Coq
as a programming language of choice
for the discriminating mathacker

Yann Régis-Gianas
IRIF (PPS) – INRIA ($\pi.r^2$)

Avril 2018 – Fontainebleau

How to help the rooster to fly away from ML
to become a **general purpose** programming language?

[1] Mathematicians with a taste for hacking are already Coq users, right?

How to help the rooster to fly away from ML
to become a **general purpose** programming language?

- Follow Haskell path, but go beyond!
- Attract mathacker (hacker with a taste for Mathematics[1]).

[1] Mathematicians with a taste for hacking are already Coq users, right?

A patchwork talk

4. Meta
   - Metaprogramming Coq in Coq with Mtac2
3. Infinity
   - Which language to define infinite objects?
   - Duality for code reuse
2. Differences
   - Certified differential functional programming
   - Relational analysis using bi-interpreters
1. Effects
   - Compositional Effects
   - Building effect interfaces

Effects

"Haskell is the world's finest imperative programming language."
– Simon Peyton Jones, 2000

Effects

"~~Haskell~~ Coq is the world's finest imperative programming language."
– ~~Simon Peyton Jones~~ Pierre-Marie Pédrot, 2021

An effectful program written in Coq

```
1  Equations play
2            {ix:       Type -> Type} `{Use CONSOLE ix}
3            (attempt:  game_steps)
4            (target:   nat)
5    : Program ix unit :=
6  play GameOver t :=
7    echo "Looks like you've lost.\n";
8
9  play (OneMoreTime n) t :=
10   echo "What number do you think it is? " ;;
11   x <- readNat <$> scan                    ;
12   match x with
13   | Some x
14     => if x =? t
15        then echo "You won!\n"
16        else if x <? t
17        then echo "More!\n"                 ;;
18             play n t
19        else echo "Less!"                   ;;
20             play n t
21   | Nothing
22     => play n t
23   end.
```

Under the hood : A free monad

```
1  Definition Interface := Type -> Type.
2
3  Inductive Program (I: Interface) (A: Type) :=
4  | Pure (a:  A)
5    : Program I A
6  | Request (e:  I A)
7    : Program I A
8  | Bind {B:  Type}
9         (p:  Program I B)
10        (f:  B -> Program I A)
11   : Program I A.
```

Concrete effect interfaces

```
1  Inductive CONSOLE
2    : Type -> Type :=
3  | Echo (str:  string)
4    : CONSOLE unit
5  | Log  (str:  string)
6    : CONSOLE unit
7  | Scan
8    : CONSOLE string.
```

Effect interpreter

```
1  CoInductive Semantics
2              (I: Interface)
3    : Type :=
4  | handler (f: forall {A: Type}, I A -> A * Semantics I)
5    : Semantics I.
6
7  Fixpoint runProgram
8           {I:    Interface}
9           {A:    Type}
10          (sig:  Semantics I)
11          (p:    Program I A)
12   : (A * Semantics I) :=
13   match p with
14   | Pure a
15     => (a, sig)
16   | Request e
17     => handle sig e
18   | Bind q f
19     => runProgram (snd (runProgram sig q)) (f (fst (runProgram sig q)))
20   end.
```

Design leitmotiv: Keep it stupidly simple!

Compositionality of interfaces

- If `P : Program I A`, then lift to `P : Program (I + J) A`.
- By construction, effect interfaces are separated.

Verification rules

- A specification is an abstract model for an interface:

```
1  Record Specification (W:  Type) (I:  Type -> Type) :=
2    { abstract_step (A:  Type) (e:  I A) (x:  A) : W -> W
3    ; precondition (A:  Type)  : I A -> W -> Prop
4    ; postcondition (A:  Type) : I A -> A -> W -> Prop
5    }.
```

- Compliance of a semantics wrt to a specification.
- Correctness of a program wrt to a compliant semantics.

Layered architecture for effectful applications

Certified software engineering principles

- Layer n provides the semantics for the effects used by layer $n + 1$.
- A model written in Coq allows reasoning about effects.
- The realization of this model in OCaml allows executing the application.

Results and ongoing work

History

- 2015: Guillaume Claret designed **`CoqIO`**.
- 2017: Collaboration with Thomas Letan (ANSSI), birth of **`FreeSpec`**[2]

Ongoing work

- Thomas Letan is working on his PhD manuscript but will stay in this adventure!
- Starting a collaboration with Albert Servan (University of Barcelona) to design a **`Unix`** model based on **`FreeSpec`**.

Open fundamental questions

- Can we encode all algebraic effects? (Loïc Peyrot, Master thesis)

[2] `https://gitlab.inria.fr/freespec`, paper accepted at FM'18.

Differences

From the dynamics of programs and proofs,
to the dynamics of programming and proving.

Change structures

Definition (Change action)

A **change action** is a triple $(A, \Delta A, \oplus)$ such that A is a set, ΔA is a monoid and $\oplus : A \to \Delta A \to A$ is a monoid action on A.

Definition (Change structure)

A **change structure** is a change action extended with an operator $\ominus : A \to A \to \Delta A$ such that $\forall xy, x \oplus (y \ominus x) = y$.

Definition (Nil change)

For each value $x : A$, such that A is equipped with a change structure, we write 0_x for the **nil change** of x.

Definition (Differentiable function)

A function $f : A \to B$ is **differentiable** if there exists $f' : A \to \Delta A \to \Delta B$ such that $f\,(x \oplus dx) = f\,x \oplus f'\,x\,dx$. In that case, f' is said to be a derivative of f.

Example

Let S be a (large) multiset of natural numbers with weights in $\mathbb{Z}$.
Let δS be a (small) multiset of the same kind.

$$\sum_{w_x \cdot x \in S \oplus \delta S} w_x \cdot x = \sum_{w_x \cdot x \in S} w_x \cdot x \oplus \sum_{w_x \cdot x \in \delta S} w_x \cdot x$$

The cost of updating the sum is proportional to the size of the small multiset.

Context

Questions

- Can we define a change action over functional types?
- Under which conditions is f differentiable?
- What is differential functional programming?
- Why is Coq a good candidate for differential programming?
- What do you mean by “the dynamic of programming”?

What’s the point?

- Incrementality : minimize recomputation after “small” changes.
- Proof of function stability: show that the derivative of a function is nil.
- Proof of invariant: show that the derivative of the invariant is nil.
- Program synthesis: explore a function change space to refine programs.
- Parallelization: exploit orthogonality of changes to separate computations.

What would be a change action over functional types?

Let us define the set of valid changes for $f : A \to B$ as the functions df such that:

- $f\ a \oplus df\ a\ da = f\ (a \oplus da) \oplus df\ (a \oplus da)\ 0_{a \oplus da}$
- $(f \oplus df)\ a = f\ a \oplus df\ a\ 0_a$
- $(g \ominus f)\ a\ da = g\ (a \oplus da) \ominus f\ a$ (If B is equipped with a change structure.)

Under which conditions is f differentiable?

If $f : A \to B$ is total and B is equipped with a change structure, then:

$$0_f = f' \, x \, dx = f \, (x \oplus dx) \ominus f \, x$$

but this is an inefficient derivative as it recomputes everything!

Another proposal is static differentiation (Paolo Giarrusso's PhD thesis, PLDI'13):

$$\begin{aligned} \mathsf{Derive}(x) &= dx \\ \mathsf{Derive}(\lambda x.t) &= \lambda x \, dx.\mathsf{Derive}(t) \\ \mathsf{Derive}(t \, u) &= (\mathsf{Derive}(t)) \, u \, (\mathsf{Derive}(u)) \end{aligned}$$

Recently, we improved this program transformation using Cache-Transfer-Style which allows static memoization between f and f' [3]. A more precise, less static, differentiation of λ-terms was introduced in the PhD thesis of Lourdes del Carmen Gonzalez Huesca:

$$t[x \mapsto x \oplus dx] = t \oplus \frac{\partial \, t}{\partial x} dx$$

[3] Tricky compiler correctness proofs, by the way. Look at our submitted draft!

What is differential functional programming?

Good news

We know how to compose derivatives efficiently in an higher-order setting.
(Type-checking of CTS is still open though.)

This is just the beginning of the story!

The big open question is:

> How to define efficient derivatives for actual computations (aka fixpoints)?

Problem #1: A challenging need for dependent types

Removing an element from an empty list is undefined.

Moving from $\oplus : A \to \Delta A \to A$ to $\oplus : \pi(x : A).\Delta x \to A$ seems natural but then we enter a challenging typechecking problem for every change-based computation we define...

Another solution is to move to a partiality monad, like the option monad, but then the domains of definition of change-based computations must be tracked down by the programmer.

In any case: a language with rich type system seems mandatory to write safe programs.

Problem #2: Change-based computations are large and complex

If A is an inductive datatype with m cases and ΔA has n cases, then any derivative must consider $m \times n$ cases. The programmer may forget some cases...

Besides, because of the partiality explicited in the previous slide, the programmer also has to sort out which of these cases are actually undefined because the change is not valid for the value under consideration.

Again, this cannot be seriously done without some safety net!

Problem #3: Unconventional value representations

The change-application operation must be a constant-time operation.

This does not seem compatible with recursive datatypes. Consider for instance:

```
1  Inductive dhead {A dA : Type} :=
2  | Push : A -> dhead
3  | Pop : dhead
4  | Update : dA -> dhead.
5
6  Inductive dlist (A dA : Type) :=
7  | Nil : dlist A dA
8  | Mod : @dhead A dA -> dlist A dA -> dlist A dA.
```

This definition is irrelevant because applying these changes is inefficient.

Hash-consed representations are mandatory for values.

Changes must refer to the exposed physical identities of values.

In other words, change-base computations must explicitly maintain injective hash functions. Again, without Coq as a garde-fou...

Problem #4: There are many “interesting” function changes

During the PhD of Thibaut Girka, we defined a concept of **formally verifiable semantic differences** to capture the meaning of source code patches.

This lead us to the definition of **correlating oracles**, which are bi-interpreters (written in Coq) that realize a specific relation between execution (sub)traces.

For instance, we have a difference language for each of these changes on Imp programs:

- renamings ;
- replacements of a term by an observationnally equivalent one ;
- generalization of a function ;
- fix off-by-one error in an iteration ;
- removal of a divergence ;
- etc.

We simply scratched the surface of this notion…and it is already complex because **relational analysis** of programs is hard!

Baby steps to understand differences

Current short-term projects

- For simple experiments: Δ-Caml (Olivier Martinot, P7)
- For harder experiments: Δ-Coq (Lelio Brun, Phd ENS)
- Scaling up correlating oracles to the Java language (Mentre, Mitsubishi)
- Formalize `git merge`.
- `VIT`, a collaborative synchronous programming over GIT.

Long-term project

- Capture the semantics of the dynamic of proof/software development.

Infinity

What linguistic mechanism to define and to reason about infinite objects?

Infinite objects using algebraic datatypes

« This is solved problem! » answers the Haskell programmer

Thanks to lazy evaluation, the Haskell programmer easily defines infinite sequences **using algebraic datatypes**:

```haskell
fib = 1 : 1 : (zipWith ( + ) fib (tail fib))
```

Infinite objects using algebraic datatypes

« This is solved problem! » answers the Haskell programmer

Thanks to lazy evaluation, the Haskell programmer easily defines infinite sequences **using algebraic datatypes**:

```
1  fib = 1 : 1 : (zipWith ( + ) fib (tail fib))
```

Strict evaluation and infinity, you say?

In OCaml, we can write almost the same program:

```
1  let rec fib () = 1 :: 1 :: (map2 ( + ) (fib ()) (tl (fib ())))
```

but the evaluation of `fib ()` takes the control and never gives it back!

A bit of Haskell in your OCaml?

```
1  type 'a cell = InfiniteCons of 'a * 'a stream
2  and 'a stream = 'a cell Lazy.t
3
4  let rec map2 f xs ys =
5    lazy (match Lazy.force xs, Lazy.force ys with
6          | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
7            InfiniteCons (f x y, map2 f xs ys))
8
9  let tail xs =
10   let (InfiniteCons (_, xs)) = Lazy.force xs in xs
11
12 let head xs =
13   let (InfiniteCons (x, _)) = Lazy.force xs in x
14
15 let ( +: ) x xs =
16   lazy (InfiniteCons (x, xs))
17
18 let rec fib =
19   1 +: (1 +: (map2 ( + ) fib (tail fib)))
```

A bit of Haskell in your OCaml?

```
1  type 'a cell = InfiniteCons of 'a * 'a stream
2  and 'a stream = 'a cell Lazy.t
3
4  let rec map2 f xs ys =
5    lazy (match Lazy.force xs, Lazy.force ys with
6          | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
7            InfiniteCons (f x y, map2 f xs ys))
8
9  let tail xs =
10   let (InfiniteCons (_, xs)) = Lazy.force xs in xs
11
12 let head xs =
13   let (InfiniteCons (x, _)) = Lazy.force xs in x
14
15 let ( +: ) x xs =
16   lazy (InfiniteCons (x, xs))
17
18 let rec fib =
19   1 +: (1 +: (map2 ( + ) fib (tail fib)))
20   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
21 Error: This kind of expression is not allowed
22        as right-hand side of `let rec'
```

A bit of Haskell-- in your OCaml?

```
1  type 'a cell = InfiniteCons of 'a * 'a stream
2  and 'a stream = 'a cell Lazy.t
3
4  let rec map2 f xs ys =
5    lazy (match Lazy.force xs, Lazy.force ys with
6          | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
7            InfiniteCons (f x y, map2 f xs ys))
8
9  let tail xs =
10   let (InfiniteCons (_, xs)) = Lazy.force xs in xs
11
12 let head xs =
13   let (InfiniteCons (x, _)) = Lazy.force xs in x
14
15 let rec fib =
16   lazy (InfiniteCons (1,
17     lazy (InfiniteCons (1,
18       map2 ( + ) fib (tail fib)))))
```

A bit of Haskell-- in your OCaml?

```
1  type 'a cell = InfiniteCons of 'a * 'a stream
2  and 'a stream = 'a cell Lazy.t
3
4  let rec map2 f xs ys =
5    lazy (match Lazy.force xs, Lazy.force ys with
6          | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
7            InfiniteCons (f x y, map2 f xs ys))
8
9  let tail xs =
10   let (InfiniteCons (_, xs)) = Lazy.force xs in xs
11
12 let head xs =
13   let (InfiniteCons (x, _)) = Lazy.force xs in x
14
15 let rec fib =
16   lazy (InfiniteCons (1,
17     lazy (InfiniteCons (1,
18       map2 ( + ) fib (tail fib)))))
```

« **It is almost concise**! » says the Haskell programmer (ironically)...

A bit of Haskell-- in your OCaml?

```
type 'a cell = InfiniteCons of 'a * 'a stream
and 'a stream = 'a cell Lazy.t

let rec map2 f xs ys =
  lazy (match Lazy.force xs, Lazy.force ys with
      | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
        InfiniteCons (f x y, map2 f xs ys))

let tail xs =
  let (InfiniteCons (_, xs)) = Lazy.force xs in xs

let head xs =
  let (InfiniteCons (x, _)) = Lazy.force xs in x

let rec fib =
  lazy (InfiniteCons (1, map2 ( + ) fib (tail fib))))
```

A bit of Haskell-- in your OCaml?

```
type 'a cell = InfiniteCons of 'a * 'a stream
and 'a stream = 'a cell Lazy.t

let rec map2 f xs ys =
  lazy (match Lazy.force xs, Lazy.force ys with
      | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
        InfiniteCons (f x y, map2 f xs ys))

let tail xs =
  let (InfiniteCons (_, xs)) = Lazy.force xs in xs

let head xs =
  let (InfiniteCons (x, _)) = Lazy.force xs in x

let rec fib =
  lazy (InfiniteCons (1, map2 ( + ) fib (tail fib))))
```

```
# tail fib
Exception: CamlinternalLazy.Undefined.
```

A bit of Haskell-- in your OCaml?

```
type 'a cell = InfiniteCons of 'a * 'a stream
and 'a stream = 'a cell Lazy.t

let rec map2 f xs ys =
  lazy (match Lazy.force xs, Lazy.force ys with
      | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
        InfiniteCons (f x y, map2 f xs ys))

let tail xs =
  let (InfiniteCons (_, xs)) = Lazy.force xs in xs

let head xs =
  let (InfiniteCons (x, _)) = Lazy.force xs in x

let rec fib =
  lazy (InfiniteCons (1, map2 ( + ) fib (tail fib))))
```

```
# tail fib
Exception: CamlinternalLazy.Undefined.
```

What a **cryptic** error! Better be focused!

and to reason?

Starting the proof of

```
head (map2 f xs ys) = f (head xs) (head ys)
```

and to reason?

Starting the proof of

```
head (map2 f xs ys) = f (head xs) (head ys)
```

is not very exciting, since by unfolding the definitions we get:

```
let InfiniteCons (x, _) =
  Lazy.force (lazy (match Lazy.force xs, Lazy.force ys with
    | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
      InfiniteCons (f x y, map2 f xs ys)))
in
  x
```

for the left hand side and this for the right hand side:

```
f (let InfiniteCons (x, _) = Lazy.force xs in x)
  (let InfiniteCons (y, _) = Lazy.force ys in y)
```

and to reason?

Starting the proof of

```
head (map2 f xs ys) = f (head xs) (head ys)
```

is not very exciting, since by unfolding the definitions we get:

```
let InfiniteCons (x, _) =
  Lazy.force (lazy (match Lazy.force xs, Lazy.force ys with
    | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
      InfiniteCons (f x y, map2 f xs ys)))
in
  x
```

for the left hand side and this for the right hand side:

```
f (let InfiniteCons (x, _) = Lazy.force xs in x)
  (let InfiniteCons (y, _) = Lazy.force ys in y)
```

Reasoning is **polluted**
by the elimination of `lazy`s and `Lazy.force`s.

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- Line **1** declares a **coalgebric** datatype `'a stream`.
- `Head` and `Tail` are declared **observations** for `streams`.

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- On Line **6**, `..#Head -> f xs#Head ys#Head` is read:

  « The observation of the head of *`map2 f xs ys`* triggers the evaluation of *`f`* for the head observed for *`xs`* and the head observed for *`ys`*. »

  In other words, we have by construction:

  ```
  (map2 f xs ys)#Head = f xs#Head ys#Head
  ```

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- On Line **6**, `..#Head -> f xs#Head ys#Head` is read:

  « The observation of the head of *`map2 f xs ys`* triggers the evaluation of *`f`* for the head observed for *`xs`* and the head observed for *`ys`*. »

  In other words, we have by construction:

  ```
  (map2 f xs ys)#Head = f xs#Head ys#Head
  ```

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- On Line **13**, the reference of `fib` to itself does not trigger divergence since **the evaluation is guarded by the copattern analysis**.

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- Finally, notice that, contrary to a standard pattern-matching, the typing rule of a copattern-matching accepts **branches with distinct types**.
- For instance, in `map2`, the first branch has the output type `c` of `f` where the second branch produces a value of type `c stream`.
- Of course! Observations have distinct types!

With copattern-matching

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
8
9  let corec fib : int stream with
10   | ..#Head -> 1
11   | ..#Tail : int stream with
12   | ...#Head -> 1
13   | ...#Tail -> map2 ( + ) fib fib#Tail
```

- Abel, Pientka, Thibodeau et Setzer introduced copatterns to avoid the **loss of subject reduction** of CIC with CoInductive predicates.
- Our less ambitious objective: introduce a syntactic construction of higher-level than the pair `lazy`/`Lazy.force` to define infinite values.

Can we extend OCaml with
copattern-matchings analysis (without too much pain)?

Yes! And that it is easy!

A simple macro

If your favorite language $\mathcal{L}$ is equipped with

- Generalized Algebraic Datatypes (GADTs) ;
- Higher-order polymorphism ;

then the extension of $\mathcal{L}$ with copattern-matching can be sum up to a **purely local program transformation**.

Idea

A value of a coalgebraic datatype is a function defined by case on the observations that can be made by the environmnent.

One technical difficulty

How to have the typechecker accept the heterogeneity of the observation typings?

The program transformation on an example

```
1  type 'a stream = { Head : 'a; Tail : 'a stream }
2
3  let rec map2
4  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
5  = fun f xs ys -> cofunction : c stream with
6    | ..#Head -> f xs#Head ys#Head
7    | ..#Tail -> map2 f xs#Tail ys#Tail
```

The program transformation on an example

```
1  type 'a stream =
2  | Stream of { dispatch : 'o. ('a, 'o) stream_query -> 'o }
3  and (_, _) stream_query =
4  | Head : ('a, 'a) stream_query
5  | Tail : ('a, 'a stream) stream_query
6
7  let rec map2
8  : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
9  = fun f xs ys -> cofunction : c stream with
10   | ..#Head -> f xs#Head ys#Head
11   | ..#Tail -> map2 f xs#Tail ys#Tail
```

- A coalgebraic datatype gives rise to two type declarations:
  - The type `'a stream` holds the values of a coalgebraic datatype translated as `dispatch` functions, polymorphic in the type `'a` of observations.
  - The type `'a stream_query` denotes the **reifications of observations**. That's a GADT: the second type parameter is used to specified the type of reified observation.

The program transformation on an example

```
1  type 'a stream =
2  | Stream of { dispatch : 'o. ('a, 'o) stream_query -> 'o }
3  and (_, _) stream_query =
4  | Head : ('a, 'a) stream_query
5  | Tail : ('a, 'a stream) stream_query
6
7  let head (Stream { dispatch }) = dispatch Head
8  let tail (Stream { dispatch }) = dispatch Tail
9
10 let rec map2
11 : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
12 = fun f xs ys -> cofunction : c stream with
13   | ..#Head -> f xs#Head ys#Head
14   | ..#Tail -> map2 f xs#Tail ys#Tail
```

- To observe the head of a **`stream`**, it is enough to give the control to its **`dispatch`** function to let it interpret the **`Head`** observation.

The program transformation on an example

```
1  type 'a stream =
2  | Stream of { dispatch : 'o. ('a, 'o) stream_query -> 'o }
3  and (_, _) stream_query =
4  | Head : ('a, 'a) stream_query
5  | Tail : ('a, 'a stream) stream_query
6
7  let head (Stream { dispatch }) = dispatch Head
8  let tail (Stream { dispatch }) = dispatch Tail
9
10 let rec map2
11 : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
12 = fun f xs ys ->
13   let dispatch : type o. (c, o) stream_query -> o = function
14   | Head -> f (head xs) (head ys)
15   | Tail -> map2 f (tail xs) (tail ys)
16   in
17   Stream { dispatch }
```

- A coalgebric value defined by a copattern-matching is translated into a function **`dispatch`** defined by a pattern-matching.

The program transformation on an example

```
1  type 'a stream =
2  | Stream of { dispatch : 'o. ('a, 'o) stream_query -> 'o }
3  and (_, _) stream_query =
4  | Head : ('a, 'a) stream_query
5  | Tail : ('a, 'a stream) stream_query
6
7  let head (Stream { dispatch }) = dispatch Head
8  let tail (Stream { dispatch }) = dispatch Tail
9
10 let rec map2
11 : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
12 = fun f xs ys ->
13   let dispatch : type o. (c, o) stream_query -> o = function
14   | Head -> f (head xs) (head ys)
15   | Tail -> map2 f (tail xs) (tail ys)
16   in
17   Stream { dispatch }
```

- The first case is well-typed since `o = c`, which is the output type of `f`.
- The second case is well-typed since `o = c stream`, which is the output type of the recursive call to `map2`.

That's it?

Essentially yes!

Strengths of this translation

- The translation is simple.
- This translation extends naturally to **nested copatterns** and to **indexed codatatypes**.
- The covering checking of copatterns is reduced to an exhautiveness checking already implemented by the GADT pattern-matching.

Weaknesses of this translation

- Being purely syntactic, it imposes type annotations on every copattern-matching so that the GADT typechecker has enough of them to properly compute available equations.
- Each coalgebraic value costs a closure allocation.

Current status of this line of research

Results

- An extension of OCaml (`opam switch 4.04.0+copatterns`)
- Other eluded points (see PPDP'17 paper):
  - `lazy` introduces **sharing between the observations of the same value**.
  - **The reified observations are first-class and first-order values**: they can be compared and serialized efficiently, which open hackish opportunities to design optimized representations of coalgebraic objects.

Perspectives

- Can this encoding be applied to Coq? No! `Universe inconsistency`!

```
1  Inductive natstream_obs : Type -> Type :=
2  | Head : natstream_obs nat
3  | Tail : natstream_obs natstream
4  with natstream :=
5  | Dispatch : (forall a, natstream_obs a -> a) -> natstream.
```

- Is Robert Harper LICS'18 paper a way to circumvent this problem?

Meta

Can Coq be self-sustainable?

Alice in Wonderland

A fifty years long project

- Let's use Gallina as a Metalanguage for Coq!
- Gallina to write proof scripts, to develop decision procedures, to extend the language with new constructions... and to bootstrap Coq!

Difficulties

1. Coq is not LISP: the syntax is less uniform, the language is typed.
2. Coq has no compiler: proof search, typechecking are timeconsuming!
3. Coq is complex: heuristics, incomplete algorithms, many languages...

Current (very partial) answers

1. Mtac2: do not reify Coq in Coq, only provides introspection.
2. CertiCoq (Matthieu)
3. Complexity management?

A glimpse of Mtac2 for tactic development

```
1  Definition solve_tauto : forall (l : list dyn) {P : Prop}, M P :=
2    mfix2 f (l : list dyn) (P : Prop) : M P :=
3      mtry lookup P l
4      with NotFound =>
5        mmatch P as P' return M P' with
6        | True => ret I
7        | [? Q1 Q2] Q1 ∧ Q2 =>
8          q1 <- f l Q1; q2 <- f l Q2; ret (conj q1 q2)
9        | [? Q1 Q2] Q1 ∨ Q2 =>
10         mtry q1 <- f l Q1; ret (or_introl q1)
11         with TautoFail => q2 <- f l Q2; ret (or_intror q2) end
12       | [? (Q1 Q2 : Prop)] Q1 -> Q2 =>
13         nu q1, q2 <- f (Dyn q1 :: l) Q2; abs_fun q1 q2
14       | [? X (Q : X -> Prop)] (exists x : X, Q x) =>
15         x <- evar X; q <- f l (Q x); b <- is_evar x;
16         if b then raise TautoFail
17         else ret (ex_intro Q x q)
18       | _ => raise TautoFail
19       end
20     end.
```

A glimpse of Mtac2 for proof scripts

```
1  Lemma sub_0_r : forall n, n - 0 = n.
2  MProof.
3    cintro n {- case n &> [m: idtac | intro n ] &> reflexivity -}.
4  Qed.
```

A challenge: Compiling Mtac2

Mtac2 interpreter is bipolar

- An inhabitant of MA is an (strongly-typed) abstract syntax tree.
- The interpretation of this syntax tree is done outside Coq, in OCaml.
- In this interpreter, terms in position of `mmatch` scrutinee are object-level terms.
- As long as we interpret terms, that is fine because ASTs are available.
- But in a compiled setting, there is no more ASTs!

A potential fix: schizophrenic extraction

- Turn `extract : forall A, constr A -> erasure A` into `extract : forall A, constr A -> erasure_and_reify A`.

Conclusion

Coq as a language of choice for the discriminating mathacker!

4. Meta
   - Metaprogramming Coq in Coq with Mtac2
3. Infinity
   - Which language to define infinite objects?
   - Duality for code reuse
2. Differences
   - Certified differential functional programming
   - Relational analysis using bi-interpreters
1. Effects
   - Compositional Effects
   - Building effect interfaces

Une variante pour le partage des calculs I

```
let corec fib : int stream with
  | ..#Head -> 1
  | ..#Tail : int stream with
  | ...#Head -> 1
  | ...#Tail -> map2 ( + ) fib fib#Tail
```

```
let rec fib : int stream =
  let dispatch : type o. (o, int) stream_query -> o = function
    | Head -> 1
    | Tail ->
      let dispatch : type o. (o, int) stream_query -> o = function
      | Head -> 1
      | Tail -> map2 ( + ) fib (tail fib)
      in
      Stream { dispatch }
   in
      Stream { dispatch }
```

Une variante pour le partage des calculs II

```
1  let rec lazy_fib : int stream = lazy cofunction : int stream with
2    | ..#Head -> 1
3    | ..#Tail : int stream with
4    | ...#Head -> 1
5    | ...#Tail -> lazy_map2 ( + ) lazy_fib lazy_fib#Tail
```

Une variante pour le partage des calculs III

```
let rec fib : int stream =
  let dispatch : type o. (o,int) stream_query -> o = function
    | Head -> 0
    | Tail ->
        let dispatch : type o. (o,int) stream_query -> o = function
          | Head -> 1
          | Tail -> map2 ( + ) fib (tail fib)
        in
        let hb2 = lazy (dispatch Head) in
        let tb2 = lazy (dispatch Tail) in
        let mem2 : type o. (o,int) stream_query -> o = function
          | Head -> Lazy.force hb2
          | Tail -> Lazy.force tb2
        in Stream { dispatch = mem2 }
  in
  let hb1  = lazy (dispatch Head) in
  let tb1  = lazy (dispatch Tail) in
  let mem1 : type o. (o,int) stream_query -> o = function
    | Head -> Lazy.force hb1
    | Tail -> Lazy.force tb1
  in Stream { dispatch = mem1 }
```

Imbrication des comotifs et des motifs

```
let corec cycle : nat -> nat stream with
  | (.. n)#Head -> n
  | (.. Zero)#Tail -> cycle (Succ (Succ (Succ Zero)))
  | (.. (Succ n))#Tail -> cycle n

(* est traduit en: *)

let rec cycle : nat -> nat stream = fun (x0 : nat) ->
   let dispatch : type o. (o, nat) stream_query -> o = function
      | Head  -> x0
      | Tail  -> (match x0 with
         | Zero   -> cycle (Succ (Succ (Succ Zero)))
         | Succ x1 -> cycle x1)
   in Stream { dispatch }
```

Quand les types coalgébriques rencontrent les GADTs

```
type ('a,'b) fairbistream = {
  Left  : int * (read, 'b) fairbistream <- (unread, 'b) fairbistream;
  Right : int * ('a, read) fairbistream <- ('a, unread) fairbistream;
  BTail : (unread, unread) fairbistream <- (read, read) fairbistream;
}

type ('a, 'b, 'e) twobuffer =
| E :             (read  , read  , 'e) twobuffer
| L : 'e      -> (unread, read  , 'e) twobuffer
| R :      'e -> (read  , unread, 'e) twobuffer
| F : 'e * 'e -> (unread, unread, 'e) twobuffer

let corec zfrom
: type a b. int -> (a, b, int) twobuffer -> (a, b) fairbistream with
  | (.. n  E        ) #BTail -> zfrom (n + 1) (F (n, -n))
  | (.. n (L  x    )) #Left  -> (x, zfrom n E)
  | (.. n (F (x, y))) #Left  -> (x, zfrom n (R y))
  | (.. n (R     y )) #Right -> (y, zfrom n E)
  | (.. n (F (x, y))) #Right -> (y, zfrom n (L x))
```

Tester la couverture et la redondance

```
1  let corec f : int -> int stream with
2    | (.. n)#Head -> 1
3    | (.. n)#Tail -> f (n - 1)
4    | (.. n)#Tail : int stream with
5    |          ... #Head -> n
6    |          ... #Tail -> f (n + 1)
```

Les coinductifs de Coq et la préservation du typage

```
1  CoInductive U :=
2    In : U -> U.
3
4  CoFixpoint u :=
5    In u.
6
7  Definition force (x : U) :=
8    match x with In y => In y end.
9
10 Definition eq (x : U) : x = force x :=
11   match x with In y => @eq_refl U (In y) end.
12
13 Definition eq_u : u = In u :=
14   eq u.
15
16 Compute (eq u). (* [eq u] se réduit vers [eq_refl]  *)
17 (* ... mais [eq_refl] n'est pas de type [u = In u]! *)
18 Fail Check (eq_refl : u = In u).
```

(Exemple attribué à Nicolas Oury.)

De l'utilité des observations réifiées

```
1  let trace hst (Stream {dispatch}) =
2    let new_dispatch d =
3      hst := d :: !hst;
4      dispatch d
5    in Stream {dispatch = new_dispatch}
6
7  let h0 : history = ref []
8
9  let rec zeros : nat stream =
10   let corec zs : nat stream with
11     | ..#Head  -> Zero
12     | ..#Tail  -> zeros
13   in trace h0 zs
```

Le jeu de la vie I

```
type a b. (a comonad -> b) -> a comonad -> b comonad

type `a zipper = {
  Left   :  `a stream;
  Proj   :  `a;
  Right  :  `a stream;
}

let cobind : type a b. (a zipper -> b) -> a zipper -> b zipper
= fun f z -> cofunction : b zipper with
  | .. #Left   -> map_iterate f move_left (move_left z)
  | .. #Proj   -> f z
  | .. #Right  -> map_iterate f move_right (move_right z)
```

```
let rec map_iterate : type a b. (a -> b) -> (a -> a) -> a -> b stream
  = fun f shift z -> cofunction : b stream with
  | .. # Head  -> f z
  | .. # Tail  -> map_iterate f shift (shift z)
```

Le jeu de la vie II

```
let custom_dispatch : type a.
  ((a, a) zipper_query -> a) ->
  ((a stream, a) zipper_query -> a stream) ->
  a zipper
  = fun pr dir -> cofunction : a zipper with
    | .. # Proj   -> pr Proj
    | .. # Left   -> dir Left
    | .. # Right  -> dir Right

let move : type a.
  (a stream, a) zipper_query ->
  (a zipper -> a stream) -> (a zipper -> a stream) ->
  a zipper -> a zipper
  = fun dir fwd bwd z ->
  let corec bs : a stream with
      | .. # Head -> z#Proj
      | .. # Tail -> bwd z
  in
  custom_dispatch
    (fun _ -> (fwd z)#Head)
    (fun dir' -> if dir = dir' then (fwd z)#Tail else bs)

let move_left  z = move Left left right z
let move_right z = move Right right left z
```

Le jeu de la vie III

```
let point = fun x y -> cofunction : a zipper with
  | .. # Left  -> repeat y
  | .. # Proj  -> x
  | .. # Right -> repeat y

let rec iterate : type a. (a -> a) -> a -> a stream = fun f a ->
  cofunction : a stream with
  | ..#Head -> a
  | ..#Tail -> iterate f (f a)

let rule z =
  let left  = (move_left z)#Proj
  and middle = z#Proj
  and right = (move_right z)#Proj
  in
    not (left && middle && not right || left = middle)

let game_of_life = iterate (cobind rule) (point true false)
```