when it `await`s an event. But it has a primitive to `emit` values.

  - ○ We actually implement *coarse-grained* threads (lecture 8 exercise 11), with `await` in the role of `suspend`.

- We build a module Flow with monadic type ('a, 'b) flow “storing” 'b and emitting 'a.

```
type ('a, 'b) flow
type cancellable          A handle to cancel a flow (stop further computation).
val noop_flow : ('a, unit) flow                             Same as return ().
val return : 'b -> ('a, 'b) flow                                Completed flow.
val await : 'b Froc.event -> ('a, 'b) flow             Wait and store event:
val bind :                                          the principled way to input.
  ('a, 'b) flow -> ('b -> ('a, 'c) flow) -> ('a, 'c) flow
val emit : 'a -> ('a, unit) flow                   The principled way to output.
val cancel : cancellable -> unit
val repeat :                         Loop the given flow and store the stop event.
  ?until:'a Froc.event -> ('b, unit) flow -> ('b, 'a) flow
val event_flow :
  ('a, unit) flow -> 'a Froc.event * cancellable
val behavior_flow :     The initial value of a behavior and a flow to update it.
  'a -> ('a, unit) flow -> 'a Froc.behavior * cancellable
val is_cancelled : cancellable -> bool
```

- We follow our (or *Lwt*) implementation of lightweight threads, adapting it to the need of cancelling flows.

```
module F = Froc
type ’a result =
| Return of ’a                       ↓Notifications to cancel when cancelled.
| Sleep of (’a -> unit) list * F.cancel ref list
| Cancelled
| Link of ’a state
and ’a state = {mutable state : ’a result}
type cancellable = unit state
```

- Functions `find`, `wakeup`, `connect` are as in lecture 8 (but connecting to cancelled thread cancels the other thread).

- Our monad is actually a reader monad over the result state. The reader supplies the `emit` function. (See exercise 10.)

```
type (’a, ’b) flow = (’a -> unit) -> ’b state
```

- The return and bind functions are as in our lightweight threads, but we need to handle cancelled flows: for m = bind a b, if a is cancelled then m is cancelled, and if m is cancelled then don't wake up b:

```
let waiter x =
  if not (is_cancelled m)
  then connect m (b x emit) in
...
```

- await is implemented like next, but it wakes up a flow:

```
let await t = fun emit ->
  let c = ref F.no_cancel in
  let m = {state=Sleep ([], [c])} in
  c :=
    F.notify_e_cancel t begin fun r ->
      F.cancel !c;
      c := F.no_cancel;
      wakeup m r
    end;
  m
```

- `repeat` attaches the whole loop as a waiter for the loop body.

```
let repeat ?(until=F.never) fa =
  fun emit ->
    let c = ref F.no_cancel in
    let out = {state=Sleep ([], [c])} in
    let cancel_body = ref {state=Cancelled} in
    c := F.notify_e_cancel until begin fun tv ->
        F.cancel !c;
        c := F.no_cancel;
        cancel !cancel_body; wakeup out tv
      end;
    let rec loop () =
      let a = find (fa emit) in
      cancel_body := a;
      (match a.state with
      | Cancelled -> cancel out; F.cancel !c
      | Return x ->
        failwith "loop_until: not implemented for unsuspended flows"
      | Sleep (xwaiters, xcancels) ->
        a.state <- Sleep (loop::xwaiters, xcancels)
      | Link _ -> assert false) in
    loop (); out
```

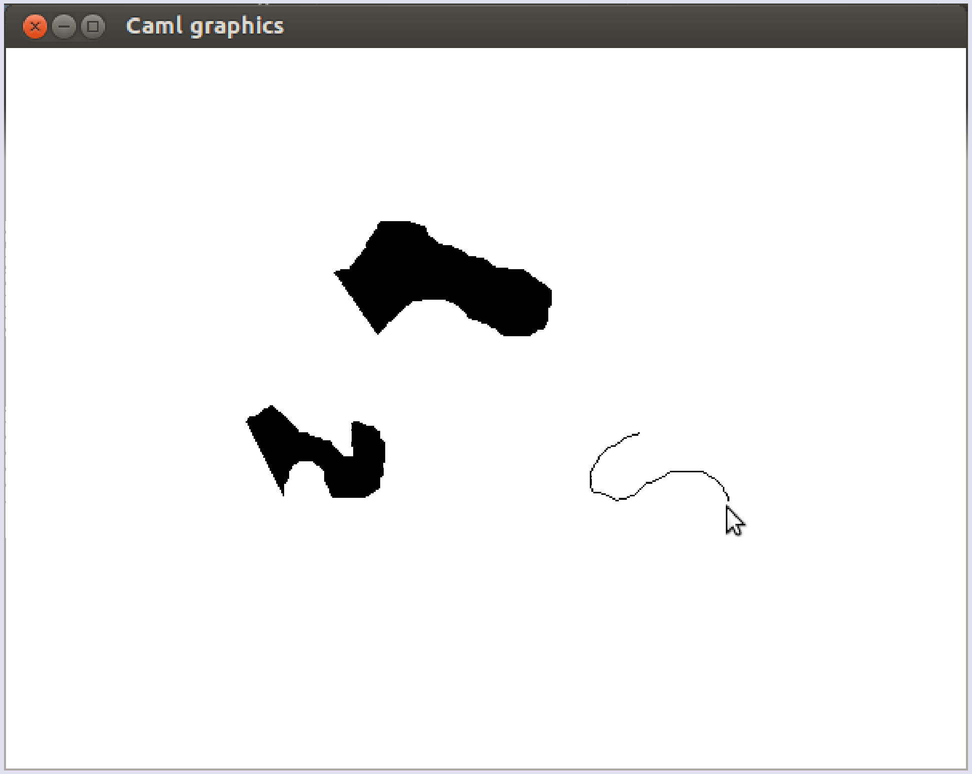
Exiting the loop consists of cancelling the loop body and waking up loop waiters.

- Example: drawing shapes. Invocation:
  `ocamlbuild Lec10d.native -pp "camlp4o monad/pa_monad.cmo" -libs froc/froc,graphics -cflags -I,+froc --`
- The event handlers and drawing/event dispatch loop `reactimate` is similar to the paddle game example (we removed unnecessary events).
- The scene is a list of shapes, the first shape is open.

```ocaml
type scene = (int * int) list list
let draw sc =
  let open Graphics in
  clear_graph ();
  (match sc with
  | [] -> ()
  | opn::cld ->
    draw_poly_line (Array.of_list opn);
    List.iter (fill_poly -| Array.of_list) cld);
  synchronize ()
```

- We build a flow and turn it into a behavior to animate.

```
let painter =
  let cld = ref [] in                                   Global state of painter.
  repeat (perform
      await mbutton_pressed;                            Start when button down.
      let opn = ref [] in
      repeat (perform
          mpos <-- await mouse_move;              ↙Add next position to line.
          emit (opn := mpos :: !opn; !opn :: !cld))
        ~until:mbutton_released;                        ↙Start new shape.
      emit (cld := !opn :: !cld; opn := []; [] :: !cld))
let painter, cancel_painter = behavior_flow [] painter
let () = reactimate painter
```

- 
Caml graphics

Flows and state

Global state and thread-local state can be used with lightweight threads, but pay attention to semantics – which computations are inside the monad and which while building the initial monadic value.

- Side effects hidden in `return` and `emit` arguments are not inside the monad. E.g. if in the “first line” of a loop effects are executed only at the start of the loop – but if after bind (“below first line” of a loop), at each step of the loop.

```
let f =
  repeat (perform
      emit (Printf.printf "[0]\n%!"; ’0’);
      () <-- await aas;
      emit (Printf.printf "[1]\n%!"; ’1’);
      () <-- await bs;
      emit (Printf.printf "[2]\n%!"; ’2’);
      () <-- await cs;
      emit (Printf.printf "[3]\n%!"; ’3’);
      () <-- await ds;
      emit (Printf.printf "[4]\n%!"; ’4’))
```

```
let e, cancel_e = event_flow f
let () =
  F.notify_e e (fun c -> Printf.printf "flow: %c\n%!" c);
  Printf.printf "notification installed\n%!"
let () =
  F.send a (); F.send b (); F.send c (); F.send d ();
  F.send a (); F.send b (); F.send c (); F.send d ()
```

```
[0]                          Only printed once – when building the loop.
notification installed        Only installed after the first flow event sent.
event: a                         Event notification (see source Lec10e.ml).
[1]                          Second emit computed after first await returns.
flow: 1                                                      Emitted signal.
event: b                                                       Next event...
[2]
flow: 2
event: c
[3]
flow: 3
event: d
[4]
```

```
flow: 4
flow: 0
```

Last signal emitted from first turn of the loop – and first signal of the second turn (but [0] not printed).

```
event: a
[1]
flow: 1
event: b
[2]
flow: 2
event: c
[3]
flow: 3
event: d
[4]
flow: 4
flow: 0
```

Program ends while flow in third turn of the loop.

Graphical User Interfaces

- In-depth discussion of GUIs is beyond the scope of this course. We only cover what's needed for an example reactive program with direct control.

- Demo of libraries *LablTk* based on optional labelled arguments discussed in lecture 2 exercise 2, and polymorphic variants, and *LablGtk* additionally based on objects. We will learn more about objects and polymorphic variants in next lecture.

Calculator Flow

```
let digits, digit = F.make_event ()            We represent the mechanics
let ops, op = F.make_event ()              of the calculator directly as a flow.
let dots, dot = F.make_event ()
let calc =                 We need two state variables for two arguments of calculation
  let f = ref (fun x -> x) and now = ref 0.0 in                                 but we
  repeat (perform              remember the older argument in partial application.
      op <-- repeat
        (perform                              Enter the digits of a number (on later turns
            d <-- await digits;                           starting from the second digit)
            emit (now := 10. *. !now +. d; !now))
        ~until:ops;                                   until operator button is pressed.
      emit (now := !f !now; f := op !now; !now);
      d <-- repeat         ↖Compute the result and “store away” the operator.
        (perform op <-- await ops; return (f := op !now))
        ~until:digits;                             The user can pick a different operator.
      emit (now := d; !now))                          Reset the state to a new number.
let calc_e, cancel_calc = event_flow calc          Notifies display update.
```

Tk: *LablTk*

- Widget toolkit ***Tk*** known from the *Tcl* language.
- Invocation:

  ```
  ocamlbuild Lec10tk.byte -cflags -I,+froc -libs froc/froc
    -pkg labltk -pp "camlp4o monad/pa_monad.cmo" --
  ```

  - For unknown reason I had build problems with `ocamlopt` (native).
- Layout of the calculator – common across GUIs.

  ```
  let layout =
   [|[|"7",‘Di 7.; "8",‘Di 8.; "9",‘Di 9.; "+",‘O (+.)|];
     [|"4",‘Di 4.; "5",‘Di 5.; "6",‘Di 6.; "-",‘O (-.)|];
     [|"1",‘Di 1.; "2",‘Di 2.; "3",‘Di 3.; "*",‘O ( *.)|];
     [|"0",‘Di 0.; ".",‘Dot;   "=", ‘O sk; "/",‘O (/.)|]|]
  ```

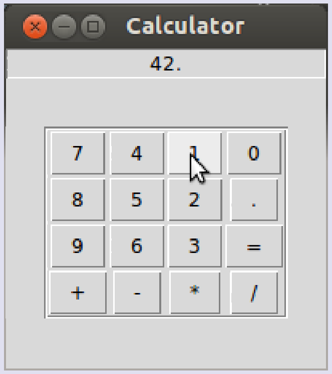
- Every *widget* (window gadget) has a parent in which it is located.
- *Buttons* have action associated with pressing them, *labels* just provide information, *entries* (aka. *edit* fields) are for entering info from keyboard.
  - Actions are *callback* functions passed as the ∼`command` argument.

- *Frames* in *Tk* group widgets.
- The parent is sent as last argument, after optional labelled arguments.

```
let top = Tk.openTk ()
let btn_frame =
  Frame.create ~relief:`Groove ~borderwidth:2 top
let buttons =
  Array.map (Array.map (function
  | text, `Dot ->
    Button.create ~text
      ~command:(fun () -> F.send dot ()) btn_frame
  | text, `Di d ->
    Button.create ~text
      ~command:(fun () -> F.send digit d) btn_frame
  | text, `O f ->
    Button.create ~text
      ~command:(fun () -> F.send op f) btn_frame)) layout
let result = Label.create ~text:"0" ~relief:`Sunken top
```

- GUI toolkits have layout algorithms, so we only need to tell which widgets hang together and whether they should fill all available space etc. – via pack, or grid for “rectangular” organization.
- ∼fill: the allocated space in ‘X, ‘Y, ‘Both or ‘None axes;
  ∼expand: maximally how much space is allocated or only as needed.
- ∼anchor: allows to glue a widget in particular direction (‘Center, ‘E, ‘Ne etc.)
- The grid packing flexibility: ∼columnspan and ∼rowspan.
- configure functions accept the same arguments as create but change existing widgets.

-
  ```
  let () =
    Wm.title_set top "Calculator";
    Tk.pack [result] ~side:`Top ~fill:`X;
    Tk.pack [btn_frame] ~side:`Bottom ~expand:true;
    Array.iteri (fun column ->Array.iteri (fun row button ->
      Tk.grid ~column ~row [button])) buttons;
    Wm.geometry_set top "200x200";
    F.notify_e calc_e
      (fun now ->
        Label.configure ~text:(string_of_float now) result);
    Tk.mainLoop ()
  ```

- 

  42.

GTk+: *LablGTk*

- ***LablGTk*** is build as an object-oriented layer over a low-level layer of functions interfacing with the *GTk+* library, which is written in *C*.
- In OCaml, object fields are only visible to object methods, and methods are called with # syntax, e.g. `window#show ()`
- The interaction with the application is reactive:
  - Our events are called signals in *GTk+*.
  - Registering a notification is called connecting a signal handler, e.g. `button#connect#clicked ~callback:hello` which takes `~callback:(unit -> unit)` and returns `GtkSignal.id`.
    - As with *Froc* notifications, multiple handlers can be attached.
  - *GTk+* events are a subclass of signals related to more specific window events, e.g. `window#event#connect#delete ~callback:delete_event`
  - *GTk+* event callbacks take more info: `~callback:(event -> unit)` for some type `event`.

- Automatic layout (aka. packing) seems less sophisticated than in *Tk*:
  - only horizontal and vertical boxes,
  - therefore `~fill` is binary and `~anchor` is replaced by `~from` `` `START `` or `` `END ``.
- Automatic grid layout is called `table`.
  - `~fill` and `~expand` take `` `X ``, `` `Y ``, `` `BOTH ``, `` `NONE ``.
- The `coerce` method casts the type of the object (in *Tk* there is `coe` function).
- Labels don't have a dedicated module – see definition of `result` widget.
- Widgets have setter methods `widget#set_X` (instead of a single `configure` function in *Tk*).
- Invocation:

  ```
  ocamlbuild Lec10gtk.native -cflags -I,+froc -libs froc/froc
    -pkg lablgtk2 -pp "camlp4o monad/pa_monad.cmo" --
  ```
- The model part of application doesn't change.

- Setup:

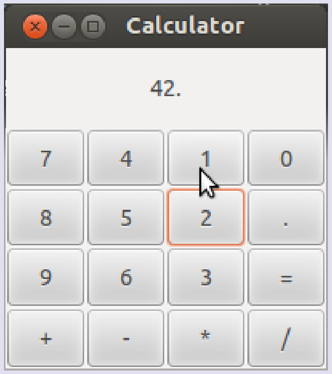
```
let _ = GtkMain.Main.init ()
let window =
  GWindow.window ~width:200 ~height:200 ~
title:"Calculator" ()
let top = GPack.vbox ~packing:window#add ()
let result = GMisc.label ~text:"0" ~packing:top#add ()
let btn_frame =
  GPack.table ~rows:(Array.length layout)
   ~columns:(Array.length layout.(0)) ~packing:top#add ()
```

- Button actions:

```
let buttons =
 Array.map (Array.map (function
 | label, ‘Dot ->
  let b = GButton.button ∼label () in
  let _ = b#connect#clicked
   ∼callback:(fun () -> F.send dot ()) in b
 | label, ‘Di d ->
  let b = GButton.button ∼label () in
  let _ = b#connect#clicked
   ∼callback:(fun () -> F.send digit d) in b
 | label, ‘O f ->
  let b = GButton.button ∼label () in
  let _ = b#connect#clicked
   ∼callback:(fun () -> F.send op f) in b)) layout
```

- Button layout, result notification, start application:

```
let delete_event _ = GMain.Main.quit (); false
let () =
  let _ = window#event#connect#delete ~
callback:delete_event in
  Array.iteri (fun column->Array.iteri (fun row button ->
    btn_frame#attach ~left:column ~top:row
      ~fill:`BOTH ~expand:`BOTH (button#coerce))
  ) buttons;
  F.notify_e calc_e
    (fun now -> result#set_label (string_of_float now));
  window#show ();
  GMain.Main.main ()
```

- 

42.