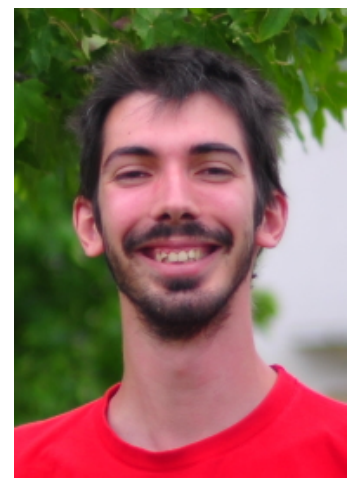
OCamlEvolution: growing Inferno

Olivier Martinot, François Pottier, **Gabriel Scherer**

May 31, 2022





Context: cleanup the OCaml type checker

low road incremental improvements to the implementation
(refactoring, documentation, etc.)
lots of work with Florian, Thomas, Jacques, Takafumi, Antal, etc.
already mentiond by Florian

high road revolution... in the ivory tower rebuild from scratch to understand better
with semantics and proofs
(and also code)

This work: part of the high road.

Inferno: the pitch (1/2)

Inferno: a type inference **library** by François Pottier.

Ideal world: for any type system, use Infero to easily write type inference.

Reality in 2018: for any ML subset with `fun` and `let`, ...

Inferno: the pitch (2/2)

Inferno: a type inference **library**.

Our current objectives:

- cover an interesting subset of the OCaml type system
- client type-checkers are pleasant:
  - for their developers (for example: good testing story)
  - for their users (for example: decent error messages)

Design of Inferno

Algorithmic building blocks:

- generic unification
- transactional union-find
- generalization with ranks

A constraint language with semantic actions
elaborates to explicitly-typed terms
... forcing you to design a typed language

A programming style to use those constraints:

- applicative functor for constraints and their actions
- explicit CPS for quantifiers (context operators)

```
let rec hastype (t : ML.term) (w : variable) : F.nominal_term co
= match t with
```

```
let rec hastype (t : ML.term) (w : variable) : F.nominal_term co
= match t with

  | ML.Abs (pos, x, u) -> (* fun x -> u *)
      let@ v1 = exist in
      let@ v2 = exist in
      let+ () = w --- arrow v1 v2
      and+ u' = def (X.Var x) v1 (hastype u v2)
      and+ ty1 = decode v1
      in F.Abs (pos, x, ty1, u') (* fun (x:ty) -> u' *)
```

```
let rec hastype (t : ML.term) (w : variable) : F.nominal_term co
= match t with

  | ML.Abs (pos, x, u) -> (* fun x -> u *)
      let@ v1 = exist in
      let@ v2 = exist in
      let+ () = w --- arrow v1 v2
      and+ u' = def (X.Var x) v1 (hastype u v2)
      and+ ty1 = decode v1
      in F.Abs (pos, x, ty1, u') (* fun (x:ty) -> u' *)

  | ML.LetProd (pos, xs, t, u) ->(* let (x1, x2...) = t in u *)
      let on_var (x:ML.tevar) : ('a, 'r) binder =
        fun (k : 'b -> 'r co) : 'r co ->
          let@ v = exist in
          def (X.Var x) v (k v)
      in
      let@ vs = mapM_now on_var xs in
      let+ t' = lift hastype t (product vs)
      and+ u' = hastype u w
      in F.LetProd(pos, xs, t', u') (* let (x1,x2..) = t' in u' *
```

Our deeds

Our project contributed to Inferno:

- The current expression style for constraints (binding operators).
- a "cram" testsuite
- "typed holes" for better counter-example shrinking
- client: inference for algebraic datatypes and pattern-matching
- constraints: rigid variables (`fun (type a) -> ...`)

Ongoing work:

- type-based disambiguation
- GADTs

Thanks!

?

Reasoning about effects

The project "Évolution d'OCaml" funds Paulo de Vilhena's PhD grant.

Paulo studies delimited control effects,
also known as effect handlers, which will appear in OCaml 5.

His results include:

- verification rules for effects (POPL 2021);
- type-checking rules for effects (ongoing work, pending submission).

An example: control inversion

A typical use of effects is control inversion,
that is, turning an `iter` function into a lazy sequence.

```
val invert :
  (('a -> unit) -> unit) -> (* an iter function: producer in control *)
  'a Seq.t                  (* a lazy sequence:  consumer in control *)
```

Using so-called deep handlers:

```
open Effect
open Effect.Deep
let invert (type a) (iter : (a -> unit) -> unit) : a Seq.t =
  let open struct type _ Effect.t += Yield : a -> unit t end in
  let yield v = perform (Yield v) in
  fun () ->
    match_with iter yield {
      retc = (fun () -> Seq.Nil);
      exnc = raise;
      effc = fun (type b) (e : b Effect.t) ->
        match e with
        | Yield v -> Some (fun (k : (b, _) continuation) ->
            Seq.Cons (v, continue k))
        | _ -> None }
```

(Beautiful syntax for effects is still missing in OCaml 5.)

Or, less mysteriously, using lower-level shallow handlers:

```
open Effect
open Effect.Shallow
let invert (type a) (iter : (a -> unit) -> unit) : a Seq.t =
  let open struct type _ Effect.t += Yield : a -> unit t end in
  let yield v = perform (Yield v) in
  let rec loop (k : (unit, unit) continuation) : a Seq.node =
    continue_with k() {
      retc = (fun () -> Seq.Nil);
      exnc = raise;
      effc = fun (type b) (e : b Effect.t) ->
        match e with
        | Yield v -> Some (fun (k : (b, _) continuation) ->
            Seq.Cons (v, fun () -> loop k))
        | _ -> None }
  in
  fun () -> loop (fiber (fun () -> iter yield))
```

How to specify and verify this code?

We use Separation Logic (Iris), extended with

- protocols, describing the effects that a computation may perform;
- new reasoning rules for performing an effect and installing a handler.

Specification of `invert`

The specification of `invert` in Iris is roughly:

```
(* For every sequence xs,
   for every function iter,
   if iter produces the sequence xs,
   then the function call invert(iter)
   performs no effect
   and returns seq
   such that seq produces the sequence xs. *)
```

This specification relies on two auxiliary definitions,

- `iter produces the sequence xs`
- `seq produces the sequence xs`

Specification of an `iter` function

What does it mean for `iter` to produce a sequence `xs`?

```
(* Pick an arbitrary definition of produced(ys) -- a loop invariant.
   Pick an arbitrary effect protocol psi.

   Suppose that, always, for every index i,
   provided x is xs[i],
   the function call f(x)
   may perform effects according to psi
   and transforms the state produced(xs[0..i))
            into the state produced(xs[0..i+1)),

   Then the function call iter(f)
   may perform effects according to psi
   and transforms the state produced([])
            into the state produced(xs). *)
```

This specification is polymorphic in `produced` and `psi`.

Specification of an `iter` function

What does it mean for `seq` to produce a sequence `xs`?

```
(* Either xs is empty and
     the function call seq()
     performs no effect
     and returns Seq.Nil
   or xs is nonempty and
     the function call seq()
     performs no effect
     and returns Seq.Cons(xs[0], seq')
     where seq' produces the sequence xs[1..]. *)
```

Summary

I have briefly shown

- an implementation of `invert` in OCaml 5;
- a specification of `invert`.

Paulo has a machine-checked proof that the code meets the spec.

- The proof involves a custom loop invariant (defined using ghost state) and a custom protocol (also tied with this ghost state).

Future work includes

- a strong type discipline for effects (ongoing work);
- support for effects in Gospel, OCaml's specification language.