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. 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, independent validation approach easier too.

1 Theory and practice

1.1 A minimal theory

We are concerned with proving the validity of logical formulas; that is, with writing a search procedure that determines whether a given goal is satisfiable. To get started, we consider a system made up of conjunctions of equalities, along with existential quantifiers. Any free variables are assumed to be universally quantified. For instance, one may want to prove the following formula:

$$\exists y.\ x = y \tag{1}$$

In order to show the validity of this judgement, one will build a proof derivation using the rules from the logic, shown in Figure 1 ($[x/y]P$ means "substitute x with y in P"). For instance, proving Equation 1 requires applying ExistsE, then Refl.

These rules embody the Truth of our logic, i.e. an omniscient reader may use them to show with absolute certainty that a given formula is true. However, if one wants to algorithmically determine whether a given formula is true, ExistsE is useless. Indeed, unless the algorithm (solver) is equipped with superpowers, it cannot magically guess, out of the blue, a suitable x in ExistsE that will ensure the remainder of the derivation succeeds. To put it another way, x is a free variable (a parameter) of ExistsE; the whole point of writing a proof search algorithm is to 1) find that ExistsE is the right rule to apply, and 2) find that x is a suitable value for instantiating y, because it will make $y = x$ succeed.

Hence, in order to build a *search procedure* for that logic, one will use another set of *algorithmic* rules, which hopefully enjoy:

$$\textsc{Refl}\ \frac{}{x = x} \qquad \textsc{And}\ \frac{P \qquad Q}{P \wedge Q} \qquad \textsc{ExistsE}\ \frac{[x/y]P}{\exists y.\ P}$$

Figure 1: A simple logic

$$\textsc{Refl}\ \frac{}{V, \sigma \vdash x = x \dashv \sigma} \qquad \textsc{Subst}\ \frac{V, \sigma \vdash \sigma P \dashv \sigma'}{V, \sigma \vdash P \dashv \sigma'} \qquad \textsc{Inst}\ \frac{x \in V \qquad y^? \in V \qquad y^? \notin \sigma \qquad V, \{y^? \mapsto x\} \circ \sigma \vdash P \dashv \sigma'}{V \vdash P \dashv \sigma'} \qquad \textsc{And}\ \frac{V, \sigma \vdash P \dashv \sigma' \qquad V, \sigma' \vdash Q \dashv \sigma''}{V \vdash P \wedge Q \dashv \sigma''}$$

$$\textsc{ExistsE}\ \frac{V \uplus y^?, \sigma \vdash P \dashv \sigma'}{V, \sigma \vdash \exists y.\ P \dashv \sigma'_{|V}}$$

Figure 2: Algorithmic proof rules

soundness : if the algorithmic rules succeed, then there exists a derivation in the logic that proves the validity of the original formula, and

completeness : if the algorithmic rules fail, then there exists no derivation in the logic that would prove the validity of the original formula.

For instance, in our logic of existentially-quantified conjunctions of equalities, one may want to use the algorithmic rules from Figure 2. These rules differ from Figure 1 in that they are algorithmic; they take an input and return an output.

In particular, in order to determine suitable values for the x parameter in ExistsE, the implementation reasons in terms of substitutions. V is a set of variables which may be substituted (recall that free variables are considered universally quantified, hence not eligible for substitution); variables that may be substituted are typeset as $y^?$. The algorithm has internal state, that is, it carries a substitution σ. Upon hitting an existential quantifier $y^?$, the algorithmic rules *open* $y^?$ and mark it as eligible for substitution. Later on (for instance, upon hitting $y^? = x$), the algorithm may pick a substitution for $y^?$ using Inst. A substitution may be applied at any time (Subst). The preconditions of Inst guarantee that the algorithm makes at most one choice for instantiating $y^?$.

In other words, the algorithmic rules *defer* the *instantiation* of the existential quantifier until the shape of the proof obligation gives us a *hint* as to what exactly this instantiation should be. This *implementation technique* is known as *flexible variables*.

The new algorithmic rules differ from the original logical rules significantly; first, there are five rules for the algorithmic system, compared to just three for the logical system. Second, these five rules do not map trivially to their counterparts in the logical system. Third, these rules are only algorithmic; the implementation that we are about to roll out uses an optimized representation for substitutions (union-find), meaning that one not only needs to check that the algorithmic rules are faithful to the logical rules, but also that the implementation is faithful to the algorithmic rules.

This paper presents a library that allows one to write an implementation of the algorithmic rules while automatically generating a derivation. The library forces the client code to lay out premises, rule applications and instantiations. The level of detail of the resulting derivation is left up to the client code; they may wish to record the very compact rules from the logic, or record more proof steps using the algorithmic rules. In any case, the derivation serves as a proof witness; if the user wishes to do so, they can write a validator that takes the witness and verifies that the derivation is, indeed, correct.

The library has been used, in a preliminary form, to implement the core of the Mezzo

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

and var = P.point

and descr =
  | Flexible
  | Rigid

and state = descr P.state
```

Figure 3: Formulas and state

type-checker. This paper presents a cleaned-up, isolated version of this library.

1.2 An implementation with flexible variables and union-find

The logic we present is a much simplified version of the logic (type system) of Mezzo. In particular, this paper only mentions the right-exists quantifier; Mezzo has all four possible combinations of left/right exists/forall. Generally, in proof search, *flexible variables* stem from the right-elimination of existential quantifiers, or the left-elimination of universal quantifiers. The right-elimination of universal quantifiers, or the left-elimination of existential quantifiers give rise to universally-quantified variables, which are called *rigid*.

In order to simplify the problem, we assume that all existential variables have been introduced as flexible variables already. That way, we won't be sidetracked, talking about binders and the respective merits of De Bruijn *vs.* locally nameless. Furthermore, we assume that all instantiations of flexible variables are legal. This is not true in general: for instance, if the goal is $\forall x, \exists y^?, \forall z.\ P$, picking $y^? = z$ makes no sense. Mezzo forbids this choice using *levels*; in the present document, we skip this discussion altogether and assume that "all is well". Finally, although in a general setting, several rules may trigger for a given goal (this is the case in Mezzo), the algorithmic set of rules we use is syntax-driven: the syntactic shape of the goal determines which rule should be applied.

We thus restrict our formulas to conjunctions of equalities between variables. The plan is to write a solver that takes, as an input, a formula, and outputs a valid substitution, if any. That is, write an algorithm that abides by the rules from Figure 2. For instance, one may want to solve: $x = y^? \wedge z = z$. A solution exists: the solver outputs $\sigma = \{y^? \mapsto x\}$ as a valid substitution that solves the input problem. However, if one attempts to solve: $x = y^? \wedge y^? = z$, the solver fails to find a proper substitution, and returns nothing. Indeed, the first clause demands that $y^?$ substitutes to x, meaning that the second clause becomes $x = z$, which always evaluates to false (x and z are two distinct rigid variables).

Once the algorithm has run, we obtain an output substitution σ. One can, if they wish to do so, take the reflexive-transitive closure σ^*, and apply it to a flexible variable (say, $y^?$) to recover the parameter of ExistsE that should be used in the logical rules (here, x). This is all very informal – the point of the subsequent sections is to formalize the claim that the proof search algorithm produces a proof witness.

We implement proof search in OCaml (Figure 3); we do not use explicit substitutions, but rather an optimized representation based on a union-find data structure. 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, σ parameters in our rules are embodied by the `state` type; just like the σ parameter is chained from one premise to another (And), `state` is an input and an output to the solver. Just like the σ 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 σ parameter. It represents a

```
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 4: Solver for the simplified problem

substitution; in other words, this is what we want our solver to compute.

The choice of a 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 4 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 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, the outputs substitution *is* the proof witness: one can just apply the substitution to the original 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 ExistsE rule, and proof witnesses are attached to arbitrary nodes of the tree. We thus need to build a properly annotated proof tree in the general case.

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 5).

We record applications of Inst, Refl and And. This produces a derivation tree that makes sure that the algorithm follows the algorithmic rules from Figure 2. Section 4 shows how to

```
(* These two modules belong to the library. *)
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

(* This is the client code using modules from the library. *)
module MyLogic = struct
  type formula = ... (* as before *)
  type rule_name = R_And | R_Refl | R_Inst
end
module MyProofTree = MakeProofTree(MyLogic)
```

Figure 5: The functor of proof trees (library and client code)

generate a slightly different tree that matches the rules from Figure 1.

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 5, rule names are just constant constructors, since the 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 4). 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 6) 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

```
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 6: The writer monad transformer (library code)

is focused on a given rule, has reached a certain state, after proving a certain list of premises.

We provide a convenience `qed` combinator: once one has obtained the final state, it pairs the state with the name of the rule we want to conclude with. It makes the implementation of `solve` (Figure 8) more elegant.

Once all the premises have been proven, 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 4), 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 wrapped in `M.t` (here, `option`): proving a formula may fail.

The type `outcome` (Figure 7) 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 8 demonstrates an implementation of `solve` in the new style. Compared to the previous implementation (Figure 4):

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

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

let qed r e =
  return (e, r)

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

Figure 7: The high-level combinators for building proof derivations (library code)

- `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.

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 10). 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 10), 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 9), then a right-elimination (Or-R).

The solver can now solve problems of the form $x = z \vee y^? = z$. It fails, however, to solve problems of the form $(y^? = x \vee y^? = z) \wedge y^? = z$. The reason is, the option monad is not powerful

```
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 begin
        premise (prove_equality state goal v1 v2) >>=
        qed R_Instantiate
      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 begin
        premise (solve state g1) >>= fun state ->
        premise (solve state g2) >>=
        qed R_And
      end
```

Figure 8: A solver written using the high-level combinators (client code)

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

Figure 9: New proof rules for disjunction

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 to $\sigma = \{y^? \mapsto x\}$, instead of keeping $\sigma = \{y^? \mapsto z\}$ 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 path in the search tree of derivation trees.

The monad of non-determinism is implemented using lists; OCaml is a strict language, so

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

(* The logic is also extended 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 [choice] library combinator... *)
let choice (goal: goal) (args: 'a list) (f: 'a -> ('b * rule_name) m): 'b outcome =
  M.search (fun x -> prove goal (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 (r, g) ->
        premise (solve state g) >>=
        qed r
      )
```

Figure 10: The choice combinator (library and client code)

```
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 11: The exploration monad

we write the non-determinism monad (also known as the exploration or backtracking monad) using lazy lists (Figure 11).

The reader can now go back and replace `module M = MOption` with `module M = MExplore`

in Figure 6. 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 $(y^? = x \vee y^? = z) \wedge y^? = z$ 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 Generating a proof tree using the logical rules

5 Conclusion

We presented a support library for writing a proof search engine using backtracking in any given logic; indeed, the library is parameterized by: the type of formulas; the type of rule applications; the internal state type of the client. By merely using the combinators of the library, the client gets derivations built for free; this allows a separate verifier to independently check the steps required to prove the formula. By opting into the library, the client gets to rewrite their code in a new syntactic style that makes rule application explicits, forbids "bundled" applications of multiple rules at the same time and clearly lays out the premises required to prove a judgement. Since the code resembles the logical rules, mistakes are easier to spot.

The logic presented in this paper is as simple as it gets. It does, however, highlight the main concepts. A version of this library is used in the core of Mezzo's type-checker. The version of the library used in Mezzo also builds failed derivations; these failed derivations stop at the first failed premise or, in case of a choice, list all the failed attempts. We have not yet explained this last feature as a clean combination of monads and operators, but hope to do so in the near future.

References