Sémantiques mécanisées, sixième cours

L'éternité c'est long…
Sémantiques de la divergence: théorie des domaines et approches coinductives

Xavier Leroy

2020-01-30

Collège de France, chaire de sciences du logiciel

Pourquoi s'intéresser aux programmes qui divergent ?

Dans des langages sans entrées/sorties (IMP, fonctionnel pur) :

- Aucun intérêt pratique.
- Utile pour la théorie : calculabilité, équivalences de programmes.

Dans la pratique de l'informatique :

- Bien des programmes utiles ne sont pas censés s'arrêter ! (noyaux, contrôle-commande, serveurs, …)
- Divergence réactive : calculs finis entre deux entrées-sorties.

Comment formaliser la divergence ?

De manière négative, ≪en creux≫ :

- Les programmes qui divergent sont ceux qui ne terminent pas (ni normalement, ni sur une erreur).

De manière positive, voire constructive : (parties 2 et 3)

- Des caractérisations co-inductives de la divergence.
  (La terminaison est fondamentalement inductive.)

En même temps que la terminaison : (parties 1 et 3)

- Un des buts de la sémantique dénotationnelle.
- Classiquement (théorie des domaines) ou constructivement (monade de partialité)

D’un interpréteur borné à une sémantique dénotationnelle

Un interpréteur de référence

Rappel du 1er cours : il n'est pas possible de définir la sémantique d'une commande IMP par une fonction

$$\text{état mémoire } \ll\text{avant}\gg \to \text{état mémoire } \ll\text{après}\gg$$

puisque cette fonction serait partielle (non-terminaison).

Nous pouvons cependant définir une approximation de cette fonction en bornant *a priori* la profondeur de récursion avec un paramètre `fuel` de type `nat`.

Un interpréteur de référence, borné

```
Fixpoint cinterp (fuel: nat) (c: com) (s: store)
                                          : option store :=
  ...
```

Un résultat `Some s'` signifie que `c` termine sur `s'` à coup sûr.

Un résultat `None` est non concluant : ou bien `c` diverge, ou bien il faut plus de fuel pour terminer le calcul de `c`.

Ces résultats forment une monade ($\approx$ monade d'erreurs) :

```
Definition ret {A: Type} (v: A) := Some v.

Definition bind {A B: Type} (x: option A) (f: A -> option B) :=
  match x with None => None | Some v => f v end.
```

L'interpréteur de référence, borné

```
Fixpoint cinterp (fuel: nat) (c: com) (s: store) : option store :=
  match fuel with
  | O => None
  | S n =>
      match c with
      | SKIP => Some s
      | ASSIGN x a => Some (update x (aeval a s) s)
      | SEQ c1 c2 => bind (cinterp n c1 s) (cinterp n c2)
      | IFTHENELSE b c1 c2 =>
          cinterp n (if beval b s then c1 else c2) s
      | WHILE b c1 =>
          if beval b s
          then bind (cinterp n c1 s) (cinterp n (WHILE b c1))
          else Some s
      end
  end.
```

L'ordre partiel sur les résultats

L'ordre $r \sqsubseteq r'$, lire : «r' est plus défini que r»

$$\texttt{None} \sqsubseteq r' \qquad\qquad \texttt{Some}(v) \sqsubseteq \texttt{Some}(v)$$

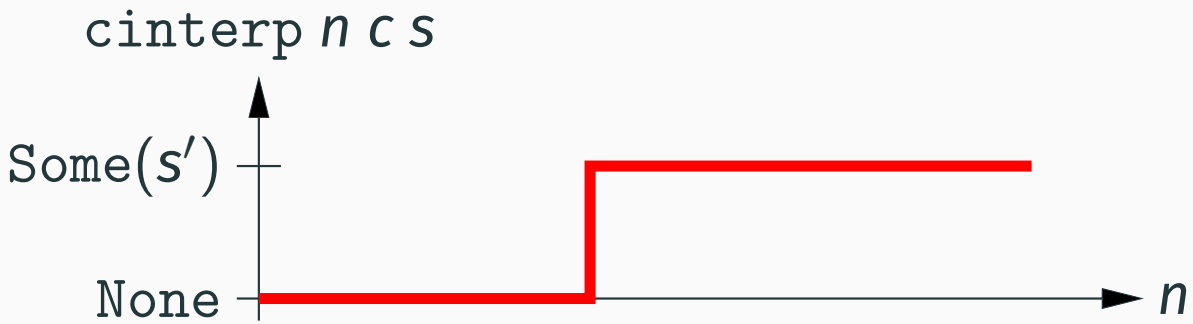
Une propriété cruciale : l'interpréteur borné est croissant.
(Plus de fuel $\Rightarrow$ résultat plus défini.)

$$i \leq j \quad \Rightarrow \quad \texttt{cinterp}\, i\, s\, c \sqsubseteq \texttt{cinterp}\, j\, s\, c$$

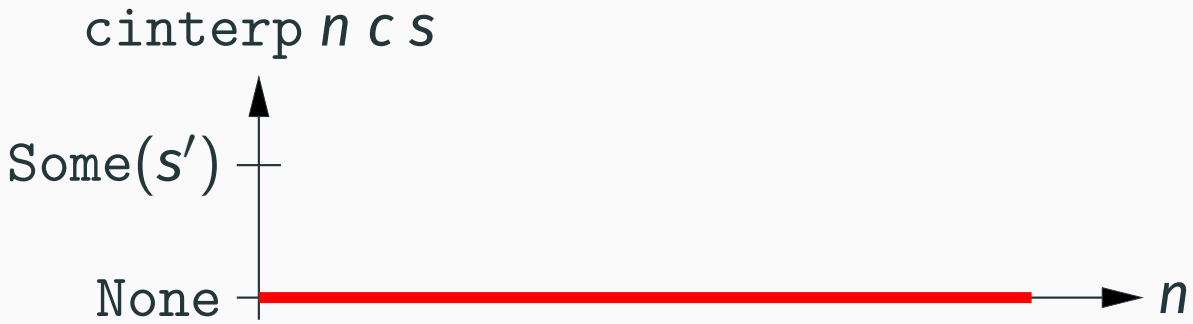
Vers une sémantique dénotationnelle

Que se passe-t'il si on fait ≪tendre le fuel vers l'infini≫ ?

Pour une commande c qui termine depuis l'état s :



Pour une commande c qui diverge depuis l'état s :



Limites

Toute suite croissante $f : \texttt{nat} \to \texttt{option}\ A$ admet une limite $\lim f$, qui n'est autre que sa borne supérieure, caractérisée par

$$\exists i, \forall j, i \leq j \Rightarrow f\ j = \lim f$$

Cette propriété n'est pas constructive : le développement Coq (fichier `Divergence.v`) utilise

- l'axiome du tiers exclu pour montrer l'existence de la limite (ou bien $\forall i,\ f\ i = \texttt{None}$ ou bien $\exists i,\ f\ i \neq \texttt{None}$);
- un axiome de description pour définir la limite $\lim f$ comme une fonction de la suite f.

Une sémantique dénotationnelle pour IMP

On définit la dénotation $[\![c]\!]$ d'une commande c comme la limite des exécutions par l'interpréteur de référence :

$$[\![c]\!]\, s \quad \stackrel{def}{=} \quad \lim(\texttt{fun}\ n \Rightarrow \texttt{cinterp}\ n\ c\ s)$$

Cette définition satisfait les équations attendues :

$$[\![\texttt{skip}]\!]\, s = \texttt{Some}(s)$$

$$[\![x := a]\!]\, s = \texttt{Some}(s\{x \leftarrow [\![a]\!]\, s\})$$

$$[\![c_1; c_2]\!]\, s = \texttt{bind}\ ([\![c_1]\!]\, s)\ [\![c_2]\!]$$

$$[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\, s = \begin{cases} [\![c_1]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{true} \\ [\![c_2]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

Une sémantique dénotationnelle pour IMP

Pour les boucles :

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\, s = \begin{cases} \texttt{bind}\ ([\![c]\!]\, s)\ [\![\texttt{while}\ b\ \texttt{do}\ c]\!] & \text{si } [\![b]\!]\, s = \texttt{true} \\ \texttt{Some}(s) & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

De plus, $[\![\texttt{while b do c}]\!]$ est la plus petite fonction $F : \texttt{store} \to \texttt{option store}$ qui satisfait l'équation

$$F\, s = \begin{cases} \texttt{bind}\ ([\![c]\!]\, s)\ F & \text{si } [\![b]\!]\, s = \texttt{true} \\ \texttt{Some}(s) & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

Exemple : $[\![\texttt{while true do}\ c]\!]\, s = \texttt{None}$ puisque la fonction `fun s => None` satisfait l'équation.

Équivalence entre sém. naturelle et sém. dénotationnelle

$$c/s \Downarrow s' \quad \Rightarrow \quad [\![c]\!]\, s = \mathtt{Some}(s')$$

$$[\![c]\!]\, s = \mathtt{Some}(s') \quad \Rightarrow \quad \mathtt{cinterp}\; n\; c\; s = \mathtt{Some}(s') \quad \Rightarrow \quad c/s \Downarrow s'$$

(Démonstrations dans le fichier `Divergence.v`.)

Pour aller plus loin : la théorie des domaines

Un domaine est un ensemble A muni d'un ordre partiel $\sqsubseteq$

$$x \sqsubseteq x \qquad \text{(réflexif)}$$

$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad \text{(transitif)}$$

$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \qquad \text{(antisymétrique)}$$

qui est ω-complet : toute suite croissante a une borne supérieure.

$$u_0 \sqsubseteq u_1 \sqsubseteq \cdots \sqsubseteq u_n \sqsubseteq \cdots \quad \Rightarrow \quad \sup u \in A$$

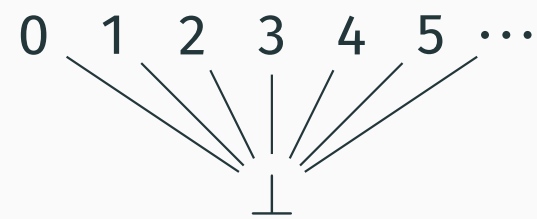
(Dans la littérature en anglais : ω-cpo ou juste cpo.)

Exemples de domaines

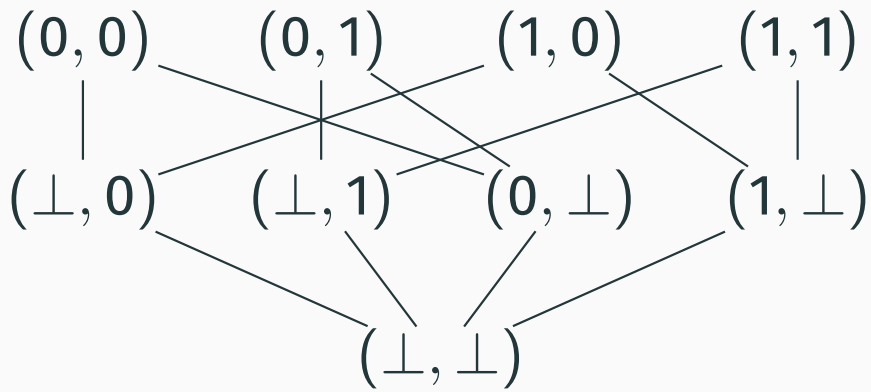
Domaine plat
(≈ valeur d'un type de base)

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \cdots$$

Domaine plat pointé
(≈ calcul d'un type de base)

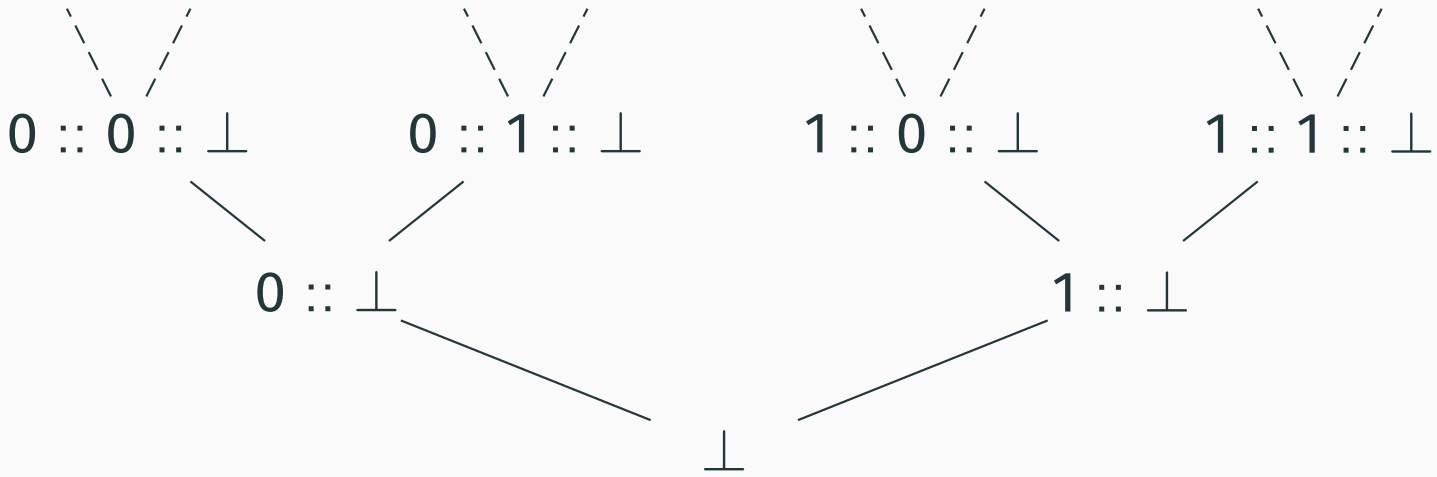


Paires paresseuses (≈ le type OCaml `bool lazy * bool lazy`)



Exemples de domaines

Flux de booléens :



Combinaisons de domaines :

- ≪Pointage≫ (ajout d'un élément minimal) : $D_\perp = D \uplus \{\perp\}$
- Produit $D_1 \times D_2$, somme $D_1 + D_2$.
- Fonctions continues $[D_1 \to D_2]$.

Fonctions continues

Une fonction $f : D_1 \to D_2$ est continue si elle préserve les bornes supérieures des suites croissantes :

$$\sup f(u_i) = f(\sup u_i)$$

Toute fonction continue est croissante. La réciproque est fausse.

Exemple : la fonction flux fini $\mapsto 0,$ flux infini $\mapsto 1$ est croissante mais discontinue. Elle n'est d'ailleurs pas calculable.

Théorème (de point fixe de Scott)

Si D est un domaine pointé, toute fonction $F : D \to D$ continue admet un plus petit point fixe $\mu F = \sup_n F^n(\bot)$.

Types récursifs

Pour interpréter les types de données récursifs comme des domaines, il faut résoudre des équations (à isomorphisme près) entre domaines, p.ex. :

- Listes d'entiers : $D_{list} \simeq \{nil\} + Nat \times D_{list}$
- Lambda-termes purs : $D_\infty \simeq [D_\infty \to D_\infty]$

C'est possible à condition de se placer dans des domaines algébriques (domaines de Scott), où tout élément est limite d'une suite d'éléments compacts ($\approx$ finis).

(Voir les notes de cours de Plotkin citées dans la bibliographie.)

Prédicats coinductifs et sémantique naturelle de la divergence

Prédicats définis par axiomes et règles d'inférence

$$P(\texttt{skip}, s) \qquad \frac{c/s \to c'/s' \qquad P(c', s')}{P(c, s)}$$

Jusqu'ici nous avons toujours interprété une telle définition de prédicat de manière inductive :

- comme un plus petit point fixe d'un opérateur ;
- en termes de dérivations finies.

Une autre interprétation existe, l'interprétation coinductive :

- comme un plus grand point fixe d'un opérateur ;
- en termes de dérivations infinies.

Opérateur associé à une définition

$$P(\texttt{skip}, s) \qquad \frac{c/s \to c'/s' \qquad P(c', s')}{P(c, s)}$$

À la définition ci-dessus est associé un opérateur

$$F(X) = \{(\texttt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

Intuitivement : $F(X)$ est l'ensemble des faits que l'on peut déduire en supposant les faits X vrais et en appliquant un axiome ou une règle d'inférence.

L'opérateur F est croissant, donc il admet un plus petit point fixe et un plus grand point fixe.

Plus petit point fixe

$$F(X) = \{(\texttt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

Le plus petit point fixe est

$$\mu F \stackrel{def}{=} \bigcap \{X \mid F(X) \subseteq X\}$$

C'est aussi la limite de la suite croissante $\emptyset, F(\emptyset), \ldots, F^n(\emptyset), \ldots$

Dans l'exemple, $F^n(\emptyset)$ est l'ensemble des (c, s) qui se réduisent en $\texttt{skip}$ en au plus n réductions. Donc, μF est l'ensemble des (c, s) qui terminent ($c/s \stackrel{*}{\to} \texttt{skip}/s'$).

Dérivations finies

Dérivation = arbre avec axiomes aux feuilles, règles d'inférence aux noeuds.

L'interprétation inductive μF correspond aux faits qui sont conclusion d'un arbre de dérivation dont toutes les branches sont finies.

(Si toutes les règles ont un nombre fini de prémisses, ce sont les arbres finis.)

Dérivations finies

Un exemple de dérivation finie :

$$\dfrac{c_1/s_1 \to c_2/s_2 \qquad \dfrac{c_2/s_2 \to c_3/s_3 \qquad \dfrac{\dfrac{c_n/s_n \to \texttt{skip}/s_{n+1}}{\vdots}}{P(c_3, s_3)}}{P(c_2, s_2)}}{P(c_1, s_1)}$$

$P(c, s)$ est dérivable par un arbre de hauteur n si et seulement si c/s se réduit en `skip` en n étapes.

Plus grand point fixe

$$F(X) = \{(\texttt{skip}, s)\} \cup \{(c, s) \mid c/s \to c'/s' \wedge (c', s') \in X\}$$

Le plus grand point fixe est

$$\nu F \stackrel{def}{=} \bigcup \{X \mid X \subseteq F(X)\}$$

C'est la limite de la suite décroissante $U, F(U), \ldots, F^n(U), \ldots$ où U est l'univers de toutes les paires (c, s).

Dans l'exemple, νF se compose de

- tous les (c, s) qui terminent : $c/s \stackrel{*}{\to} \texttt{skip}/s'$
- tous les (c, s) qui divergent : $c/s \stackrel{*}{\to} c_n/s_n \to \cdots$

Dérivations infinies

L'interprétation coinductive νF correspond aux faits qui sont conclusion d'un arbre de dérivation fini ou infini.

Un exemple de dérivation infinie :

$$\dfrac{c_1/s_1 \to c_2/s_2 \qquad \dfrac{c_2/s_2 \to c_3/s_3 \qquad \dfrac{\dfrac{c_n/s_n \to c_{n+1}/s_{n+1} \quad \vdots}{\vdots}}{P(c_3, s_3)}}{P(c_2, s_2)}}{P(c_1, s_1)}$$

La divergence, co-inductivement

$$\frac{c/s \to c'/s' \qquad \mathtt{div}(c', s')}{\mathtt{div}(c, s)}$$

Interprétation inductive : faux partout ! (pas d'axiome…)

Interprétation coinductive (double barre horizontale) : correspond à l'existence d'une suite infinie de réductions.

En Coq :

```
CoInductive div: com * state -> Prop :=
  | div_intro: forall c s c' s',
      red (c, s) (c', s') -> div (c', s') ->
      div (c, s).
```

Principe de coinduction

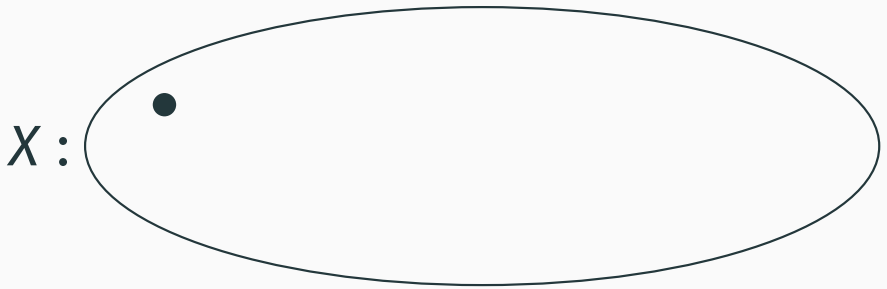
Par définition du plus grand point fixe $\nu F = \cup\{X \mid X \subseteq F(X)\}$, tout X tel que $X \subseteq F(X)$ est inclus dans νF.

Donc : si on a un prédicat $X : \texttt{com} * \texttt{store} \to \texttt{Prop}$ tel que

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

alors $X\ (c, s)$ implique $\texttt{div}\ (c, s)$.

Graphiquement :



Principe de coinduction

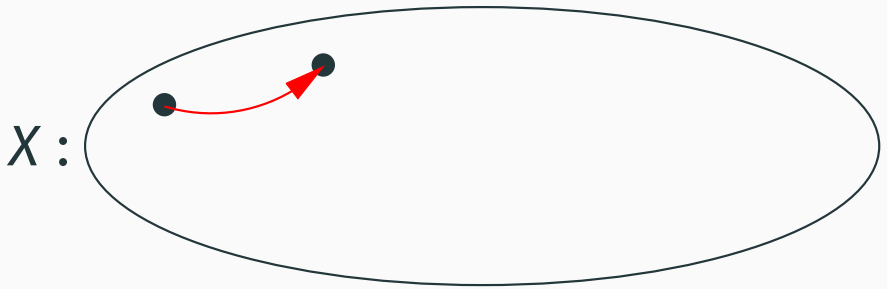
Par définition du plus grand point fixe $\nu F = \cup\{X \mid X \subseteq F(X)\}$, tout X tel que $X \subseteq F(X)$ est inclus dans νF.

Donc : si on a un prédicat X : $\texttt{com} * \texttt{store} \to \texttt{Prop}$ tel que

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

alors $X\ (c, s)$ implique $\texttt{div}\ (c, s)$.

Graphiquement :



Principe de coinduction

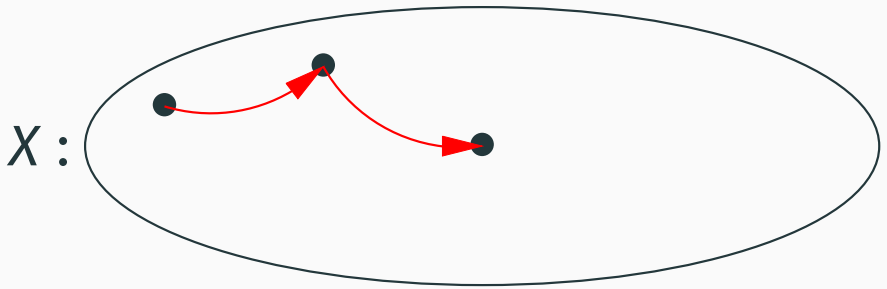
Par définition du plus grand point fixe $\nu F = \cup\{X \mid X \subseteq F(X)\}$, tout X tel que $X \subseteq F(X)$ est inclus dans νF.

Donc : si on a un prédicat X : $\texttt{com} * \texttt{store} \to \texttt{Prop}$ tel que

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

alors $X\ (c, s)$ implique $\texttt{div}\ (c, s)$.

Graphiquement :



Principe de coinduction

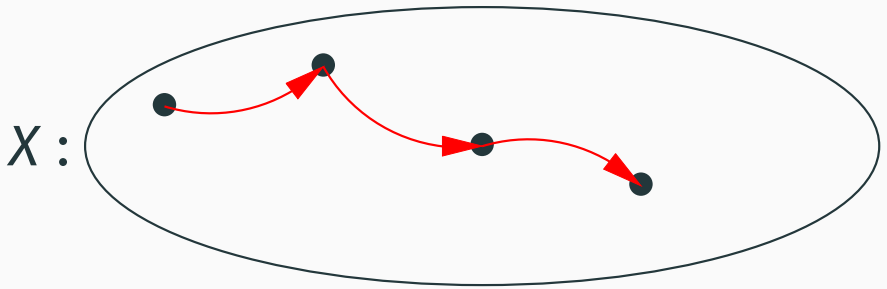
Par définition du plus grand point fixe $\nu F = \cup\{X \mid X \subseteq F(X)\}$, tout X tel que $X \subseteq F(X)$ est inclus dans νF.

Donc : si on a un prédicat X : $\texttt{com} * \texttt{store} \to \texttt{Prop}$ tel que

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

alors $X\ (c, s)$ implique $\texttt{div}\ (c, s)$.

Graphiquement :



Principe de coinduction

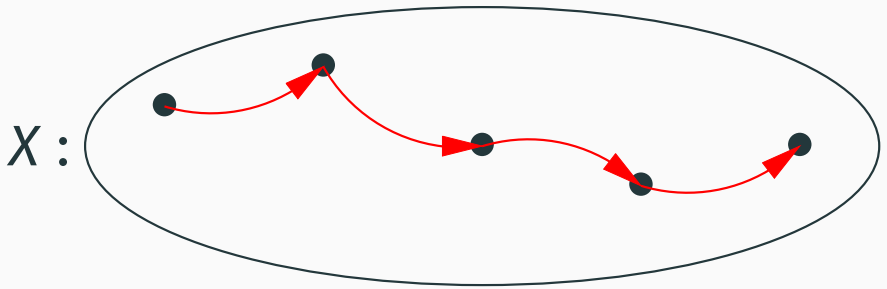
Par définition du plus grand point fixe $\nu F = \cup\{X \mid X \subseteq F(X)\}$, tout X tel que $X \subseteq F(X)$ est inclus dans νF.

Donc : si on a un prédicat X : $\texttt{com} * \texttt{store} \to \texttt{Prop}$ tel que

$$\forall c, \forall s,\ X\ (c, s) \Rightarrow \exists c', \exists s',\ c/s \to c'/s' \wedge X\ (c', s')$$

alors $X\ (c, s)$ implique $\texttt{div}\ (c, s)$.

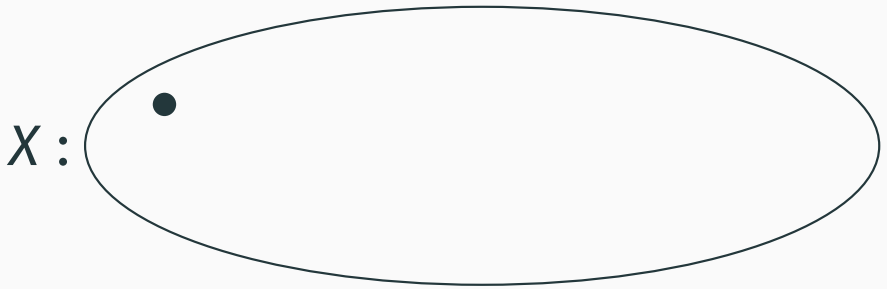
Graphiquement :



Principe de coinduction

De ce principe de coinduction on peut dériver un second principe, très utile, avec $\xrightarrow{+}$ (une ou plusieurs réductions) au lieu de $\rightarrow$ (une réduction) :

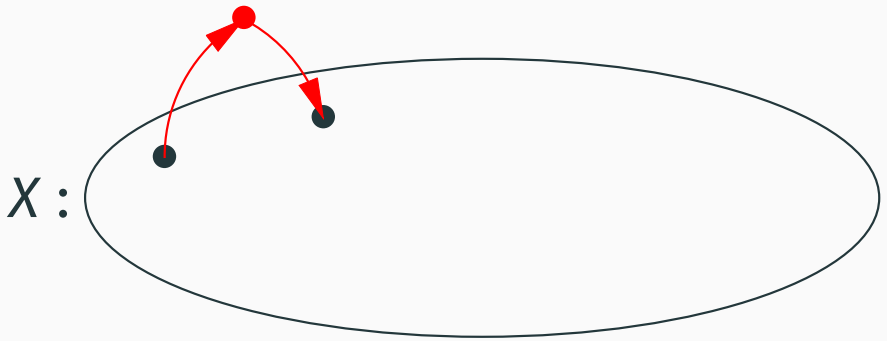
Si $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
alors $X\ c\ s$ implique $\texttt{div}\ c\ s$.



Principe de coinduction

De ce principe de coinduction on peut dériver un second principe, très utile, avec $\overset{+}{\to}$ (une ou plusieurs réductions) au lieu de $\to$ (une réduction) :

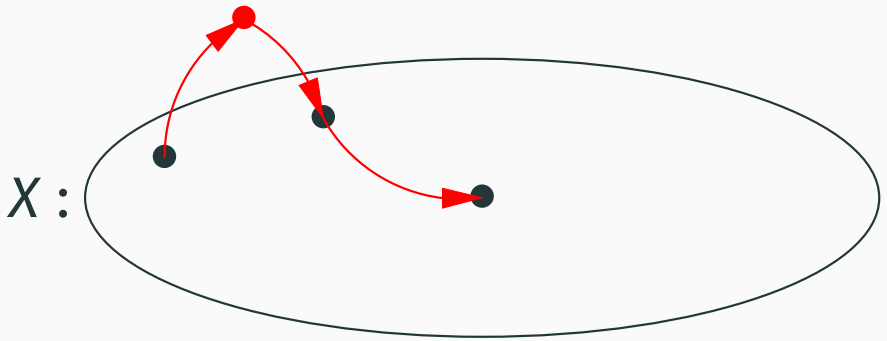
Si $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \overset{+}{\to} c'/s' \wedge X\ c'\ s'$,
alors $X\ c\ s$ implique $\texttt{div}\ c\ s$.



Principe de coinduction

De ce principe de coinduction on peut dériver un second principe, très utile, avec $\xrightarrow{+}$ (une ou plusieurs réductions) au lieu de $\rightarrow$ (une réduction) :

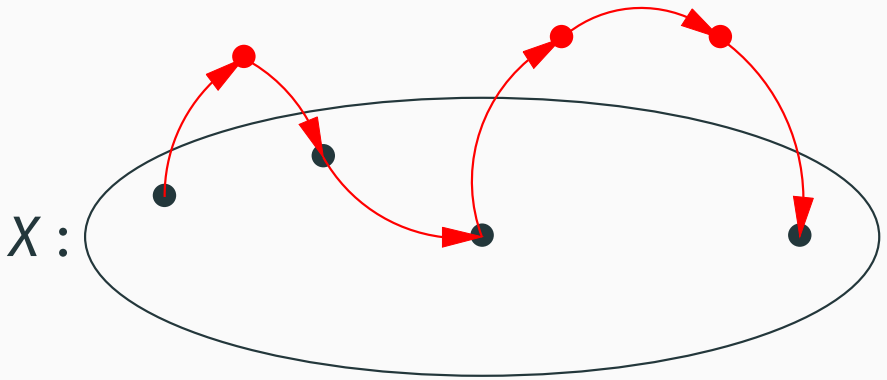
Si $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
alors $X\ c\ s$ implique $\texttt{div}\ c\ s$.



Principe de coinduction

De ce principe de coinduction on peut dériver un second principe, très utile, avec $\xrightarrow{+}$ (une ou plusieurs réductions) au lieu de $\rightarrow$ (une réduction) :

Si $\forall c, \forall s,\ X\ c\ s \Rightarrow \exists c', \exists s',\ c/s \xrightarrow{+} c'/s' \wedge X\ c'\ s'$,
alors $X\ c\ s$ implique $\texttt{div}\ c\ s$.



Retour sur la sémantique naturelle

Au 1er cours, nous avons introduit la sémantique naturelle comme moyen de structurer les réductions vers `skip`.

Par exemple, si la commande $c; c'$ termine, sa suite de réductions a nécessairement la forme suivante :

$$\begin{aligned}(c; c')/s \to (c_1; c')/s_1 \to \cdots \to (\texttt{skip}; c')/s' \\ \to c'/s' \to \cdots \to \texttt{skip}/s''\end{aligned}$$

Cette forme est reflétée par la règle de la sémantique naturelle pour la séquence :

$$\frac{c/s \Downarrow s' \qquad c'/s' \Downarrow s''}{c; c'/s \Downarrow s''}$$

Une structure pour les suites infinies de réductions

De même, si la commande $c; c'$ diverge, sa suite infinie de réductions est nécessairement d'une des deux formes suivantes :

$$(c; c')/s \to \cdots \to (c_n; c')/s_n \to \cdots \tag{1}$$

$$(c; c')/s \overset{*}{\to} (\texttt{skip}; c')/s' \to c'/s' \to \cdots c'_n/s'_n \to \cdots \tag{2}$$

Dans le cas (1) : c diverge, c' ne s'exécute pas.
Dans le cas (2) : c termine, puis c' diverge.

Essayons de refléter cette structure dans des règles pour un prédicat $c/s \Uparrow$, «la commande c diverge depuis l'état s».

La sémantique naturelle pour la divergence

$$\frac{c_1/s \Uparrow}{c_1; c_2/s \Uparrow} \qquad \frac{c_1/s \Downarrow s' \quad c_2/s' \Uparrow}{c_1; c_2/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \quad c_1/s \Uparrow}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)/s \Uparrow} \qquad \frac{[\![b]\!]\, s = \texttt{false} \quad c_2/s \Uparrow}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \quad c/s \Uparrow}{(\texttt{while } b \texttt{ do } c)/s \Uparrow}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \quad c/s \Downarrow s' \quad (\texttt{while } b \texttt{ do } c)/s' \Uparrow}{(\texttt{while } b \texttt{ do } c)/s \Uparrow}$$

Un exemple de divergence

La boucle $c \stackrel{def}{=} \texttt{while true do x} := \texttt{x} + 1$ diverge.

$$\dfrac{x := x + 1/s_0 \Downarrow s_1 \qquad \dfrac{x := x + 1/s_1 \Downarrow s_2 \qquad \dfrac{x := x + 1/s_2 \Downarrow s_3 \qquad \vdots}{c/s_2 \Uparrow}}{c/s_1 \Uparrow}}{c/s_0 \Uparrow}$$

(Avec $s_i = s_0[x \leftarrow s_0(x) + i]$.)

Équivalence avec la sémantique à réductions

Théorème

Si $c/s \Uparrow$, alors c/s se réduit infiniment.

Démonstration (constructive).

On montre $c/s \Uparrow \Rightarrow \exists c', \exists s', c/s \xrightarrow{+} c'/s' \wedge c'/s' \Uparrow$.
On conclut par le second principe de coinduction appliqué à l'ensemble $X = \{(c, s) \mid c/s \Uparrow\}$. □

Théorème

Si c/s se réduit infiniment, alors $c/s \Uparrow$.

Démonstration (classique).

Par coinduction et par examen de la forme des suites de réductions.
On utilise le tiers exclu : ou bien c/s se réduit finiment en `skip`, ou bien c/s se réduit infiniment. □

Application à la vérification de compilateurs

Au 2e cours, nous avions utilisé la sémantique naturelle pour montrer la correction du code compilé pour une commande IMP qui termine :

```
Lemma compile_com_correct_terminating:
  forall s c s', cexec s c s' ->
  forall C pc σ, code_at C pc (compile_com c) ->
  transitions C
      (pc, σ, s)
      (pc + codelen (compile_com c), σ, s').
```

Une récurrence sur la dérivation de `cexec s c s'` nous avait donné une démonstration assez simple.

Paradis perdu : la démonstration ne s'étendait pas aux commandes qui divergent.

Application à la vérification de compilateurs

Paradis retrouvé : la sémantique naturelle coinductive nous permet une démonstration assez simple de la correction du compilateur pour les commandes qui divergent.

On considère l'ensemble des configurations de la machine qui correspondent à des commandes qui divergent :

$$X \stackrel{def}{=} \{(pc, \sigma, s) \mid \exists c,\ c/s \Uparrow\ \wedge\ \texttt{code\_at}\ C\ pc\ (\texttt{compile\_com}\ c)\}$$

On montre que cet ensemble est ≪productif≫ :

$$\forall (pc, \sigma, s) \in X,\ \exists (pc', \sigma', s') \in X,\ (pc, \sigma, s) \xrightarrow{+} (pc', \sigma', s')$$

On conclut que, démarrée dans tout état appartenant à X, la machine effectue une infinité de transitions.

Monade de partialité et interpréteur de référence coinductif

Calculs partiels en théorie des types

(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.

```
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`.

✘
```
Fixpoint remainder (a b: nat) : nat :=
  if a <? b then a else remainder (a - b) b.
```

✘
```
CoFixpoint remainder (a b: nat) : delay nat :=
  if a <? b then now a else remainder (a - b) b.
```

✔
```
CoFixpoint remainder (a b: nat) : delay nat :=
  if a <? b then now a else later (remainder (a - b) b).
```

Récursion et corécursion en Coq

Définition récursive de fonction (`Fixpoint`) :

- L'argument est d'un type inductif.
- Condition de garde : $f\ x$ peut appeler récursivement $f\ y$ si l'argument y est un sous-terme strict de l'argument x.

Définition corécursive de fonction (`CoFixpoint`) :

- Le résultat est d'un type coinductif.
- Condition de productivité : $f\ x$ peut appeler récursivement $f\ y$ si le résultat $f\ y$ est un sous-terme strict de $f\ x$.
  ($f\ x$ est $f\ y$ enveloppé dans un ou plusieurs constructeurs.)

Récursion générale

```
CoFixpoint remainder (a b: nat) : delay nat :=
    if a <? b then now a else later (remainder (a - b) b).
```

On peut raisonner sur la terminaison ou la divergence de la fonction après l'avoir définie.

```
Theorem remainder_Euclid:
  forall a b, b > 0 ->
  exists q r, terminates (remainder a b) r ∧ r < b ∧ a = b*q+r.

Theorem remainder_divergence:
  forall a, diverges (remainder a 0).
```

Équivalence observationnelle

Une définition constructive de l'équi-terminaison :

```
CoInductive equi {A: Type} : delay A -> delay A -> Prop :=
  | equi_terminates: forall x y v,
      terminates x v -> terminates y v -> equi x y
  | equi_later: forall x y,
      equi x y -> equi (later x) (later y).
```

En logique classique, c'est équivalent à

$(\exists v,\ \texttt{terminates}\ x\ v \wedge \texttt{terminates}\ y\ v) \vee (\texttt{diverges}\ x \wedge \texttt{diverges}\ y)$

mais en logique constructive c'est plus fort. (Pas besoin de ≪deviner à l'avance≫ si les 2 calculs terminent ou divergent.)

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.

La monade de partialité

Les trois lois monadiques sont satisfaites à équivalence observationnelle près (`equi`, notée $\approx$ par la suite) :

$$\texttt{bind}\ (\texttt{now}\ v)\ f \approx f\ v$$

$$\texttt{bind}\ a\ \texttt{now} \approx a$$

$$\texttt{bind}\ (\texttt{bind}\ a\ f)\ g \approx \texttt{bind}\ a\ (\texttt{fun}\ x \Rightarrow \texttt{bind}\ (f\ x)\ g)$$

De plus,

$$\texttt{bind}\ a\ f \approx \texttt{bind}\ a'\ f' \text{ si } a \approx a' \text{ et } \forall x, f\ x \approx f'\ x$$

Un interpréteur dans la monade de partialité

À l'aide de la monade de partialité, essayons d'écrire un interpréteur récursif général pour IMP.

```
CoFixpoint cinterp (c: com) (s: store) : delay store :=
  match c with
  | SKIP => now s
  | ASSIGN x a => now (update x (aeval a s) s)
  | SEQ c1 c2 => bind (cinterp c1 s) (cinterp c2)
  | IFTHENELSE b c1 c2 =>
      later (cinterp (if beval b s then c1 else c2) s)
  | WHILE b c =>
      if beval b s then bind (cinterp c s) (cinterp (WHILE b c))
                   else ret s
  end.
```

Problème : la définition n'est pas productive !

Passer par la monade libre

(Une application de la technique de 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
  | Later: forall {A: Type}, mon A -> mon A
  | Bind: forall {A B: Type}, mon A -> (A -> mon B) -> mon B
```

En d'autres termes : la monade libre (plus `later`).

En d'autres termes : une syntaxe abstraite pour le métalangage monadique de Moggi (avec en plus `later`).

Un interpréteur dans la monade libre

```
CoFixpoint cinterp (c: com) (s: store) : mon store :=
  match c with
  | SKIP => Ret s
  | ASSIGN x a => Ret (update x (aeval a s) s)
  | SEQ c1 c2 => Bind (cinterp c1 s) (cinterp c2)
  | IFTHENELSE b c1 c2 =>
      Later (cinterp (if beval b s then c1 else c2) s)
  | WHILE b c =>
      if beval b s then Bind (cinterp c s) (cinterp (WHILE b c))
                   else Ret s
  end.
```

Cette définition est productive !

Interpréter la monade libre

Un terme de type `mon A` décrit un calcul de type `delay A`.

```
CoFixpoint run {A: Type} (m: mon A) : delay A :=
  match m with
  | Ret v => now v
  | Later m => later (run m)
  | Bind (Ret v) f => later (run (f v))
  | Bind (Later m) f => later (run (Bind m f))
  | Bind (Bind m f) g =>
      later (run (Bind m (fun x => Bind (f x) g)))
  end.
```

On note l'utilisation ≪au vol≫ de la première et de la troisième loi des monades.

Les surprises de la productivité

Le critère de productivité est une approximation syntaxique.

Il n’est pas compositionnel.

La fonction `run` comme sémantique dénotationnelle

On peut voir la fonction `run` comme une sémantique dénotationnelle du métalangage monadique :

`run` : syntaxe (type `mon A`) $\to$ signification (type `delay A`).

Équivalences satisfaites par `run` :

$$\texttt{run}\ (\texttt{Later}\ m) \approx \texttt{later}(\texttt{run}\ m) \quad \text{(dénotation de Later)}$$

$$\texttt{run}\ (\texttt{Bind}\ m\ f) \approx \texttt{bind}\ (\texttt{run}\ m)\ (\texttt{fun}\ x \Rightarrow \texttt{run}\ (f\ x)) \quad \text{(dénotation de Bind)}$$

$$\texttt{run}\ (\texttt{Bind}\ (\texttt{Ret}\ v)\ f) \approx \texttt{run}\ (f\ v) \quad \text{(1re loi monadique)}$$

$$\texttt{run}\ (\texttt{Bind}\ m\ \texttt{Ret}) \approx \texttt{run}\ m \quad \text{(2e loi monadique)}$$

$$\texttt{run}\ (\texttt{Bind}\ (\texttt{Bind}\ m\ f)\ g) \approx \texttt{run}\ (\texttt{Bind}\ m\ (\texttt{fun}\ x \Rightarrow \texttt{Bind}\ (f\ x)\ g)) \quad \text{(3eloi monadique)}$$

L'interpréteur coinductif comme sémantique dénotationnelle

On définit la dénotation d'une commande c comme

$$[\![c]\!]\, s \stackrel{def}{=} \texttt{run}\,(\texttt{cinterp}\; c\; s) \qquad (\text{de type } \texttt{delay}\ \ \texttt{store})$$

Cette définition vérifie les équations attendues :

$$[\![\texttt{skip}]\!]\, s \approx \texttt{now}(s)$$

$$[\![x := a]\!]\, s \approx \texttt{now}(s\{x \leftarrow [\![a]\!]\, s\})$$

$$[\![c_1; c_2]\!]\, s \approx \texttt{bind}\,([\![c_1]\!]\, s)\,[\![c_2]\!]$$

$$[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\, s \approx \begin{cases} [\![c_1]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{true} \\ [\![c_2]\!]\, s & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\, s \approx \begin{cases} \texttt{bind}\,([\![c]\!]\, s)\,[\![\texttt{while}\ b\ \texttt{do}\ c]\!] & \text{si } [\![b]\!]\, s = \texttt{true} \\ \texttt{now}(s) & \text{si } [\![b]\!]\, s = \texttt{false} \end{cases}$$

Point d’étape

Point d'étape

La coinduction comme outil fondamental pour penser la divergence, du trivial (suites infinies de réductions) au subtil (sémantiques naturelles de la divergence, monade de partialité).

Les sémantiques dénotationnelles ont besoin d'une structure mathématique adaptée. Classiquement ce sont les domaines ; constructivement ce pourrait être le type quotient $\texttt{delay}\ A/\approx$.

Pour le moment, les approches esquissées dans ce cours ne passent pas aussi bien ≪à l'échelle≫ que les sémantiques à transitions.

Bibliographie

Bibliographie

La théorie des domaines et sa mécanisation :

- G. Plotkin, *Domains*, 1983, `http://homepages.inf.ed.ac.uk/gdp/publications/Domains_a4.ps`
- N. Benton, A. Kennedy, C. Varming, *Some Domain Theory and Denotational Semantics in Coq*, TPHOLs 2009.

Les sémantiques naturelles coinductives :

- X. Leroy et H. Grall, *Coinductive big-step operational semantics*, Inf&Comp 207(2), 2009.

Monade de partialité et sémantiques dénotationnelles :

- V. Capretta, *General recursion via coinductive types*, LMCS(1), 2005.
- N. A. Danielsson, *Operational Semantics Using the Partiality Monad*, ICFP 2012,