# Analyse des Programmes et Sémantique (5)

Prof[r] Jacques Malenfant

(cours de J.Malenfant modifié et enseigné en 2013 par Basile Starynkévitch)

janvier-avril 2013

MI030 - APS

Cours (en M1) du Professeur Jacques Malenfant
http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/ Professeur en informatique au Laboratoire d'Informatique de Paris 6

Enseigné en 2013 par Basile Starynkevitch
http://starynkevitch.net/Basile/
basile@starynkevitch.net & basile.starynkevitch@cea.fr
ingénieur chercheur au CEA, LIST - http://www-list.cea.fr/
travaille sur gcc-melt.org

## ♠ Nota Bene

Les transparents à fond rose (pages numérotées ♠) et les mots ♠ signalés ainsi ♣ sont de Basile Starynkevitch (dont les opinions n'engagent que lui) ♠

La plupart des transparents sont [recopiés de ceux] de J.Malenfant 2012, que je remercie. Ces transparents sont disponibles sous http://starynkevitch.net/Basile/

# Rappels sur le λ-calcul

1. Variables (lettres minuscules) $v$
2. Constantes prédéfinies $c$ (valeurs, opérateurs, ..., *i.e.*, les $\delta$-*règles*)
3. Application de fonctions $(f\ a)$ où $f$ est la fonction et $a$ son argument (combinateurs)
4. $\lambda$-abstractions $\lambda v. \ldots$ (définitions de fonctions)

- variable liée par $\lambda$
- $\alpha$-conversion : $\lambda v.E \ \Rightarrow_\alpha \ \lambda w.E[v \rightarrow w]$
- $\beta$-redex : un terme $(\lambda v.E)\ E_1$
- $\beta$-réduction : $(\lambda v.E)\ E_1 \ \Rightarrow_\beta \ E[v \rightarrow E_1]$
- $\delta$-règles (pour les constantes comme `1` ou `+`)
- tous les termes ne sont pas réductibles en forme normale (unique à $\alpha$-conv. près)

# Thèse de Church

## Thèse de Church

toutes les fonctions calculables sur les entiers sont celles définissables par le $\lambda$-calcul pur et correspondent aux fonctions calculables par une machine de Turing.

♠ Toute méthode de calcul correspond à une machine de Turing ou à une réduction dans le $\lambda$-calcul. ♣

Rappelons que Turing a démontré l'indécidabilité du problème d'arrêt...

# Relation avec le passage de paramètres

Il existe donc, comme mentionné précédemment, une relation étroite entre stratégie de réduction et mode de passage de paramètres aux fonctions :

- Appel par *nom* : réduction en ordre normal [1], sauf qu'aucun redex apparaissant dans une abstraction n'est jamais réduit.
- Appel par *valeur* : réduction en ordre applicatif [2], sauf qu'aucun redex apparaissant dans une abstraction n'est jamais réduit.

1. ♠ Ordre normal : on réduit toujours le β-redex le plus à gauche et le plus « extérieur » dans l'arborescence formée par le terme ♣
2. ♠ Ordre applicatif : on réduit toujours le β-redex le plus à gauche et le plus « profond » dans l'arborescence formée par le terme ♣

# Comment écrire une fonction récursive en λ-calcul

- En λ-calcul, les fonctions n'ont pas de nom, elles ne peuvent donc pas se désigner elle-même de manière à s'appeler récursivement.
- On a vu une notation pour donner des noms à des λ-expressions (*Twice* = ...), mais
  - c'est une notation qui est « hors » du λ-calcul, et
  - elle suppose qu'on peut faire une substitution textuelle, ce qui n'est pas possible si le terme se mentionne lui-même dans son corps.
- Le λ-calcul utilise donc plutôt un terme particulier qui permet, lors de sa réduction, de réappliquer une fonction à l'intérieur de son corps autant de fois que nécessaire.
- C'est le *combinateur de point fixe* !
  ♠ $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ qui est une chose bizarre appliquant une fonction à elle-même ! ♣

# à propos des points fixes

Soit $f$ une fonction (ou une application) d'un ensemble $E$ vers lui-même $f : E \to E$ alors un element $x$ de $E$ est **un point fixe de** $f$ ssi $f(x) = x$

Ainsi 2 est un point fixe de $x \to 5x - 8$ car $5 \times 2 - 8 = 2$

Intuitivement si on a une suite $u_{n+1} = f(u_n)$ et $u_0$ donnés alors "souvent" $\lim_{i \to \infty} u_i$ est[3] un point fixe de $f$

Par exemple (algorithme de Héron, méthode de Newton Raphson) calcul de $\sqrt{a}$ (où $a > 0$) via $u_{n+1} = \frac{1}{2}(u_n + \frac{a}{u_n})$ car $\sqrt{a}$ est un point fixe de $f(x) = \frac{1}{2}(x + \frac{a}{x})$

Les points fixes sont intuitivement liés aux itérations et aux récursions.

3. Ce n'est pas toujours vrai, par contre-exemple $u_0 = 1$ et $f(x) = -x$ dont 0 est un point fixe vers lequel ne converge pas la suite $u_{n+1} = f(u_n)$ ...

# Théorème du point fixe I

## Théorème (point fixe)

Pour tout $\lambda$-terme $F$, il existe un $\lambda$-terme $X$ tel que $F\ X = X$. $X$ est appelé *point fixe* de $F$.

**Preuve.** Soit $W \equiv \lambda x.F\ (x\ x)$ et $X \equiv W\ W$. Alors

$$X \equiv W\ W \equiv (\lambda x.F\ (x\ x))\ W \Rightarrow_\beta F\ (W\ W) \equiv F\ X$$

- Les points fixes nous sont familiers. Par exemple, 1 est le point fixe de la fonction racine carrée, car $\sqrt{1} = 1$.
- En $\lambda$-calcul, la fonction identité est son propre point fixe :

$$(\lambda x.x)\ (\lambda x.x) \Rightarrow_\beta (\lambda x.x)$$

- Il est plus surprenant d'apprendre que *tout terme* a un point fixe !

# Théorème du point fixe II

De fait, le théorème du point fixe possède une preuve constructive, c'est-à-dire que la preuve exhibe le terme qui est le point fixe, et ainsi fournit une *recette* pour le construire le point fixe de tout terme. Pour $F \equiv \lambda x.\lambda y.(x\ y)$, il suffit d'utiliser le terme $W$ et d'y substituer $F$ :

$$W \equiv \lambda z.F\ (z\ z) \equiv \lambda z.((\lambda x.\lambda y.(x\ y))\ (z\ z)) \equiv \lambda z.\lambda y.((z\ z)\ y)$$

Le terme $X \equiv W\ W \equiv ((\lambda z.\lambda y.((z\ z)\ y))\ (\lambda z.\lambda y.((z\ z)\ y)))$ est le point fixe cherché. En effet,

$$\begin{aligned} X &\equiv (W\ W) \\ &\equiv ((\lambda z.\lambda y.((z\ z)\ y))\ W) \\ &\Rightarrow_\beta \lambda y.((W\ W)\ y) \\ &\Rightarrow_\beta ((\lambda x.\lambda y.(x\ y))\ (W\ W)) \\ &\equiv (F\ (W\ W)) \\ &\equiv F\ X \end{aligned}$$

# Un combinateur de point fixe

- Le théorème du point fixe nous dit qu'on peut trouver le point fixe de tout terme $F$ en substituant ce terme à l'intérieur du terme $W\ W$.
- Cette observation inspire la définition suivante qui applique simplement le $\lambda$-calcul à ce procédé.

**Définition (combinateur de point fixe)**

$$Y \equiv \lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))))$$

- Notez que pour tout terme $F$, l'expression $Y\ F$ va donner son point fixe, d'où le nom de combinateur de point fixe pour $Y$.

# Combinateur de point fixe et fonction récursive I

- Considérons la fonction factorielle que nous pourrions définir dans un langage de programmation quelconque sous la forme :

```
fact n := if n = 0 then 1 else n * (fact n-1)
```

- Comme nous l'avons vu, il serait tentant d'écrire :

$$Fact = \lambda n.if\ (=\ n\,0)\ then\ 1\ else\ (\times\ n\ (Fact\ (-\ n\,1)))$$

mais ce n'est pas un $\lambda$-terme...

- Notons cependant que nous pourrions faire apparaître le terme *Fact* dans le $\lambda$-terme comme un paramètre d'une $\lambda$-expression :

$$Fact = \lambda f.\lambda n.if\ (=\ n\,0)\ then\ 1\ else\ (\times\ n\ (f\ (-\ n\,1)))$$

et la récursivité pourrait alors être obtenue en appliquant ce terme sur lui-même !

- C'est exactement ce que le combinateur de point fixe va permettre. Calculons $T = ((Y\ Fact)\ 2)$ :

# Combinateur de point fixe et fonction récursive II

$$
\begin{aligned}
T &\equiv ((Y\ Fact)\ 2)\\
&\equiv ((\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))))\ Fact)\ 2)\\
&\Rightarrow_\beta (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ 2)\\
&\Rightarrow_\beta ((Fact\ ((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x)))))\ 2)\\
&\Rightarrow_\beta ((\lambda n.if\ (=\ n\ 0)\ then\ 1\ else\ (\times\ n\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ n\ 1))))\ 2)\\
&\Rightarrow_\beta if\ (=\ 2\ 0)\ then\ 1\ else\ (\times\ 2\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ 2\ 1)))\\
&\Rightarrow_\delta (\times\ 2\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ 2\ 1)))\\
&\Rightarrow_\delta (\times\ 2\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ 1))\\
&\Rightarrow_\beta (\times\ 2\ ((Fact\ ((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x)))))\ 1))\\
&\Rightarrow_\beta (\times\ 2\ ((\lambda n.if\ (=\ n\ 0)\ then\ 1\ else\ (\times\ n\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ n\ 1)))\ 1))\\
&\Rightarrow_\beta (\times\ 2\ (if\ (=\ 1\ 0)\ then\ 1\ else\ (\times\ 1\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ 1\ 1)))))
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow_\delta (\times\ 2\ (\times\ 1\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ 1\ 1))))\\
&\Rightarrow_\delta (\times\ 2\ (\times\ 1\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ 0)))\\
&\Rightarrow_\beta (\times\ 2\ (\times\ 1\ ((Fact\ ((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x)))))\ 0)))\\
&\Rightarrow_\beta (\times\ 2\ (\times\ 1\ ((\lambda n.if\ (=\ n\ 0)\ then\ 1\ else\ (\times\ n\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ n\ 1))))\ 0)))\\
&\Rightarrow_\delta (\times\ 2\ (\times\ 1\ (if\ (=\ 0\ 0)\ then\ 1\ else\ (\times\ 0\ (((\lambda x.(Fact\ (x\ x)))\ (\lambda x.(Fact\ (x\ x))))\ (-\ 0\ 1))))))\\
&\Rightarrow_\delta (\times\ 2\ (\times\ 1\ 1)) \Rightarrow_\delta (\times\ 2\ 1) \Rightarrow_\delta 2
\end{aligned}
$$

# Combinateur de point fixe et fonction récursive III

- Notons que l'obtention de la forme normale dans cette réduction dépend de manière cruciale sur l'utilisation de l'ordre normal dans les $\beta$-réductions, sinon il y aurait divergence (essayez !).
- Cette observation sera importante quand il s'agira d'implanter le $\lambda$-calcul sur ordinateur.

## ♠ NB

Cf `http://en.wikipedia.org/wiki/Fixed-point_combinator` et en Ocaml

```
let rec fix f x = f (fix f) x (* note the extra x *) ;;
let factabs fact = fun x -> if x <= 0 then 1 else x * fact (x - 1);;
(fix factabs) 5
```

♠

# Point fixe et récursion en pratique

Un `letrec` n'admet (dans un langage strict c.à.d. non paresseux) que des expressions **constructives** (dont les λ-abstractions), pratiquement :

1. on alloue en parallèle toutes les zones pour les expressions constructives
2. on "lie les variables" donc construit un environment
3. on remplit les zones
4. on récurre, donc on peut entrer dans le corps du `letrec`

Langages paresseux (à la Haskell)
`http://en.wikipedia.org/wiki/Lazy_evaluation` : les expressions y sont des "thunks" `http://en.wikipedia.org/wiki/Thunk_(functional_programming)` qui ne calculeront que si/quand nécessaire

# Activités complémentaires

- Regarder le tutoriel de Barendregt et Barendsen sur le λ-calcul.

# Premiers contacts

Sémantique formelle : établit une correspondance entre programmes et un domaine d'objets abstraits mais rigoureusement définis donnant de manière non-ambigüe la signification des programmes en termes de calculs réalisés.

Sémantique dénotationnelle : les objets utilisés pour donner la sémantique sont des objets *mathématiques*.

*N.B. : à l'origine, la sémantique dénotationnelle s'appelait sémantique* ***mathématique****.*

# Principes fondamentaux

- La SD est fondée sur l'idée que les programmes et les objets qu'ils manipulent ne sont que des réalisations symboliques (syntaxiques) d'objets mathématiques sous-jacents.

  Exemple : séquences de caractères de chiffres représentent, réalisent symboliquement ou syntaxiquement les nombres ;
  les fonctions et les sous-programmes réalisent des fonctions mathématiques (au moins un sous-ensemble de ces dernières...).

- Le principe de la SD est donc de définir des règles permettant d'associer un objet mathématique à toute construction syntaxique pouvant apparaître dans un programme.
  On dira alors que la construction syntaxique **dénote** l'objet mathématique et que ce dernier est la **dénotation** de la construction.

  D'où le terme **sémantique dénotationnelle**.

# Compositionnalité

Il s'agit d'une propriété fondamentale, que nous avons évoquée pour les SOS, mais qui est considérée comme partie intégrante des SD.

Rappel : la syntaxe abstraite décompose un programme en constructions et sous-constructions selon une grammaire précise.

## Définition (compositionnalité)

Une sémantique dénotationnelle est *compositionnelle* si toute dénotation d'une construction syntaxique est la composition des dénotations de ses sous-constructions strictes.

- Pragmatiquement, cela veut dire que la dénotation d'une construction ne peut dépendre d'autres constructions qui ne sont pas ses sous-constructions.
- Par exemple, la sémantique d'un `while` se compose de la sémantique de son expression de test et de celle de son corps, mais ne peut pas dépendre de la sémantique des constructions syntaxiques qui le précède ou le suit, ni même d'elle-même.
- Rappelez-vous que la règle SOS du `while` pour BOPL utilise récursivement sa propre sémantique dans son antécédent ; ce ne sera plus permis en SD.
- L'un des intérêts de cette restriction, observée rigoureusement, est de permettre les preuves par induction structurelle (nous avons un peu triché avec les SOS...).

# Les doubles crochets

- Syntaxe versus sémantique : les définitions de SD établissent une frontière beaucoup plus rigoureuse que les SOS entre le monde syntaxique et le monde sémantique.
- Les « doubles crochets », une des caractéristiques visuelles les plus marquantes des SD, isolent le monde syntaxique du mon sémantique.
- Exemple de l'entier 8 pouvant être la dénotation de plusieurs constructions syntaxiques :

$$\mathit{meaning}[\![2*4]\!] = \mathit{meaning}[\![5+3]\!] = \mathit{meaning}[\![8]\!] = 8$$

# Les fonctions

- Les fonctions jouent un rôle crucial en SD à la fois :
  - pour dénoter les constructions syntaxiques et donc exprimer ce qu'elles calculent, mais aussi
  - pour donner une représentation aux éléments de contexte nécessaires à la définition de la sémantique, comme les environnements, la mémoire, etc.
- La raison pour laquelle les SD insistent pour utiliser des fonctions, là où on aurait envie d'utiliser des structures de données classiques, tient au fait que le monde sémantique ne doit contenir que des objets mathématiques rigoureusement définis ; l'utilisation de fonctions permet d'atteindre cet objectif.
- Comment définir les fonctions ?
  - On peut adopter une vision en *extension*, ensembliste, comme la définition suivante de la fonction successeur : $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle\}$, c'est-à-dire l'ensemble des paires $\langle i, i+1\rangle, i \in \{0,1,2\}$.
  - Cette vision est cependant très limitée, puisqu'elle ne s'applique qu'aux fonctions sur des domaines finis.
  - La SD utilise aussi (et plus souvent) une définition en *intention* utilisant un formalisme mathématiquement bien fondé : le $\lambda$-calcul.
    Exemple : la fonction successeur est définie par $\lambda n.n+1$.

# Structure d'une SD

- Formaliser n'exclut pas d'être structuré et compréhensible.
- Depuis les débuts de la sémantique dénotationnelles, un certain nombre de bonnes pratiques se sont imposées :
  - utilisation d'une structure standard pour définir les SD ;
  - utilisation de noms de domaines standards pour désigner les valeurs utilisables dans différents contextes : résultat d'expressions, valeurs affectables en mémoire, passables en paramètres à des fonctions, ou susceptibles d'être retournées comme résultats, ...
- La structure d'une sémantique dénotationnelle se décompose donc en trois parties :
  1. Le « monde » syntaxique : la syntaxe abstraite.
  2. Le « monde » sémantique : la définition des objets mathématiques utilisés par la suite dans les fonctions de valuation, et des fonctions qui les manipulent.
  3. Les fonctions de valuation et leurs équations : connexion entre monde syntaxique et sémantique, généralement à raison d'une équation au moins par catégorie syntaxique.

# Le « monde » syntaxique

- Le « monde » syntaxique est défini autour de *domaines* ou *catégories* syntaxiques, comme les instructions, les expressions, ...
- La grammaire abstraite utilise des règles de production pour montrer comment se construisent les arbres de syntaxe abstraite à partir des catégories syntaxiques préalablement définies.

Exemple :
Syntaxe abstraite

---

Catégories syntaxiques

$$
\begin{array}{lcl}
i & \in & \textit{Instructions} \\
e & \in & \textit{Expressions} \\
n & \in & \textit{Nombres} \\
v & \in & \textit{Variables}
\end{array}
$$

Grammaire

$$
\begin{array}{lcl}
i & ::= & v := e \mid \texttt{if}\ e\ i_1\ i_2 \mid \texttt{while}\ e\ i \\
e & ::= & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid \\
 & & e_1 \div e_2 \mid n \mid v
\end{array}
$$

---

# Le « monde » sémantique

- Appelé *algèbre sémantique*, le « monde » sémantique est défini autour de *domaines* sémantiques, c'est-à-dire l'ensemble des objets mathématiques utilisés comme dénotations ou dans la construction de ces domaines.
- Il comporte également un ensemble d'*opérations*, qui sont des fonctions auxiliaires utilisées dans la définition de la sémantique, le plus souvent pour manipuler les éléments des domaines sémantiques.

Exemple :
Algèbre sémantique

---

Domaines sémantiques

$$\begin{aligned} T &= \{true, false\} \\ \mathbb{Z} &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\ \Sigma &= Variables \to \mathbb{Z} \end{aligned}$$

Opérations

$$not : T \to T$$

$$plus, minus, times, divide : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$store : \Sigma \times Variables \times \mathbb{Z} \to \Sigma$$

---

# Fonctions sémantiques et équations

- Les fonctions sémantiques établissent la correspondance entre le « monde » syntaxique et le monde « sémantique ».
- Elles sont définies pour chaque catégorie syntaxique et de manière dirigée par la syntaxe avec généralement une équation par production de la grammaire abstraite.
- Les équations utilisent les fameux doubles crochets pour isoler ♠ mais aussi lier ♣ le « monde » syntaxique du « monde » sémantique.

Exemple :
Fonctions sémantiques

---

$$eval : Expressions \rightarrow \Sigma \rightarrow \mathbb{Z}$$

$$eval[\![e_1 + e_2]\!] = \lambda\sigma.plus(eval[\![e_1]\!]\sigma, eval[\![e_2]\!]\sigma)$$

---

♠ Certains notent pédantiquement $[\![+]\!]$ notre *plus*, d'autres se lâchent à la noter aussi $+$ alors qu'il s'agit de mondes différents ♣

# Retour sur la compositionnalité

- Trois raisons pour des définitions compositionnelles :
  1. Les définitions compositionnelles ne dépendent pas du contexte. Si on arrive à prouver, par exemple, l'équivalence sémantique entre deux constructions, on pourra substituer l'une par l'autre *indépendamment du contexte dans lequel elles apparaîssent* (optimisation, ...).
  2. Elles permettent l'utilisation de la technique de preuve par induction structurelle.
  3. Elles présentent une certaine « élégance » par le fait de suivre la structure syntaxique du langage, ce qui en facilite également l'extension à de nouvelles constructions.
- Mathématiquement, on peut aussi montrer que la fonction de valuation définie par une telle expression compositionnelle de ses équations sémantiques est un *homomorphisme*, c'est-à-dire qu'elle respecte les opérations dans le sens suivant :

**Définition (homomorphisme)**

Pour les fonctions $f : A \times A \to A$ et $g : B \times B \to B$, $\mathcal{H} : A \to B$ est un homomorphisme si $(\forall x, y \in A)\ \mathcal{H}(f(x,y)) = g(\mathcal{H}(x), \mathcal{H}(y))$.

Cette notion sera utile quand on parlera de la sémantique par point fixe.

# Syntaxe abstraite

## Catégories syntaxiques

| | | |
|---|---|---|
| $i$ | $\in$ | *Instruction* |
| $e$ | $\in$ | *Expression* |
| $be$ | $\in$ | *BExpression* |
| $n$ | $\in$ | *Nombre* |
| $d$ | $\in$ | *Chiffre* |
| $v$ | $\in$ | *Variable* |

## Grammaire

| | | |
|---|---|---|
| $i$ | $::=$ | `seq` $i_1$ $i_2$ \| $v$ `:=` $e$ \| `if` $be$ $i_1$ $i_2$ \| |
| | | `while` $be$ $i$ |
| $e$ | $::=$ | $e_1$ `+` $e_2$ \| $e_1$ `-` $e_2$ \| $e_1$ `*` $e_2$ \| |
| | | $e_1$ `/` $e_2$ \| $n$ \| $v$ |
| $be$ | $::=$ | $be_1$ `&` $be_2$ \| $be_1$ `\|` $be_2$ \| |
| | | $e_1$ `<` $e_2$ \| $e_1$ `=` $e_2$ \| `not` $be$ |
| $v$ | $::=$ | *non-spécifié* |
| $n$ | $::=$ | $-n$ \| $n\,d$ \| $d$ |
| $d$ | $::=$ | `zero` \| `un` \| `deux` \| `trois` \| `quatre` \| |
| | | `cinq` \| `six` \| `sept` \| `huit` \| `neuf` |

# Algèbre sémantique — domaines

## Algèbre sémantique

### Domaines sémantiques

$$
\begin{aligned}
z \in \mathbb{Z} &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
t \in \mathbf{T} &= \{\mathit{true}, \mathit{false}\} \\
n \in \mathbb{N} &= \{0, 1, 2, \ldots\} \\
\rho \in \mathbf{Env} &= \mathit{Variable} \to \mathbb{N} \\
\sigma \in \Sigma &= \mathbb{N} \to \mathbb{Z}
\end{aligned}
$$

♠ On est tenté d'ajouter un $\bot$ ("bottom") dans chaque domaine pour représenter le "non-défini" ; certains exagèrent jusqu'à noter $\top$ ("top") pour "n'importe-quelle valeur" ; Voir aussi l'interprétation abstraite où ces notations $\bot$ et $\top$ ont un sens bien précis. ♣

# Algèbre sémantique — opérations

$$\mathit{emptyEnv} : \mathbf{Env}$$
$$\mathit{emptyEnv} = \lambda v.\bot_{\mathbb{N}}$$
$$\mathit{extendEnv} : \mathbf{Env} \to \mathit{Variable} \to \mathbb{N} \to \mathbf{Env}$$
$$\mathit{extendEnv} = \lambda \rho.\lambda v.\lambda n.\lambda v_1.\mathbf{if}\ v_1 = v\ \mathbf{then}\ n\ \mathbf{else}\ (\rho\ v_1)$$
$$\mathit{emptyStore} : \Sigma$$
$$\mathit{emptyStore} = \lambda n.\bot_{\mathbb{Z}}$$
$$\mathit{updateStore} : \Sigma \to \mathbb{N} \to \mathbb{Z} \to \Sigma$$
$$\mathit{updateStore} = \lambda \sigma.\lambda n.\lambda z.\lambda n_1.\mathbf{if}\ n_1 = n\ \mathbf{then}\ z\ \mathbf{else}\ (\sigma\ n_1)$$

# Fonctions sémantiques — chiffres et nombres

$$\mathit{chiffre} : \mathit{Chiffre} \to \mathbb{Z}$$

$$\mathit{chiffre}[\![\texttt{zero}]\!] = 0 \qquad \mathit{chiffre}[\![\texttt{un}]\!] = 1 \qquad \mathit{chiffre}[\![\texttt{deux}]\!] = 2 \qquad \mathit{chiffre}[\![\texttt{trois}]\!] = 3$$

$$\mathit{chiffre}[\![\texttt{quatre}]\!] = 4 \qquad \mathit{chiffre}[\![\texttt{cinq}]\!] = 5 \qquad \mathit{chiffre}[\![\texttt{six}]\!] = 6 \qquad \mathit{chiffre}[\![\texttt{sept}]\!] = 7$$

$$\mathit{chiffre}[\![\texttt{huit}]\!] = 8 \qquad \mathit{chiffre}[\![\texttt{neuf}]\!] = 9$$

$$\mathit{valeur} : \mathit{Nombre} \to \mathbb{Z}$$

$$\mathit{valeur}[\![d]\!] = \mathit{chiffre}[\![d]\!]$$

$$\mathit{valeur}[\![n\ d]\!] = \mathit{valeur}[\![n]\!] \times 10 + \mathit{chiffre}[\![d]\!]$$

$$\mathit{valeur}[\![-\ n]\!] = -\mathit{valeur}[\![n]\!]$$

♠ NB : pas le même $-$ à gauche et à droite ♣

# Fonctions sémantiques — expressions numériques

$$eval : Expression \rightarrow Env \rightarrow \Sigma \rightarrow \mathbb{Z}$$

$$eval[\![e_1 + e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma + eval[\![e_2]\!]\rho\sigma$$

$$eval[\![e_1 - e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma - eval[\![e_2]\!]\rho\sigma$$

$$eval[\![e_1 * e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma \times eval[\![e_2]\!]\rho\sigma$$

$$eval[\![e_1 / e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma \div eval[\![e_2]\!]\rho\sigma$$

$$eval[\![n]\!]\rho\sigma = valeur[\![n]\!]$$

$$eval[\![v]\!]\rho\sigma = (\sigma\ (\rho\ v))$$

# Fonctions sémantiques — expressions booléennes

$$beval : BExpression \to Env \to \Sigma \to \mathbf{T}$$

$$beval[\![be_1 \texttt{ \& } be_2]\!]\rho\sigma = beval[\![be_1]\!]\rho\sigma \wedge beval[\![be_2]\!]\rho\sigma$$

$$beval[\![be_1 \texttt{ | } be_2]\!]\rho\sigma = beval[\![be_1]\!]\rho\sigma \vee beval[\![be_2]\!]\rho\sigma$$

$$beval[\![e_1 < e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma < eval[\![e_2]\!]\rho\sigma$$

$$beval[\![e_1 = e_2]\!]\rho\sigma = eval[\![e_1]\!]\rho\sigma = eval[\![e_2]\!]\rho\sigma$$

$$beval[\![\texttt{not } be]\!]\rho\sigma = \neg beval[\![be]\!]\rho\sigma$$

# Fonctions sémantiques — instructions

$$execute : Instruction \to Env \to \Sigma \to \Sigma$$

$$execute[\![\texttt{seq}\ i_1\ i_2]\!]\rho\sigma = execute[\![i_2]\!]\rho(execute[\![i_1]\!]\rho\sigma)$$

$$execute[\![v := e]\!]\rho\sigma = (((updateStore\ \sigma)\ (\rho\ v))\ eval[\![e]\!]\rho\sigma)$$

$$execute[\![\texttt{if}\ be\ i_1\ i_2]\!]\rho\sigma = \textbf{if}\ beval[\![be]\!]\rho\sigma\ \textbf{then}\ execute[\![i_1]\!]\rho\sigma\ \textbf{else}\ execute[\![i_2]\!]\rho\sigma$$

$$execute[\![\texttt{while}\ be\ i]\!]\rho\sigma = (loop\ \sigma)$$

$$\textbf{where}\ loop\ = (\textbf{fix}\ \lambda f.\lambda\sigma.\textbf{if}\ beval[\![be]\!]\rho\sigma\ \textbf{then}\ (f\ execute[\![i]\!]\rho\sigma)\ \textbf{else}\ \sigma)$$

♠ où **fix** est l'opérateur de point fixe $Y$ du $\lambda$-calcul ♣

# SD et domaines

- La SD utilise le $\lambda$-calcul pour définir les fonctions.
- Pour être bien fondée, il faut une sémantique du $\lambda$-calcul, c'est-à-dire un *modèle*.
- D'autre part, les langages de programmation utilisent très souvent la récursivité.
- Pour dénoter ces constructions récursives, il faudrait recourir à des fonctions récursives, mais en $\lambda$-calcul, on a vu que la récursivité ne s'exprime que par des points fixes.
- Il faut donc se fonder sur un modèle du $\lambda$-calcul qui garantisse l'existence de points fixes à chaque fois qu'on utilisera un opérateur de point fixe (c'est-à-dire **fix**, ou $Y$).

# De la récursivité au point fixe — rappel

- Soient deux définitions de fonctions récursives simples :

```
f(n) = if n = 0 then 1 else f(n-1)
g(n) = if n = 0 then 1 else g(n+1)
```

- En λ-calcul avec définition de méta-variables, on serait tenté d'écrire :

$$\begin{aligned} F &= \lambda n.\textbf{if } (= n\,0) \textbf{ then } 1 \textbf{ else } (F\,(-n1)) \\ G &= \lambda n.\textbf{if } (= n\,0) \textbf{ then } 1 \textbf{ else } (G\,(+n1)) \end{aligned}$$

- Mais ceci n'est pas admissible, puisque la définition de méta-variables en λ-calcul n'admet que les cas où on peut faire une substition textuelle de la méta-variable par sa définition, ce qui n'est pas possible si la méta-variable apparaît dans sa propre définition.
- Pour définir ces méta-variables correctement, il faudrait trouver les termes $\Lambda_1$ et $\Lambda_2$ tels que

$$\begin{aligned} \Lambda_1 &= \lambda n.\textbf{if } (= n\,0) \textbf{ then } 1 \textbf{ else } (\Lambda_1\,(-n1)) \\ \Lambda_2 &= \lambda n.\textbf{if } (= n\,0) \textbf{ then } 1 \textbf{ else } (\Lambda_1\,(+n1)) \end{aligned}$$

- On constate que dans les deux cas, la méta-variable ($\Lambda_1$ ou $\Lambda_2$) apparaît à la fois à gauche et à droite de l'équation, ce qui demande de trouver dans chaque cas le λ-terme qui rend les deux côtés de l'équation égaux.
- Mathématiquement, cela revient à résoudre deux équations de récurrence, de la même manière qu'on peut devoir résoudre une équation de récurrence comme $x = x^2 - 4x + 6$. Cette équation a deux solutions : $x = 2$ et $x = 3$ (essayez !).

# Calcul du point fixe

Idée du calcul du point fixe :

- Un opérateur est défini tel que, si on lui soumet une valeur, il rend une nouvelle valeur strictement plus près du point fixe.
- On peut alors utiliser cet opérateur en l'appliquant répétitivement à partir d'une valeur initiale jusqu'à convergence sur le point fixe, c'est-à-dire quand l'opérateur rend exactement la même valeur que celle qui lui est soumise.

Mais pour que ça fonctionne, il faut :

1. Se donner une mesure de proximité pour prouver que l'opérateur rend bien une valeur strictement plus proche.
2. Prouver que la limite de la séquence de valeurs partant de la valeur initiale et produite par les applications successives de l'opérateur existe bel et bien

Dans les mathématiques du continu, les espaces mesurables (par exemple, les espaces de Banach) ont servi à démontrer ce genre de propriétés pour des calculs de points fixes sur des équations réelles.
La *théorie des domaines* cherche à donner des fondations mathématiques permettant de prouver cela pour le $\lambda$-calcul.

# Les domaines

- Les domaines sont des structures d'ensembles munis de relations d'ordre leur donnant des propriétés similaires à des treillis.

**Définition (treillis)**

Un *treillis* (en anglais : lattice) est un ensemble partiellement ordonné dans lequel chaque couple d'éléments admet une borne supérieure et une borne inférieure.

- Un domaine est une version « affaiblie » d'un treillis, dont nous allons donner une définition algébrique.

# Vers une définition des domaines

## Définition (ordre partiel)

Un *ordre partiel* sur un ensemble $S$ est une relation $\preccurlyeq$ telle que :

1. $\preccurlyeq$ est réflexive, c'est-à-dire $(\forall x \in S)\ \ x \preccurlyeq x$.
2. $\preccurlyeq$ est transitive, c'est-à-dire $(\forall x, y, z \in S)\ \ x \preccurlyeq y$ *et* $y \preccurlyeq z \implies x \preccurlyeq z$.
3. $\preccurlyeq$ est antisymétrique, c'est-à-dire $(\forall x, y \in S)\ \ x \preccurlyeq y$ *et* $y \preccurlyeq x \implies x = y$.

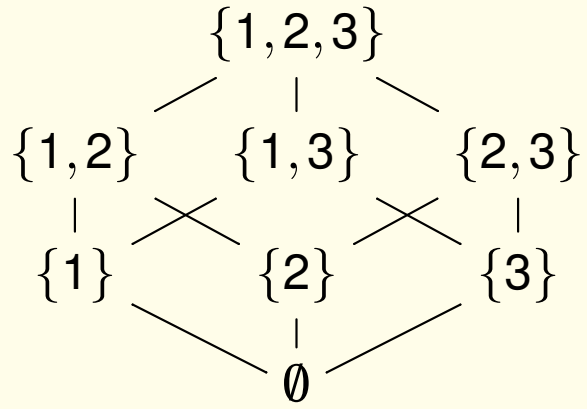
## Définitions

Soit $A$ un sous-ensemble de $S$ muni d'un ordre partiel $\preccurlyeq$,

1. Une *borne inférieure* de $A$ est un élément $l \in S$ tel que $(\forall x \in A)\ \ l \preccurlyeq x$.
2. Une *borne supérieure* de $A$ est un élément $u \in S$ tel que $(\forall x \in A)\ \ x \preccurlyeq u$.
3. Une *plus petite borne supérieure* de $A$, notée $lub(A)$, est une borne supérieure de $A$ telle que pour toute borne supérieure $m$ de $A$, $lub(A) \preccurlyeq m$.

# Un exemple d'ordre partiel

Exemple : L'ensemble $\mathscr{P}(\{1,2,3\})$ muni de $\subseteq$, l'inclusion ensembliste.




Pour l'ensemble $A = \{\{1,2\},\{2\},\{1,3\}\} \subseteq \mathscr{P}(\{1,2,3\})$, on a $lub(A) = \{1,2,3\}$.
*Nota : en fait* $\mathscr{P}(\{1,2,3\})$ *muni de* $\subseteq$ *forme un treillis.*

**Définition (chaîne ascendante)**

Soit un ensemble $S$ muni d'un ordre partiel $\preccurlyeq$, une *chaîne ascendante* est une séquence d'éléments $x_1, x_2, x_3, \ldots$ de $S$ telle que $x_1 \preccurlyeq x_2 \preccurlyeq x_3 \preccurlyeq \ldots$.

Pour l'exemple de $\mathscr{P}(\{1,2,3\})$, les ensembles $\emptyset$, $\{1\}$ et $\{1,3\}$ forment une chaine ascendante selon l'ordre d'inclusion.

# Ordre partiel complet “CPO”

Soit $\perp$ une valeur indéfinie pouvant dénoter l'inconnu ou la non-terminaison.

**Définition (ordre partiel complet)**

Un *ordre partiel complet* sur un ensemble $S$ est un ordre partiel tel que :

1. Il existe un élément $\perp \in S$ tel que $(\forall x \in S)\ \perp \preccurlyeq x$.
2. Toute chaîne ascendante a une plus petite borne supérieure dans $S$.

*Nota : les conditions d'un ordre partiel complet (CPO, “complete partial order”) sont strictement moins fortes que pour un treillis.*

# Les domaines

- Les ensembles munis d'un CPO fournissent une bonne base pour la sémantique dénotationnelle car ils vont permettre le calcul des points fixes sur la base d'une notion (proximité) de « plus ou moins défini » fondée sur la relation $\preccurlyeq$, c'est-à-dire
  - $x \preccurlyeq y$ est interprété comme le fait que $y$ contient plus d'information que $x$, tout en étant consistant avec $x$ ;
  - les chaînes ascendantes représentent une accumulation d'information rendant la valeur de mieux en mieux définie ;
  - l'existence d'une plus petite borne supérieure à toute chaîne ascendante permet de garantir l'existence d'un point fixe.
- Dans certains cas, la notion de plus ou moins défini sera triviale.
- Pour les entiers, par exemple, ce sera simplement le fait de connaître (un certain $n \in \mathbb{N}$) ou non ($\perp$) l'entier.
- Pour les fonctions, ce sera plus riche : la chaine ascendante sera formée de fonctions partielles de mieux en mieux définies, c'est-à-dire couvrant de mieux en mieux leur domaine de définition, et ayant pour point fixe la fonction totale consistante avec toutes les fonctions partielles précédentes.
- L'objectif est maintenant de définir un catalogue de domaines susceptibles de représenter toutes les valeurs à manipuler dans les sémantiques de langages de programmation.

# Domaines de base

- Extension des ensembles de valeurs discrètes comme les entiers, les booléens, ...

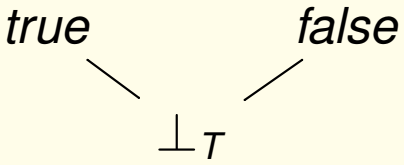
**Définition (domaines discrets)**

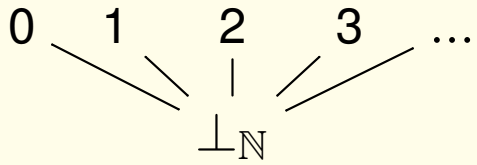
Soit l'ensemble $S$, on construit le domaine **S** par :

- ajout d'un élément $\bot_S$ ;
- définition d'un ordre partiel discret $\preccurlyeq_S$ de la manière suivante :

$$\forall x, y \in S \quad x \preccurlyeq_S y \quad \textit{ssi} \quad x = y \;\; \textit{ou} \;\; x = \bot_S$$

- Exemples : Booléens $T$ et Entiers $\mathbb{N}$



# Domaines produits

## Définition (domaines produits)

Soient les domaines $\mathbf{A}$ muni de $\preccurlyeq_{\mathbf{A}}$ et $\mathbf{B}$ muni de $\preccurlyeq_{\mathbf{B}}$, le domaine produit $\mathbf{A} \otimes \mathbf{B}$ est l'ensemble $\{(a,b) \mid a \in A, b \in B\}$ muni de l'ordre $\preccurlyeq_{\mathbf{A}\otimes\mathbf{B}}$ défini par :

$$(a,b) \preccurlyeq_{\mathbf{A}\otimes\mathbf{B}} (c,d) \quad \textit{ssi} \quad a \preccurlyeq_{\mathbf{A}} c \ \textit{et} \ b \preccurlyeq_{\mathbf{B}} d$$

avec $\bot_{\mathbf{A}\times\mathbf{B}} = (\bot_{\mathbf{A}}, \bot_{\mathbf{B}})$

*Nota : s'étend immédiatement au cas* $\mathbf{D}_1 \otimes \ldots \otimes \mathbf{D}_n$ *pour un n fini.*

## Définitions (fonctions de projections)

$$\begin{aligned} \mathit{first} &= \lambda(a,b).a \in \mathbf{A}\otimes\mathbf{B} \to \mathbf{A} \\ \mathit{second} &= \lambda(a,b).b \in \mathbf{A}\otimes\mathbf{B} \to \mathbf{B} \end{aligned}$$

*Nota : s'étend aussi au cas* $\mathbf{D}_1 \otimes \ldots \otimes \mathbf{D}_n$ *pour un n fini avec les fonctions ith,* $3 \leq i \leq n$ *(c'est-à-dire* 3*th,* 4*th, ...).*

# Domaines sommes (unions disjointes)

## Définition (domaines sommes)

Soient les domaines $\mathbf{A}$ muni de $\preccurlyeq_{\mathbf{A}}$ et $\mathbf{B}$ muni de $\preccurlyeq_{\mathbf{B}}$, le domaine somme $\mathbf{A} \oplus \mathbf{B}$ est l'ensemble $\{\langle a,1\rangle \mid a \in \mathbf{A}\} \cup \{\langle b,2\rangle \mid b \in \mathbf{B}\} \cup \{\bot_{\mathbf{A}\oplus\mathbf{B}}\}$ avec la relation d'ordre $\preccurlyeq_{\mathbf{A}\oplus\mathbf{B}}$ défini par :

$$\langle a,1\rangle \preccurlyeq_{\mathbf{A}\oplus\mathbf{B}} \langle c,1\rangle \quad ssi \quad a \preccurlyeq_{\mathbf{A}} c$$

$$\langle b,2\rangle \preccurlyeq_{\mathbf{A}\oplus\mathbf{B}} \langle d,2\rangle \quad ssi \quad b \preccurlyeq_{\mathbf{B}} d$$

$$\bot_{\mathbf{A}\oplus\mathbf{B}} \preccurlyeq_{\mathbf{A}\oplus\mathbf{B}} \langle a,1\rangle \quad \forall a \in \mathbf{A}$$

$$\bot_{\mathbf{A}\oplus\mathbf{B}} \preccurlyeq_{\mathbf{A}\oplus\mathbf{B}} \langle b,2\rangle \quad \forall b \in \mathbf{B}$$

$$\bot_{\mathbf{A}\oplus\mathbf{B}} \preccurlyeq_{\mathbf{A}\oplus\mathbf{B}} \bot_{\mathbf{A}\oplus\mathbf{B}}$$

*Nota : le choix des étiquettes 1 et 2 est purement arbitraire.*

*Nota : s'étend immédiatement au cas $\mathbf{D}_1 \oplus \ldots \oplus \mathbf{D}_n$ pour un n fini.*

## Définitions (fonctions d'injection)

Soit $\mathbf{S} = \mathbf{A} \oplus \mathbf{B}$

$$in\mathbf{S}_1 = \lambda a.\langle a,1\rangle \in \mathbf{A} \to \mathbf{A} \oplus \mathbf{B}$$

$$in\mathbf{S}_2 = \lambda b.\langle b,2\rangle \in \mathbf{B} \to \mathbf{A} \oplus \mathbf{B}$$

# Domaines sommes 2

## Définitions (fonctions de projection)

$$out\mathbf{A} = \lambda s. \begin{cases} a & si\ s = \langle a, 1\rangle, \forall a \in \mathbf{A} \\ \bot_{\mathbf{A}} & si\ s = \langle b, 2\rangle, \forall b \in \mathbf{B}\ ou\ s = \bot_{\mathbf{A}\oplus\mathbf{B}} \end{cases}$$

$$out\mathbf{B} = \lambda s. \begin{cases} b & si\ s = \langle b, 2\rangle, \forall b \in \mathbf{B} \\ \bot_{\mathbf{B}} & si\ s = \langle a, 1\rangle, \forall a \in \mathbf{A}\ ou\ s = \bot_{\mathbf{A}\oplus\mathbf{B}} \end{cases}$$

## Définitions (fonctions d'inspection)

$$is\mathbf{A} = \lambda s. \begin{cases} true & si\ s = \langle a, 1\rangle, \forall a \in \mathbf{A} \\ false & si\ s = \langle b, 2\rangle, \forall b \in \mathbf{B} \\ \bot_{\mathbf{T}} & si\ s = \bot_{\mathbf{A}\oplus\mathbf{B}} \end{cases}$$

$$is\mathbf{B} = \lambda s. \begin{cases} true & si\ s = \langle b, 2\rangle, \forall b \in \mathbf{B} \\ false & si\ s = \langle a, 1\rangle, \forall a \in \mathbf{A} \\ \bot_{\mathbf{T}} & si\ s = \bot_{\mathbf{A}\oplus\mathbf{B}} \end{cases}$$

# Domaines sommes 3

*Nota : s'étendent aussi au cas* $\mathbf{D}_1 \oplus \ldots \oplus \mathbf{D}_n$ *pour un n fini avec les fonctions* $inS_i$, $outD_i$ *et* $isD_i$, $1 \leq i \leq n$.

# Domaines séquences

- La construction $D^* = \{nil\} \oplus D \oplus D^1 \oplus D^2 \oplus \ldots$, permet de représenter l'ensemble des séquences (listes) de valeurs du domaine $D$.
- Pour plus de commodité, on adjoint la longueur de la liste.

**Définition (domaines séquences)**

La construction $D^* = \bigoplus_{k=0}^{\infty} \langle D^k, k \rangle$ représente l'ensemble des séquences ♠ finies ♣ (listes) de valeurs du domaine $D$.

où

$$D^n = \underbrace{D \otimes \ldots \otimes D}_{n \text{ fois}} \quad et \quad D^0 = nil$$

# fonctions sur les séquences

## Définitions (fonctions sur les séquences)

Soient $L \in D^*$, $e \in D$, avec $L = \langle d, k \rangle$ pour $d \in D^k$, $k \geq 0$ et où $D^0 = \{nil\}$, alors :

1. $head : D^* \to D$, $head(L) = first(outD^k(L))$, si $k > 0$ et $\perp$ sinon.
2. $tail : D^* \to D^*$, $tail(L) = inD^*(\langle\langle second(outD^k(L)), third(outD^k(L)), ...\rangle, k-1\rangle)$, si $k > 0$ et $\perp$ sinon.
3. $null : D^* \to T$, $null(\langle nil, 0\rangle) = true$ et $null(\langle L, k\rangle) = false, k > 0$.
4. $prefix : D \times D^* \to D^*$, $prefix(e, L) = inD^*(\langle\langle e, first(outD^k(L)), second(outD^k(L)), ...\rangle, k+1\rangle)$, avec $prefix(e, \langle nil, 0\rangle) = \langle e, 1\rangle$.
5. $affix : D^* \times D \to D^*$, $affix(L, e) = inD^*(\langle\langle first(outD^k(L)), second(outD^k(L)), ..., e\rangle, k+1\rangle)$, avec $affix(\langle nil, 0\rangle, e) = \langle e, 1\rangle$.