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 the formal rules closely. The user calls functions such as \texttt{premise}, \texttt{prove} or \texttt{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 derivation 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:

\[ \exists y. \, x = y \] (1)

In order to show the validity of this judgement, one usually builds a proof derivation using rules from the logic. In our case, the rules are given in Figure 1, where \([x/y]P\) means “substitute \(x\) with \(y\) in \(P\)”. For instance, proving Equation 1 requires applying \texttt{ExistsE}, then \texttt{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, \texttt{ExistsE} is useless. Indeed, unless the algorithm (solver) is equipped with superpowers, it cannot magically guess, out of the blue, a suitable \(x\) in \texttt{ExistsE} that will ensure the remainder of the derivation succeeds. To put it another way, \(x\) is a free variable (a parameter) of \texttt{ExistsE}; the whole point of writing a proof search algorithm is to 1) find that \texttt{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:

\[
\begin{array}{c}
\text{Refl} \\
\hline
x = x
\end{array}
\quad
\begin{array}{c}
\text{AND} \\
\hline
P \\
Q
\end{array}
\quad
\begin{array}{c}
\text{ExistsE} \\
\hline
[x/y]P
\end{array}
\]

\[ \exists y. \, P \]

Figure 1: A simple logic
### Algorithmic proof rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ref</strong></td>
<td>$V, \sigma \vdash x = x \sigma$</td>
<td>$V, \sigma \vdash P \sigma'$</td>
</tr>
<tr>
<td><strong>Subst</strong></td>
<td>$V, \sigma \vdash P \sigma'$</td>
<td>$V, {y^? \mapsto x} \circ \sigma \vdash P \circ \sigma'$</td>
</tr>
<tr>
<td><strong>Inst</strong></td>
<td>$x \in V$, $y^? \in V$, $y^? \notin \sigma$</td>
<td>$V \vdash P \sigma'$</td>
</tr>
<tr>
<td><strong>And</strong></td>
<td>$V, \sigma \vdash P \sigma'$</td>
<td>$V, \sigma' \vdash Q \sigma''$</td>
</tr>
<tr>
<td><strong>ExistsE</strong></td>
<td>$V \cup y^?, \sigma \vdash P \sigma'$</td>
<td>$V, \sigma \vdash \exists y^?, P \sigma'</td>
</tr>
</tbody>
</table>

![Figure 2: Algorithmic proof rules](image-url)

**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 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 $\sigma$. Upon hitting an existential quantifier $y^?$, the algorithmic rules open $y^?$ and mark it as eligible for substitution (**ExistsE**). 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 some sub-goal, later on, 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 still very much abstract; the implementation that we are about to roll out uses an optimized representation for substitutions (union-find) that is not formalized in Figure 2. Phrased differently, one not only needs to check that the algorithmic rules are faithful to the proof rules, but also that the implementation itself 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; the user may wish to record a proof derivation using the proof rules, or record a trace of the algorithm using the algorithmic rules. In any case, the derivation serves as a witness; in the case of a proof derivation, a validator may certify that the proof is valid, while in the case of an algorithmic trace, the user may verify the algorithm, or inspect the trace for debugging or feedback purposes.
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 [9]. In the present document, we only mention the right-exists quantifier. General systems such as Mezzo have all four possible combinations of left/right exists/forall. The right-elimination of existential quantifiers, or the left-elimination of universal quantifiers gives flexible variables, while the right-elimination of universal quantifiers, or the left-elimination of existential quantifiers gives universally-quantified variables, also called rigid variables.

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 [10]; 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: \( x = y' \wedge y'^{\prime} = z \), the solver fails to find a proper substitution, and returns nothing. Indeed, the first clause demands that \( y'^{\prime} \) 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 \( \sigma \). One can, if they wish to do so, take the reflexive-transitive closure \( \sigma^* \), and apply it to a flexible variable (say, \( y' \)) to recover the parameter of \( \exists y' \) that should be used in the logical rules (here, \( x \)). This way of checking correctness is not satisfactory and does not scale if nested quantifiers appear in the goal; the point of the subsequent sections is to make sure the search algorithm produces a proper proof witness (derivation) that has a tree-like structure and does not leak implementation details (such as substitutions).

We implement proof search in OCaml [7] (Figure 3); we implement substitutions using a union-find data structure [1, 12]. The data type of formulas is self-explanatory. Variables are implemented as equivalence classes in the persistent union-find data structure, which the module P implements. The \( V, \sigma \) parameters in our rules are embodied by the state type; just

```
| type formula = and descr |
| | Equals of var * var |
| | Flexible |
| | And of formula * formula |
| | Rigid |
| and var = P.point |
| and state = descr P.state |
```

Figure 3: Formulas and state
module MOption = struct
  (* ... defines [return], [nothing] and [>>=] *)
end

let unify state v1 v2 =
  let rec solve state formula =
    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 v1 v2 state)
    | Rigid, Rigid ->
      if P.same v1 v2 state then
        return state
      else
        nothing
    match formula with
    | Flexible, Flexible
    | Equals (v1, v2) ->
      unify state v1 v2
    | And (f1, f2) ->
      solve state f1 >>= fun state ->
        solve state f2
  state

Figure 4: Solver for the simplified problem

like the $\sigma$ parameter is chained from one premise to another (AND). $\text{state}$ is an input and an output to the solver. Just like the $\sigma$ parameter in the rules, a $\text{state}$ of the persistent union-find represents a set of equations between variables. The algorithmic rules mentioned a theoretical $\sigma$ parameter; the $\text{state}$ is our specific implementation choice.

The choice of a union-find (as opposed to explicit substitutions) is irrelevant. All that matters is that we pick a data structure that models substitutions, and that the structure be persistent.

Figure 4 implements a solver for our minimal problem; since we perform computations that either return a result or a failure, the code leads itself well to an implementation using monads \([13, 14]\), in our case, the MOption monad. The Some $\text{state}$ is for success, meaning a substitution has been found, while the None case means 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 EXISTE 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 (algorithm trace) that makes sure that the algorithm follows the algorithmic rules from Figure 2. Section 4 shows how to generate a different, more compact tree that matches the rules from Figure 1.

A derivation tree is a pair of a formula (the goal we wish to 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 (Section 4), 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. The tell combinator appends a new element to the log; this 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
The Proof Search Monad

J. Protzenko

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)

---

monads, we apply the WriterT monad transformer to the MOption monad (Figure 6) and obtain MWriter.

The type of computations 'a MWriter.t boils down (after functor application) to (derivation list * state) option.

A computation in the monad represents a given point in the proof: the solver is focused on a rule; has proved a number of premises so far (the derivation list); has reached a certain state (threaded through the premises). The option type accounts for failure; in case a premise cannot be proved, the computation aborts and becomes None.

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 MWriter monad).

An outcome is 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): if the computation of premises is a failure, then the proof of the desired goal is a failure too.

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 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.
• qed is a convenience combinator that pairs the state with the name of the rule we want to conclude with; it makes the implementation of solve (Figure 8) more elegant.

---

Figure 6: The writer monad transformer (library code)
The Proof Search Monad

J. Protzenko

(* 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)

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):

- 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 us 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 of type a, each of which has a given outcome. We extend solve with an extra case,
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 -> 
  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_Ref1 
  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)

\[
\begin{align*}
\text{OR-L} & : \quad V \vdash P \rightarrow V' \\
               & \Rightarrow V \vdash P \lor Q \rightarrow V' & \text{OR-R} & : \quad V \vdash Q \rightarrow V' \\
               & \Rightarrow V \vdash P \lor Q \rightarrow V'
\end{align*}
\]

Figure 9: New proof rules for disjunction

which attempts to prove a disjunction by first trying a left-elimination (\textbf{OR-L}, Figure 9), then a right-elimination (\textbf{OR-R}).

The solver can now solve problems of the form \(x = z \lor y = z\). It fails, however, to solve problems of the form \((y = x \lor y = z) \land y = z\). 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, \texttt{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 \texttt{MOption} with the non-determinism monad that will implement backtracking.
The Proof Search Monad

J. Protzenko

(* We extend formulas with disjunctions. *)

```ocaml
type formula =
  (* ... *)
  | Or of formula * formula
```

(* The logic is also extended with two rules. *)

```ocaml
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... *)

```ocaml
let choice (goal: formula) (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: *)

```ocaml
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)

### 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 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 \((y^2 = x \lor y^2 = z) \land y^2 = z\) is now successfully solved by the library. Thanks to laziness, no extra computations occur; further solutions down the lazy
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

\[
\text{FORALL E} \\
\frac{}{\forall y. P}
\]

Figure 12: Extra rule for the universal quantifier

list are only evaluated if the first ones failed.

4 Extension: quantifiers and proof trees

We mentioned earlier that the derivation we were building tracked the application of algorithmic rules; that is, we were building a trace of the algorithm. While the trace is useful to extract information for the user, one may also want to build a proper proof witness in order to certify the validity of the formula.

In order to make a proof tree relevant and not just provide the substitution as the proof witness, we introduce quantifiers to the language, and construct proof trees that apply the proof rules from Figure 1, Figure 9, Figure 12. The nodes of the proof tree are rules; each node is annotated, if applicable, by its implicit parameters. That is, \text{REFL} and \text{EXISTS E} are annotated with their implicit \text{x} parameter.

The updates to the library required to implement quantifiers are minimal; the bulk of the work is essentially writing substitution and a proper treatment of binders on the client-side.

Figure 13 presents in an informal style the series of updates required.

i) We augment the data type of formulas with quantifiers; we replace the type of rules with the rules from the logic. Furthermore, we demand that the \text{REFL} and \text{EXISTS E} rules be annotated with their argument.

Bound variables are globally-unique atoms (strings); open variables are equivalence classes of the union-find, as before (not shown here).

ii) We previously did not distinguish between a goal and a formula; this was only possible because we assumed all variables were initially open, meaning that we could deference an open variable in any state. Now, we open binders and substitute variables, through the allocation of new points in the union-find (the state). Therefore, a given formula only makes sense when paired with a specific state.

iii) We update the prove combinator to record the state upon creating a new node in the derivation tree. More precisely, the prove combinator records the state after the premises
The Proof Search Monad

J. Protzenko

(* i) Update of the [MyLogic] module. *)

type formula =
  (* ... *)
  | Exists of string * formula
  | Forall of string * formula

(* ii) Update of the [Derivations] module. *)

type derivation =
goal * rule

and goal =
L.state * L.formula

and (* ... *)

(* iii) Update of the [Combinators] module. *)

let prove (goal: Logic.formula) (x: ('a * rule_name) m): 'a outcome =
M.[x >>= fun (premises, (env, rule)) ->
  return (env, ((env, goal), (rule, Premises premises))))

(* iv) Update of the client. *)

let rec solve (state: state) (goal: formula): state outcome =
  (* ... *)
  | Exists (atom, g) ->
    let var, g, state = open_flexible state atom g in
    let var = assert_open var in
    prove goal begin
      premise (solve state g) >>= fun state ->
      qed (R_ExistsE (name var state)) state
    end

Figure 13: Dealing with quantifiers

have been satisfied.

iv) Only a slight is needed on the client side to record a proper proof witness: in the Exists case, the solver prods the union-find state to discover the instantiation choice that has been made for the existentially-quantified variable, and records it in the proof tree.

The sample code in the library comes with a pretty-printer. Here is the output for a simple formula that combines all features from our formula language.

prove ∀x. ∀z. ∃y. (y = x \ y = z) \ y = z using [forall]
| prove ∀z. ∃y. (y = x \ y = z) \ y = z using [forall]
| | prove ∃y. (y = x \ y = z) \ y = z using [exists[z]]

11
prove (z = x ∨ z = z) ∧ z = z using [\]
prove z = x ∨ z = z using [\_r]
prove z = z using axiom [refl[z]]
prove z = z using axiom [refl[z]]

5 Source code
The library is available online at https://github.com/msprotz/proof-search-monad. The file example01.ml contains the full implementation of the primitive solver described in Section 1. The file example02.ml contains the backtracking solver written within the proof search monad, as described in Section 3. One can get the non-backtracking version, described in Section 2, by replacing MExplore with MOption. Finally, the file example03.ml contains the final algorithm described in Section 4. The representation of binders adopted in the last example is suboptimal; bound variables are represented using globally-unique atoms. One may want to use locally nameless, with De Bruijn indices for bound variables (as used in Mezzo). It allows keeps the boilerplate to a minimum, though.

6 Related work
An article titled “The Proof Monad” already exists [6]; in spite of the closely related title, the article is concerned with a slightly different problem, namely giving an operational semantics to tactic languages used in theorem provers. In that sense, the article is related to Mtac [15], which is also concerned with a proper monad for writing tactics in Coq.

Hedges [3] compares various explorations monads, notably using the continuation monad, the selection monad [2] and their respective monad transformers. The main focus of the article seems to be the relationship between backtracking and game theory.

Hinze [4] shows how to use the backtracking monad transformer, i.e. add backtracking to any existing monad. It would be interesting to determine whether our library can be re-implemented using a backtracking monad transformer, rather than the writer transformer applied to the monad of non-determinism. The (draft) version of the library used in Mezzo also builds failed derivations (as error messages) that list all attempted proofs, along with the first premise that failed; doing so would not be possible using exceptions.

The choice operator is related to polarization and focusing [8]. For instance, in the problem y² = x ∨ y² = z, depending on which side of the disjunction the algorithm considers first, the outcome is going to be different. This is analogous to a synchronous phase (where the order of the rules matters, and where a particular choice may have consequences on the rest of the search). Similarly, one may swap premises chained by the >>= operator, as the order doesn’t matter. This is analogous to a asynchronous phase.

7 Conclusion
We presented a support library for writing a proof search engine using backtracking. The library is parameterized by: the type of formulas; the type of rule applications; the internal state type of the client. This leaves complete freedom for the client to define their own logic. 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


