Functional Pearl: the Proof Search Monad

Jonathan Protzenko

Microsoft Research
protz@microsoft.com

Abstract

We present the proof search monad, a set of combinators that allows one to write a proof search engine in a style that resembles closely the inference rules. The user calls functions such as `premise`, `prove` or `choice`; the library then takes care of generating a derivation tree, which serves either as a witness (success) or as an error message (failure). Proof search engines written in this style enjoy: first, a one-to-one correspondence between the implementation and the theoretical rules, which makes manual inspection easier; second, proof witnesses "for free", which makes a verified-validation approach easier too.

1 A minimal problem

We consider conjunctions of equalities of the form $\mathcal{V}x_1, \ldots, \mathcal{V}x_n.\ \bigwedge_k x_i = x_j$, that is, conjunctions of equalities between variables. Some variables are *flexible*, meaning they may be substituted for other variables. Their binder $\mathcal{V}x_i$ is of the form $\mathcal{F}\mathbf{x}_i$ (and their are typeset in bold font). Other variables are *rigid*, and may not be substituted. Their binder $\mathcal{V}x_i$ is of the form $\mathcal{R}x_i$. The problem consists in computing a substitution of *flexible* variables that evaluate the conjunction to `true`. If no such substitution exists, the solver outputs nothing.

For instance, one may want to solve: $\mathcal{R}x, \mathcal{R}z, \mathcal{F}\mathbf{y}.\ x = \mathbf{y} \wedge z = z$. A solution exists: the solver outputs $\sigma = \{\mathbf{y} \mapsto x\}$ as a valid substitution that solves the input problem. However, if one attempts to solve: $\mathcal{R}x, \mathcal{R}z, \mathcal{F}\mathbf{y}.\ x = \mathbf{y} \wedge \mathbf{y} = z$, the solver fails to find a proper substitution, and return nothing. Indeed, the first clause demands that $\mathbf{y}$ substitutes to x, meaning that the second clause becomes $x = z$, which always evaluates to false (x and z are two distinct rigid variables).

In proof search, rigid variables stem from the right-elimination of universal quantifiers, or left-elimination of existential quantifiers, which both result in abstract variables. When left-eliminating a universal quantifier (respectively right-eliminating an existential quantifier), one must provide an argument to the type application (resp. an existential witness). If the argument (resp. the witness) cannot be found on the spot, one typically uses *flexible variables*, which allow the proof search procedure to *defer* the choice until a later point in search (where the choice may be guessed). Flexible variables are thus an *implementation technique*.

In order to simplify the discussion, we skip elimination rules and quantifiers altogether, and just assume that our input problem contains a combination of flexible and rigid variables. Furthermore, we assume that any instantiation of a flexible variable is legal. The only crucial point is that the solver outputs something.

(The library that we describe here has been used in the implementation of the Mezzo type-checker. There, all introduction and elimination rules are present; furthermore, the implementation uses *levels* to guarantee that only sound instantiation choices are performed. This is, however, orthogonal to the present discussion; the minimal problem we consider already conveys all the important points, so we will skip quantifiers in the remainder of the discussion.)

Solving conjunctions of equalities requires three rules, which make up our logic (Figure 1). Refl embodies the reflexivity axiom; And highlights that the proof produces an *output* V'

$$\textsc{Refl}\ \frac{}{V \vdash x = x \dashv V} \qquad \textsc{Inst}\ \frac{x \in V_i \qquad V_1, \mathcal{F}(\mathbf{y} = x), V_2 \vdash [\mathbf{y}/x]P \dashv V'}{V_1, \mathcal{F}(\mathbf{y}), V_2 \vdash P \dashv V'} \qquad \textsc{And}\ \frac{V \vdash P \dashv V' \qquad V' \vdash Q \dashv V''}{V \vdash P \wedge Q \dashv V''}$$

Figure 1: Proof rules for the first system

```
type formula =
| Equals of var * var
| And of formula * formula

and var = P.point
```

```
and descr =
  | Flexible
  | Rigid

and state = descr P.state
```

Figure 2: Formulas and state

which is *chained* from one premise to another; Inst embodies the "instantiation" mechanism: in essence, if y has not been instantiated so far, one may instantiate it onto any other variable and perform the substitution in order to prove the goal ($[\mathbf{y}/x]P$ stands for "replace y with x in P").

Once the proof tree has been laid out, one obtains an output (e.g. V'' in And) where the instantiation choices (e.g. $\mathcal{F}(\mathbf{x}_i = x_j)$) determine the desired substitution (e.g. $\sigma = \{\mathbf{x}_i \mapsto x_j\}^*$).

We implement proof search in OCaml (Figure 2). The data type of formulas is self-explanatory. Variables are implemented as equivalence classes in a *persistent* union-find data structure, which the module `P` implements. The V parameter in our rules is embodied by the `state` type; just like the V parameter is chained from one premise to another (And), `state` is an input and an output to the solver. Just like the V parameter in the rules, a `state` of the persistent union-find represents a set of equations between variables. In a sense, `state` is a specific implementation of the theoretical V parameter. It represents a substitution; in other words, this is what we want our solver to compute.

The choice of a persistent union-find is irrelevant. All that matters is that we pick a data structure that models substitutions, and that is *persistent*. Had we picked an explicit substitution instead of a union-find, the rest of the discussion would have been the same.

Figure 3 implements a solver for our minimal problem; written within the `MOption` monad, it returns either `Some state` (in case a successful substitution has been found), or `None` if no solution exists. The solver is complete.

The solver uses `MOption.>>=` to sequence premises in the `And` case. It doesn't keep track of premises; it just ensures (thanks to `>>=`) that if the first premise evaluates to `nothing`, the second premise is not evaluated, since it is suspended behind a `fun` expression (OCaml is a strict language).

2 Building derivations

There are two shortcomings with this solver. First, the `unify` sub-routine conflates several rules of the logic together. Indeed, the `return (P.union ...)` expression hides a combination of Inst and Refl. Second, we have no way to replay the proof to verify it independently. One may argue that in this simplified example, one can just apply the substitution to the original

```
module MOption = struct
  (* ... defines [return], [nothing] and [>>=] *)
end

let unify state v1 v2 =
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
      return (P.union v1 v2 state)
  | Rigid, Flexible ->
      return (P.union v2 v1 state)
  | Rigid, Rigid ->
      if P.same v1 v2 state then
        return state
      else
        nothing
```

```
let rec solve state formula =
  match formula with
  | Equals (v1, v2) ->
      unify state v1 v2
  | And (f1, f2) ->
      solve state f1 >>= fun state ->
      solve state f2
```

Figure 3: Solver for the simplified problem

```
module type LOGIC = sig
  type formula
  type rule_name
end
```

```
module MakeProofTree (L: LOGIC) = struct
  type derivation = L.formula * rule
  and rule = L.rule_name * premises
  and premises = Premises of derivation list
end
```

```
module MyLogic = struct
  type formula = ... (* as before *)
  type rule_name = R_And | R_Refl | R_Inst
end
module MyProofTree = MakeProofTree(MyLogic)
```

Figure 4: The functor of proof trees

formula and verify that all the clauses are of the form $x = x$, without the need for a proof tree. In the general case, however, the proof tree contains the elimination witnesses for quantifiers; this allows one to independently verify a proof without relying on inference techniques, such as flexible variables.

2.1 Defining proof trees

One way to make the solver better is to make sure each step it performs corresponds in an obvious manner to the application of an admissible rule. To that effect, we define the data type of all three rules in our system, which we apply to the functor of *proof trees* (Figure 4).

A `derivation` tree is a pair of a `formula` (the goal we wish the prove) and a `rule` (that we apply in order to prove the goal). A `rule` has a name and `premises`; the `premises` type is simply a `derivation list` (the `Premises` constructor is here to prevent a non-constructive type abbreviation). When using the library, the client is expected to make sure that each `rule_name` is paired with the proper number of premises (0 for Refl, 1 for Inst and 2 for And); this is not enforced by the type system.

In the (simplified) sketch from Figure 4, rule names are just constant constructors, since the

```
module WriterT (M: MONAD) (L: MONOID): sig
  type 'a m = (L.a * 'a) M.m
  val return: 'a -> 'a m
  val ( >>= ): 'a m -> ('a -> 'b m) -> 'b m

  val tell: L.a -> unit m
end = ...

module M = MOption
module MWriter = WriterT(M)(L)
```

```
module L = struct
  type a = MyProofTree.derivation list
  let empty = []
  let append = List.append
end
```

Figure 5: The writer monad transformer

rule parameters (such as x and `y` in INST) can be recovered from the `formula`. In the general case, the various constructors of `rule_name` do have parameters that record how one specific rule was instantiated.

2.2 Proof tree combinators

We previously used the `>>=` operator from the `MOption` monad in order to chain premises (Figure 3). We now need a new operator, that not only *binds* the result (i.e. stops evaluating premises after a failure, as before), but also *records* the premises in sequence, in order to build a proper derivation. The former is still faithfully implemented by the option monad; the latter is implemented by the writer monad.

Computations in the writer monad return a result (of type `'a`) along with a log of elements (of type `L.a`). The (usual) `>>=` and `return` combinators operate on the result part of the computation, while the (new) `tell` combinator operates on the logging part of the computation. This `tell` combinator appends a new element to the log. Appending elements to the log is done by way of the `MONOID` module type, which essentially demands a value for the empty log, and a function to append new entries into the log.

In order to get a new `>>=` operator that combines the features of the option and writer monads, we apply the `WriterT` monad transformer to the `MOption` monad (Figure 5) and obtain `MWriter`, a monad whose computations represent a sequence of derivations (the premises we have proved so far) along with a result (the `state` that we chain through the premises). These computations are wrapped in `MOption.m`, that is, are wrapped within an `option` to account for a possible proof failure.

A computation within this new monad has type (simplified after functor applications) `(derivation list * state) option`. It represents a given point in the proof; the solver is focused on a given rule, has reached a certain state, after proving a certain list of premises.

Once all the premises have been proved, one needs to draw a horizontal line and reach the conclusion of the proof. That is, take the final state and the list of premises, and generate a `derivation` that stands for the application of the entire rule.

Contrary to the first implementation (Figure 3), where the working state and the return value of `solve` both had type `state option`, we now distinguish between an `outcome` (the result of a call to `solve`) and a working state (a computation in the monad).

An outcome is, as we mentioned earlier, the pair of a final `state` along with a `derivation` that justifies that we reached this state. The pair is optional since, after all, not all formulas are provable.

```
(* This snippet is in the [MWriter(M)(L)] monad. Upon a first reading, think
   [module M = MOption]. *)
type 'a outcome = ('a * derivation) M.m

let premise (outcome: 'a outcome): 'a m =
  M.bind outcome (fun (state, derivation) ->
    tell [ derivation ] >>= fun () ->
    return state
  )

let prove (goal: goal) (rule: rule_name) (x: 'a m): 'a outcome =
  M.(x >>= fun (premises, state) ->
    return (state, (goal, (rule, Premises premises))))

let axiom (state: 'a) (goal: goal) (axiom: rule_name): 'a outcome =
  prove goal axiom (return state)

let fail: 'a outcome =
  M.nothing
```

Figure 6: The high-level combinators for building proof derivations

The type `outcome` (Figure 6) is parametric: it works for any state that the client code uses. In other words, our library is generic with regards to the particular `state` type the client uses.

We now have a duality between the `outcome` type (the result of solving a sub-goal) and the `m` type (a computation within the monad, i.e. a working state between two premises). Therefore, we introduce two high-level combinators: `premise` and `prove`. The former goes from `outcome` to `m`: it injects a new sub-goal as a premise of the rule we are currently trying to prove. The latter goes from `m` to `outcome`: if all premises have been satisfied, it draws the horizontal line that builds a new node in the derivation tree.

- `premise` is the composition of `tell`, which records the derivation for this sub-goal, and `return`, which passes the state on to the next sub-goal.
- `prove` is a computation in the `M` monad (here, `MOption`). If all the premises have been satisfied, it bundles them as a new node of the derivation tree. If a premise failed, then `x` is `M.nothing`, and `prove` also returns a failed outcome.
- `axiom` is short-hand for a rule that requires no premises.
- `fail` is for situations where no rule applies: this is a failed outcome.

2.3 A solver in the new style

Figure 7 demonstrates an implementation of `solve` in the new style. Compared to the previous implementation (Figure 3):

- `prove_equality` makes it explicit which rules are applied, and singles out two distinct rule applications in the flexible-rigid case;
- the premises of each rule are clearly identified;
- axioms and failure conditions are explicit,
- the `And` case is easy to review manually, to make sure that no premise was forgotten.

```
let rec prove_equality (state: state) (goal: formula) (v1: var) (v2: var) =
  let open MOption in
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
      let state = P.union v1 v2 state in
      prove goal R_Instantiate begin
        premise (prove_equality state goal v1 v2) >>=
        return
      end
  (* ... *)
  | Rigid, Rigid ->
      if P.same v1 v2 state then
        axiom state goal R_Refl
      else
        fail

let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  | Equals (v1, v2) ->
      prove_equality state goal v1 v2
  | And (g1, g2) ->
      prove goal R_And begin
        premise (solve state g1) >>= fun state ->
        premise (solve state g2) >>=
        return
      end
```

Figure 7: A solver written using the high-level combinators

This is, as mentioned previously, a minimal example that showcases the usage of the library. In the implementation of Mezzo, switching the core of the type-checker to this style revealed several bugs where premises were not properly chained or simply forgotten.

3 Backtracking

3.1 Limitations of the option monad

We now extend our formulas with disjunctions (Figure 9). A consequence is that we now need our base monad `M` to offer a new operation; namely, one that, among several possible choices, picks the first one that is not a failure. We thus augment `MOption` with a search combinator (Figure 9), which in turn allows one to implement a high-level `choice` combinator for our library. The `choice` combinator attempts to prove a `goal` by trying a function `f` on several arguments `a`, each of which is associated to a given rule. We can add one more branch to the `solve` function, which attempts to prove a disjunction by first trying a left-elimination (Or-L, Figure 8), then a right-elimination (Or-R).

The solver can now solve problems of the form $\mathcal{R}x, \mathcal{R}z, \mathcal{F}\mathbf{y}.\ x = z \vee \mathbf{y} = z$. It fails, however, to solve problems of the form $\mathcal{R}x, \mathcal{R}z, \mathcal{F}\mathbf{y}.\ (\mathbf{y} = x \vee \mathbf{y} = z) \wedge \mathbf{y} = z$. The reason is, the option monad is not powerful enough: upon finding a suitable choice in the disjunction case, it commits to it and drops the other one. In other words, when hitting the disjunction, `MOption` commits

$$\textsc{Or-L}\ \frac{V \vdash P \dashv V'}{V \vdash P \vee Q \dashv V'} \qquad \textsc{Or-R}\ \frac{V \vdash Q \dashv V'}{V \vdash P \vee Q \dashv V'}$$

Figure 8: New proof rules for disjunction

```
(* We extend formulas with disjunctions. *)
type formula =
  (* ... *)
  | Or of formula * formula

(* The logic is also extend with two rules. *)
type rule_name =
  (* ... *)
  | R_OrL
  | R_OrR

module MOption = struct
  (* ... *)
  let rec search f = function
    | [] -> None
    | x :: xs ->
        match f x with
        | Some x -> Some x
        | None -> search f xs
end

(* Equipped with [search], we define the high-level [choice] combinator... *)
let choice (goal: goal) (args: (rule_name * 'a) list) (f: 'a -> 'b m): 'b outcome =
  M.search (fun (r, x) -> prove goal r (f x)) args

(* ...which one uses as follows: *)
let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  (* ... *)
  | Or (g1, g2) ->
      choice goal [ R_OrL, g1; R_OrR, g2 ] (fun g ->
        premise (solve state g) >>=
        return
      )
```

Figure 9: The choice combinator

```
module LL = LazyList
module MExplore
  type 'a m = 'a LL.t
  let return = LL.one
  let ( >>= ) = LL.flattenl (LL.map f x)
  let nothing = LL.nil
  let search f l = LL.bind (LL.of_list l) f
end
```

Figure 10: The exploration monad

to $\sigma = \{\mathbf{y} \mapsto x\}$, instead of keeping $\sigma = \{\mathbf{y} \mapsto z\}$ as a backup solution. Phrased yet again differently, we need to replace `MOption` with the non-determinism monad that will implement *backtracking*.

3.2 The exploration monad

Conceptually, we want to change our way of thinking; instead of thinking of `solve` as a function that returns *a solution*, we now think of it as a function that returns *several possible solutions*. The state is now a set of states, each of which represent a possible choice.

The monad of non-determinism is implemented using lists; OCaml is a strict language, so we write the non-determinism monad (also known as the exploration or backtracking monad) using lazy lists (Figure 10).

The reader can now go back and replace `module M = MOption` with `module M = MExplore` in Figure 5. The rest of the library remains unchanged; the `solve` function (the client code) is also unchanged; and the combinators of the library now implement backtracking.

In particular, the earlier example of $\mathcal{R}x, \mathcal{R}z, \mathcal{F}\mathbf{y}.\ (\mathbf{y} = x \vee \mathbf{y} = z) \wedge \mathbf{y} = z$ is now successfully solved by the library. Thanks to laziness, no extra computations occur; further solutions down the lazy list are only evaluated if the first ones failed.

4 Conclusion

References