Nondeterministic and probabilistic programming with monads and iterative deepening

MPRI 2-4 programming project

Version 1.0, December 18, 2014

Prologue

The band U2 has a concert that starts in 17 minutes and they must all cross a bridge to get there. They stand on the same side of the bridge. It is night. There is one flashlight. A maximum of two people can cross at one time, and they must have the flashlight with them. The flashlight must be walked back and forth (no throwing the flashlight across the bridge). A pair walks together at the rate of the slower man's pace:

Bono	1 minute to cross
Edge	2 minutes to cross
Adam	5 minutes to cross
Larry	10 minutes to cross

For example: if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.

How would you schedule the moves of the band members so that they have all crossed the bridge in 17 minutes or less? Can you find an answer within 5 minutes?

(Turn the page after 5 minutes.)

Introduction

The purpose of this programming project is to develop libraries and tools to help solve problems like the "bridge crossing" puzzle. More precisely, we are going to support nondeterministic or probabilistic programming in a functional language through the development of appropriate monads, following and extending the approach outlined in X. Leroy's MPRI 2-4 lectures.

Nondeterministic programming focuses on the generation of possible candidate results, using choice between various possibilities, combined with elimination of choices that do not satisfy the constraints of the problem. In the end, all combinations of choices that satisfy the constraints are produced.

Probabilistic programming is similar, but probabilities are attached to the choices and tracked during evaluation. The constraints of the problem are observations that eliminate impossible cases and affect the a posteriori distribution of the results. In the end, the a posteriori probabilities of the results are produced.

Logistics

A number of templates and test files, written in OCaml, are provided in directory `src/` and described in this document. The templates contain a number of holes, marked `failwith "TODO"`. Your goal is to fill the holes in the templates and obtain a working implementation.

This assignment is structured in a number of tasks. Each task is designed so that it can be tested independently, without having completed the following tasks.

The tasks of sections 3, 4, and 5 are independent and can be completed in any order.

We strongly suggest you use OCaml as the programming language, if only because the templates are written in OCaml and provide a lot of boring code (e.g. printing of results) that you don't need to implement. If you really must, you can use another functional programming language, but make sure to reimplement the same functionality as the OCaml code, e.g. with respect to printing and formatting of the test results.

No Makefile is provided but you can just use `ocamlbuild`. E.g. `ocamlbuild Bools.native` in directory `src/` will build the executable corresponding to the test `Bools.ml`.

What to turn in Once you are done, mail `Xavier.Leroy@inria.fr` a tar or zip archive containing:

- the `.ml` and `.mli` files from `src/` with your code inside;
- preferably, a short `README` or `LISEZMOI` file telling which tasks you completed, how well you think your code works, and any other comment we should know about.

The following command will build the correct archive:

```
tar czf Firstname_Lastname.tgz README src/*.ml src/*.mli
```

1 The naive monad for nondeterminism

Study file `src/Nondet.ml`. It defines the generic interface for a nondeterminism monad (module type `NONDET`) and one naive implementation based on the "list" monad shown in the lecture. (Two more advanced implementation are outlined in the file and will be worked on in sections 2 and 5.) The interface declares:

- A type `'a mon` of monadic computations that produce values of type `'a`.
- The monadic operations `ret` and `bind`. You can write

  ```
  a >>= fun x -> b
  ```

  instead of `bind a (fun x -> b)`.
- The `choice` function, which provides nondeterministic choice between zero, one or several possibilities, represented as a list of monadic computations.
- Derived functions `fail` and `either`. `fail` represents failure and is equivalent to `choice []` (no possibilities). `either a b` is choice between `a` and `b`; it is equivalent to `choice [a;b]`. You can write `a ||| b` instead of `either a b`.
- Two fixpoint combinators, `fix` and `fixparam`, whose need will become apparent in section 2. You can ignore them for now.
- A `run` operation that takes a monadic computation of type `'a mon` and produces 1- a list of possible `'a` result values, and 2- a boolean indicating whether the list contains all possible results (the boolean is `true`) or whether some results were possibly missed (the boolean is `false`). `run` has an extra integer parameter representing the maximal depth of exploration of the results. Its use will become apparent in section 2.
- A `print_run` operation that calls `run` and displays the results with the help of a printing function of type `'a -> unit` provided as first parameter. All our example will use `print_run` to run monadic computations and print the results.

The module `Naive` is our first implementation of this `NONDET` monadic interface. Like in the lecture, it implements `'a mon` as `'a list`. In other words, a nondeterministic computation producing results of type `'a` is just the list of all possible results (an `'a list`). Read the implementation to reacquaint you with this monad.

Task 1 Use the `Nondet.Naive` monad to solve the "bridge crossing" puzzle. A template is given in `src/Puzzle.ml`. Fill the missing parts in `src/Puzzle.ml` and run your program to find out the answer to the puzzle. There are exactly two answers.

2 Choice trees

The representation of nondeterministic computations by a list of values (with strict evaluation) is inefficient if we need to generate a lot of possibilities, or even an infinite number of them, then filter out a few. For example, we cannot ask for "any positive integer" or "any list of booleans", then select among these sets.

We now develop a different implementation of nondeterminism, where monadic computations produce a tree-shaped, lazily-evaluated representation of the possible choices. Thunks, i.e. functions of type `unit` $\to \tau$, are used heavily to delay evaluation until absolutely necessary. Given this tree representation, `run` and `print_run` can explore the tree up to a given depth to collect possible results. This exploration is often incomplete, if only because the tree is often infinite. By varying the depth of exploration, we can tune the amount of results obtained (iterative deepening).

Consider module `Tree` in file `src/Nondet.ml`. It implements the type `'a mon` of monadic computations as:

```
type 'a mon = unit -> 'a case list
```

Note the `unit ->` that corresponds to a thunk, in order to delay evaluation. Each element of the list returned by the thunk is not a value of type `'a` but an element of type

```
and 'a case = Val of 'a | Susp of 'a mon
```

The `Val` case corresponds to a fully-known value. The `Susp` case corresponds to further possibilities that have not been explored yet. The `Susp` constructor carries a thunk that, when applied, produces further cases. So, the argument of `Susp` has type `unit -> 'a case list`, which is just `'a mon`.

Example Consider the following element of type `int mon`:

```
m = fun () -> [Val 1; Susp p; Susp q]
p = fun () -> [Val 2]
q = fun () -> [Val 3; Susp r]
r = fun () -> []
```

`m` is a thunk that, when applied to `()`, produces immediately one possible result value `1` and two suspensions `p` and `q`. Applying `p` and `q` to `()` gives two more possible values, `2` and `3`, and yet another thunk `r`. Applying `r` to `()` produces no other possibility.

Task 2.1 Implement the monadic operations `ret`, `bind`, `choice`, `fail` and `either` in the `Tree` module. Hint: in several places you will have a choice between eagerly constructing lists of cases or lazily generating a `Susp` that delays this construction. It is recommended to be as lazy as possible, so that all nontrivial monadic computation steps are materialized as `Susp` nodes in the choice tree.

Now that we have constructed our choice trees, it remains to explore them to finite depth. The key function to implement is

```
flatten: int -> 'a mon -> 'a case list
```

which forces the evaluation of the thunks contained in a tree (second argument) up to a given depth (first argument), collecting the results in a single list of cases.

Example Continuing the `m` example above,

```
flatten 0 m = [Val 1; Susp p; Susp q]
flatten 1 m = [Val 1; Val 2; Val 3; Susp r]
flatten 2 m = [Val 1; Val 2; Val 3]
```

The `run` operation is, then, defined by post-processing the list of cases produced by `flatten`. `Val` elements of this list are collected to form the first result of `run` (the list of possible values). The second result of `run`, the boolean indicating exhaustiveness, depend on whether the result of `flatten` contains `Susp` elements (as in `flatten 1 m` above) or not (as in `flatten 2 m` above).

Task 2.2 Implement `run` and `print_run` in monad `Tree` as outlined above. Test your implementation on the following two examples: 1- `src/Bools.ml` (a simple combinatorial exploration of 3 booleans), 2- `src/Puzzle.ml` (your solution to the puzzle, after replacing `open Nondet.Naive` with `open Nondet.Tree` at the top of the file).

Despite its lazyness, the `Tree` monad still has problems defining infinite enumerations such as "all lists of booleans" or "all integers greater or equal to n". Consider:

```
let rec any_bool_list =
    ret []
||| (any_bool >>= fun hd ->
     any_bool_list >>= fun tl -> ret (hd :: tl))

let rec any_int n =
    ret n ||| any_int (n + 1)
```

The first recursive definition is rejected by Caml because the right-hand side is not syntactically a function. The second recursive definition is accepted but diverges when applied, because of strict evaluation.

We can work around these issues with eta-expansion:

```
let rec any_bool_list () =
    (ret [] ||| ...) ()
let rec any_int n () =
    (ret n ||| any_int (n + 1)) ()
```

However, it is nicer (and more resilient to changes in implementation of the monad) to use *fixed-point combinators* provided by the monad, instead. This is the role of the `fix` and `fixparam` combinators of the `Tree` monad. They let us write

```
let any_bool_list = fix (fun any_bool_list ->
    choice [ret [];
            (any_bool_list >>= fun tl -> ret (false :: tl));
            (any_bool_list >>= fun tl -> ret (true :: tl))]

let any_int = fixparam (fun any_int n ->
    ret n || any_int (n + 1)
```

Task 2.3 Implement the `fix` and `fixparam` combinators of the `Tree` monad. Test them on the `src/Sumless.ml` example (which involves enumerating all positive integers) and on the `src/Append.ml` example (discussed in the next section).

3 A taste of Prolog

Logic programming languages such as Prolog have an advantage over functional languages: a given Prolog predicate can be used in several "modes", thus implementing several functionalities that would each require a specific function definition in a functional language. Consider for example list concatenation:

```
append([],L,L).
append([X|L1],L2,[X|L3])  :-  append(L1,L2,L3).
```

(`[X|L]` means "`H` cons `L`".) If we give ground lists as the first two arguments, Prolog computes the third argument, which is the concatenation of the two lists. However, if we give the first and third arguments, Prolog will compute their "difference". Finally, if we give the third argument only, Prolog computes all the ways to split it into two lists that concatenate back to the argument.

Nondeterministic functional programming can emulate some of this "reversibility". Consider the example `src/Append.ml`. It shows how to find all the ways to decompose a given list L into the concatenation of two lists, just by enumerating all pairs of lists (L_1, L_2) and keeping only those such that $L_1 \mathbin{@} L_2 = L$. This works only for short lists L, as the complexity of the enumeration is exponential. (But see section 6 for further improvements.)

Another area where "reversibility" comes handy is type checking and type inference. Consider the simply-typed, implicitly-typed λ-calculus with constants:

Terms: $a ::= N \mid x \mid \lambda x.a \mid a_1\ a_2$

Types: $\tau ::= \texttt{int} \mid \tau_1 \to \tau_2$

$$\Gamma \vdash N : \texttt{int} \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma, x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \lambda x.a : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \qquad a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau}$$

Determining whether a term has a given type, or determining the type(s) of a term, both appear to involve some amount of "guessing" the types involved in the typing derivation(s). The clever way to eliminate this "guessing" is to perform constraint-based or unification-based type inference, as shown in the MPRI 2-4 lectures. However, in a pinch, generating all the possible types for type unknowns can do.

Task 3.1 Consider file `src/Typing.ml`. Fill in the blanks so as to implement nondeterministic functions that 1- infer the type(s) of a term, and 2- check that a term has a given type.

Pleasantly, we can also "reverse-execute" those functions to e.g. find (closed) terms that have a given type.

Task 3.2 In file `src/Typing.ml`, add a nondeterministic generator for terms. Use this generator to find many closed terms that have type $\texttt{int} \to \texttt{int}$.

Interlude: a game of Bayesian Cluedo

Zoe is found murdered in her house. Alice and Bob are the two suspects. Based on their respective feelings towards Zoe, the police estimates that Alice has a 30% chance of being the murderer and Bob 70%. Then, a length of pipe is found near Zoe's body. Bob, being into firearms, has a 80% probability of having used a gun and only 20% of having used a pipe. Alice, on the other hand, likes to practice plumbing, hence she has a 97% chance of having used a pipe and only 3% of having used a gun. Who is the more likely culprit?

4 Probabilistic programming

Probabilistic programming is very much like nondeterministic programming, except that the various alternatives of a choice are weighted by the probabilities of taking the alternative. For example, a biased coin flip is expressed as

```
distr [(Heads, 0.4); (Tails, 0.6)]
```

meaning that we get `Heads` with probability 0.4 and `Tails` with probability 0.6.

Study file `src/Proba.ml`. It defines the generic interface for a monad of probability distributions (module type `PROBA`). Instead of just computing the set of all possible results, we want to compute the distribution of the result: every possible result, and for each one, its probability. The type of distributions is, therefore

```
type 'a distribution = ('a * prob) list
with prob = float
```

Your goal is to implement this monad using the lazy choice tree approach of section 2. The type `'a mon` of monadic computations is, now,

```
type 'a mon = unit -> 'a case distribution
and 'a case = Val of 'a | Susp of 'a mon
```

That is, every case is now weighted by its probability. Note that in the returned `'a case distribution`, the probabilities do not necessarily sum to 1, because failure cases are not represented in the list.

Task 4.1 Implement the `Tree` module in file `src/Proba.ml`. In the `flatten` function, be very careful to correctly combine probabilities when expanding a `Susp` case. In the `run` function, remember to normalize the probabilities in the final distribution so that they sum to 1. Test your implementation with the simple examples in file `src/Probatests.ml`.

Task 4.2 Use your probability distribution monad to model and solve the game of Bayesian Cluedo above. Put your solution in file `src/Cluedo.ml`.

5 Imperative nondeterministic programming

Perhaps surprisingly, mutable references and other imperative programming features are compatible with nondeterministic and probabilistic programming. For simplicity, we now forget about probabilities and return to plain nondeterminism.

Consider the following example of imperative nondeterministic programming, where we write `getref` and `setref` for the usual dereferencing and assignment operations over references:

```
(setref r false ||| setref r true) >>= fun _ -> ...
```

or, equivalently,

```
(ret false ||| ret true) >>= fun b -> setref r b >>= fun _ -> ...
```

The net effect is to evaluate the `...` expression in two different states, one where `getref r` returns `false`, the other where it returns `true`.

As the example above suggests, we cannot use Caml's built-in references here, because they are global and hold only one value at any given time. Instead, we must materialize the state as an explicit store component in our nondeterminism monad, so that different states can be associated to different cases of an alternative. This leads to the following type for our monadic computations:

```
type 'a mon = Store.t -> ('a case * Store.t) list
and 'a case = Val of 'a | Susp of 'a mon
```

A monadic computation is now a store transformer, taking the initial store and returning a list of possible cases, along with the corresponding final stores. Again, it is crucial that different cases can have different final stores.

A somewhat clever implementation of stores (finite maps from references to values) is provided in files `src/Store.mli` and `src/Store.ml`. Read the interface `src/Store.mli` to understand the store operations and their semantics.

Task 5 Implement the `TreeState` monad in `src/Nondet.ml`. It extends the `Tree` monad with the operations `newref` (to create a fresh reference), `getref` (to query the value of a reference) and `setref` (to change the value of a reference). Note that references do not need to be initialized when created: doing `getref` on a fresh reference over which no `setref` has been done simply fails. Test your implementation with the `src/Sumless_imp.ml` example.

6 Monadic data structures and memoization

This part of the project is significantly more difficult than the preceding parts. It is for extra credit.

We now return to Prolog-style reversibility (section 3). The generate-then-filter approach we followed is quite inefficient because our generation strategy for recursive data structures is inefficient. Taking lists of booleans as an example, we basically generate a huge alternative

```
    []
||| [false]
||| [true]
```

```
||| [false; false]
||| [false; true]
||| [true; false]
||| ...
```

The number of cases is exponential in the length of the list. Each of these cases is, then, filtered by applying constraints, but it is too late: we have already generated too many cases.

We are looking for alternative approaches where we could filter over partially-defined lists such as "`true` cons any list" and discard them if the head of the list does not match the constraints, without generating the tail. One such alternative is to define lists as follows (see file `src/Mlist.ml`):

```
type 'a mlist = 'a mlist_content mon
and 'a mlist_content = Nil | Cons of 'a * 'a mlist
```

Every list cell is a monadic (= nondeterministic) computation that produces either `Nil` or `Cons` of a value and another monadic computation for the tail of the list. The familiar list constructors are:

```
let nil : 'a mlist = ret Nil
let cons (hd: 'a) (tl: 'a mlist) : 'a mlist = ret (Cons(hd, tl))
```

and the generator for boolean mlists is:

```
let any_bmlist : bool mlist = fix (any_bmlist ->
  choose [nil; cons false any_bmlist; cons true any_bmlist])
```

As a trivial example of efficient filtering over mlists, consider:

```
let isnil (l: 'a mlist) : unit mon =
  l >>= function Nil -> ret () | Cons(_, _) -> fail

let _ =
  print_run (fun () -> printf "OK!") 1000 (isnil any_bmlist)
```

Despite the big depth 1000, this run executes instantly, and moreover it is exhaustive. That's because the `isnil` test eliminated all non-nil mlists in one single test!

Task 6.1 Implement the following functions over mlists: conversions to and from ordinary lists, concatenation `append`, equality. See `src/Mlist.ml` for the types of these functions. Using these functions, solve the question "given a mlist L, does there exists two mlists L_1, L_2 such that $L =$ `append` L_1 L_2 ?". Check that your solver is relatively efficient, i.e. quadratic (not exponential) in the length of L.

So far, mlists give us efficient answers to questions of the form "does there exist an mlist such that ...?". However, they do not help answering questions of the form "what are the mlists such that ...?". Continuing the trivial `isnil` example, assume we want to observe the mlists that pass the `isnil` test:

```
let nil_mlists =
  let l = any_bmlist in
  isnil l >>= fun _ -> list_of_mlist l
```

Running `nil_mlists` will attempt to print all boolean mlists, not just the empty one! The reason is *use-site choice*: even though `any_bmlist` is let-bound to `l`, the two uses of `l` represent independent generation processes. The first use, as argument to `isnil`, is filtered so that only the empty mlist remains. But the second use is independent and still produces all mlists.

We can, however, implement *definition-site choice* by memoizing the choices made as some use site and sharing them with the other use sites. For this memoization and sharing, we can use the store of the `TreeState` monad. Memoization is presented as a combinator `memo` that takes any `'a mon` and returns an equivalent `'a mon` with memoization of the possible results. The following example illustrates the effect of `memo`:

```
let without_memo : bool mon =
  let any_b = ret false ||| ret true in
  any_b >>= fun b -> if b then any_b else fail

let with_memo : bool mon =
  let any_b = memo (ret false ||| ret true) in
  any_b >>= fun b -> if b then any_b else fail
```

`with_memo` produces only the boolean `true`, while `without_memo` produces both `false` and `true`.

Task 6.2 Implement the `memo` combinator in the `TreeState` monad of file `src/Nondet.ml`. Test it using the simple examples in `src/Testmemo.ml`.

The next step is to return to mlists and add memoization in the generator for boolean mlists. Ideally we would like to write

```
let any_bmlist () : bool mlist = fix (fun any_bmlist ->
  memo (choice [nil; cons false any_bmlist; cons true any_bmlist]))
```

You can try that with the examples in `src/Mlist.ml`, but chances are that `fix` and `memo` don't play well together, with `fix` delaying the creation of the memo location and therefore losing sharing. In the end, you will probably need to implement a specific `fixmemo` combinator for fixed points with sharing, and write

```
let any_bmlist () : bool mlist = fixmemo (fun any_bmlist ->
  choice [nil; cons false any_bmlist; cons true any_bmlist])
```

Task 6.3 Implement the `fixmemo` combinator in the `TreeState` monad of file `src/Nondet.ml`. Test it with the `nil_mlists` example from `src/Mlist.ml`. Define `split_mlist` in the same file and make sure that it produces all splits efficiently.

Can we play the same game with the type inference example? That is, represent type expressions using the monad, like we did for mlists?

Task 6.4 Fill in the blanks in `src/Mtyping.ml` and see if this alternate representation of types improves the speed and completeness of the typing function.