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 =                  and descr =
| Equals of var * var             | Flexible
| And of formula * formula        | Rigid

and var = P.point               and env = descr P.state
```

Figure 2: Formulas and environments

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 `env` type ("environment"), which is an input and an output to the solver. An `env` represents a given state of the union-find data structure, that is, a set of equations between variables. This is another way of representing a substitution, i.e. 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 env` (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 formula and verify that all the clauses are of the form $x = x$, without the need for a proof tree.

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

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

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

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.

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

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

Figure 5: The writer monad transformer

In the (simplified) sketch from Figure 4, 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 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 `env` 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.

References