Quantified Applicatives – API design for type-inference constraints

Olivier Martinot, Gabriel Scherer

July 24, 2020

Abstract

The Inferno library (Pottier, 2014a,b) uses an applicative functor `'a co` to represent constraint-based inference problems that elaborate into explicitly-typed term representations. A central operation of the Inferno API is the `exist` quantifier, which generates a fresh inference variable scoped in an existential constraint. `exist` returns the result of solving the constraint, as well as the elaborated type inferred for the inference variable: `val exist : (variable -> 'a co) -> ('a * ty) co`

We found that programming with this interface is difficult and leads to program that are hard to read. The difficulty is specific to applicative functors, there would be standard solutions with a monad.

We report on our API design work to solve this issue. We start by explaining the programming difficulties and why the Inferno functor is not a monad, then a first experiment with a codensity monad, and finally a simpler solution that avoids mixing several functors in the same codebase. Our approach makes good use of the "binding operators" recently introduced in OCaml to provide flexible syntactic sugar for monad or applicative-like structures.

We believe that the API design pattern we propose, "turning direct outputs into modal inputs", may apply to any applicative functors with quantifiers. It may already be in use, and its discussion may benefit other projects. Experience sharing with the ML Workshop audience could be very valuable.

1 Inferno

François Pottier's Inferno (Pottier, 2014b) is a software library for constraint-based type inference. It exposes low-level data structures and algorithms for efficient inference (union-find graphs for unification, level-based generalization), a term representation for inference constraints (SolverLo.mli)[1], and a higher-level API (SolverHi.mli) exposing an applicative functor `'a co` designed to write constraint-based type inference engines in a declarative style (Pottier, 2014a).

A value of type `'a co` pairs together an inference constraint with a function that will be called with the solution of the constraint to construct an `'a` "witness", that inference has suceeded. For example, a type-inference procedure for ML programs could have type `ML.term -> F.term co`: it takes an ML term and generates a constraint that, once solved, will produce an explicity-typed representation of the input program as a System F term. The implementation of Inferno comes with exactly such an example inference program, for a core fragment of ML: Client.ml.

In this document we will use the following running example, adapted from a fragment of the Inferno inference example. On the right, we show the types of the combinators we use, provided by Inferno's high-level layer:

[1] We give links to source code for the interested reader, which are versioned to point at the versions of the software at the time of writing (git revision `d02e4c23430df0b8cd15a99adfa41fd00086b925`).

```
let rec hastype (t : ML.term) (w : Infer.variable) : F.term co =
  match t with
  | [...]
  | ML.Pair (u1, u2) ->
    exist (fun v1 ->
      exist (fun v2 ->
        w --- Infer.Prod(v1, v2) ^^
        hastype u1 v1 ^&
        hastype u2 v2
    )) <$$> fun (ty1, (ty2, (u'1, u'2))) ->
    F.Pair ((u'1, ty1), (u'2, ty2))
```

```
val exist : (Infer.variable -> 'a co) -> (F.ty * 'a) co
val (---) : Infer.variable -> Infer.ty -> unit co
val (^^) : unit co -> 'b co -> 'b co
val (^&) : 'a co -> 'b co -> ('a * 'b) co
val (<$$>) : 'a co -> ('a -> 'b) -> 'b co
```

We define a function `hastype : ML.term -> Infer.variable -> F.term co` which takes an ML term `t` and a "destination inference variable" `w`; the returned constraint will unify the inferred type of `t` with `w`, and return the elaborated System F term. We show the inference code for the `Pair(u1,u2)` construct, representing pairs in ML; for illustration purposes we use a "very explicit" syntax for pairs in the System F side, which expects the two terms but also their types: `F.Pair : (F.term * F.ty) * (F.term * F.ty) -> F.term`. (In practice our pair syntax does not require explicit types, but the Church-style syntax for λ-abstractions does.)

The infix operation `(---)` inserts a unification constraint (`Infer.ty` is not the source representation of ML or F types, but an inference-specific representation of types, which the engine knows how to traverse for unification, etc.). The infix combinators `(^^), (^&)` pair constraints together. Finally, `<$$>` is a flipped `map` function for constraints.

All type-inference cases have roughly this structure: using `exist` to create inference variables for subterms, combining type-checking constraints to follow our type-checking rule, concluded by a final "map" that takes all the witnesses of the constraint-solving pieces, and combines them into a final witness for the inferred term.

1.1 The problem

Consider extending this code to handle not just binary pairs, but n-ary tuples `Tuple us`. Instead of a statically-known number of inference variables (2), we want to generate one for each subterm of the list, to build the result type. How would you program this with the proposed API?

If `'a co` was an inference/elaboration *monad*, it would be natural to decompose the problem with a function of type `'a list -> Infer.variable list co`, or possibly `mapM : ('a -> 'b co) -> 'a list -> 'b list co`, and then `bind` the result to continue. We write this hypothetical program below on the left column. But `'a co` is *not* a monad for a good reason – we explain this in the next section. Instead, we have to write code in continuation-passing style, on the right column below.

```
| ML.Tuple us ->
  let on_each u =
    exist (fun v -> v ^& hastype u v)
    <$$> fun (ty, (v, u')) -> (v, (u', ty))
  in
  mapM on_each us >>= fun vutys ->
  let vs, utys = List.split vutys in
  w --- Infer.product vs ^^
  return (ML.tuple utys)
```

```
| ML.Tuple us ->
  let rec traverse us (k : variable list -> 'r co)
    : ((F.term * F.ty) list * 'r) co
  = match us with
    | [] -> k [] <$$> fun r -> ([], r)
    | u::us ->
      exist (fun v ->
        traverse us (fun vs ->
          hastype u v ^&
          k (v :: vs)
      )) <$$> fun (ty, (u', (utys, rs))) ->
        ((u', ty) :: utys, r)
  in
  traverse us (fun vs ->
    w --- Infer.product vs
  ) <$$> fun (utys, ()) ->
    F.Tuple utys
```

Note: `traverse` is called with a single continuation; it is possible to inline this continuation in the empty-

list case, and get a simpler, less generic function, that could not be reused elsewhere. We wish to discuss the reusable presentation in this document, but the API design improvements also help in the non-reusable case.

There are two difficulties with the code we had to write:

1. It is much harder to figure out how to write this code than in the monadic version, which allows a natural problem decomposition. Programmers may not know how to generate dynamically many inference variables. If they know that some things can be done with CPS, they may not know exactly which, or which signatures to use for CPS functions.

2. The code, in either versions, is hard to read. In particular, values are *named* far from where they are *expressed* in the program; notice for example how `u'` binds the witness of `hastype u v`, but is placed farther away in the program. This problem gets worse when writing inference code for more complex constructs.

   When we explain this code, we say that the `Infer.ty` result of `exist` or the `(F.term * F.ty) list` result of `traverse` is "pushed on the stack" – which is then later shuffled around by the `(<$$>)` operator. We would rather avoid this particular style of point-free programming.

Our contribution is to identify, explain this problem, and propose a partial solution: a systematic approach to programming with "binders" that can be explained and documented, and a different API that lets us name terms in a more natural, local way – at the cost of "unpacking" operations later on, as we will explain.

While our work is specific to Inferno (and OCaml), we believe that this API problem arises in the more general case of an applicative functor with quantifiers, as `exist` in our example.

1.2 Aside: binding operators

OCaml recently adopted a flexible syntactic sugar for composing computations in monadic- and applicative-like structures called "binding operators". Just like we allow user-defined "infix operators" and "prefix operators" in a limited set of symbols (`(^^)`, `(^&)` were examples), we allow "binding operators" of the form `let<op>` and `and<op>`, where `<op>` is a user-chosen sequence of symbols, with the following desugaring rules:

```
let<op1> p1 = e1 and<op2> p2 = e2 in body
    ==>
(let<op1>) ((and<op2>) e1 e2) (fun (p1, p2) -> body)
```

The current convention is to use `(let* )` for monadic binding, and `(let+), (and+)` for applicative map and product. For example, with suitable user definitions one can write `(let* x = m in e)` for `(m >>= fun x -> e)`, and `(let+ x = foo and+ y = bar in e)` for `(pure (fun x y -> e) <$> foo <$> bar)`. Other symbols are up for grabs.

We will use binding operators for convenience, but our solution does not rely on them in an essential way.

1.3 Not a monad

Giving up on our page limit, let us explain why the `'a co` type of Inferno cannot be given the structure of a monad.

The witness `'a` is built with knowledge of the *global* solution to the inference constraints generated. Consider the applicative combinator `pair : 'a co -> 'b co -> ('a * 'b) co`; both arguments generate a part of the constraint, but the witnesses of `'a` and `'b` can only be computed once the constraints on *both* sides are solved. For example, when inferring the ML term `(x, x + 1)`, the type inferred for the first component of the pair is determined by unifications coming from the constraint of the second.

Implementing `bind : 'a co -> ('a -> 'b co) -> 'b co` would require building the witness for `'a` before the constraint arising from `'b co` is known; this cannot be done if `'a` requires the final solution.

For the abstractly inclined: internally `'a co` is defined as `raw_constraint * (solution -> 'a)` it is the composition of a "writer" monad that generate constraints and a "reader" monad consuming the solution. For the composition $W \circ R$ of two arbitrary monads to itself be a monad, a sufficient condition (Jones and Duponcheel, 1993) is that they commute: $(R \circ W) \to (W \circ R)$ – this suffices to define `join` : $(W \circ R \circ W \circ R) \to W \circ R$ from the `join` operation of W and R. Pushing the reader below the writer would require reading the solution before writing the constraint; this is not possible if we want the final solution.

2 Failed attempt: codensity

If we have to write in CPS style, a natural idea is to program in the corresponding CPS *monad*. We can define `('a, 'i) co_cps` which does have an (indexed) monadic structure:

```
type ('a, 'i) co_cps = { run : 'r . ('a -> 'r co) -> ('i * 'r) co }
val bind : ('a, 'i1) co_cps -> ('a -> ('b, 'i2) co_cps) -> ('b, 'i1 * 'i2) co_cps
val existM : (Infer.variable, F.ty) co_cps
```

This is an indexed version of the *codensity* construction: for any functor `'a t`, the type `type 'a m = 'r. ('a -> 'r t) -> 'r t` has a monadic structure. This lets us write `exist` and our `traverse` above in monadic style, and then rely on our usual monadic programming habits. Unfortunately, we found this too hard to use in practice for the following reasons:

- `exist` is a naturally *scoped* construction: besides generating a free variables, it builds a raw constraint term of the form `CExist(x,rc)`, where the inference variable scopes over the constraint sub-term `rc`.

  A `co_cps` computation builds a single ever-growing scope (by composing continuations in scope of all quantifiers effect performed so var). The scope is only "closed" when the computation is "run" by invoking it with the identity continuation – we can define a `close` combinator doing this.

  Note that lifting all existential constraints to a toplevel, global scope would be incorrect, due to the interplay with polymorphism / generalization. We need to close the scope in a more local way.

  In practice we found scoped programming with `co_cps` less natural than the syntactic/lexical scoping of the non-monadic style. It makes it harder to reason about scopes, and it forces us to write programs with hybrid usage of both `co_cps` (for quantifiers) and `co` (for scope control).

- As we have to use a mix of the two structures `co` and `co_cps` in our programs, we have to duplicate all operations to work in both of them, or to use lifting combinators from one layer to the other. This makes program more difficult to write and read.

3 Our proposal: turning direct outputs into modal inputs

We propose to define a type `'a binder` for quantifier-like definitions that resembles the `co_cps` construct, with two key differences in API design:

1. Instead of using a `bind` operator whose return type is in the `co_cps` monad, use a composition combinator whose return type is in the `co` applicative. As long as our return type remains a `co`, we only need one set of operations.

2. When quantifiers produce witnesses, instead of adding the witness type `w` to the *output* type of the continuation, have the continuation take a *modal input* of type `w co`.

   For example, we change `exist` from the first to the second type:

   ```
   val exist : (Infer.variable -> 'r co) -> (F.ty * 'r) co
   val exist : (Infer.variable * F.ty co -> 'r co) -> 'r co
   ```

Here `co` acts as a modality indicating that the value will only be available at solution time. (In particular, no constraint is produced. We could use the writer/reader decomposition of `co` to take only the reader part, but that would force us to work with two different layers again.)

More concretely, we use the following definitions:

```
type ('a, 'r) binder = ('a -> 'r co) -> 'r co
val exist : (Infer.variable * F.ty co, 'r) binder
```

(Note: we would like to define `'a binder` with a universal quantification on `'r`, but OCaml makes it painful due its limited support for universal inference; in practice binders only occur in positive positions in our APIs, so we can keep their second parameter polymorphic in each combinator, as for `exist`.)

This change in type only requires a simple change in the implementation of the quantifier.

3.1 Final API and example

Our final API looks as follows (other combinators are unchanged from our initial examples):

```
type ('a, 'r) binder = ('a -> 'r co) -> 'r co
val exist : (Infer.variable * F.ty co, 'r) binder

val (let@) : ('a, 'r) binder -> ('a -> 'r co) -> 'r co (* identity *)
val (let+) : 'a co -> ('a -> 'b) -> 'b co              (* map *)
val (and+) : 'a co -> 'b co -> ('a * 'b) co            (* pair *)
```

`('a, 'r) binder` is a type of "general quantifiers", which introduce arguments of type `'a` to build a constraint in a delimited scope returning a `'r co`. Only terms of this type are written in continuation-passing style. They are used through `let@`, that opens a quantifier which naturally scopes until the end of the lexical block. The binder syntax is a reference to OCaml's infix application operator `(@@)`: `(let@ p = a in b)` is equivalent to `(b @@ fun p -> a)`.

As a final example, let us revisit our "tuple" case with this new API. (Left: before, Right: after)

```
| ML.Tuple us ->
  let rec traverse
    us (k : variable list -> 'r co)
    : ((F.term * F.ty) list * 'r) co
  = match us with
    | [] ->
      k [] <$$> fun r -> ([], r)
    | u::us ->
      exist (fun v ->
        traverse us (fun vs ->
          hastype u v ^&
          k (v :: vs)
      )) <$$> fun (ty, (u', (utys, rs))) ->
        ((u', ty) :: utys, r)
  in
  traverse us (fun vs ->
    w --- Infer.product vs
  ) <$$> fun (utys, ()) ->
    F.Tuple utys
```

```
| ML.Tuple us ->
  let rec traverse (us : ML.term list)
    : (variable list * (F.term * F.ty) list co,
       'r) binder
  = fun k -> match us with
    | [] ->
      k ([], pure [])
    | u::us ->
      let@ v, ty = exist in
      let@ (vs, us') = traverse us in
      k (v :: vs,
         let+ u' = hastype u v
         and+ ty = ty
         and+ us' = us'
         in (u', ty) :: us')
  in
  let@ (vs, utys) = traverse us in
  let+ () = w --- Infer.product vs
  and+ utys = utys
  in F.Tuple utys
```

The CPS is now guided/documented by our types: to define a `(foo, 'r) binder` value we use CPS, and to use these binders we use the `let@` operators.

Modal inputs improve code readability by giving a name to an expression at the logical place in the code (`ty` and `us'` for example). Being in the applicative, a modal input variable `x` cannot be used as-is to build the witness, it has to be "unpacked" in the final`let+ .. and+ .. in <witness>` block by the odd construction `and+ x = x`. This unpacking construction is surprising, and the redudancy is frustrating (one could think of a proper modal-typing rule in the compiler that would implicitly unpack modal inputs); but note that the unpacking place corresponds to the place, in the previous style, where the variable would have been first named. Unpacking is a fair price to pay to recover readable code.

3.2 Another example

To conclude, we would like to demonstrate the generality of this approach to a wider family of applicative functions with "quantifier"-like combinators, by briefly showcasing the same refactoring in a different applicative functor – an artificial example.

Consider an applicative `'a cloud` for "computations in the cloud", where operating on a resources (here a remote file) comes with a "cost" (API usage, time, money, or something else) that is returned to the programmer when they conclude using the resource. In this model, opening a (remote) file for reading could be exposed by the following quantifier-like operation:

```
val with_input : path -> (in_channel -> 'r cloud) -> (cost * 'r) cloud
```

and now the user desires to generalize this into a function that opens a list of paths, operates on the correspond list of input channels, and returns the sum of all costs (we will assume that `type cost = int` for simplicity). They want to implement:

```
val with_inputs : path list -> (in_channel list -> 'r cloud) -> (cost * 'r) cloud
```

and this is tricky. Our proposal, "turning direct outputs into modal inputs", suggests the following change to the primitive operation, introducing a `binder` type that marks the place where continuation-passing-style should be used:

```
type ('a, 'r) binder = ('a -> 'r cloud) -> 'r cloud
val with_input : (in_channel * cost cloud, 'r) binder
```

An implementation of `with_inputs` using the original API is on the left, and our proposed approach is on the right.

```
let rec with_inputs paths k =
  match paths with
  | [] ->
    let+ r = k [] in (0, r)
  | p :: ps ->
    let+ (cost_p, (cost_ps, r)) =
      let@ chan = with_input p in
      let@ chans = with_inputs pts in
      k (chan :: chans)
    in (cost_p + cost_ps, r)
```

```
let rec with_inputs paths =
  fun k -> match paths with
  | [] -> k ([], pure 0)
  | p :: ps ->
    let@ chan, cost_p = with_input p in
    let@ chans, cost_ps = with_input ps in
    let+ res = k (p :: ps)
    and+ cost_p = cost_p
    and+ cost_ps = cost_ps
    in (cost_p + cost_ps, res)
```

Notice, again, that the original API pushes the cost "on the stack", and that the variables `cost_p`, `cost_ps` are bound far from the place that logically introduces them in the left solution. The issue is solved on the right, at the cost of explicit unpacking of the modal inputs.

Acknowledgments

We wish to thank François Pottier and our anonymous reviewers for their feedback.

References

Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

François Pottier. Hindley-Milner elaboration in applicative style. In *ICFP*, September 2014a. URL http://gallium.inria.fr/~fpottier/publis/fpottier-elaboration.pdf.

François Pottier. the Inferno library: https://gitlab.inria.fr/fpottier/inferno, 2014b.