Functional Programming

by Łukasz Stafiniak

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

Lecture 8: Monads

List comprehensions. Basic monads; transformers. Probabilistic Programming. Lightweight cooperative threads.

Some examples from Tomasz Wierzbicki. Jeff Newbern *"All About Monads"*.
M. Erwig, S. Kollmansberger *"Probabilistic Functional Programming in Haskell"*.
Jerome Vouillon *"Lwt: a Cooperative Thread Library"*.

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

List comprehensions

- Recall the awkward syntax we used in the Countdown Problem example:
  - Brute-force generation:

```
let combine l r =
  List.map (fun o->App (o,l,r)) [Add; Sub; Mul; Div]
let rec exprs = function
  | [] -> []
  | [n] -> [Val n]
  | ns ->
    split ns |-> (fun (ls,rs) ->
      exprs ls |-> (fun l ->
        exprs rs |-> (fun r ->
          combine l r)))
```

  - Genarate-and-test scheme:

```
let guard p e = if p e then [e] else []
let solutions ns n =
  choices ns |-> (fun ns' ->
    exprs ns' |->
      guard (fun e -> eval e = Some n))
```

- Recall that we introduced the operator

```
let ( |-> ) x f = concat_map f x
```

- We can do better with *list comprehensions* syntax extension.

```
#load "dynlink.cma";;
#load "camlp4o.cma";;
#load "Camlp4Parsers/Camlp4ListComprehension.cmo";;
```

```
let test = [i * 2 | i <- from_to 2 22; i mod 3 = 0]
```

- What it means:
  - `[expr | ]` can be translated as `[expr]`
  - `[expr | v <- generator;` *more*`]` can be translated as

    `generator |-> (fun v ->` translation of `[expr |` *more*`])`
  - `[expr | condition;` *more*`]` can be translated as

    `if condition then` translation of `[expr |` *more*`] else []`

- Revisiting the Countdown Problem code snippets:
  - Brute-force generation:

```
let rec exprs = function
  | [] -> []
  | [n] -> [Val n]
  | ns ->
    [App (o,l,r) | (ls,rs) <- split ns;
     l <- exprs ls; r <- exprs rs;
     o <- [Add; Sub; Mul; Div]]
```

  - Genarate-and-test scheme:

```
let solutions ns n =
  [e | ns' <- choices ns;
   e <- exprs ns'; eval e = Some n]
```

- Subsequences using list comprehensions (with garbage):

```
let rec subseqs l =
  match l with
    | [] -> [[]]
    | x::xs -> [ys | px <- subseqs xs; ys <- [px; x::px]]
```

- Computing permutations using list comprehensions:
  - via insertion

```
let rec insert x = function
  | [] -> [[x]]
  | y::ys' as ys ->
    (x::ys) :: [y::zs | zs <- insert x ys']
let rec ins_perms = function
  | [] -> [[]]
  | x::xs -> [zs | ys <- ins_perms xs; zs <- insert ys]
```

  - via selection

```
let rec select = function
  | [x] -> [x,[]]
  | x::xs -> (x,xs) :: [ y, x::ys | y,ys <- select xs]
let rec sel_perms = function
  | [] -> [[]]
  | xs ->
    [x::ys | x,xs' <- select xs; ys <- sel_perms xs']
```

Generalized comprehensions aka. *do-notation*

- We need to install the syntax extension `pa_monad`
  - by copying the `pa_monad.cmo` or `pa_monad400.cmo` (for OCaml 4.0) file from the course page,
  - or if it does not work, by compiling from sources at http://www.cas.mcmaster.ca/~carette/pa_monad/ and installing under a Unix-like shell (Windows: the Cygwin shell).
    - Under Debian/Ubuntu, you may need to install `camlp4-extras`
-
```
let rec exprs = function
  | [] -> []
  | [n] -> [Val n]
  | ns ->
    perform with (|->) in
      (ls,rs) <-- split ns;
      l <-- exprs ls; r <-- exprs rs;
      o <-- [Add; Sub; Mul; Div];
      [App (o,l,r)]
```

- The `perform` syntax does not seem to support guards...

```
let solutions ns n =
  perform with (|->) in
    ns' <-- choices ns;
    e <-- exprs ns';
    eval e = Some n;
    e

      eval e = Some n;
      ^^^^^^^^^^^^^^^
Error: This expression has type bool but an expression was
expected of type
         'a list
```

- So it wants a list... What can we do?

- We can decide whether to return anything

```
let solutions ns n =
  perform with (|->) in
    ns' <-- choices ns;
    e <-- exprs ns';
    if eval e = Some n then [e] else []
```

- But what if we want to check earlier...

  General "guard check" function

```
let guard p = if p then [()] else []
```

- ```
  let solutions ns n =
    perform with (|->) in
      ns' <-- choices ns;
      e <-- exprs ns';
      guard (eval e = Some n);
      [e]
  ```
  ```

Monads

- A polymorphic type `'a monad` (or `'a Monad.t`, etc.) that supports at least two operations:
  - `bind : 'a monad -> ('a -> 'b monad) -> 'b monad`
  - `return : 'a -> 'a monad`
  - `>>=` is infix syntax for `bind`: `let (>>=) a b = bind a b`
- With `bind` in scope, we do not need the `with` clause in `perform`

```
let bind a b = concat_map b a
let return x = [x]
let solutions ns n =
  perform
    ns' <-- choices ns;
    e <-- exprs ns';
    guard (eval e = Some n);
    return e
```

- Why `guard` looks this way?

```
let fail = []
let guard p = if p then return () else fail
```

  - Steps in monadic computation are composed with `>>=`, e.g. `|->`
    - as if `;` was replaced by `>>=`
  - `[] |-> ...` does not produce anything – as needed by guarding
  - `[()] |-> ...` ⇝ `(fun _ -> ...) ()` ⇝ `...` i.e. keep without change
- Throwing away the binding argument is a common practice, with infix syntax `>>` in Haskell, and supported in *do-notation* and `perform`.
- Everything is a monad?
- Different flavors of monads?
- Can `guard` be defined for any monad?

- perform syntax in depth:

`perform exp`	⟹	`exp`
`perform pat <-- exp;` `        rest`	⟹	`bind exp` `(fun pat -> perform rest)`
`perform exp; rest`	⟹	`bind exp` `(fun _ -> perform rest)`
`perform let ... in rest`	⟹	`let ... in perform rest`

`perform rpt <-- exp;`
`        rest`
	⟹	`bind exp`
		`(function`
		`| rpt -> perform rest`
		`| _ -> failwith`
		`        "pattern match")`

`perform with b [and f] in` `        body`	⟹	`perform body` but uses `b` instead of `bind` and `f` instead of `failwith` during translation

- It can be useful to redefine: `let failwith _ = fail` (*why?*)

Monad laws

- A parametric data type is a monad only if its `bind` and `return` operations meet axioms:

$$
\begin{aligned}
\mathrm{bind}\,(\mathrm{return}\,a)\;f &\approx f a \\
\mathrm{bind}\,a\,(\lambda x.\mathrm{return}\,x) &\approx a \\
\mathrm{bind}\,(\mathrm{bind}\,a\,(\lambda x.b))\,(\lambda y.c) &\approx \mathrm{bind}\,a\,(\lambda x.\mathrm{bind}\,b\,(\lambda y.c))
\end{aligned}
$$

- Check that the laws hold for our example monad

```
let bind a b = concat_map b a
let return x = [x]
```

Monoid laws and *monad-plus*

- A monoid is a type with, at least, two operations
  - `mzero : 'a monoid`
  - `mplus : 'a monoid -> 'a monoid -> 'a monoid`

  that meet the laws:

$$
\begin{aligned}
\mathrm{mplus}\,\mathrm{mzero}\,a &\approx a \\
\mathrm{mplus}\,a\,\mathrm{mzero} &\approx a \\
\mathrm{mplus}\,a\,(\mathrm{mplus}\,b\,c) &\approx \mathrm{mplus}\,(\mathrm{mplus}\,a\,b)\,c
\end{aligned}
$$

- We will define `fail` as synonym for `mzero` and infix `++` for `mplus`.
- Fusing monads and monoids gives the most popular general flavor of monads which we call *monad-plus* after Haskell.

- Monad-plus requires additional axioms that relate its "addition" and its "multiplication".

$$
\begin{aligned}
\mathrm{bind}\,\mathrm{mzero}\,f &\approx \mathrm{mzero} \\
\mathrm{bind}\,m\,(\lambda x.\mathrm{mzero}) &\approx \mathrm{mzero}
\end{aligned}
$$

- Using infix notation with $\oplus$ as `mplus`, $\mathbf{0}$ as `mzero`, $\triangleright$ as `bind` and $\mathbf{1}$ as `return`, we get monad-plus axioms

$$
\begin{aligned}
\mathbf{0} \oplus a &\approx a \\
a \oplus \mathbf{0} &\approx a \\
a \oplus (b \oplus c) &\approx (a \oplus b) \oplus c \\
\mathbf{1}\,x \triangleright f &\approx f\,x \\
a \triangleright \lambda x.\mathbf{1}\,x &\approx a \\
(a \triangleright \lambda x.b) \triangleright \lambda y.c &\approx a \triangleright (\lambda x.b \triangleright \lambda y.c) \\
\mathbf{0} \triangleright f &\approx \mathbf{0} \\
a \triangleright (\lambda x.\mathbf{0}) &\approx \mathbf{0}
\end{aligned}
$$

- The list type has a natural monad and monoid structure

```
let mzero = []
let mplus = (@)
let bind a b = concat_map b a
let return a = [a]
```

- We can define in any monad-plus

```
let fail = mzero
let failwith _ = fail
let (++) = mplus
let (>>=) a b = bind a b
let guard p = if p then return () else fail
```

Backtracking: computation with choice

We have seen `mzero`, i.e. `fail` in the countdown problem. What about `mplus`?

```
let find_to_eat n island_size num_islands empty_cells =
  let honey = honey_cells n empty_cells in

  let rec find_board s =
    (* Printf.printf "find_board: %sn" (state_str s); *)
    match visit_cell s with
    | None ->
      perform
        guard (s.been_islands = num_islands);
        return s.eaten
    | Some (cell, s) ->
      perform
        s <-- find_island cell (fresh_island s);
        guard (s.been_size = island_size);
        find_board s
```

```
and find_island current s =
  let s = keep_cell current s in
  neighbors n empty_cells current
  |> foldM
      (fun neighbor s ->
        if CellSet.mem neighbor s.visited then return s
        else
          let choose_eat =
            if s.more_to_eat <= 0 then fail
            else return (eat_cell neighbor s)
          and choose_keep =
            if s.been_size >= island_size then fail
            else find_island neighbor s in
          mplus choose_eat choose_keep)
      s in

let cells_to_eat =
  List.length honey - island_size * num_islands in
find_board (init_state honey cells_to_eat)
```

Monad “flavors”

- Monads “wrap around” a type, but some monads need an additional type parameter.
  - Usually the additional type does not change while within a monad – we will therefore stick to `'a monad` rather than parameterize with an additional type `('s, 'a) monad`.
- As monad-plus shows, things get interesting when we add more operations to a basic monad (with `bind` and `return`).
  - Monads with access:

    ```
    access : 'a monad -> 'a
    ```

    Example: the lazy monad.
  - Monad-plus, non-deterministic computation:

    ```
    mzero : 'a monad
    mplus : 'a monad -> 'a monad -> 'a monad
    ```

- Monads with environment or state – parameterized by type `store`:

```
get : store monad
put : store -> unit monad
```

  There is a “canonical” state monad. Similar monads: the writer monad (with get called `listen` and put called `tell`); the reader monad, without put, but with get (called `ask`) and `local`:

```
local : (store -> store) -> 'a monad -> 'a monad
```

- The exception / error monads – parameterized by type `excn`:

```
throw : excn -> 'a monad
catch : 'a monad -> (excn -> 'a monad) -> 'a monad
```

- The continuation monad:

```
callCC : (('a -> 'b monad) -> 'a monad) -> 'a monad
```

  We will not cover it.

- Probabilistic computation:

```
choose : float -> 'a monad -> 'a monad -> 'a monad
```

satisfying the laws with $a \oplus_p b$ for `choose p a b` and $p\,q$ for `p*.q`, $0 \leqslant p, q \leqslant 1$:

$$\begin{array}{rcl} a \oplus_0 b & \approx & b \\ a \oplus_p b & \approx & b \oplus_{1-p} a \\ a \oplus_p (b \oplus_q c) & \approx & \left(a \oplus_{\frac{p}{p+q-pq}} b\right) \oplus_{p+q-pq} c \\ a \oplus_p a & \approx & a \end{array}$$

- Parallel computation as monad with access and parallel bind:

```
parallel :
'a monad-> 'b monad-> ('a -> 'b -> 'c monad) -> 'c monad
```

Example: lightweight threads.

Interlude: the module system

- I provide below much more information about the module system than we need, just for completeness. You can use it as reference.
  - Module system details will **not** be on the exam – only the structure / signature definitions as discussed in lecture 5.
- Modules collect related type definitions and operations together.
- Module "values" are introduced with `struct ... end` – structures.
- Module types are introduced with `sig ... end` – signatures.
  - A structure is a package of definitions, a signature is an interface for packages.
- A source file `source.ml` or `Source.ml` defines a module `Source`.

  A source file `source.mli` or `Source.mli` defines its type.
- We can create the initial interface by entering the module in the interactive toplevel or by command `ocamlc -i source.ml`

- In the "toplevel" – accurately, module level – modules are defined with `module ModuleName = ...` or `module ModuleName : MODULE_TYPE = ...` syntax, and module types with `module type MODULE_TYPE = ...` syntax.
    - Corresponds to `let v_name = ...` resp. `let v_name : v_type = ...` syntax for values and `type v_type = ...` syntax for types.
- Locally in expressions, modules are defined with `let module M = ... in ...` syntax.
    - Corresponds to `let v_name = ... in ...` syntax for values.
- The content of a module is made visible in the remainder of another module by `open Module`
    - Module `Pervasives` is initially visible, as if each file started with `open Pervasives`.
- The content of a module is made visible locally in an expression with `let open Module in ...` syntax.

- Content of a module is included into another module – i.e. made part of it – by `include Module`.
  - Just having `open Module` inside `Parent` does not affect how `Parent` looks from outside.
- Module functions – functions from modules to modules – are called *functors* (not the Haskell ones!). The type of the parameter has to be given.

```
module Funct = functor (Arg : sig … end) -> struct … end
module Funct (Arg : sig … end) = struct … end
```

  - Functors can return functors, i.e. modules can be parameterized by multiple modules.
  - Modules are either structures or functors.
  - Different kind of thing than Haskell functors.
- Functor application always uses parentheses: `Funct (struct ... end)`
- We can use named module type instead of signature and named module instead of structure above.
- Argument structures can contain more definitions than required.

- A signature `MODULE_TYPE with type t_name = ...` is like `MODULE_TYPE` but with `t_name` made more specific.
- We can also include signatures into other signatures, by `include MODULE_TYPE`.
    - `include MODULE_TYPE with type t_name := ...` will substitute type `t_name` with provided type.
- Modules, just as expressions, are **not** recursive or mutually recursive by default. Syntax for recursive modules:
  `module rec ModuleName : MODULE_TYPE = ... and ...`
- We can recover the type – i.e. signature – of a module by
  `module type of Module`

- Finally, we can pass around modules in normal functions.
    - `(module Module)` is an expression
    - `(val module_v)` is a module
    - ```
      # module type T = sig val g : int -> int end
      let f mod_v x =
        let module M = (val mod_v : T) in
        M.g x;;

      val f : (module T) -> int -> int = <fun>

      # let test = f (module struct let g i = i*i end : T);;

      val test : int -> int = <fun>
      ```
      ```

The two metaphors

- Monads can be seen as **containers**: `'a monad` contains stuff of type `'a`
- and as **computation**: `'a monad` is a special way to compute `'a`.
  - A monad fixes the sequence of computing steps – unless it is a fancy monad like parallel computation monad.

Monads as containers

- A monad is a *quarantine container*:
  - we can put something into the container with `return`
  - we can operate on it, but the result needs to stay in the container

    ```
    let lift f m = perform x <-- m; return (f x)
    val lift : ('a -> 'b) -> 'a monad -> 'b monad
    ```

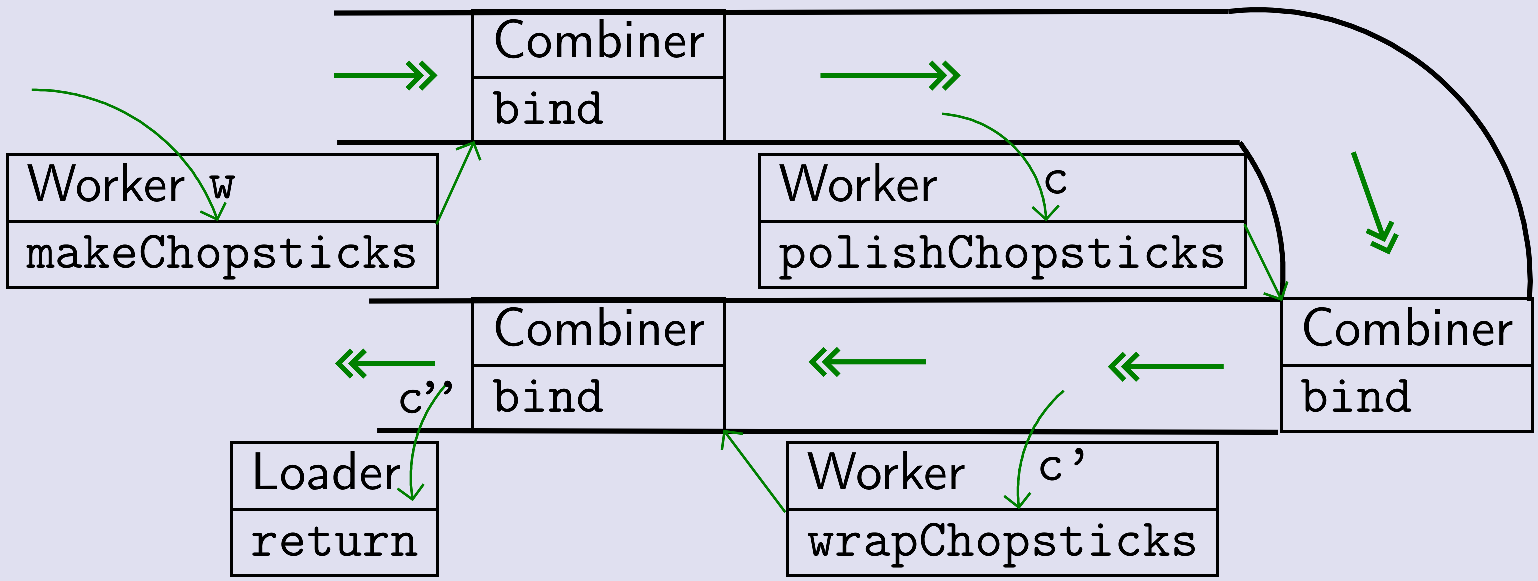
  - We can deactivate-unwrap the quarantine container but only when it is in another container so the quarantine is not broken

    ```
    let join m = perform x <-- m; x
    val join : ('a monad) monad -> 'a monad
    ```

- The quarantine container for a **monad-plus** is more like other containers: it can be empty, or contain multiple elements.
- Monads with access allow us to extract the resulting element from the container, other monads provide a `run` operation that exposes "what really happened behind the quarantine".

Monads as computation

- To compute the result, perform instructions, naming partial results.
- Physical metaphor: **assembly line**




```
let assemblyLine w =
    perform
      c <-- makeChopsticks w
      c' <-- polishChopsticks c
      c'' <-- wrapChopsticks c'
      return c''
```

- Any expression can be spread over a monad, e.g. for λ-terms:

$$
\begin{aligned}
[\![N]\!] &= \text{return}\, N && \text{(constant)} \\
[\![x]\!] &= \text{return}\, x && \text{(variable)} \\
[\![\lambda x.a]\!] &= \text{return}(\lambda x.[\![a]\!]) && \text{(function)} \\
[\![\text{let}\, x = a \,\text{in}\, b]\!] &= \text{bind}\, [\![a]\!]\, (\lambda x.[\![b]\!]) && \text{(local definition)} \\
[\![a\, b]\!] &= \text{bind}\, [\![a]\!]\, (\lambda v_a.\text{bind}\, [\![b]\!]\, (\lambda v_b.v_a\, v_b)) && \text{(application)}
\end{aligned}
$$

- When an expression is spread over a monad, its computation can be monitored or affected without modifying the expression.

Monad classes

- To implement a monad we need to provide the implementation type, `return` and `bind` operations.

  ```
  module type MONAD = sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
  end
  ```

  - Alternatively we could start from `return`, `lift` and `join` operations.
  - For monads that change their additional type parameter we could define:

    ```
    module type MONAD = sig
      type ('s, 'a) t
      val return : 'a -> ('s, 'a) t
      val bind :
        ('s, 'a) t -> ('a -> ('s, 'b) t) -> ('s, 'b) t
    end
    ```

- Based on just these two operations, we can define a whole suite of general-purpose functions. We look at just a tiny selection.

```
module type MONAD_OPS = sig
  type 'a monad
  include MONAD with type 'a t := 'a monad
  val ( >>= ) :'a monad -> ('a -> 'b monad) -> 'b monad
  val foldM :
    ('a -> 'b -> 'a monad) -> 'a -> 'b list -> 'a monad
  val whenM : bool -> unit monad -> unit monad
  val lift : ('a -> 'b) -> 'a monad -> 'b monad
  val (>>|) : 'a monad -> ('a -> 'b) -> 'b monad
  val join : 'a monad monad -> 'a monad
  val ( >=> ) :
    ('a ->'b monad) -> ('b ->'c monad) -> 'a -> 'c monad
end
```

- Given a particular implementation, we define these functions.

```
module MonadOps (M : MONAD) = struct
  open M
  type 'a monad = 'a t
  let run x = x
  let (>>=) a b = bind a b
  let rec foldM f a = function
    | [] -> return a
    | x::xs -> f a x >>= fun a' -> foldM f a' xs
  let whenM p s = if p then s else return ()
  let lift f m = perform x <-- m; return (f x)
  let (>>|) a b = lift b a
  let join m = perform x <-- m; x
  let (>=>) f g = fun x -> f x >>= g
end
```

- We make the monad "safe" by keeping its type abstract. But `run` exposes "what really happened".

```
module Monad (M : MONAD) :
sig
  include MONAD_OPS
  val run : 'a monad -> 'a M.t
end = struct
  include M
  include MonadOps(M)
end
```

  - Our `run` function does not do anything at all. Often more useful functions are called `run` but then they need to be defined for each implementation separately. Our `access` operation (see section on monad flavors) is often called `run`.

- The monad-plus class of monads has a lot of implementations. They need to provide `mzero` and `mplus`.

```
module type MONAD_PLUS = sig
  include MONAD
  val mzero : 'a t
  val mplus : 'a t -> 'a t -> 'a t
end
```

- Monad-plus class also has its general-purpose functions:

```
module type MONAD_PLUS_OPS = sig
  include MONAD_OPS
  val mzero : 'a monad
  val mplus : 'a monad -> 'a monad -> 'a monad
  val fail : 'a monad
  val (++) : 'a monad -> 'a monad -> 'a monad
  val guard : bool -> unit monad
  val msum_map : ('a -> 'b monad) -> 'a list -> 'b monad
end
```

- We again separate the “implementation” and the “interface”.

```
module MonadPlusOps (M : MONAD_PLUS) = struct
  open M
  include MonadOps(M)
  let fail = mzero
  let (++) a b = mplus a b
  let guard p = if p then return () else fail
  let msum_map f l = List.fold_right
    (fun a acc -> mplus (f a) acc) l mzero
end
```

```
module MonadPlus (M : MONAD_PLUS) :
sig
  include MONAD_PLUS_OPS
  val run : 'a monad -> 'a M.t
end = struct
  include M
  include MonadPlusOps(M)
end
```

- We also need a class for computations with state.

```
module type STATE = sig
  type store
  type 'a t
  val get : store t
  val put : store -> unit t
end
```

The purpose of this signature is inclusion in other signatures.

Monad instances

- We do not define a class for monads with access since accessing means running the monad, not useful while in the monad.
- Notation for laziness heavy? Try a monad! (Monads with access.)

```
module LazyM = Monad (struct
  type 'a t = 'a Lazy.t
  let bind a b = lazy (Lazy.force (b (Lazy.force a)))
  let return a = lazy a
end)
```

```
let laccess m = Lazy.force (LazyM.run m)
```

- Our resident list monad. (Monad-plus.)

```
module ListM = MonadPlus (struct
  type 'a t = 'a list
  let bind a b = concat_map b a
  let return a = [a]
  let mzero = []
  let mplus = List.append
end)
```

Backtracking parameterized by monad-plus

```
module Countdown (M : MONAD_PLUS_OPS) = struct
  open M
  let rec insert x = function
    | [] -> return [x]
    | y::ys as xs ->
      return (x::xs) ++
        perform xys <-- insert x ys; return (y::xys)
  let rec choices = function
    | [] -> return []
    | x::xs -> perform
        cxs <-- choices xs;
        return cxs ++ insert x cxs
```

Open the module to make monad operations visible.

All choice-introducing operations need to happen in the monad.

Choosing which numbers in what order and now whether with or without x.

```
type op = Add | Sub | Mul | Div

let apply op x y =
  match op with
  | Add -> x + y
  | Sub -> x - y
  | Mul -> x * y
  | Div -> x / y

let valid op x y =
  match op with
  | Add -> x <= y
  | Sub -> x > y
  | Mul -> x <= y && x <> 1 && y <> 1
  | Div -> x mod y = 0 && y <> 1
```

```
type expr = Val of int | App of op * expr * expr

let op2str = function
  | Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
let rec expr2str = function
  | Val n -> string_of_int n
  | App (op,l,r) ->"("^expr2str l^op2str op^expr2str r^")"
```

We will provide solutions as strings.

```
let combine (l,x) (r,y) o = perform
    guard (valid o x y);
    return (App (o,l,r), apply o x y)
```

Try out an operator.

```
let split l =
  let rec aux lhs = function
    | [] | [_] -> fail
    | [y; z] -> return (List.rev (y::lhs), [z])
    | hd::rhs ->
      let lhs = hd::lhs in
      return (List.rev lhs, rhs)
        ++ aux lhs rhs in
  aux [] l
```

Another choice: which numbers go into which argument.

Both arguments need numbers.

```
  let rec results = function          Build possible expressions once numbers
    | [] -> fail                                           have been picked.
    | [n] -> perform
        guard (n > 0); return (Val n, n)
    | ns -> perform
        (ls, rs) <-- split ns;
        lx <-- results ls;
        ly <-- results rs;                  Collect solutions using each operator.
        msum_map (combine lx ly) [Add; Sub; Mul; Div]

  let solutions ns n = perform                              Solve the problem:
      ns' <-- choices ns;                          pick numbers and their order,
      (e,m) <-- results ns';                          build possible expressions,
      guard (m=n);                      check if the expression gives target value,
      return (expr2str e)                                  "print" the solution.
end
```

Understanding laziness

- We will measure execution times:

```
#load "unix.cma";;
let time f =
  let tbeg = Unix.gettimeofday () in
  let res = f () in
  let tend = Unix.gettimeofday () in
  tend -. tbeg, res
```

- Let's check our generalized `Countdown` solver using original operations.

```
module ListCountdown = Countdown (ListM)
let test1 () = ListM.run (ListCountdown.solutions
[1;3;7;10;25;50] 765)
let t1, sol1 = time test1
```

- ```
  val t1 : float = 2.2856600284576416
  val sol1 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  ```
  ```

- What if we want only one solution? Laziness to the rescue!

```
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t
let rec ltake n = function
 | LCons (a, lazy l) when n > 0 -> a::(ltake (n-1) l)
 | _ -> []
let rec lappend l1 l2 =
  match l1 with LNil -> l2
  | LCons (hd, tl) ->
    LCons (hd, lazy (lappend (Lazy.force tl) l2))
let rec lconcat_map f = function
  | LNil -> LNil
  | LCons (a, lazy l) ->
    lappend (f a) (lconcat_map f l)
```

- That is, another monad-plus.

  ```
  module LListM = MonadPlus (struct
    type 'a t = 'a llist
    let bind a b = lconcat_map b a
    let return a = LCons (a, lazy LNil)
    let mzero = LNil
    let mplus = lappend
  end)
  ```

- ```
  module LListCountdown = Countdown (LListM)
  let test2 () = LListM.run (LListCountdown.solutions
  [1;3;7;10;25;50] 765)
  ```

- ```
  # let t2a, sol2 = time test2;;
  val t2a : float = 2.51197600364685059
  val sol2 : string llist = LCons ("((25-(3+7))*(1+50))",
  <lazy>)
  ```

  Not good, almost the same time to even get the lazy list!

- ```
  # let t2b, sol2_1 = time (fun () -> ltake 1 sol2);;
  val t2b : float = 2.86102294921875e-06
  val sol2_1 : string list = ["((25-(3+7))*(1+50))"]
  # let t2c, sol2_9 = time (fun () -> ltake 10 sol2);;
  val t2c : float = 9.059906005859375e-06
  val sol2_9 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  # let t2d, sol2_39 = time (fun () -> ltake 49 sol2);;
  val t2d : float = 4.00543212890625e-05
  val sol2_39 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  ```

  Getting elements from the list shows they are almost already computed.
  ```

- Wait! Perhaps we should not store all candidates when we are only interested in one.

```
module OptionM = MonadPlus (struct
  type 'a t = 'a option
  let bind a b =
    match a with None -> None | Some x -> b x
  let return a = Some a
  let mzero = None
  let mplus a b = match a with None -> b | Some _ -> a
end)
```

- ```
  module OptCountdown = Countdown (OptionM)
  let test3 () = OptionM.run (OptCountdown.solutions
  [1;3;7;10;25;50] 765)
  ```

- ```
  # let t3, sol3 = time test3;;
  val t3 : float = 5.00679016113281 25e-06
  val sol3 : string option = None
  ```

  It very quickly computes... nothing. Why?

  - What is the `OptionM` monad (`Maybe` monad in Haskell) good for?

- Our lazy list type is not lazy enough.
  - Whenever we “make” a choice: `a ++ b` or `msum_map` ..., it computes the first candidate for each choice path.
  - When we bind consecutive steps, it computes the second candidate of the first step even when the first candidate would suffice.

- We want the whole monad to be lazy: it's called *even lazy lists*.
    - Our `llist` are called *odd lazy lists*.

```
type 'a lazy_list = 'a lazy_list_ Lazy.t
and 'a lazy_list_ = LazNil | LazCons of 'a * 'a lazy_list
let rec laztake n = function
 | lazy (LazCons (a, l)) when n > 0 ->
   a::(laztake (n-1) l)
 | _ -> []
let rec append_aux l1 l2 =
  match l1 with lazy LazNil -> Lazy.force l2
   | lazy (LazCons (hd, tl)) ->
     LazCons (hd, lazy (append_aux tl l2))
let lazappend l1 l2 = lazy (append_aux l1 l2)
let rec concat_map_aux f = function
   | lazy LazNil -> LazNil
   | lazy (LazCons (a, l)) ->
     append_aux (f a) (lazy (concat_map_aux f l))
let lazconcat_map f l = lazy (concat_map_aux f l)
```

- ```
  module LazyListM = MonadPlus (struct
    type 'a t = 'a lazy_list
    let bind a b = lazconcat_map b a
    let return a = lazy (LazCons (a, lazy LazNil))
    let mzero = lazy LazNil
    let mplus = lazappend
  end)
  ```
- ```
  module LazyCountdown = Countdown (LazyListM)
  let test4 () = LazyListM.run (LazyCountdown.solutions
  [1;3;7;10;25;50] 765)
  ```

- ```
  # let t4a, sol4 = time test4;;
  val t4a : float = 2.86102294921875e-06
  val sol4 : string lazy_list = <lazy>
  # let t4b, sol4_1 = time (fun () -> laztake 1 sol4);;
  val t4b : float = 0.367874860763549805
  val sol4_1 : string list = ["((25-(3+7))*(1+50))"]
  # let t4c, sol4_9 = time (fun () -> laztake 10 sol4);;
  val t4c : float = 0.234670877456665039
  val sol4_9 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  # let t4d, sol4_39 = time (fun () -> laztake 49 sol4);;
  val t4d : float = 4.0594940185546875
  val sol4_39 : string list =
    ["((25-(3+7))*(1+50))"; "(((25-3)-7)*(1+50))"; ...
  ```
  - Finally, the first solution in considerably less time than all solutions.
  - The next 9 solutions are almost computed once the first one is.
  - But computing all solutions takes nearly twice as long as without the overhead of lazy computation.
  ```

The exception monad

- Built-in non-functional exceptions in OCaml are more efficient (and more flexible).
- Instead of specifying a type of exceptional values, we could use OCaml open type `exn`, restoring some flexibility.
- Monadic exceptions are safer than standard exceptions in situations like multi-threading. Monadic lightweight-thread library `Lwt` has `throw` (called `fail` there) and `catch` operations in its monad.

```
module ExceptionM(Excn : sig type t end) : sig
  type excn = Excn.t
  type 'a t = OK of 'a | Bad of excn
  include MONAD_OPS
  val run : 'a monad -> 'a t
  val throw : excn -> 'a monad
  val catch : 'a monad -> (excn -> 'a monad) -> 'a monad
end = struct
  type excn = Excn.t
```

```
  module M = struct
    type 'a t = OK of 'a | Bad of excn
    let return a = OK a
    let bind m b = match m with
      | OK a -> b a
      | Bad e -> Bad e
  end
  include M
  include MonadOps(M)
  let throw e = Bad e
  let catch m handler = match m with
    | OK _ -> m
    | Bad e -> handler e
end
```

The state monad

```
module StateM(Store : sig type t end) : sig
  type store = Store.t         Pass the current store value to get the next value.
  type 'a t = store -> 'a * store
  include MONAD_OPS
  include STATE with type 'a t := 'a monad
                and type store := store
  val run : 'a monad -> 'a t
end = struct
  type store = Store.t
  module M = struct
    type 'a t = store -> 'a * store
    let return a = fun s -> a, s          Keep the current value unchanged.
    let bind m b = fun s -> let a, s' = m s in b a s'
  end                 To bind two steps, pass the value after first step to the second step.
  include M include MonadOps(M)
  let get = fun s -> s, s        Keep the value unchanged but put it in monad.
  let put s' = fun _ -> (), s'       Change the value; a throwaway in monad.
end
```

- The state monad is useful to hide passing-around of a "current" value.
- We will rename variables in λ-terms to get rid of possible name clashes.
  - This does not make a λ-term safe for multiple steps of β-reduction. Find a counter-example.
- ```
  type term =
  | Var of string
  | Lam of string * term
  | App of term * term
  ```
- ```
  let (!) x = Var x
  let (|->) x t = Lam (x, t)
  let (@) t1 t2 = App (t1, t2)
  let test = "x" |-> ("x" |-> !"y" @ !"x") @ !"x"
  ```
- ```
  module S =
    StateM(struct type t = int * (string * string) list end)
  open S
  ```

  Without opening the module, we would write `S.get`, `S.put` and `perform with S in`...
  ```

- ```
  let rec alpha_conv = function
    | Var x as v -> perform
      (_, env) <-- get;
      let v = try Var (List.assoc x env)
        with Not_found -> v in
      return v
    | Lam (x, t) -> perform
      (fresh, env) <-- get;
      let x' = x ^ string_of_int fresh in
      put (fresh+1, (x, x')::env);
      t' <-- alpha_conv t;
      (fresh', _) <-- get;
      put (fresh', env);
      return (Lam (x', t'))
    | App (t1, t2) -> perform
      t1 <-- alpha_conv t1;
      t2 <-- alpha_conv t2;
      return (App (t1, t2))
  ```

  Function from terms to `StateM` monad.
  Seeing a variable does not change state but we need its new name.
  Free variables don't change name.
  We rename each bound variable.
  We need a fresh number.
  Remember new name, update number.
  We need to restore names, but keep the number fresh.
  Passing around of names and the currently fresh number is done by the monad.
  ```

- ```
  val test : term = Lam ("x", App (Lam ("x", App (Var "y",
  Var "x")), Var "x"))
  # let _ = StateM.run (alpha_conv test) (5, []);;
  - : term * (int * (string * string) list) =
  (Lam ("x5", App (Lam ("x6", App (Var "y", Var "x6")), Var
  "x5")), (7, []))
  ```
- If we separated the reader monad and the state monad, we would avoid the lines:

  ```
  (fresh', _) <-- get;
  put (fresh', env);
  ```

  Restoring the "reader" part `env` but preserving the "state" part `fresh`.
- The elegant way is to define the monad locally:

```
let alpha_conv t =
  let module S = StateM
    (struct type t = int * (string * string) list end) in
  let open S in
```
```

```
let rec aux = function
  | Var x as v -> perform
    (fresh, env) <-- get;
    let v = try Var (List.assoc x env)
      with Not_found -> v in
    return v
  | Lam (x, t) -> perform
    (fresh, env) <-- get;
    let x' = x ^ string_of_int fresh in
    put (fresh+1, (x, x')::env);
    t' <-- aux t;
    (fresh', _) <-- get;
    put (fresh', env);
    return (Lam (x', t'))
  | App (t1, t2) -> perform
    t1 <-- aux t1; t2 <-- aux t2;
    return (App (t1, t2)) in
run (aux t) (0, [])
```

Monad transformers

- Based on: http://lambda.jimpryor.net/monad_transformers/
- Sometimes we need merits of multiple monads at the same time, e.g. monads `AM` and `BM`.
- Straightforwad idea is to nest one monad within another:
  - either `'a AM.monad BM.monad`
  - or `'a BM.monad AM.monad`.
- But we want a monad that has operations of both `AM` and `BM`.
- It turns out that the straightforward approach does not lead to operations with the meaning we want.
- A *monad transformer* `AT` takes a monad `BM` and turns it into a monad `AT(BM)` which actually wraps around `BM` on both sides. `AT(BM)` has operations of both monads.

- We will develop a monad transformer `StateT` which adds state to a monad-plus. The resulting monad has all: `return`, `bind`, `mzero`, `mplus`, `put`, `get` and their supporting general-purpose functions.
  - There is no reason for `StateT` not to provide state to any flavor of monads. Our restriction to monad-plus is because the type/module system makes more general solutions harder.
- We need monad transformers in OCaml because "monads are contagious": although we have built-in state and exceptions, we need to use monadic state and exceptions when we are inside a monad.
  - The reason *Lwt* is both a concurrency and an exception monad.
- Things get *interesting* when we have several monad transformers, e.g. `AT`, `BT`, ... We can compose them in various orders: `AT(BT(CM))`, `BT(AT(CM))`, ... achieving different results.
  - With a single trasformer, we will not get into issues with multiple-layer monads...
  - They are worth exploring – especially if you plan a career around programming in Haskell.

- The state monad, using `(fun x -> ...) a` instead of `let x = a in ...`

```
type 'a state =
    store -> ('a * store)

let return (a : 'a) : 'a state =
    fun s -> (a, s)

let bind (u : 'a state) (f : 'a -> 'b state) : 'b state =
    fun s -> (fun (a, s') -> f a s') (u s)
```

- Monad `M` transformed to add state, in pseudo-code:

```
type 'a stateT(M) =
    store -> ('a * store) M
(* notice this is not an ('a M) state *)

let return (a : 'a) : 'a stateT(M) =
    fun s -> M.return (a, s)
```

Rather than returning, M.return

```
let bind(u:'a stateT(M))(f:'a->'b stateT(M)):'b stateT(M)=
    fun s -> M.bind (u s) (fun (a, s') -> f a s')
```

Rather than let-binding, M.bind

State transformer

```
module StateT (MP : MONAD_PLUS_OPS) (Store : sig type t end)
: sig
  type store = Store.t
  type 'a t = store -> ('a * store) MP.monad
  include MONAD_PLUS_OPS
  include STATE with type 'a t := 'a monad
                 and type store := store
  val run : 'a monad -> 'a t
  val runT : 'a monad -> store -> 'a MP.monad
end = struct
  type store = Store.t
```

Functor takes two modules – the second one provides only the storage type.

Exporting all the monad-plus operations and state operations.

Expose "what happened" – resulting states.

Run the state transformer – get the resulting values.

```
module M = struct
  type 'a t = store -> ('a * store) MP.monad
  let return a = fun s -> MP.return (a, s)
  let bind m b = fun s ->
    MP.bind (m s) (fun (a, s') -> b a s')
  let mzero = fun _ -> MP.mzero
  let mplus ma mb = fun s -> MP.mplus (ma s) (mb s)
end
include M
include MonadPlusOps(M)
let get = fun s -> MP.return (s, s)
let put s' = fun _ -> MP.return ((), s')
let runT m s = MP.lift fst (m s)
end
```

Lift the monad-plus operations.

Instead of just returning, MP.return.

Backtracking with state

For use with list monad or lazy list monad.

```
module HoneyIslands (M : MONAD_PLUS_OPS) = struct
  type state = {
    been_size: int;
    been_islands: int;
    unvisited: cell list;
    visited: CellSet.t;
    eaten: cell list;
    more_to_eat: int;
  }

  let init_state unvisited more_to_eat = {
    been_size = 0;
    been_islands = 0;
    unvisited;
    visited = CellSet.empty;
    eaten = [];
    more_to_eat;
  }
```

```
module BacktrackingM =
  StateT (M) (struct type t = state end)
open BacktrackingM

let rec visit_cell () = perform
    s <-- get;
    match s.unvisited with
    | [] -> return None
    | c::remaining when CellSet.mem c s.visited -> perform
      put {s with unvisited=remaining};
      visit_cell ()
  | c::remaining (* when c not visited *) -> perform
      put {s with
        unvisited=remaining;
        visited = CellSet.add c s.visited};
      return (Some c)
```

State update actions.

Throwaway argument because of recursion. See (*)

This action returns a value.

```
let eat_cell c = perform
    s <-- get;
    put {s with eaten = c::s.eaten;
      visited = CellSet.add c s.visited;
      more_to_eat = s.more_to_eat - 1};
    return ()
```

Remaining state update actions just affect the state.

```
let keep_cell c = perform
    s <-- get;
    put {s with
      visited = CellSet.add c s.visited;
      been_size = s.been_size + 1};
    return ()

let fresh_island = perform
    s <-- get;
    put {s with been_size = 0;
      been_islands = s.been_islands + 1};
    return ()
```

```
let find_to_eat n island_size num_islands empty_cells =
  let honey = honey_cells n empty_cells in
```

OCaml does not realize that `'a monad` with state is actually a function – it's an abstract type.(*)

```
 let rec find_board () = perform
      cell <-- visit_cell ();
      match cell with
      | None -> perform
          s <-- get;
          guard (s.been_islands = num_islands);
          return s.eaten
      | Some cell -> perform
          fresh_island;
          find_island cell;
          s <-- get;
          guard (s.been_size = island_size);
          find_board ()
```

```
and find_island current = perform
    keep_cell current;
    neighbors n empty_cells current
    |> foldM
        (fun () neighbor -> perform
            s <-- get;
            whenM (not (CellSet.mem neighbor s.visited))
              (let choose_eat = perform
                  guard (s.more_to_eat > 0);
                  eat_cell neighbor
              and choose_keep = perform
                  guard (s.been_size < island_size);
                  find_island neighbor in
              choose_eat ++ choose_keep)) () in
```

The partial answer sits in the state – throwaway result.

```
    let cells_to_eat =
      List.length honey - island_size * num_islands in
    init_state honey cells_to_eat
    |> runT (find_board ())

end

module HoneyL = HoneyIslands (ListM)
let find_to_eat a b c d =
  ListM.run (HoneyL.find_to_eat a b c d)
```

Probabilistic Programming

- Using a random number generator, we can define procedures that produce various output. This is **not functional** – mathematical functions have a deterministic result for fixed arguments.
- Similarly to how we can "simulate" (mutable) variables with state monad and non-determinism (i.e. making choices) with list monad, we can "simulate" random computation with probability monad.
- The probability monad class means much more than having randomized computation. We can ask questions about probabilities of results. Monad instances can make tradeoffs of efficiency vs. accuracy (exact vs. approximate probabilities).
- Probability monad imposes limitations on what approximation algorithms can be implemented.
  - Efficient *probabilistic programming* library for OCaml, based on continuations, memoisation and reified search trees: http://okmij.org/ftp/kakuritu/index.html

The probability monad

- The essential functions for the probability monad class are `choose` and `distrib` – remaining functions could be defined in terms of these but are provided by each instance for efficiency.
- Inside-monad operations:
  - `choose : float -> 'a monad -> 'a monad -> 'a monad`

    `choose p a b` represents an event or distribution which is a with probability p and is b with probability $1 - p$.
  - `val pick : ('a * float) list -> 'a t`

    A result from the provided distribution over values. The argument must be a probability distribution: positive values summing to 1.
  - `val uniform : 'a list -> 'a monad`

    Uniform distribution over given values.
  - `val flip : float -> bool monad`

    Equal to `choose 0.5 (return true) (return false)`.
  - `val coin : bool monad` Equal to `flip 0.5`.

- And some operations for getting out of the monad:
  - `val prob : ('a -> bool) -> 'a monad -> float`

    Returns the probability that the predicate holds.
  - `val distrib : 'a monad -> ('a * float) list`

    Returns the distribution of probabilities over the resulting values.
  - `val access : 'a monad -> 'a`

    Samples a *random* result from the distribution – **non-functional** behavior.
- We give two instances of the probability monad: exact distribution monad, and sampling monad, which can approximate distributions.
  - The sampling monad is entirely non-functional: in Haskell, it lives in the IO monad.
- The monad instances indeed represent probability distributions: collections of positive numbers that add up to 1 – although often `merge` rather than `normalize` is used. If `pick` and `choose` are used correctly.

- ```
  module type PROBABILITY = sig
    include MONAD_OPS
    val choose : float -> 'a monad -> 'a monad -> 'a monad
    val pick : ('a * float) list -> 'a monad
    val uniform : 'a list -> 'a monad
    val coin : bool monad
    val flip : float -> bool monad
    val prob : ('a -> bool) -> 'a monad -> float
    val distrib : 'a monad -> ('a * float) list
    val access : 'a monad -> 'a
  end
  ```
  Probability monad class.
  ```

- ```
  let total dist =
    List.fold_left (fun a (_,b)->a+.b) 0. dist
  ```
  Helper functions.

  ```
  let merge dist =
    map_reduce (fun x->x) (+.) 0. dist
  ```
  Merge repeating elements.

  ```
  let normalize dist =
    let tot = total dist in
    if tot = 0. then dist
    else List.map (fun (e,w)->e,w/.tot) dist
  ```
  Normalize a measure into a distribution.

  ```
  let roulette dist =
    let tot = total dist in
    let rec aux r = function [] -> assert false
      | (e,w)::_ when w <= r -> e
      | (_,w)::tl -> aux (r-.w) tl in
    aux (Random.float tot) dist
  ```
  Roulette wheel from a distribution/measure.
  ```

- ```
  module DistribM : PROBABILITY = struct
    module M = struct      Exact probability distribution – naive implementation.
      type 'a t = ('a * float) list
      let bind a b = merge              x w.p. p and then y w.p. q happens =
        [y, q*.p | (x,p) <- a; (y,q) <- b x]        y results w.p. pq.
      let return a = [a, 1.]                                    Certainly a.
    end
    include M include MonadOps (M)
    let choose p a b =
      List.map (fun (e,w) -> e, p*.w) a @
        List.map (fun (e,w) -> e, (1. -.p)*.w) b
    let pick dist = dist
    let uniform elems = normalize
      (List.map (fun e->e,1.) elems)
    let coin = [true, 0.5; false, 0.5]
    let flip p = [true, p; false, 1. -. p]
  ```
  ```

```
  let prob p m = m
    |> List.filter (fun (e,_) -> p e)          All cases where p holds,
    |> List.map snd |> List.fold_left (+.) 0.             add up.
  let distrib m = m
  let access m = roulette m
end
```

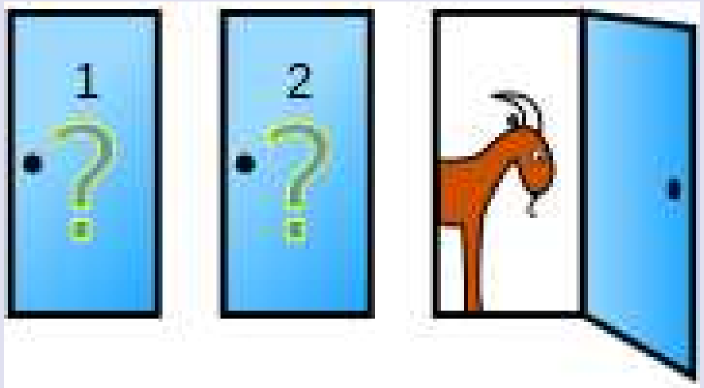
- ```
  module SamplingM (S : sig val samples : int end)
    : PROBABILITY = struct                 Parameterized by how many samples
    module M = struct                      used to approximate prob or distrib.
      type 'a t = unit -> 'a               Randomized computation – each call a()
      let bind a b () = b (a ()) ()        is an independent sample.
      let return a = fun () -> a           Always a.
    end
    include M include MonadOps (M)
    let choose p a b () =
      if Random.float 1. <= p then a () else b ()
    let pick dist = fun () -> roulette dist
    let uniform elems =
      let n = List.length elems in
      fun () -> List.nth (Random.int n) elems
    let coin = Random.bool
    let flip p = choose p (return true) (return false)
  ```
  ```

```
  let prob p m =
    let count = ref 0 in
    for i = 1 to S.samples do
      if p (m ()) then incr count
    done;
    float_of_int !count /. float_of_int S.samples
  let distrib m =
    let dist = ref [] in
    for i = 1 to S.samples do
      dist := (m (), 1.) :: !dist done;
    normalize (merge !dist)
  let access m = m ()
end
```

Example: The Monty Hall problem

- http://en.wikipedia.org/wiki/Monty_Hall_problem:

  > In search of a new car, the player picks a door, say 1. The game host then opens one of the other doors, say 3, to reveal a goat and offers to let the player pick door 2 instead of door 1.



- ```
  module MontyHall (P : PROBABILITY) = struct
    open P
    type door = A | B | C
    let doors = [A; B; C]

    let monty_win switch = perform
        prize <-- uniform doors;
        chosen <-- uniform doors;
        opened <-- uniform
          (list_diff doors [prize; chosen]);
        let final =
          if switch then List.hd
            (list_diff doors [opened; chosen])
          else chosen in
        return (final = prize)
  end
  ```
- ```
  module MontyExact = MontyHall (DistribM)
  module Sampling1000 =
    SamplingM (struct let samples = 1000 end)
  module MontySimul = MontyHall (Sampling1000)
  ```

- ```
  # let t1 = DistribM.distrib (MontyExact.monty_win false);;
  val t1 : (bool * float) list =
    [(true, 0.333333333333333315); (false, 0.66666666666666663)]
  # let t2 = DistribM.distrib (MontyExact.monty_win true);;
  val t2 : (bool * float) list =
    [(true, 0.66666666666666663); (false, 0.333333333333333315)]
  # let t3 = Sampling1000.distrib (MontySimul.monty_win false);;
  val t3 : (bool * float) list = [(true, 0.313); (false, 0.687)]
  # let t4 = Sampling1000.distrib (MontySimul.monty_win true);;
  val t4 : (bool * float) list = [(true, 0.655); (false, 0.345)]
  ```
  ```

Conditional probabilities

- Wouldn't it be nice to have a monad-plus rather than a monad?
- We could use `guard` – conditional probabilities!
  - $P(A|B)$
    - Compute what is needed for both A and B.
    - Guard B.
    - Return A.
- For the exact distribution monad it turns out very easy – we just need to allow intermediate distributions to be unnormalized (sum to less than 1).
- For the sampling monad we use rejection sampling.
  - `mplus` has no straightforward correct implementation.
- We implemented `PROBABILITY` separately for educational purposes only, as `COND_PROBAB` introduced below supersedes it.

- ```
  module type COND_PROBAB = sig
    include PROBABILITY
    include MONAD_PLUS_OPS with type 'a monad := 'a monad
  end
  ```
  Class for conditional probability monad, where guard cond conditions on cond.

- ```
  module DistribMP : COND_PROBAB = struct
    module MP = struct
      type 'a t = ('a * float) list
     let bind a b = merge
        [y, q*.p | (x,p) <- a; (y,q) <- b x]
      let return a = [a, 1.]
      let mzero = []
      let mplus = List.append
    end
    include MP include MonadPlusOps (MP)
    let choose p a b =
      List.map (fun (e,w) -> e, p*.w) a @
        List.map (fun (e,w) -> e, (1. -.p)*.w) b
    let pick dist = dist
  ```
  The measures no longer restricted to probability distributions:

  Measure equal 0 everywhere is OK.

  It isn't a w.p. p & b w.p. $(1-p)$ since a and b are not normalized!

```
  let uniform elems = normalize
    (List.map (fun e->e,1.) elems)
  let coin = [true, 0.5; false, 0.5]
  let flip p = [true, p; false, 1. -. p]
  let prob p m = normalize m
    |> List.filter (fun (e,_) -> p e)
    |> List.map snd |> List.fold_left (+.) 0.
  let distrib m = normalize m
  let access m = roulette m
end
```

Final normalization step.

- We write the rejection sampler in mostly imperative style:

```
module SamplingMP (S : sig val samples : int end)
  : COND_PROBAB = struct
  exception Rejected                              For rejecting current sample.
  module MP = struct          Monad operations are exactly as for SamplingM
    type 'a t = unit -> 'a
    let bind a b () = b (a ()) ()
    let return a = fun () -> a
    let mzero = fun () -> raise Rejected        but now we can fail.
    let mplus a b = fun () ->
      failwith "SamplingMP.mplus not implemented"
  end
  include MP include MonadPlusOps (MP)
```

```ocaml
let choose p a b () =
  if Random.float 1. <= p then a () else b ()
let pick dist = fun () -> roulette dist
let uniform elems =
  let n = List.length elems in
  fun () -> List.nth elems (Random.int n)
let coin = Random.bool
let flip p = choose p (return true) (return false)
```

Inside-monad operations don't change.

```ocaml
let prob p m =
  let count = ref 0 and tot = ref 0 in
  while !tot < S.samples do
    try
      if p (m ()) then incr count;
      incr tot
    with Rejected -> ()
  done;
  float_of_int !count /. float_of_int S.samples
```

Getting out of monad: handle rejected samples.

Count up to the required number of samples.

`m()` can fail.

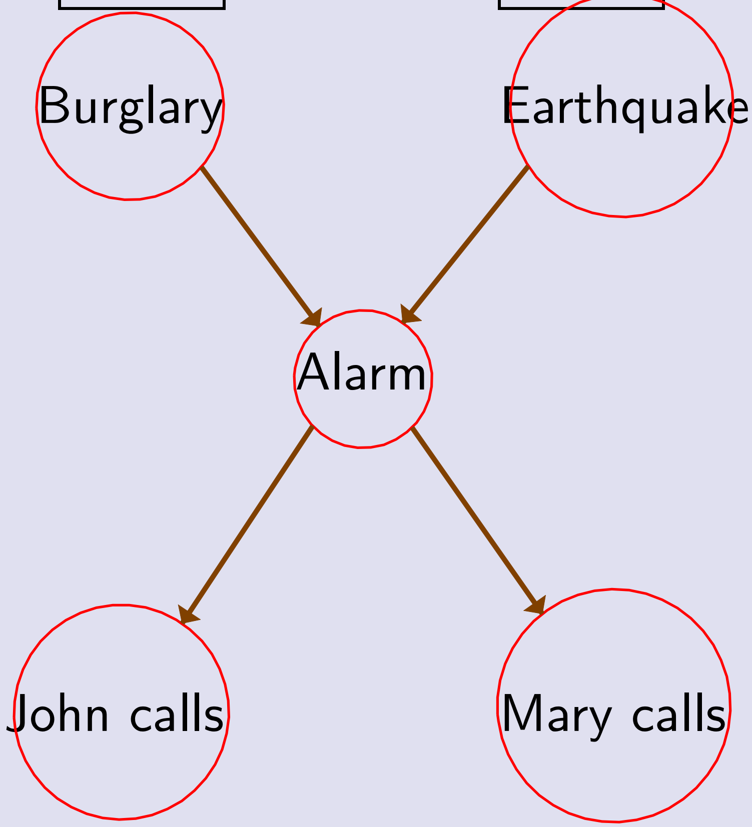
But if we got here it hasn't.

Rejected, keep sampling.

```
  let distrib m =
    let dist = ref [] and tot = ref 0 in
    while !tot < S.samples do
      try
        dist := (m (), 1.) :: !dist;
        incr tot
      with Rejected -> ()
    done;
    normalize (merge !dist)
  let rec access m =
    try m () with Rejected -> access m
end
```

Burglary example: encoding a Bayes net

- We're faced with a problem with the following dependency structure:



$P(B)$
0.001

$P(E)$
0.002

B	E	$P(A\|B,E)$
F	F	0.001
F	T	0.29
T	F	0.94
T	T	0.95

A	$P(J\|A)$
F	0.05
T	0.9

A	$P(M\|A)$
F	0.01
T	0.7

- Alarm can be due to either a burglary or an earthquake.
- I've left on vacations.
- I've asked neighbors John and Mary to call me if the alarm rings.
- Mary only calls when she is really sure about the alarm, but John has better hearing.
- Earthquakes are twice as probable as burglaries.
- The alarm has about 30% chance of going off during earthquake.
- I can check on the radio if there was an earthquake, but I might miss the news.

- ```
  module Burglary (P : COND_PROBAB) = struct
    open P
    type what_happened =
      Safe | Burgl | Earthq | Burgl_n_earthq

    let check ~john_called ~mary_called ~radio = perform
      earthquake <-- flip 0.002;
      guard (radio = None || radio = Some earthquake);
      burglary <-- flip 0.001;
      let alarm_p =
        match burglary, earthquake with
        | false, false -> 0.001
        | false, true -> 0.29
        | true, false -> 0.94
        | true, true -> 0.95 in
      alarm <-- flip alarm_p;
  ```
  ```

```
    let john_p = if alarm then 0.9 else 0.05 in
    john_calls <-- flip john_p;
    guard (john_calls = john_called);
    let mary_p = if alarm then 0.7 else 0.01 in
    mary_calls <-- flip mary_p;
    guard (mary_calls = mary_called);
    match burglary, earthquake with
    | false, false -> return Safe
    | true, false -> return Burgl
    | false, true -> return Earthq
    | true, true -> return Burgl_n_earthq
end
```

- ```
  module BurglaryExact = Burglary (DistribMP)
  module Sampling2000 =
    SamplingMP (struct let samples = 2000 end)
  module BurglarySimul = Burglary (Sampling2000)
  ```
  ```

```
# let t1 = DistribMP.distrib
  (BurglaryExact.check ~john_called:true ~mary_called:false
     ~radio:None);;
    val t1 : (BurglaryExact.what_happened * float) list =
  [(BurglaryExact.Burgl_n_earthq, 1.03476433660005444e-05);
   (BurglaryExact.Earthq, 0.00452829235738691407);
   (BurglaryExact.Burgl, 0.00511951049003530299);
   (BurglaryExact.Safe, 0.99034184950921178)]
# let t2 = DistribMP.distrib
  (BurglaryExact.check ~john_called:true ~mary_called:true
     ~radio:None);;
    val t2 : (BurglaryExact.what_happened * float) list =
  [(BurglaryExact.Burgl_n_earthq, 0.000574372565004057794);
   (BurglaryExact.Earthq, 0.175492465840075218);
   (BurglaryExact.Burgl, 0.283597462799388911);
   (BurglaryExact.Safe, 0.540335698795532)]
# let t3 = DistribMP.distrib
  (BurglaryExact.check ~john_called:true ~mary_called:true
     ~radio:(Some true));;
    val t3 : (BurglaryExact.what_happened * float) list =
  [(BurglaryExact.Burgl_n_earthq, 0.00326224160214999262);
   (BurglaryExact.Earthq, 0.99673775839785006)]
```

```
# let t4 = Sampling2000.distrib
  (BurglarySimul.check ~john_called:true ~mary_called:false
     ~radio:None);;
    val t4 : (BurglarySimul.what_happened * float) list =
  [(BurglarySimul.Earthq, 0.0035); (BurglarySimul.Burgl, 0.0035);
   (BurglarySimul.Safe, 0.993)]
# let t5 = Sampling2000.distrib
  (BurglarySimul.check ~john_called:true ~mary_called:true
     ~radio:None);;
    val t5 : (BurglarySimul.what_happened * float) list =
  [(BurglarySimul.Burgl_n_earthq, 0.0005); (BurglarySimul.Earthq, 0.1715);
   (BurglarySimul.Burgl, 0.2875); (BurglarySimul.Safe, 0.5405)]
# let t6 = Sampling2000.distrib
  (BurglarySimul.check ~john_called:true ~mary_called:true
     ~radio:(Some true));;
    val t6 : (BurglarySimul.what_happened * float) list =
  [(BurglarySimul.Burgl_n_earthq, 0.0015); (BurglarySimul.Earthq, 0.9985)]
```

Lightweight cooperative threads

- `bind` is inherently sequential: `bind a (fun x -> b)` computes `a`, and resumes computing `b` only once the result `x` is known.
- For concurrency we need to "suppress" this sequentiality. We introduce

  ```
  parallel :
  'a monad-> 'b monad-> ('a -> 'b -> 'c monad) -> 'c monad
  ```

  where `parallel a b (fun x y -> c)` does not wait for `a` to be computed before it can start computing `b`.
- It can be that only accessing the value in the monad triggers the computation of the value, as we've seen in some monads.
  - The state monad does not start computing until you "get out of the monad" and pass the initial value.
  - The list monad computes right away – the `'a monad` value is the computed results.

  In former case, a "built-in" `parallel` is necessary for concurrency.

- If the monad starts computing right away, as in the *Lwt* library, `parallel` e_a e_b c is equivalent to

```
perform
  let a = e_a in
  let b = e_b in
  x <-- a;
  y <-- b;
  c x y
```

  - We will follow this model, with an imperative implementation.
  - In any case, do not call `run` or `access` from within a monad.

- We still need to decide on when concurrency happens.
  - Under **fine-grained** concurrency, every `bind` is suspended and computation moves to other threads.
    - It comes back to complete the `bind` before running threads created since the `bind` was suspended.
    - We implement this model in our example.
  - Under **coarse-grained** concurrency, computation is only suspended when requested.
    - Operation `suspend` is often called `yield` but the meaning is more similar to `Await` than `Yield` from lecture 7.
    - Library operations that need to wait for an event or completion of IO (file operations, etc.) should call `suspend` or its equivalent internally.
    - We leave coarse-grained concurrency as exercise 11.

- The basic operations of a multithreading monad class.

```
module type THREADS = sig
  include MONAD
  val parallel :
    'a t -> 'b t -> ('a -> 'b -> 'c t) -> 'c t
end
```

- Although in our implementation `parallel` will be redundant, it is a principled way to make sure subthreads of a thread are run concurrently.

- All within-monad operations.

```
module type THREAD_OPS = sig
  include MONAD_OPS
  include THREADS with type 'a t := 'a monad
  val parallel_map :
    'a list -> ('a -> 'b monad) -> 'b list monad
  val (>||=) :
    'a monad -> 'b monad -> ('a -> 'b -> 'c monad) ->
    'c monad
  val (>||) :
    'a monad -> 'b monad -> (unit -> 'c monad) ->
    'c monad
end
```

- Outside-monad operations.

```
module type THREADSYS = sig
  include THREADS
  val access : 'a t -> 'a
  val kill_threads : unit -> unit
end
```

- Helper functions.

```
module ThreadOps (M : THREADS) = struct
  open M
  include MonadOps (M)
  let parallel_map l f =
    List.fold_right (fun a bs ->
      parallel (f a) bs
        (fun a bs -> return (a::bs))) l (return [])
  let (>||=) = parallel
  let (>||) a b c = parallel a b (fun _ _ -> c ())
end
```

- Put an interface around an implementation.

```
module Threads (M : THREADSYS) :
sig
  include THREAD_OPS
  val access : 'a monad -> 'a
  val kill_threads : unit -> unit
end = struct
  include M
  include ThreadOps(M)
end
```

- Our implementation, following the *Lwt* paper.

```
module Cooperative = Threads(struct
  type 'a state =
  | Return of 'a                      The thread has returned.
  | Sleep of ('a -> unit) list        When thread returns, wake up waiters.
  | Link of 'a t                      A link to the actual thread.
  and 'a t = {mutable state : 'a state}   State of the thread can change
                                      – it can return, or more waiters can be added.
  let rec find t =
    match t.state with                Union-find style link chasing.
    | Link t -> find t
    | _ -> t

  let jobs = Queue.create ()          Work queue – will store
                                      unit -> unit procedures.
```

```
let wakeup m a =
  let m = find m in
  match m.state with
  | Return _ -> assert false
  | Sleep waiters ->
    m.state <- Return a;
    List.iter ((|>) a) waiters
  | Link _ -> assert false

let return a = {state = Return a}
```

Thread `m` has actually finished – updating its state.

Set the state, and only then wake up the waiters.

```
let connect t t' =
  let t' = find t' in
  match t'.state with
  | Sleep waiters' ->
    let t = find t in
    (match t.state with
    | Sleep waiters ->
      t.state <- Sleep (waiters' @ waiters);
      t'.state <- Link t
    | _ -> assert false)
  | Return x -> wakeup t x
  | Link _ -> assert false
```

t was a placeholder for t'.

If both sleep, collect their waiters and link one to the other.

If t' returned, wake up the placeholder.

```
let rec bind a b =
  let a = find a in
  let m = {state = Sleep []} in
  (match a.state with
  | Return x ->
    let job () = connect m (b x) in
    Queue.push job jobs
  | Sleep waiters ->
    let job x = connect m (b x) in
    a.state <- Sleep (job::waiters)
  | Link _ -> assert false);
  m
```

The resulting monad.

If a returned, we suspend further work. (In exercise 11, this should only happen after suspend.)

If a sleeps, we wait for it to return.

```
let parallel a b c = perform
  x <-- a;
  y <-- b;
 c x y
```

Since in our implementation the threads run as soon as they are created, `parallel` is redundant.

```
  let rec access m =
    let m = find m in
    match m.state with
    | Return x -> x
    | Sleep _ ->
      (try Queue.pop jobs ()
       with Queue.Empty ->
         failwith "access: result not available");
      access m
    | Link _ -> assert false

  let kill_threads () = Queue.clear jobs
end)
```

Accessing not only gets the result of `m`, but spins the thread loop till `m` terminates.

No further work.

Perform suspended work.

Remove pending work.

- ```
  module TTest (T : THREAD_OPS) = struct
    open T
    let rec loop s n = perform
      return (Printf.printf "-- %s(%d)\n%!" s n);
      if n > 0 then loop s (n-1)
      else return ()
  end
  module TT = TTest (Cooperative)
  ```

  We cannot use whenM because the thread would be created regardless of condition.

- ```
  let test =
    Cooperative.kill_threads ();
    let thread1 = TT.loop "A" 5 in
    let thread2 = TT.loop "B" 4 in
    Cooperative.access thread1;
    Cooperative.access thread2
  ```

  Clean-up after previous tests.

  We ensure threads finish computing before we proceed.

```
# let test =
    Cooperative.kill_threads ();
    let thread1 = TT.loop "A" 5 in
    let thread2 = TT.loop "B" 4 in
    Cooperative.access thread1;
    Cooperative.access thread2;;
-- A(5)
-- B(4)
-- A(4)
-- B(3)
-- A(3)
-- B(2)
-- A(2)
-- B(1)
-- A(1)
-- B(0)
-- A(0)
val test : unit = ()
```