Programming = proving?

The Curry-Howard correspondence today

Ninth lecture

Sisyphus happy: infinite data types, proofs by coinduction, and reactive programming

Xavier Leroy

Collège de France

2019-01-16

Inductive types, inductive predicates

The lecture of Nov 28th 2018 presented inductive types and inductive predicates, a powerful mechanism to

- define data types and logical predicates
- that are finitely generated by constructors
- and used by structural induction and case analysis.

Example: the natural numbers in Coq.

```
Inductive nat : Type := O: nat | S: nat -> nat.

Fixpoint add (n m: nat) {struct n} : nat :=
  match n with O => m | S n' => S (add n' m) end.

Inductive even : nat -> Prop :=
  | even_O: even O
  | even_S: forall n, even n -> even (S (S n)).
```

Infinite data

How can we declare and work with infinite data structures?

For instance:

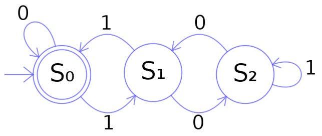
- Streams, that is, infinite lists.
  A stream = a pair of a value and of a stream.
- Infinite binary trees.
  An infinite binary tree = a triple (left subtree, value, right subtree).

Note: “infinite” here means “potentially infinite”: a terminating program will only traverse a finite part of the structure.

Computer representation of infinite data

1- By directed graphs with cycles.

Example: a finite deterministic automaton over the alphabet $\{0, 1\}$ encodes an infinite binary tree (with, at each node, a Boolean “accepting / not accepting”).



A finite graph can only represent regular infinite structures, that is, those having finitely many different sub-structures.

Regular infinite: the stream 0.1.2.0.1.2.0.1.2. . . .

Non-regular infinite: the stream of integers 0.1.2.3.4.5.6.7.8.9 . . .

Computer representation of infinite data

2- By delayed / on-demand evaluation of sub-structures.

Evaluation explicitly delayed by a function:

```
type 'a stream = unit -> 'a cell
and 'a cell = Cons of 'a * 'a stream
let tail s = match s() with Cons(h,t) -> t
```

On-demand evaluation via an explicit `lazy` type, as in OCaml:

```
type 'a stream = 'a cell Lazy.t
and 'a cell = Cons of 'a * 'a stream
let tail s = match Lazy.force s with Cons(h,t) -> t
```

On-demand evaluation by default, as in Haskell:

```
data Stream a = Cons a (Stream a)   (* implicitly "lazy" *)
```

This lecture

How to model infinite data structures and reason upon them?

- Classic set-theoretic approach: greatest fixed points.
- Proof theoretic approach: infinite trees and infinite derivations.
- Coalgebraic approach: “codata” defined by their observations.

Two applications:

- The partiality monad, to do general recursion in type theory.
- Reactive programming, viewed as programming over infinite streams, or maybe not…

I

Greatest fixed points

Least fixed point, greatest fixed point

Let A be a set and $F : \mathcal{P}(A) \to \mathcal{P}(A)$ a monotonically increasing function: if $X \subseteq Y$ then $F(X) \subseteq F(Y)$.

If $F(X) \subseteq X$ we say that X is stable by F.
If $X \subseteq F(X)$ we say that X is consistent for F.

Theorem (Knaster, Tarski, Kleene)

The set $\{x \mid x = F(x)\}$ of fixed points of F is a complete lattice.
In particular,

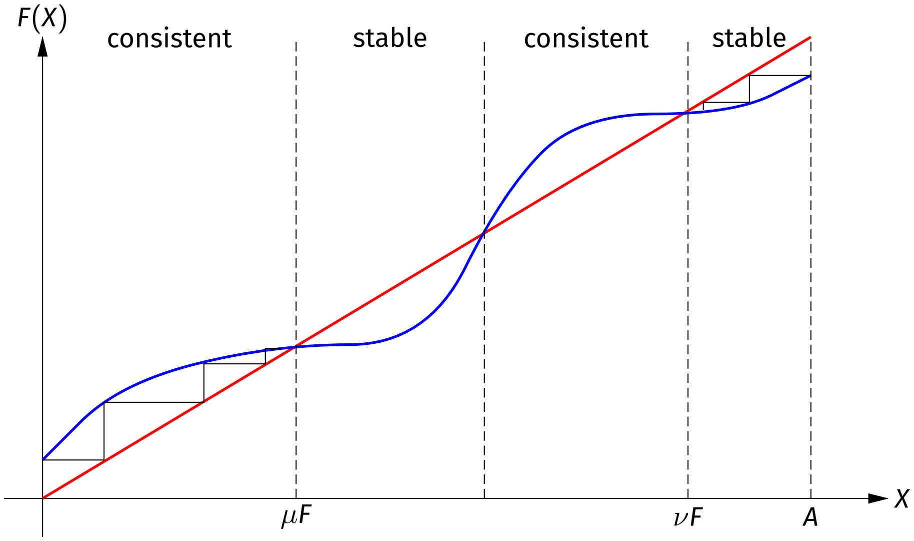
$$\mu F \overset{\mathit{def}}{=} \bigcap \{X \mid X \text{ stable by } F\} \quad \text{is the least fixed point of } F$$

$$\text{and it is the limit of the increasing sequence } \emptyset, F(\emptyset), F(F(\emptyset)), \ldots$$

$$\nu F \overset{\mathit{def}}{=} \bigcup \{X \mid X \text{ consistent for } F\} \quad \text{is the greatest fixed point of } F$$

$$\text{and it is the limit of the decreasing sequence } A, F(A), F(F(A)), \ldots$$

Fixed points, graphically



Induction and co-induction

Induction principle: if X is stable by F, then $\mu F \subseteq X$.

In other words: to show $\forall a \in \mu F,\ a \in X$,
it suffices to show $\forall a \in F(X),\ a \in X$.

Coinduction principle: if X is consistent for F, then $X \subseteq \nu F$.

In other words: to show $\forall a \in X,\ a \in \nu F$,
it suffices to show $\forall a \in X,\ a \in F(X)$.

Alternative: to show $a \in \nu F$, it suffices to find a set X
such that $a \in X$ and $\forall b \in X,\ b \in F(X)$.

Inductive types, coinductive types

Consider an algebraic type declaration, such as

```
type nat = O | S of nat
```

The $F(X)$ operator is defined as “the values we can construct by applying a constructor once, taking arguments of the algebraic type from X”.
In our example:

$$F(X) = \{\mathtt{O}\} \cup \{\mathtt{S}(x) \mid x \in X\}$$

This operator is increasing if all occurrences of the algebraic type in the types of constructors are strictly positive.

The inductive type defined by this declaration is μF, the least fixed point of operator F.

The coinductive type defined by this declaration is νF, the greatest fixed point of operator F.

The inductive interpretation

```
Inductive nat: Type := O: nat | S: nat -> nat
```

$$F(X) = \{\mathtt{O}\} \cup \{\mathtt{S}(x) \mid x \in X\}$$

`nat` is the least fixed point of F.

It is also the limit of the sequence $\emptyset$, $F(\emptyset) = \{\mathtt{O}\}$, $F(F(\emptyset)) = \{\mathtt{O}, \mathtt{S}(\mathtt{O})\}, \ldots$

More generally, `nat` is the set of terms $X \overset{def}{=} \{\mathtt{S}^n(\mathtt{O}) \mid n \in \mathbb{N}\}$.

- For every n, $\mathtt{S}^n(\mathtt{O}) \in F^{n+1}(\emptyset) \subseteq \mu F$.
- Conversely, the set X is stable by F, since $F(X) = \{\mathtt{O}\} \cup \{\mathtt{S}(\mathtt{S}^n(\mathtt{O})) \mid n \in \mathbb{N}\} = X$, hence, by the induction principle, $\mu F \subseteq X$.

The coinductive interpretation

```
CoInductive conat: Type := O: conat | S: conat -> conat
```

$$F(X) = \{\mathtt{O}\} \cup \{\mathtt{S}(x) \mid x \in X\}$$

`conat` is the greatest fixed point of F.

It is also the limit of the sequence U, $F(U) = \{\mathtt{O}\} \cup \{\mathtt{S}(x) \mid x \in U\}, \ldots,$
$F^n(U) = \{\mathtt{O}, \mathtt{S}(\mathtt{O}), \ldots, \mathtt{S}^{n-1}(\mathtt{O})\} \cup \{\mathtt{S}^n(x) \mid x \in U\}, \ldots$

Every $\mathtt{S}^n(\mathtt{O})$ is in `conat`, since $\mu F \subseteq \nu F$.

However, `conat` also contained the infinite term ω defined by $\omega = \mathtt{S}\ \omega$.

That's because $\{\omega\} \subseteq F(\{\omega\}) = \{\mathtt{O}, \omega\}$, hence, by the coinduction principle, $\{\omega\} \subseteq \nu F$.

Inductive predicates, coinductive predicates

The same approach extends to inductive families, and in particular to predicates defined by axioms and inference rules.

Example: the predicate "being even" over natural numbers.

$$\mathtt{even}(\mathtt{0}) \qquad\qquad \frac{\mathtt{even}(n)}{\mathtt{even}(\mathtt{S}(\mathtt{S}(n)))}$$

This set of rules corresponds to an operator $F(X)$ that computes the facts we can deduce from the facts X by applying one rule:

$$F(X) = \{\mathtt{even}(\mathtt{0})\} \cup \{\mathtt{even}(\mathtt{S}(\mathtt{S}(n))) \mid \mathtt{even}(n) \in X\}$$

The inductive interpretation

```
Inductive even: conat -> Prop :=
  | even_O: even O
  | even_S: forall n, even n -> even (S(S n)).
```

$$F(X) = \{\mathtt{even}(\mathtt{O})\} \cup \{\mathtt{even}(\mathtt{S}(\mathtt{S}(n))) \mid \mathtt{even}(n) \in X\}$$

`even` is defined as μF, the least fixed point F.

Induction principle: to show $\forall n,\ \mathtt{even}(n) \Rightarrow P(n)$,
it suffices to show $P(\mathtt{O})$ and $\forall n, P(n) \Rightarrow P(\mathtt{S}(\mathtt{S}(n)))$.

We can characterize `even`: (with $f : \mathtt{nat} \to \mathtt{conat}$ canonical injection)

- $\forall n : \mathtt{nat}, \mathtt{even}(f(n+n))$ by induction over $n : \mathtt{nat}$
- $\forall m, \mathtt{even}(m) \Rightarrow \exists n,\ m = f(n+n)$ by induction over $\mathtt{even}(m)$
- $\forall m, \mathtt{even}(m) \Rightarrow m \neq \omega$ by induction over $\mathtt{even}(m)$

The coinductive interpretation

```
CoInductive coeven: conat -> Prop :=
  | coeven_O: coeven O
  | coeven_S: forall n, coeven n -> coeven (S(S n)).
```

$$F(X) = \{\mathtt{coeven}(\mathtt{O})\} \cup \{\mathtt{coeven}(\mathtt{S}(\mathtt{S}(n))) \mid \mathtt{coeven}(n) \in X\}$$

`coeven` is defined as νF, the greatest fixed point of F.

Coinduction principle: to show $\forall n,\ P(n) \Rightarrow \mathtt{coeven}(n)$,
it suffices to show $\forall n, P(n) \Rightarrow n = \mathtt{O} \vee \exists m, n = \mathtt{S}(\mathtt{S}(m)) \wedge P(m)$.

We can therefore show:

- $\forall n, \mathrm{even}(n) \Rightarrow \mathtt{coeven}(n)$ by coinduction, taking $P(n) = \mathrm{even}(n)$.
- $\mathtt{coeven}(\omega)$ by coinduction, taking $P(n) = (n = \omega)$.

II

Infinite trees, infinite derivations

An interpretation in terms of trees

A value of a (co-)inductive type can be viewed as a tree, with
at leaves: constant constructors,
at nodes: constructors of arity *n*, where *n* = number of sub-trees.

(We treat `C of nat -> t` as a constructor with infinite arity.)

Values of a co-inductive type are finite or infinite trees.

Values of an inductive type are trees where all branches are finite.

Values of an inductive type whose constructors have finite arities are finite trees.

$$\begin{tikzcd} \mathtt{S} \arrow[d] \\ \mathtt{O} \end{tikzcd} \qquad \begin{tikzcd} \mathtt{S} \arrow[d] \\ \mathtt{S} \arrow[d] \\ \mathtt{S} \arrow[d] \\ \mathtt{S} \\ \vdots \end{tikzcd}$$

An interpretation in terms of derivations

Likewise, a proof term for a (co-)inductive predicate can be viewed as a derivation tree, with at leaves: instances of axioms,
at nodes: instances of inference rules with *n* premises, *n* = number of sub-trees.

A coinductive predicate holds iff it is the conclusion of a finite or infinite derivation.

An inductive predicate holds iff it is the conclusion of a derivation where all branches are finite.

An inductive predicate where all rules have a finite number of premises holds iff it is the conclusion of a finite derivation.

$$\frac{\texttt{even}(\texttt{0})}{\texttt{even}(\texttt{S}(\texttt{S}(\texttt{0})))} \qquad \begin{array}{c} \vdots \\ \hline\hline \texttt{coeven}(\omega) \\ \hline\hline \texttt{coeven}(\omega) \\ \hline\hline \texttt{coeven}(\omega) \end{array}$$

Structural recursion, productive corecursion

A recursive function $f : \mathit{ind} \rightarrow t$ over an inductive type *ind* must be structural: recursive calls $f(y)$ are done over strict sub-terms y of the argument.

```
Fixpoint f(n: nat) : t :=
  match n with
  | O => ...
  | S p => ... f p ✔ ... f n ✘ ... end
```

Consequently: the computation $f(n)$ always terminates.

Structural recursion, productive corecursion

A recursive function $f : ind \to t$ over an inductive type *ind* must be structural: recursive calls $f(y)$ are done over strict sub-terms y of the argument.

A corecursive function $f : t \to coind$ returning a coinductive type must be productive: recursive calls $f(y)$ must be strict sub-terms of the result.

```
CoFixpoint f(x: t) : conat :=
  match x with
  | ... => O ✔
  | ... => S(f(...)) ✔
  | ... => f(...) ✘
  | ... => g(f(...)) ✘
```

Consequently: the computation of the head constructor of $f(x)$ always terminates.

Examples of corecursive definitions

The number “positive infinity”:

```
    CoFixpoint omega : conat := S omega.  ✔
```

The sum of two natural numbers, possibly infinite:

```
    CoFixpoint add (p q: conat) : conat :=
      match p with
      | O => q                    ✔ (* no recursive call *)
      | S p’ => S (add p’ q) ✔ (* recursive call guarded by S *)
```

Subtraction cannot be defined: `sub omega omega` would diverge while determining the head constructor.

```
    CoFixpoint sub (p q: conat) : conat :=
      match p, q with
      | O, _ => O                 ✔ (* no recursive call *)
      | _, O => p                 ✔ (* no recursive call *)
      | S p’, S q’ => sub p’ q’ ✘ (* unguarded recursive call *)
      end.
```

Examples of coinductive proofs

In the spirit of Curry-Howard, proofs by coinduction are corecursive definitions of proof terms.

For instance, let's define the coinductive predicate `infinite` as

```
CoInductive infinite: conat -> Prop :=
  | infinite_S: forall n, infinite n -> infinite (S n).
```

Then, the proof of `infinite omega` is, morally,

```
CoFixpoint omega_infinite: infinite omega :=
    infinite_S omega omega_infinite.
```

This is not quite right because `omega` is not convertible with `S omega` (see next slide), hence we must explicitly apply the (provable) equality `omega_eq: omega = S omega`.

```
CoFixpoint omega_infinite: infinite omega :=
    eq_ind_r _ (infinite_S omega omega_infinite) omega_eq.
```

Reduction rules

Just like recursive definitions (`Fixpoint`), corecursive definitions (`CoFixpoint`) cannot be arbitrarily "unrolled", since this would cause nontermination:

$$\texttt{omega} \to \texttt{S(omega)} \to \texttt{S(S(omega))} \to \cdots \qquad ✘$$

The reduction rule used by Coq: a corecursive definition $\nu x.\, a$ expands to $a\{x \leftarrow \nu x.a\}$ only when it is argument of a `match`:

$$\texttt{match}\ \nu x.\, a\ \texttt{with}\ \ldots \quad \to \quad \texttt{match}\ a\{x \leftarrow \nu x.a\}\ \texttt{with}\ \ldots$$

Compare with the rule for recursive definitions $\mu f.\, \lambda x.\, a$, which expand only when applied to a constructor:

$$(\mu f.\, \lambda x.\, a)\ (C\ b) \quad \to \quad a\{f \leftarrow \mu f.\, \lambda x.\, a,\ x \leftarrow C\ b\}$$

Unrolling equalities

Even if it does not hold by conversion, we can show the equality `omega = S omega` by proof.

First we show an extensionality property for values of type `conat`:

```
Lemma unroll: forall (n: conat),
    n = match n with O => O | S m => S m end.
```

Then, `unroll omega` has the following type, which reduces:

```
    omega = match omega with O => O | S m => S m end
→   omega = match S omega with O => O | S m => S m end
→   omega = S omega
```

Typing is not preserved by reductions

(E. Giménez, *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants*, PhD thesis, ENS Lyon, 1996)

We have the following reductions and typings:

$$\begin{tikzcd}
\texttt{unroll omega} \arrow[d, "*"'] \arrow[r, phantom, ":"] & \texttt{omega} = \texttt{S omega} \arrow[d, phantom, "\not\updownarrow \text{ (not convertible)}"] \\
\texttt{eq\_refl omega} \arrow[r, phantom, ":"] & \texttt{omega} = \texttt{omega}
\end{tikzcd}$$

This invalidates preservation of typing by reduction (*subject reduction*):
if $M : t$ and $M \to M'$ then $M' : t$.

This doesn't mean that Coq + coinductive types is inconsistent; only that a consistency proof is more difficult. (See Giménez's PhD.)

III

Coinductive types, general recursion, and the partiality monad

Partial computations

(V. Capretta, *General recursion via coinductive types*, LMCS(1), 2005)

```
CoInductive delay (A: Type) : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

`delay` *A* represents computations that return a value of type *A* if they terminate.

The `later` constructor materializes one step of computation.

The type `delay` being coinductive, we can have infinitely many computation steps, and therefore a computation that does not terminate. Example:

```
CoFixpoint bottom (A: Type) : delay A := later (bottom A).
```

Partial computations

```
    CoInductive delay (A: Type) : Type :=
      | now: A -> delay A
      | later: delay A -> delay A.
```

We characterize inductively the terminating computations, coinductively the diverging computations:

```
Inductive terminates (A: Type) : delay A -> A -> Prop :=
  | terminates_now:
      forall v, terminates (now v) v
  | terminates_later:
      forall a v, terminates a v -> terminates (later a) v.

CoInductive diverges (A: Type) : delay A -> Prop :=
  | diverges_later:
      forall a, diverges a -> diverges (later a).
```

General recursion

We can define arbitrary general recursive functions with a `delay` result type, provided all recursive calls are guarded by a `later` constructor.

```
CoFixpoint syracuse (n: nat): delay unit :=
  if Nat.eqb n 1 then now tt
  else if Nat.even n then later (syracuse (Nat.div n 2))
  else later (syracuse (3 * n + 1)).
```

We can, then, reason over termination or divergence of the function.

```
Conjecture Collatz_1:
  forall n, n >= 1 -> terminates (syracuse n) tt.
Conjecture Collatz_2:
  exists n, n >= 1 ∧ diverges (syracuse n).
```

A fixed-point combinator

Consider $F : (A \to \texttt{delay}\ B) \to (A \to \texttt{delay}\ B)$. We can construct a function $Y\ F : A \to \texttt{delay}\ B$ by iterating F from $\lambda x.\,\texttt{bottom}$ and taking "the first defined result":

$$\begin{array}{rcccccccccc}
(\lambda x.\,\texttt{bottom})\ a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
F(\lambda x.\,\texttt{bottom})\ a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & & \\
F^2(\lambda x.\,\texttt{bottom})\ a & \cdot & \cdot & \cdot & \cdot & \color{red}{v} & & & & & \\
F^3(\lambda x.\,\texttt{bottom})\ a & \cdot & \cdot & \cdot & \cdot & v & & & & &
\end{array}$$

Under some hypotheses over F, the fixed-point equation holds, namely the equivalence between $Y\ F\ a$ and $F\ (Y\ F)\ a$.

The partiality monad

The `delay` type is a monad, with the `now` constructor as `ret` operation, and the `bind` operation defined as sequencing of two computations.

```
CoFixpoint bind (A B: Type)
                (a: delay A) (f: A -> delay B) : delay B :=
  match a with
  | now v => later (f v)
  | later a’ => later (bind a’ f)
  end.
```

The expected properties of sequencing hold, for instance `bind a f` diverges iff `a` diverges or `a` terminates with `v` and `f v` diverges.

Productivity issues

Owing to the syntactic productivity criterion, the `bind` we just defined is often unusable inside a coinductive definition.

Example: McCarthy's 91 function.

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

```
CoFixpoint M (n: nat) : delay nat :=
  if Nat.leb n 100
  then bind (M (n + 11)) (fun x => M x)
  else now (n - 10).
```

The corecursive calls `M (n + 11)` and `M x` are rejected because they occur under the `bind`, which is not a constructor of type `delay`.

Going through the free monad

(N. A. Danielsson, *Beating the Productivity Checker Using Embedded Languages*, 2010)

We can work around the issue by presenting the monad as a coinductive type, whose constructors are the operations of the partiality monad: `ret`, `bind`, and `later`.

```
CoInductive mon: Type -> Type :=
  | ret: forall (A: Type), A -> mon A
  | bind: forall (A B: Type), mon A -> (A -> mon B) -> mon B
  | latr: forall (A: Type), mon A -> mon A.
```

Going through the free monad

Now, function 91 is productive, indeed:

```
CoFixpoint M (n: nat) : mon nat :=
  if Nat.leb n 100
  then bind (M (n + 11)) (fun x => M x)
  else ret (n - 10).
```

Going through the free monad

The description of a computation, of type `mon A`, is turned into a computation, of type `delay A`, by the following function:

```
CoFixpoint interp (A: Type) (m: mon A) : delay A :=
  match m with
  | ret v => now v
  | latr m => later (interp m)
  | bind (ret v) f => later (interp (f v))
  | bind (latr m) f => later (interp (bind m f))
  | bind (bind a f) g =>
                  later (interp (bind a (fun v => bind (f v) g)))
  end.
```

Note the use of the monadic laws `bind-ret` and `bind-bind` so as to reduce every `bind` to `later`(`interp` a)

IV

Codata and copatterns

Types defined by observations

Inductive types are finitely generated by their constructors. Likewise, we can say that coinductive types are finitely determined by their projections, that is, by the observations we can make over the values of those types.

Example: streams have two projections,

$$\mathtt{hd} : \mathtt{stream}\, A \to A \qquad \mathtt{tl} : \mathtt{stream}\, A \to \mathtt{stream}\, A$$

A stream s is entirely determined by the results of the observations $\mathtt{hd}(s), \mathtt{hd}(\mathtt{tl}(s)), \ldots, \mathtt{hd}(\mathtt{tl}^n(s)), \ldots$

Remark: a function $f : A \to B$ is also a "black box" determined by the results of the applications (observations) $f\ a_1, \ldots, f\ a_n, \ldots$

Data and codata

This suggests to classify types into

- Data types: integers, Booleans, $A \times B$, $A + B$, all inductive types.
- Codata types: functions $A \to B$, streams, all coinductive types.

Inductive types are presented as sums labeled by constructors. Likewise, coinductive types are presented as products labeled by projections (i.e. as records).

```
  list A := ⟨ nil | cons of A × list A ⟩
stream A := { hd: A; tl: stream A }
```

Defining codata

Codata is used by applying projections. But how is codata defined? By equations that define how it reacts to projections!

Examples: the stream `from` n of integers $n, n + 1, n + 2, \ldots$:

```
(from n).hd = n
(from n).tl = from (n + 1)
```

Mapping a function f over every element of a stream:

```
(map f s).hd = f (s.hd)
(map f s).tl = map f (s.tl)
```

Definitions by equations and pattern-matching

This style of definition is also present in Haskell and Agda to define recursive functions over inductive data, such as

```
double O = O
double (S x) = S (S (double x))
```

Symmetry: in the case of data, we need one equation per possible constructor of the argument; in the case of codata, we need one equation per possible projection of the result.

```
(map f s).hd = f (s.hd)          map f nil = nil
(map f s).tl = map f (s.tl)      map f (cons h t) =
                                         cons (f h) (map f t)
```

Patterns and copatterns

(Abel, Pientka, Thibodeau, Setzer, *Copatterns*, POPL 2013)

Abel et al propose to unify these two styles of definitions by equations with the help of patterns and copatterns.

Patterns (= possible shapes for data)

$$\begin{array}{lll} p ::= & x & \text{variable} \\ & \mid \mathit{Cstr}\ p_1 \ \cdots\ p_n & \text{constructor} \end{array}$$

Copattern (= possible observations over codata)

$$\begin{array}{lll} q ::= & \Box & \text{the object being defined} \\ & \mid q\ p & \text{function application} \\ & \mid q\ .\mathit{Proj} & \text{projection} \end{array}$$

A piece of codata is, then, a list of (copattern, expression) cases.
For instance, $\lambda x.a$ is $[(\Box\ x, a)]$
and $\{\mathtt{hd} = a, \mathtt{tl} = b\}$ is $[(\Box.\mathtt{hd}, a); (\Box.\mathtt{tl}, b)]$.

Examples of nested copatterns

Fibonacci numbers, inductively:

```
fib O         = S O
fib (S O)     = S O
fib (S (S x)) = fib x + fib (S x)
```

Fibonacci numbers, as a stream:

```
fibs .hd     = S O
fibs .tl .hd = S 0
fibs .tl .tl = zipWith plus fibs (fibs.tl)
```

The stream $n, n-1, \ldots, 1, 0, N, N-1, \ldots, 0, N, \ldots$:

```
(cycle n) .hd     = n
(cycle O) .tl     = cycle N
(cycle (S n)) .tl = cycle n
```

Reduction rules

Each equation is interpreted as a reduction rule (from left to right):

$$\begin{aligned}\texttt{cycle}\ (\texttt{S}(\texttt{S}(\texttt{S}\ \texttt{O}))).\texttt{tl}.\texttt{tl}.\texttt{hd} &\to \texttt{cycle}\ (\texttt{S}(\texttt{S}\ \texttt{O})).\texttt{tl}.\texttt{hd}\\ &\to \texttt{cycle}\ (\texttt{S}\ \texttt{O}).\texttt{hd}\\ &\to \texttt{S}\ \texttt{O}\end{aligned}$$

The issue in Coq (non-preservation of typing) is avoided because there are no equalities between a piece of codata and its expansion.

For instance, s and $\{\texttt{hd} = s.\texttt{hd}; \texttt{tl} = s.\texttt{tl}\}$ react identically to projections, but have no reasons to be equal.

Productivity and structural recursion

To guarantee normalization, simple syntactic conditions seem sufficient:

- Productivity: every copattern must start with a projection
- Structural recursion: if we make a recursive call over a variable *x*, it must appear under a constructor in the copattern.

```
F x = F (S x) ✗      (F x).tl = F (S x) ✓      F (S x) = G(F x) ✓
```

There are complex copatterns where productivity is less obvious:

```
        fibs .tl .tl = zipWith plus fibs (fibs.tl)
```

Agda uses sized types for finer control of productivity.

(Abel and Pientka, *Wellfounded Recursion with Copatterns and Sized Types*, JFP(26), 2016.)

V

Reactive programming and guarded recursion

Reactive programming

Broadly speaking: any program whose purpose is to react via computations to events coming from the outside world.

Example: a spreadsheet in “automatic recalculate” mode.

In this lecture: any program that receives a sequence of input values $i_0, i_1, i_2, \ldots$ and computes incrementally and “at the same pace” a sequence of output values $o_0, o_1, o_2, \ldots$

Reactive programming = programming with streams?

It is convenient to use streams to represent the sequences of inputs, of outputs, or of intermediate results.

However, a reactive program is not any function from streams to streams. The function must be causal: the n-th output o_n depends on the past inputs $i_0, \ldots, i_n$ only, not on the future inputs $e_{n+1}, e_{n+2}, \ldots$.

Examples of causal functions (✔) and non-causal functions (✘):

```
✔  F s = map f s
✔  F s = cons 0 s
✘  F s = tl s
✘  F s = cons (hd s) (F (tl (tl s)))
✔  F s = cons (hd s) (cons (hd s) (F (tl (tl s))))
```

Lustre, a causal-by-construction language

A Lustre program defines a set of streams $X_1, X_2, \ldots$ by a set of mutually-recursive equations $X_i = E_i$.

The expression language E_i is sufficiently restricted to guarantee causality. It includes:

- Pointwise computations: `X + Y` is `zipWith (+) X Y`.
- Temporal operators that reduce to `cons` over streams:

```
cst fby X   =  cons cst X
pre X       =  cons nil X
X -> Y      =  cons (hd X) Y
```

- Sampling operators over sub-clocks (represented as Boolean streams).

```
X when C  =  cons (if hd C then Pre(hd X) else Abs)
                  ((tl X) when (tl C))
```

Enforcing causality by typing

For additional flexibility (higher-order functions), we can replace syntactic causality conditions by type constraints.

Krishnaswami and Benton (2011–2013) extend types with a “later” modality, written $\triangleright A$.

Intuitively, values of type $\triangleright A$ are values of type A that must not be used immediately (causality would be violated), but can be used at the next time step.

$$\begin{aligned} \texttt{hd} &: \texttt{stream}\,A \to A \\ \texttt{tl} &: \texttt{stream}\,A \to \triangleright(\texttt{stream}\,A) \\ \texttt{cons} &: A \to \triangleright(\texttt{stream}\,A) \to \texttt{stream}\,A \end{aligned}$$

Examples of typings

(In the language of Clouston et al, *The guarded lambda-calculus*, LMCS(12), 2016)

With $\texttt{fix} : (\triangleright A \to A) \to A$, $\texttt{next} : A \to \triangleright A$, and
$\circledast : \triangleright(A \to B) \to \triangleright A \to \triangleright B$

$$\begin{aligned} \texttt{map} &= \lambda f.\, \texttt{fix}(\lambda m.\, \lambda s.\, \texttt{cons}\,(f\,(\texttt{hd}\,s))\,(m \circledast (\texttt{tl}\,s))) \\ &: (A \to B) \to \texttt{stream}\,A \to \texttt{stream}\,B \\ &\qquad\qquad \text{with } m : \triangleright(\texttt{stream}\,A \to \texttt{stream}\,B) \end{aligned}$$

$$\begin{aligned} \texttt{nats} &= \texttt{fix}(\lambda s.\, \texttt{cons}\,\texttt{O}\,(\texttt{next}\,(\texttt{map}\,\texttt{S}) \circledast s)) \\ &: \texttt{stream}\,\texttt{nat} \\ &\qquad\qquad \text{with } s : \triangleright(\texttt{stream}\,\texttt{nat}) \end{aligned}$$

Typing guarantees causality and productivity, while the definition

```
CoFixpoint nats := cons O (map S nats)
```

doesn't even pass Coq's syntactic productivity check.

Guarded recursive types

(Nakano, *A modality for recursion*, LICS 2000.)

We can form the recursive type $\mu\alpha.\,\tau$ provided α is guarded in τ : all occurrences of α are “under” a $\triangleright$ modality.

Examples:

$\mu\alpha.\,A \times \triangleright\alpha$	streams (infinite lists)
$\mu\alpha.\,\texttt{unit} + A \times \triangleright\alpha$	finite or infinite lists
$\mu\alpha.\,\texttt{bool} \times \triangleright\alpha \times \triangleright\alpha$	infinite binary trees (= deterministic automata)
$\mu\alpha.\texttt{unit} + \alpha$ ✘	not guarded

Guarded recursive types

(Clouston et al, *The guarded lambda-calculus*, LMCS(12), 2016)

In the “topos of trees” (lectures of Jan 9th 2019), these guarded recursive types are interpreted by time-indexed families of non-recursive types (hence not coinductive).

Example: the type of streams $\mu\alpha.\, A \times \triangleright\alpha$.

$$\begin{array}{ll}
\text{at time 0:} & \texttt{unit} \\
\text{at time 1:} & A \times \texttt{unit} \\
\text{at time 2:} & A \times A \times \texttt{unit} \\
\vdots & \\
\text{at time } n\text{:} & A \times \cdots A \times \texttt{unit} \text{ (lists of length } n\text{)}
\end{array}$$

Example: $\mu\alpha.\, \texttt{unit} + A \times \triangleright\alpha$ at time n = lists of length $\leq n$.

From guarded recursive types to coinductive types

We recover the usual coinductive types, freely usable at any time, via another modality: $\Box$, “forever”.

Semantically, $\Box A$ is the “constant object” that is the limit of the interpretations of A at time n when $n \to \infty$. For instance:

$$\begin{aligned}
\Box(\mu\alpha.\, A \times \triangleright\alpha) &\approx \lim_{n\to\infty} (\text{lists of } A \text{ of length } n) \\
&\approx \text{infinite lists of } A \\
\Box(\mu\alpha.\, \texttt{unit} + A \times \triangleright\alpha) &\approx \lim_{n\to\infty} (\text{lists of } A \text{ of length} \leq n) \\
&\approx \text{finite or infinite lists of } A
\end{aligned}$$

Restriction: A must be closed in $\Box A$.
(For example, this prohibits $\mu\alpha \ldots \Box \triangleright \alpha$)

From guarded recursive types to coinductive types

The $\Box$ modality behaves globally like the "necessarily" modality in S4 intuitionistic logic.

$$\frac{\Gamma \vdash a : \Box A}{\Gamma \vdash \texttt{unbox}\, a : A} \qquad \frac{\vec{x} : \vec{C} \vdash a : A}{\vec{x} : \vec{C} \vdash \texttt{box}(a) : \Box A} \qquad \frac{\vec{x} : \vec{C} \vdash a : \triangleright A}{\vec{x} : \vec{C} \vdash \texttt{prev}(a) : A}$$

C stands for a type constant over time: $C ::= \texttt{nat} \mid C \to C \mid \Box A$.

The box and unbox operators make it possible to define non-causal functions over coinductive types, such as

$$\begin{aligned} \texttt{hd}_c : \Box(\texttt{stream}\, A) \to A \quad &= \lambda s.\, \texttt{hd}(\texttt{unbox}(s)) \\ \texttt{tl}_c : \Box(\texttt{stream}\, A) \to \Box(\texttt{stream}\, A) \quad &= \lambda s.\, \texttt{box}(\texttt{prev}(\texttt{tl}(\texttt{unbox}(s)))) \end{aligned}$$

From guarded recursion to corecursion

There is no fixed point operator corresponding to corecursion. Instead, we must write guarded recursions, then apply `box`, thus guaranteeing productivity.

Example: the stream function "take every other element". We first define the function that produces a guarded recursive stream

$$\begin{aligned} F &: \Box(\mathtt{stream}\, A) \to \mathtt{stream}\ \mathtt{A} \\ &= \mathtt{fix}(\lambda f.\, \lambda s.\, \mathtt{cons}(\mathtt{hd}_c(s))(g \circledast \mathtt{next}(\mathtt{tl}_c(\mathtt{tl}_c(s)))))) \end{aligned}$$

then, using `box`, the function that produces a coinductive stream:

$$\begin{aligned} G &: \Box(\mathtt{stream}\, A) \to \Box(\mathtt{stream}\ \mathtt{A}) \\ &= \lambda s.\, \mathtt{box}(F(s)) \end{aligned}$$

VI

Concluding remarks

Happy like Sisyphus?

Using these techniques for coinductive definitions and proofs, we can "look infinity square in the eye" and program and reason directly with infinite objects.

All this also works in type theory and in systems such as Agda and Coq.

However, productivity conditions are very strict, and coinductive reasoning remains difficult.
$\Rightarrow$ more flexible proof principles, such as parameterized coinduction (PACO, C. K. Hur et al, 2013) and coinduction "all the way up" (D. Pous, 2016).

As shown by the example of reactive programming, "finitary" approaches (step-indexing, finite approximations) are sometimes preferable to coinductive approaches.

VII

Further reading

Further reading

A tutorial on set-theoretic coinduction, with application to subtyping:

- B. C. Pierce: *Types and Programming Languages*, chapter 21, MIT Press, 2002.

Coinduction in Coq:

- Y. Bertot and P. Castéran: *Interactive Theorem Proving and Program Development*, chapter 14.

The codata approach:

- A. Abel, B. Pientka, D. Thibodeau, A. Setzer: *Copatterns: programming infinite structures by observations*. POPL 2013. `http://www.cse.chalmers.se/~abela/popl13.pdf`

The guarded recursion approach:

- R. Clouston, A. Bizjak, H. B. Grathwohl, L. Birkedal: *The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types*. LMCS 12(3) 2016, `https://arxiv.org/abs/1606.09455`