(functional pearl)

the Proof Search Monad

Jonathan Protzenko (Microsoft Research)
protz@microsoft.com

The Proof Search Monad

A library developed to implement the type-checker of *Mezzo*.

In this talk:

- what is *Mezzo* (from a very high-level)
- generalize (when is this library suitable)
- the library itself (combinators and implementation).

Where it all came from

My thesis in one slide

"Separation logic as a type system".

- *Mezzo*: *barely* a type system (flow-sensitive, structural information, keeps track of local aliasing)
- Hindley-Milner: nope
- Because: undecidable (System F, entailment, framing, higher-order logic)
- But: we *still* want inference
- So: *heuristics* (it "mostly" works)

Flow-sensitivity

Mezzo has permissions, of the form $x @ t$, separated by $*$.

In ML: $\Gamma = x : t, y : u$

In *Mezzo*: $P = x @ t * y @ u$

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Flow-sensitivity

Mezzo has permissions, of the form $x @ t$, separated by $*$.

$$\text{In ML: } \Gamma = x : t, y : u$$

$$\text{In } \textit{Mezzo}\text{: } P = x @ t * y @ u$$

P₁

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Flow-sensitivity

Mezzo has permissions, of the form $x @ t$, separated by $*$.

In ML: $\Gamma = x : t, y : u$

In *Mezzo*: $P = x @ t * y @ u$

P_2

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Flow-sensitivity

Mezzo has permissions, of the form $x @ t$, separated by $*$.

In ML: $\Gamma = x : t, y : u$

In *Mezzo*: $P = x @ t * y @ u$

```
val f (x: ...): ... =
  let y = ... in
  ...
```

P_3

Flow-sensitivity

Mezzo has permissions, of the form $x @ t$, separated by $*$.

In ML: $\Gamma = x : t, y : u$

In *Mezzo*: $P = x @ t * y @ u$

```
val f (x: ...): ... =
  let y = ... in
  ...
```

This allows keeping track of ownership.

Why ownership?

At each program point, the programmer knows which objects they own, and how they own them. Ownership is either shared (duplicable) or unique (exclusive).

Data-race freedom *Mutable* objects have a unique owner. Therefore, at most one thread may mutate an object at any given time. *Mezzo* programs are data-race free.

State change If I'm the sole owner of an object, I'm allowed to break its invariants (type). Important because of fine-grained aliasing tracking.

Ownership example

Data-race freedom *and* state change.

```
let r = newref x in
(* r @ Ref { contents = x } *)
r := y;
(* r @ Ref { contents = y } *)
```

A rich type system…

Singleton types	`x @ (=y)`: `x` is `y` Written as: $x = y$
Constructor types	`xs @ Cons { head: t; tail: u }` (special-case: `t` is a singleton, we write `xs @ Cons { head = …; tail = … }`)
Decomposition	via unfolding (named fields), refinement (matching) and folding (subtyping)
Several possible types	`x @ (int, int)`, `x @ ∃(y,z: value). (=y | y @ int, =z | z @ int)`, `x @ ∃t.t`, etc.

...that still isn't quite a logic

Mezzo remains a type system.

- far less connectives and rules
- $f @ t \to u * x @ t \not\leq \exists (y : \text{value})\ y @ u$ (no implicitly callable ghost functions)
- no built-in disjunction (only tagged sums)

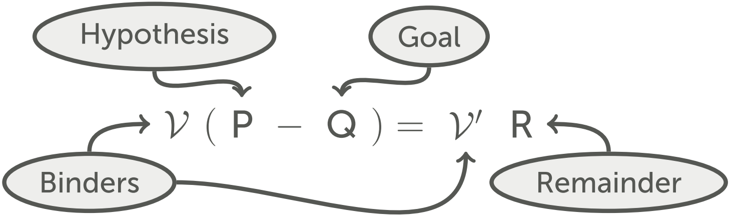
Mezzo's type system feels like a limited fragment of intuitionistic logic.

Subtraction: an unusual algorithm

- Subtyping needs to be decided for function calls and for function bodies.
- Blurs the frontier between type-checking and logics.
- The subtyping algorithm *has to* perform inference

More about subtraction

The operation is written $P - Q = R$ and assumes P has been *normalized* (all left-invertible rules have been applied).



This means:

"with the instantiation choices from $\mathcal{V}'$, we get $P \leq Q * R$".

Subtraction example (1)

In order to type-check the application $f\ x$, we ask:

$$P - f @ \alpha \to \beta * x @ \alpha \quad = \quad ?$$

The algorithm must guess α, β, and the remainder.

Subtraction example (2)

$$
\begin{aligned}
& \ell \,@\, \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\} * \\
& h \,@\, \mathsf{ref\ int} * t \,@\, \mathsf{list}\,(\mathsf{ref\ int}) \\
- & \\
& \ell \,@\, \mathsf{list}\,(\mathsf{ref\ int}) \\
= & \\
& \ell \,@\, \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\}
\end{aligned}
$$

Subtraction example (3)

$$
\begin{array}{ll}
 & \mathbf{z} \mathbin{@} \exists\alpha, \forall\beta.(\alpha, \beta) \\
- & \\
 & \mathbf{z} \mathbin{@} \exists\alpha', \forall\beta'.(\alpha', \beta') \\
= & \\
 & ?
\end{array}
$$

Backtracking

Inference uses *flexible* variables.

There may be several solutions:

$$x\,@\,\mathsf{int} - x\,@\,\alpha = x\,@\,\mathsf{int} \quad \text{with} \quad \begin{cases} \alpha = \mathsf{int} \\ \alpha = {=}x \\ \alpha = \top \end{cases}$$

Not all solutions are explored: α could be $(\beta \mid p)$.

There are many other backtracking points: right-introduction *vs.* left-introduction, which atomic permission to focus...

How to implement all that?

A module that takes care of running the proof search and providing an answer to:

$$P - Q \quad = \quad ?$$

```
| TyQ (Forall, binding1, _, t'1), TyQ (Exists, binding2, _, t'2) ->
    par env judgement "Intro-Flex" [
      try_proof_root "Forall-L" begin
        let env, t'1, _ = bind_flexible_in_type env binding1 t'1 in
        sub_type env t'1 t2 >>=
        qed
      end;
      try_proof_root "Exists-R" begin
        let env, t'2, _ = bind_flexible_in_type env binding2 t'2 in
        sub_type env t1 t'2 >>=
        qed
      end
    ]
```

The proof search monad

- *Mezzo* is not decidable; however
- we *know* where we want to backtrack; (and where we do not want to go, e.g. $\pi * \pi' * \pi'' \leq p * q * r$)
- we *know* which branch is most likely to succeed; ($\alpha =$ int usually better than $\alpha = \top$)
- we *know* that sub-branches *terminate* (no need to interleave)

In that case, adopt the library style where:

- the proof search algorithm looks like the paper rules;
- you get a derivation for free;
- the library works for any logic (not in *Mezzo*, hence this paper)

The library, dissected

From the library's perspective (1)

The client's logic must satisfy **`LOGIC`**.

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

From the library's perspective (2)

The library defines derivation trees for **LOGIC**.

```
module Derivations = functor (L: LOGIC) -> struct
  type derivation = goal * rule
  and goal = L.state * L.formula
  and rule = L.rule_name * premises
  and premises = Premises of derivation list
end
```

From the library's perspective (3)

An `'a m` is the working state of a rule application.

```
module Make(Logic: LOGIC)(M: MONAD) = struct
  module Proofs = Derivations(Logic)
  include Proofs

  module L: MONOID with type t = Proofs.derivation list = struct
    type t = Proofs.derivation list
    let empty = []
    let append = List.append
  end

  include WriterT(M)(L) (*
    type 'a m = (L.t * 'a) M.m
    val return : 'a -> 'a m
    val tell : L.a -> unit m
    val ( >>= ) : 'a m -> ('a -> 'b m) -> 'b m
  *)

  ...
```

From the library's perspective (4)

An **'a outcome** is the result of a rule application.

```
  ...

  type 'a outcome = ('a * derivation) M.m

  (* _Record_ a proof in the premises. *)
  val premise: 'a outcome -> 'a m
  (* _Conclude_ from the given premises. *)
  val prove:
    Logic.formula ->
    (Logic.state * rule_name) m ->
    Logic.state outcome
end
```

From the client's perspective (1)

```
module MyLogic = struct ... end
module MyMonad = ProofSearchMonad.Make(MyLogic)(MExplore)

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

Let's refine (1)

```
let qed rule = fun state -> state, rule

let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  ...
  | And (g1, g2) ->
      prove goal begin
        premise (solve state g1) >>= fun state ->
        premise (solve state g2) >>=
        qed R_And
      end
  ...
```

Let's refine (2)

```
val choice :
  Logic.formula ->
  'a list -> ('a -> (Logic.state * rule_name) m) -> Logic.state outcome

let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  ...
  | Or (g1, g2) ->
      choice goal [ R_OrL, g1; R_OrR, g2 ] (fun (rule, g) ->
        premise (solve state g) >>=
        qed rule
      )
  ...
```

In conclusion

- A powerful library for writing the proof search of *Mezzo*
- Generalizes if your problem fits the earlier description
- Proof derivations for free
- We used them mostly for debugging (In *Mezzo*, a post-processing phase tries to extract *relevant* parts of a *failed* derivation.)

Next

- Extend the library to record *failed* derivations (`choice` now records *all* the things we tried; >>= records up to the first *failed* premise)
- See if compatible with more complex exploration strategies.