Programmer = démontrer?

La correspondance de Curry-Howard aujourd'hui

Neuvième cours

Sisyphe heureux: types de données infinies, démonstrations par coinduction, et programmation réactive

Xavier Leroy

Collège de France

2019-01-16

COLLÈGE
DE FRANCE
1530

Types et prédicats inductifs

Dans le cours du 28 novembre 2018, nous avons vu les types et prédicats inductifs, un mécanisme puissant pour

- définir des types de données et des prédicats logiques ;
- qui sont finiment engendrés par des constructeurs ;
- que l'on utilise par récurrence structurelle et analyse de cas.

Exemple : les entiers naturels en 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)).
```

Données infinies

Comment déclarer et travailler sur des structures de données infinies ?

Par exemple :

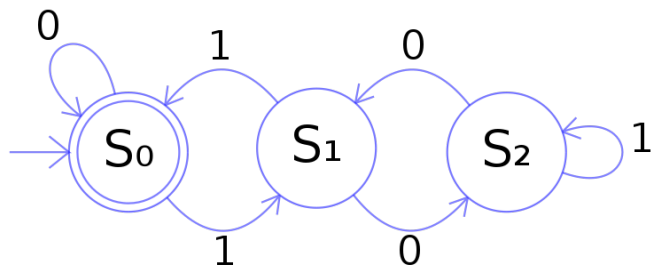
- Les flux, c.à.d. les listes infinies.
  Un flux = une paire d'une valeur et du flux suivant.
- Les arbres binaires infinis.
  Un a.b.i = un triplet (sous-arbre gauche, valeur, sous-arbre droit).

N.B. par ≪infini≫ on veut dire ≪potentiellement infinies≫ : un programme qui termine n'explorera qu'une partie finie de la structure.

Représentations informatiques des données infinies

1- Par des graphes orientés avec des cycles.

Exemple : un automate fini déterministe sur l'alphabet $\{0, 1\}$ encode un arbre binaire infini (avec à chaque noeud un Booléen «acceptant»/«non acceptant»).



Un graphe fini ne peut représenter que les structures infinies régulières, ayant un nombre fini de sous-structures différentes.

Infini régulier : le flux 0.1.2.0.1.2.0.1.2. . . .

Infini non régulier : le flux de tous les entiers 0.1.2.3.4.5.6.7.8.9 . . .

Représentations informatiques des données infinies

2- Par évaluation retardée / à la demande des sous-structures.

Évaluation retardée explicitement par une fonction :

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

Évaluation à la demande via un type `lazy` explicite, comme en 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
```

Évaluation à la demande par défaut, comme en Haskell

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

Objectifs du cours

Comment modéliser les structures de données infinies et raisonner dessus ?

- Approche ensembliste classique : plus grands points fixes.
- Approche théorie de la démonstration : arbres et dérivations infinies.
- Approche coalgébrique : ≪codonnées≫ définies par leurs observations.

Deux applications :

- La monade de partialité, pour faire de la récursion générale en théorie des types.
- La programmation réactive, vue comme programmation sur les flux infinis, ou pas…

I

Plus grands points fixes

Plus petit point fixe, plus grand point fixe

Soit A un ensemble et $F : \mathcal{P}(A) \to \mathcal{P}(A)$ une fonction croissante : si $X \subseteq Y$ alors $F(X) \subseteq F(Y)$.

Si $F(X) \subseteq X$ on dit que X est stable par F.
Si $X \subseteq F(X)$ on dit que X est cohérent pour F.

Théorème (Knaster, Tarski, Kleene)

L'ensemble $\{x \mid x = F(x)\}$ des points fixes de F est un treillis complet. En particulier,

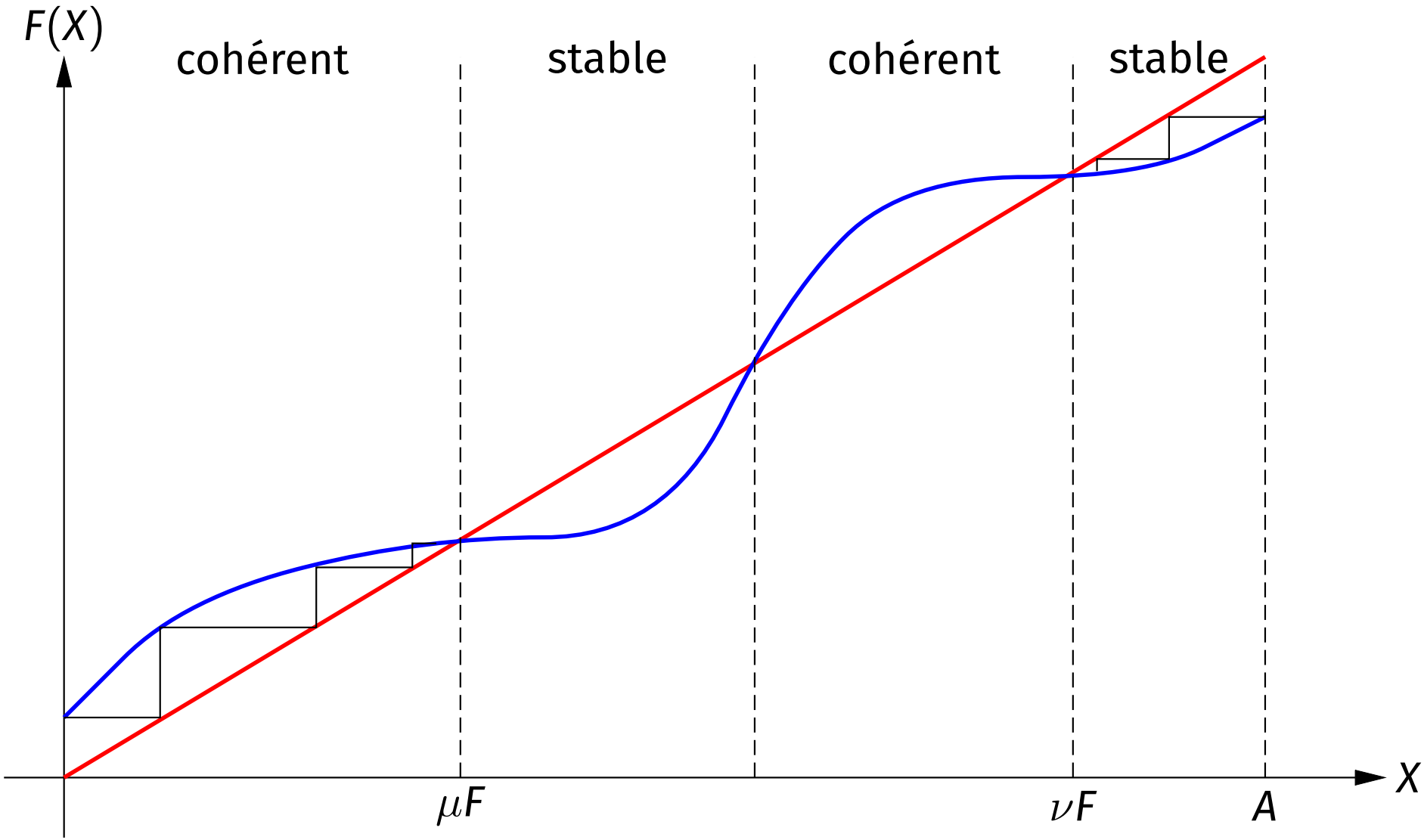
$$\mu F \stackrel{def}{=} \bigcap \{X \mid X \text{ stable par } F\} \quad \text{est le plus petit point fixe de } F$$

et c'est la limite de la suite croissante $\emptyset, F(\emptyset), F(F(\emptyset)), \ldots$

$$\nu F \stackrel{def}{=} \bigcup \{X \mid X \text{ cohérent pour } F\} \quad \text{est le plus grand point fixe de } F$$

et c'est la limite de la suite décroissante $A, F(A), F(F(A)), \ldots$

Les points fixes, graphiquement



Induction et co-induction

Principe d'induction : si X est stable par F, alors $\mu F \subseteq X$.

Autrement dit : pour montrer que $\forall a \in \mu F,\ a \in X$,
il suffit de montrer que $\forall a \in F(X),\ a \in X$.

Principe de coinduction : si X est cohérent pour F, alors $X \subseteq \nu F$.

Autrement dit : pour montrer que $\forall a \in X,\ a \in \nu F$,
il suffit de montrer que $\forall a \in X,\ a \in F(X)$.

Variante : pour montrer que $a \in \nu F$, il suffit de trouver un ensemble X
tel que $a \in X$ et tel que $\forall b \in X,\ b \in F(X)$.

Types inductifs, types coinductifs

On considère une déclaration de type algébrique, p.ex.

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

L'opérateur $F(X)$ est défini comme ≪les valeurs que l'on peut construire en appliquant une fois un constructeur et en prenant dans X les valeurs du type algébrique≫. Dans l'exemple :

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

Cet opérateur est croissant si toutes les occurrences du type récursif dans les types des constructeurs sont strictement positives.

Le type inductif défini par la déclaration est μF, le plus petit point fixe de l'opérateur F.

Le type coinductif défini par la déclaration est νF, le plus grand point fixe de l'opérateur F.

L'interprétation inductive

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

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

`nat` est le plus petit point fixe de F.

C'est aussi la limite de la suite $\emptyset$, $F(\emptyset) = \{\mathtt{O}\}$, $F(F(\emptyset)) = \{\mathtt{O}, \mathtt{S}(\mathtt{O})\}$, ...

Plus généralement, `nat` est l'ensemble de termes $X \stackrel{def}{=} \{\mathtt{S}^n(\mathtt{O}) \mid n \in \mathbb{N}\}$.

- Pour tout n, $\mathtt{S}^n(\mathtt{O}) \in F^{n+1}(\emptyset) \subseteq \mu F$.
- Réciproquement, l'ensemble X est stable par F, puisque $F(X) = \{\mathtt{O}\} \cup \{\mathtt{S}(\mathtt{S}^n(\mathtt{O})) \mid n \in \mathbb{N}\} = X$, donc, par le principe d'induction, $\mu F \subseteq X$.

L'interprétation coinductive

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

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

`conat` est le plus grand point fixe de F.

C'est aussi la limite de la suite U, $F(U) = \{\mathtt{O}\} \cup \{\mathtt{S}(x) \mid x \in U\}$, …, $F^n(U) = \{\mathtt{O}, \mathtt{S}(\mathtt{O}), \ldots, \mathtt{S}^{n-1}(\mathtt{O})\} \cup \{\mathtt{S}^n(x) \mid x \in U\}$.

Tout $\mathtt{S}^n(\mathtt{O})$ est dans `conat` puisque $\mu F \subseteq \nu F$.

Mais `conat` contient aussi le terme infini ω défini par $\omega = \mathtt{S}\,\omega$, car $\{\omega\} \subseteq F(\{\omega\}) = \{\mathtt{O}, \omega\}$ et donc, par le principe de coinduction, $\{\omega\} \subseteq \nu F$.

Prédicats inductifs, prédicats coinductifs

La même approche s'étend aux familles inductives, et en particulier aux prédicats définis par axiomes et règles d'inférence.

Exemple : le prédicat ≪être pair≫ sur les entiers.

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

À ce système de règles est associé un opérateur $F(X)$ qui calcule les faits que l'on peut déduire en appliquant une règle à partir des faits X :

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

L'interprétation inductive

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

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

even est défini comme μF, le plus petit point fixe de F.

Principe d'induction : pour montrer $\forall n,\ \texttt{even}(n) \Rightarrow P(n)$,
il suffit de montrer $P(\texttt{O})$ et $\forall n, P(n) \Rightarrow P(\texttt{S}(\texttt{S}(n)))$.

On peut donc caractériser even : (avec $f : \texttt{nat} \to \texttt{conat}$ canonique)

- $\forall n : \texttt{nat}, \texttt{even}(f(n+n))$ par induction sur $n : \texttt{nat}$
- $\forall m, \texttt{even}(m) \Rightarrow \exists n,\ m = f(n+n)$ par induction sur $\texttt{even}(m)$
- $\forall m, \texttt{even}(m) \Rightarrow m \neq \omega$ par induction sur $\texttt{even}(m)$

L'interprétation coinductive

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

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

coeven est défini comme νF, le plus grand point fixe de F.

Principe de coinduction : pour montrer $\forall n,\ P(n) \Rightarrow \texttt{coeven}(n)$,
il suffit de montrer $\forall n, P(n) \Rightarrow n = \texttt{O} \vee \exists m, n = \texttt{S}(\texttt{S}(m)) \wedge P(m)$.

On peut donc montrer :

- $\forall n, \texttt{even}(n) \Rightarrow \texttt{coeven}(n)$ par coinduction avec $P(n) = \texttt{even}(n)$.
- $\texttt{coeven}(\omega)$ par coinduction avec $P(n) = (n = \omega)$.

II

Arbres infinis, dérivations infinies

Interprétation en termes d'arbres

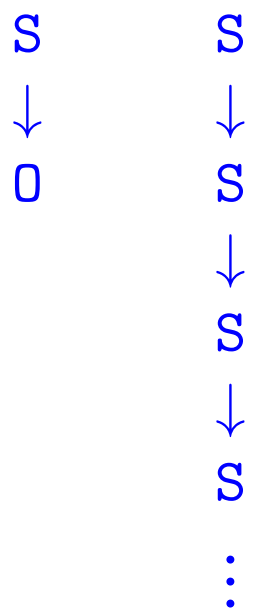
Une valeur d'un type (co-)inductif peut être vue comme un arbre avec
aux feuilles des constructeurs constants,
aux noeuds des constructeurs d'arité n, avec n = nombre de sous-arbres.

(On traite `C of nat -> t` comme un constructeur d'arité infinie.)

Les valeurs d'un type co-inductif sont des arbres finis ou infinis.

Les valeurs d'un type inductif sont des arbres dont toutes les branches sont finies.

Les valeurs d'un type inductif dont tous les constructeurs sont d'arité finie sont des arbres finis.



Interprétation en termes de dérivations

De même, un terme de preuve pour un prédicat (co-)inductif peut être vu comme un arbre de dérivation avec aux feuilles des instances d'axiomes, aux noeuds des instances de règles d'inférences ayant n prémisses, avec n = nombre de sous-arbres.

Un prédicat coinductif est vrai ss'il est conclusion d'une dérivation finie ou infinie.

Un prédicat inductif est vrai ss'il est conclusion d'une dérivation dont toutes les branches sont finies.

Un prédicat inductif dont toutes les règles ont un nombre fini de prémisses est vrai ss'il est conclusion d'une dérivation finie.

$$\dfrac{\dfrac{\texttt{even}(\texttt{O})}{\texttt{even}(\texttt{S}(\texttt{S}(\texttt{O})))}}{\texttt{even}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{S}(\texttt{O})))))}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\texttt{coeven}(\omega)}}{\texttt{coeven}(\omega)}}{\texttt{coeven}(\omega)}$$

Récursion structurelle, corécursion productive

Une fonction récursive $f : ind \to t$ sur un type inductif *ind* doit être structurelle : les appels récursifs $f(y)$ se font sur des sous-termes stricts y de l'argument.

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

Conséquence : le calcul de $f(n)$ termine toujours.

Récursion structurelle, corécursion productive

Une fonction récursive $f : ind \to t$ sur un type inductif *ind* doit être structurelle : les appels récursifs $f(y)$ se font sur des sous-termes stricts y de l'argument.

Une fonction corécursive $f : t \to coind$ renvoyant un type coinductif doit être productive : les appels récursifs $f(y)$ doivent être des sous-termes stricts du résultat.

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

Conséquence : le calcul du constructeur de tête de $f(x)$ termine toujours.

Exemples de définitions corécursives

L'entier ≪plus l'infini≫ :

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

L'addition de deux entiers éventuellement infinis :

```
CoFixpoint add (p q: conat) : conat :=
  match p with
  | O => q                  ✔ (* pas d'appel récursif *)
  | S p' => S (add p' q)    ✔ (* appel récursif gardé par S *)
```

La soustraction n'est pas définissable : `sub omega omega` divergerait avant de déterminer le constructeur de tête.

```
CoFixpoint sub (p q: conat) : conat :=
  match p, q with
  | O, _ => O                   ✔ (* pas d'appel récursif *)
  | _, O => p                   ✔ (* pas d'appel récursif *)
  | S p', S q' => sub p' q'     ✘ (* appel récursif non gardé *)
  end.
```

Exemples de démonstrations coinductives

Dans l'esprit de Curry-Howard, les démonstrations par coinduction sont des définitions corécursives de termes de preuve.

Par exemple, on définit le prédicat coinductif `infinite` par

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

Alors, la démonstration de `infinite omega` est, moralement,

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

Ce n'est pas tout à fait exact, car `omega` n'est pas convertible en `S omega` (voir transp. suivant), donc il faut appliquer explicitement l'égalité (démontrable) `omega_eq: omega = S omega`.

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

Règles de réduction

Tout comme les définitions récursives (`Fixpoint`), les définitions corécursives (`CoFixpoint`) ne peuvent pas être «déroulée» arbitrairement, sous peine de non-terminaison :

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

La règle de réduction utilisée par Coq : une définition corécursive $\nu x.\, a$ s'expanse en $a\{x \leftarrow \nu x.a\}$ seulement quand elle est argument d'un `match` :

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

À rapprocher de la règle pour les définitions récursives $\mu f.\, \lambda x.\, a$ qui s'expanse seulement quand elle est appliquée à un constructeur :

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

Égalités de déroulage

Même si elle n'est pas vraie par conversion, on peut montrer l'égalité `omega = S omega` par démonstration.

On montre d'abord une propriété d'extensionnalité sur les valeurs de type `conat` :

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

Dès lors, `unroll omega` est du type suivant, qui se réduit :

```
    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
```

Non-préservation du typage par réduction

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

On a les réductions et les typages suivants :

$$\begin{array}{rcl} \texttt{unroll omega} & : & \texttt{omega} = \texttt{S omega} \\ * \downarrow & & \not\downarrow \text{ (non convertibles)} \\ \texttt{eq\_refl omega} & : & \texttt{omega} = \texttt{omega} \end{array}$$

Ceci invalide la propriété de préservation du typage (*subject reduction*) : si $M : t$ et $M \to M'$ alors $M' : t$.

Cela ne veut pas dire que Coq + coinductifs est incohérent ; juste qu'une preuve de cohérence est plus difficile. (Cf. thèse de Giménez.)

III

Types coinductifs, récursion générale, et monade de partialité

Calculs partiels

(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* représente les calculs qui, s'ils terminent, renvoient une valeur de type *A*.

Le constructeur `later` matérialise une étape de calcul.

Le type `delay` étant coinductif, on peut avoir une infinité d'étapes de calcul, et donc un calcul qui ne termine jamais. Exemple :

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

Calculs partiels

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

On caractérise inductivement les calculs qui terminent, coinductivement ceux qui divergent :

```
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).
```

Récursion générale

On peut définir des fonctions récursives générales arbitraires, à résultats dans un type `delay`, à condition de protéger tous les appels récursifs par `later`.

```
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)).
```

On peut alors raisonner sur la terminaison ou la divergence de la fonction.

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

Un combinateur de point fixe

Soit $F : (A \to \texttt{delay}\ B) \to (A \to \texttt{delay}\ B)$. On peut construire une fonction $Y\ F : A \to \texttt{delay}\ B$ en itérant F à partir de $\lambda x.\,\texttt{bottom}$ et en prenant ≪le premier résultat défini≫ :

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

Sous certaines hypothèses sur F, on a l'équation de point fixe, c.à.d. l'équivalence entre $Y\ F\ a$ et $F\ (Y\ F)\ a$.

La monade de partialité

Le type `delay` est une monade, avec le constructeur `now` comme opération `ret`, et l'opération `bind` défini comme le séquencement de deux calculs.

```
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.
```

On a les propriétés attendues d'un séquencement, p.ex. `bind a f` diverge ssi `a` diverge ou `a` termine sur `v` et `f v` diverge.

Problèmes de productivité

À cause du critère syntaxique de productivité, le `bind` ainsi défini est souvent inutilisable à l'intérieur d'une définition coinductive.

Exemple : la fonction 91 de McCarthy,

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

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

Les appels corécursifs `M (n + 11)` et `M x` sont rejetés, car ils apparaissent sous `bind`, qui n'est pas un constructeur de `delay`.

Passer par la monade libre

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

On peut contourner le problème en présentant la monade comme un type coinductif ayant pour constructeurs les opérations de la monade de partialité : `ret`, `bind`, et `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.
```

Passer par la monade libre

La fonction 91 est alors bien productive :

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

Passer par la monade libre

Une description de calcul, de type `mon A`, se transforme en calcul, de type `delay A`, par la fonction suivante :

```
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.
```

On remarque l'utilisation des lois monadiques `bind-ret` et `bind-bind` afin de réduire tous les `bind` en `later(interp` a`)`

IV

Codonnées et comotifs

Types définis par observations

De même que les types inductifs sont finiment engendrés par leurs constructeurs, on peut dire que les types coinductifs sont finiment caractérisés par leurs projections, c.à.d. par les observations que l'on peut faire sur les valeurs de ces types.

Exemple : les flux ont deux projections,

$$\texttt{hd} : \texttt{stream}\ A \to A \qquad \texttt{tl} : \texttt{stream}\ A \to \texttt{stream}\ A$$

Un flux s est entièrement caractérisé par les résultats des observations $\texttt{hd}(s), \texttt{hd}(\texttt{tl}(s)), \ldots, \texttt{hd}(\texttt{tl}^n(s)), \ldots$

Remarque : une fonction $f : A \to B$ est elle aussi une ≪boîte noire≫ caractérisée par les résultats des applications (observations) $f\ a_1, \ldots, f\ a_n, \ldots$

Données et codonnées

Cela suggère de classer les types en

- Types de données : entiers, Booléens, $A \times B$, $A + B$, tous les types inductifs.
- Types de codonnées : fonctions $A \to B$, flux, tous les types coinductifs.

Les types inductifs se présentent comme des sommes étiquetées par des constructeurs; les types coinductifs comme des produits étiquetés par des projections (c.à.d. des enregistrements).

$$\texttt{list A} := \langle\ \texttt{nil} \mid \texttt{cons of A} \times \texttt{list A}\ \rangle$$
$$\texttt{stream A} := \{\ \texttt{hd: A; tl: stream A}\ \}$$

Définir des codonnées

Une codonnée s'utilise en appliquant des projections. Mais comment définir une codonnée ? Par des équations définissant ses réactions aux projections !

Exemples : le flux `from` n des entiers $n, n+1, n+2, \ldots$:

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

L'application d'une fonction f en chaque point d'un flux :

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

Définitions par équations et filtrage

Ce style de définitions par équations existe aussi en Haskell et Agda pour définir des fonctions récursives opérant sur des données inductives, p.ex.

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

Symétrie : dans le cas des données, il faut une équation par constructeur possible pour l'argument ; dans le cas des codonnées, il faut une équation par projection possible sur le résultat.

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

```
map f nil = nil
map f (cons h t) =
        cons (f h) (map f t)
```

Motifs et comotifs

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

Abel et al proposent d'unifier ces deux styles de définitions par équations à l'aide de motifs (*patterns*) et de comotifs (*copatterns*).

Motifs (= forme possible pour une donnée)

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

Comotifs (= observations possibles sur une codonnée)

$$\begin{array}{lll} q ::= \Box & & \text{l'objet en cours de définition} \\ \quad \mid q\ p & & \text{application de fonction} \\ \quad \mid q\ .\mathit{Proj} & & \text{projection} \end{array}$$

Une codonnée est alors une liste de cas (comotif, expression).
P.ex. $\lambda x.a$ est $[(\Box\ x, a)]$ et $\{\mathtt{hd} = a, \mathtt{tl} = b\}$ est $[(\Box.\mathtt{hd}, a); (\Box.\mathtt{tl}, b)]$.

Exemples de comotifs imbriqués

Les nombres de Fibonacci, inductifs :

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

Les nombres de Fibonacci, sous forme de flux :

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

Le flux $n, n-1, \ldots, 1, 0, N, N-1, \ldots, 0, N, \ldots$:

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

Règles de réduction

Chaque équation est interprétée comme une règle de réduction (de la gauche vers la droite) :

$$\texttt{cycle (S(S(S O))).tl.tl.hd} \to \texttt{cycle (S(S O)).tl.hd}$$
$$\to \texttt{cycle (S O).hd}$$
$$\to \texttt{S O}$$

Le problème de Coq (non-préservation du typage) est évité car il n'y a pas d'égalités entre une codonnée et son expansion.

P.ex. s et $\{\texttt{hd} = s.\texttt{hd}; \texttt{tl} = s.\texttt{tl}\}$ réagissent pareil aux projections, mais n'ont aucune raison d'être égaux.

Productivité et récurrence structurelle

Pour garantir la normalisation, des conditions syntaxiques simples semblent suffire :

- Productivité : chaque comotif doit commencer par une projection
- Récurrence structurelle : si on fait un appel récursif sur une variable x, elle doit apparaître dans le comotif sous un constructeur.

```
F x = F (S x) ✘      (F x).tl = F (S x) ✔      F (S x) = G(F x) ✔
```

Pour certains comotifs complexes, la productivité est moins évidente :

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

Agda utilise les types avec tailles (*sized types*) pour contrôler la productivité plus finement.

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

V

Programmation réactive et récursion gardée

La programmation réactive

Au sens très large : tout programme dont le but est de réagir, en calculant, aux événements qui lui proviennent de l'extérieur.

Exemple : une feuille de calcul en mode ≪recalcul automatique≫.

Pour la suite de ce cours : tout programme qui reçoit une suite de valeurs d'entrées $e_0, e_1, e_2, \ldots$ et calcule incrémentalement et ≪au même rythme≫ une suite de valeurs de sorties $s_0, s_1, s_2, \ldots$.

Programmation réactive = programmation sur des flux ?

Il est commode de représenter par des flux les suites d'entrées, de sorties, ou de résultats intermédiaires.

Cependant, un programme réactif n'est pas n'importe quelle fonction des flux dans les flux. Il faut que la fonction soit causale :
la n^{e} sortie s_n dépend uniquement des entrées passées $e_0, \ldots, e_n$ mais pas des entrées futures $e_{n+1}, e_{n+2}, \ldots$.

Exemples de fonctions causales (✔) et non causales (✘) :

✔ `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, un langage causal par construction

Un programme Lustre définit des flux $X_1, X_2, \ldots$ par un ensemble d'équations $X_i = E_i$ mutuellement récursives.

Le langage des expressions E_i est suffisamment restreint pour garantir la causalité. Il inclut :

- Des calculs point par point : `X + Y` est `zipWith (+) X Y`.
- Des opérateurs temporels qui se ramènent au `cons` des flux :

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

- Des opérations d'échantillonage sur des sous-horloges (représentées par des flux de Booléens).

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

Assurer la causalité par typage

Pour plus de flexibilité (fonctions d'ordre supérieur), on peut remplacer les conditions syntaxiques par des vérifications de types.

N. Krishnaswami et N. Benton (2011–2013) ajoutent aux types une modalité ≪plus tard≫ que nous noterons $\triangleright A$.

Intuitivement, les valeurs de type $\triangleright A$ sont des valeurs de type A qui ne doivent pas être utilisées tout de suite (au risque de violer la causalité), mais peuvent être utilisées à partir du prochain pas de temps.

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

Exemples de typage

(Dans le langage de Clouston et al, *The guarded lambda-calculus*, LMCS(12), 2016)

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

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

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

Le typage garantit la causalité et la productivité, alors que la définition

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

ne passe même pas le critère syntaxique de productivité de Coq.

Types récursifs gardés

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

On peut former le type récursif $\mu\alpha.\,\tau$ à condition que α soit gardé dans τ : toutes les occurrences de α sont ≪sous≫ une modalité $\triangleright$.

Exemples :

$\mu\alpha.\,A \times \triangleright\alpha$	flux (listes infinies)
$\mu\alpha.\,\mathtt{unit} + A \times \triangleright\alpha$	listes finies ou infinies
$\mu\alpha.\,\mathtt{bool} \times \triangleright\alpha \times \triangleright\alpha$	arbres binaires infinis
	(= automates déterministes)
$\mu\alpha.\mathtt{unit} + \alpha$ ✘	non gardé

Types récursifs gardés

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

Dans le ≪topos des arbres≫ (cf. les cours du 9 janvier 2019), ces types récursifs gardés s'interprètent comme des familles indexées par le temps de types non récursifs (et donc non coinductifs).

Exemple : le type des flux $\mu\alpha.\, A \times \triangleright\alpha$.

au temps 0 :	$\mathtt{unit}$
au temps 1 :	$A \times \mathtt{unit}$
au temps 2 :	$A \times A \times \mathtt{unit}$
	$\vdots$
au temps n :	$A \times \cdots A \times \mathtt{unit}$ (les listes de longueur n)

Exemple : $\mu\alpha.\, \mathtt{unit} + A \times \triangleright\alpha$ au temps n = listes de longueur $\leq n$.

Des types récursifs gardés aux types coinductifs

On retrouve les types coinductifs habituels, utilisables librement à tout instant, via une autre modalité : $\Box$, «pour toujours».

Sémantiquement, $\Box A$ est «l'objet constant» qui est la limite des interprétations de A au temps n quand $n \to \infty$. P.ex. :

$$\begin{aligned}
\Box(\mu\alpha.\, A \times \triangleright\alpha) &\approx \lim_{n\to\infty} \text{(listes de } A \text{ de longueur } n\text{)} \\
&\approx \text{listes infinies de } A \\
\Box(\mu\alpha.\, \mathtt{unit} + A \times \triangleright\alpha) &\approx \lim_{n\to\infty} \text{(listes de } A \text{ de longueur} \leq n\text{)} \\
&\approx \text{listes finies ou infinies de } A
\end{aligned}$$

Restriction : A doit être clos dans $\Box A$ (interdit : $\mu\alpha \ldots \Box \triangleright \alpha$ p.ex.)

Des types récursifs gardés aux types coinductifs

La modalité $\Box$ se comporte globalement comme la modalité «nécessairement» de la logique S4 intuitionniste.

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

C dénote un type constant dans le temps : $C ::= \mathtt{nat} \mid C \to C \mid \Box A$.

Les opérateurs $\mathtt{box}$ et $\mathtt{unbox}$ permettent de définir des fonctions non causales sur des types coinductifs, p.ex.

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

De la récursion gardée à la corécursion

Il n'y a pas d'opérateur de point fixe correspondant à la corécursion.
Il faut faire des récursions gardées puis appliquer `box`, ce qui garantit la productivité.

Exemple : la fonction «prendre un élément sur deux». On définit d'abord la fonction qui produit un flux récursif gardé

$$F : \Box(\texttt{stream}\ A) \to \texttt{stream}\ A$$
$$= \texttt{fix}(\lambda f.\ \lambda s.\ \texttt{cons}(\texttt{hd}_c(s))(g \circledast \texttt{next}(\texttt{tl}_c(\texttt{tl}_c(s)))))$$

puis, avec `box`, la fonction qui produit un flux coinductif :

$$G : \Box(\texttt{stream}\ A) \to \Box(\texttt{stream}\ A)$$
$$= \lambda s.\ \texttt{box}(F(s))$$

VI

Conclusion

Heureux comme Sisyphe ?

Avec ces techniques de définitions et de preuves coinductives, on peut ≪regarder l'infini droit dans les yeux≫ et programmer et raisonner directement sur les objets infinis.

Ceci y compris en théorie des types et dans des systèmes comme Agda et Coq.

Cependant, les conditions de productivité sont très strictes, et raisonner par coinduction reste difficile.
$\Rightarrow$ des principes de preuves plus souples comme la coinduction paramétrée (PACO, C. K. Hur et al, 2013) et la coinduction *all the way up* (D. Pous, 2016).

Comme dans l'exemple de la programmation réactive, les approches ≪finitistes≫ (step-indexing, approximations finies) sont parfois préférables aux approches coinductives.

VII

Bibliographie

Bibliographie

Un tutoriel sur la coinduction ensembliste, avec application au sous-typage :

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

La coinduction en Coq :

- Y. Bertot et P. Castéran : *Le Coq'Art*, chapitre 14, `http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`, traduction française du livre *Interactive Theorem Proving and Program Development*.

L'approche par codonnées définies par leurs projections :

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

L'approche par récursion gardée :

- 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`