MI030 — APS
Analyse des programmes et sémantique

© Jacques Malenfant, 2010–2014

avec la participation initiale d'**Olena Rogovchenko**

Université Pierre et Marie Curie
UFR 919 Ingénierie
Jacques.Malenfant@lip6.fr

Cours 6 et 7

Introduction à la sémantique dénotationnelle

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 I

- 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...).

Principes fondamentaux II

- 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 I

- 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.

Compositionnalité II

- 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.

Compositionnalité III

- 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 :

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

Les fonctions I

- 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.

Les fonctions II

- 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 λ-calcul. Exemple : la fonction successeur est définie par $\lambda n.n+1$.

Structure d'une SD I

- 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 :

Structure d'une SD II

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 I

- 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.

Le « monde » syntaxique II

Exemple :

Syntaxe abstraite

Catégories syntaxiques			Grammaire		
i	$\in$	*Instructions*	i	$::=$	$v := e \mid \texttt{if}\ e\ i_1\ i_2 \mid \texttt{while}\ e\ i$
e	$\in$	*Expressions*	e	$::=$	$e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid$
n	$\in$	*Nombres*			$e_1 \div e_2 \mid n \mid v$
v	$\in$	*Variables*			

Le « monde » sémantique I

- 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.

Le « monde » sémantique II

Exemple :

Algèbre sémantique

Domaines sémantiques			Opérations
T	$=$	$\{true, false\}$	$not : T \to T$
$\mathbb{Z}$	$=$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	$plus, minus, times, divide : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
Σ	$=$	$Variables \to \mathbb{Z}$	$store : \Sigma \times Variables \times \mathbb{Z} \to \Sigma$

Fonctions sémantiques et équations I

- 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 le « monde » syntaxique du « monde » sémantique.

Fonctions sémantiques et équations II

Exemple :

Fonctions sémantiques

$$eval : Expressions \to \Sigma \to \mathbb{Z}$$
$$eval[\![e_1 + e_2]\!] = \lambda\sigma.plus(eval[\![e_1]\!]\sigma, eval[\![e_2]\!]\sigma)$$

Retour sur la compositionnalité I

- 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.

Retour sur la compositionnalité II

- 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

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

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

Algèbre sémantique — opérations

Opérations

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

Fonctions sémantiques — chiffres et nombres

Fonctions sémantiques

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

$$chiffre[\![\texttt{zero}]\!] = 0 \quad chiffre[\![\texttt{un}]\!] = 1 \quad chiffre[\![\texttt{deux}]\!] = 2 \quad chiffre[\![\texttt{trois}]\!] = 3$$
$$chiffre[\![\texttt{quatre}]\!] = 4 \quad chiffre[\![\texttt{cinq}]\!] = 5 \quad chiffre[\![\texttt{six}]\!] = 6 \quad chiffre[\![\texttt{sept}]\!] = 7$$
$$chiffre[\![\texttt{huit}]\!] = 8 \quad chiffre[\![\texttt{neuf}]\!] = 9$$

$$valeur : Nombre \to \mathbb{Z}$$
$$valeur[\![d]\!] = chiffre[\![d]\!]$$
$$valeur[\![n\ d]\!] = valeur[\![n]\!] \times 10 + chiffre[\![d]\!]$$
$$valeur[\![-n]\!] = -valeur[\![n]\!]$$

Fonctions sémantiques — expressions numériques

$$eval : Expression \to Env \to \Sigma \to \mathbb{Z}$$
$$eval[\![e_1 \texttt{ + } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma + eval[\![e_2]\!]\rho\sigma$$
$$eval[\![e_1 \texttt{ - } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma - eval[\![e_2]\!]\rho\sigma$$
$$eval[\![e_1 \texttt{ * } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma \times eval[\![e_2]\!]\rho\sigma$$
$$eval[\![e_1 \texttt{ / } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma \div eval[\![e_2]\!]\rho\sigma$$
$$eval[\![n]\!] = \lambda\rho.\lambda\sigma.valeur[\![n]\!]$$
$$eval[\![v]\!] = \lambda\rho.\lambda\sigma.(\sigma\ (\rho\ v))$$

Fonctions sémantiques — expressions booléennes

$$beval : BExpression \to Env \to \Sigma \to \mathbf{T}$$
$$beval[\![be_1 \texttt{ \& } be_2]\!] = \lambda\rho.\lambda\sigma.beval[\![be_1]\!]\rho\sigma \wedge beval[\![be_2]\!]\rho\sigma$$
$$beval[\![be_1 \texttt{ | } be_2]\!] = \lambda\rho.\lambda\sigma.beval[\![be_1]\!]\rho\sigma \vee beval[\![be_2]\!]\rho\sigma$$
$$beval[\![e_1 \texttt{ < } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma < eval[\![e_2]\!]\rho\sigma$$
$$beval[\![e_1 \texttt{ = } e_2]\!] = \lambda\rho.\lambda\sigma.eval[\![e_1]\!]\rho\sigma = eval[\![e_2]\!]\rho\sigma$$
$$beval[\![\texttt{not } be]\!]\rho\sigma = \lambda\rho.\lambda\sigma.\neg beval[\![be]\!]\rho\sigma$$

Fonctions sémantiques — instructions

$$execute : Instruction \to Env \to \Sigma \to \Sigma$$
$$execute[\![\texttt{seq } i_1\ i_2]\!] = \lambda\rho.\lambda\sigma.execute[\![i_2]\!]\rho(execute[\![i_1]\!]\rho\sigma)$$
$$execute[\![v \texttt{ := } e]\!] = \lambda\rho.\lambda\sigma.(((updateStore\ \sigma)\ (\rho\ v))\ eval[\![e]\!]\rho\sigma)$$
$$execute[\![\texttt{if } be\ i_1\ i_2]\!] = \lambda\rho.\lambda\sigma.\textbf{if } beval[\![be]\!]\rho\sigma \textbf{ then } execute[\![i_1]\!]\rho\sigma \textbf{ else } execute[\![i_2]\!]\rho\sigma$$
$$execute[\![\texttt{while } be\ i]\!] = \lambda\rho.\lambda\sigma.(loop\ \sigma)$$
$$\textbf{where } loop = (\textbf{fix } \lambda f.\lambda\sigma.\textbf{if } beval[\![be]\!]\rho\sigma \textbf{ then } (f\ execute[\![i]\!]\rho\sigma) \textbf{ else } \sigma)$$

SD et domaines

- La SD utilise le λ-calcul pour définir les fonctions.
- Pour être bien fondée, il faut une sémantique du λ-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 λ-calcul, on a vu que la récursivité ne s'exprime que par des points fixes.
- Il faut donc se fonder sur un modèle du λ-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 du cours 5 I

- 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 :

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

De la récursivité au point fixe — rappel du cours 5 II

- 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 Λ_1 et Λ_2 tels que

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

De la récursivité au point fixe — rappel du cours 5 III

- On constate que dans les deux cas, la méta-variable (Λ_1 ou Λ_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 I

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 :

Calcul du point fixe II

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 λ-calcul.

Les domaines

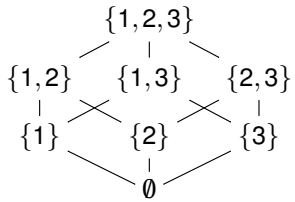
- Les domaines sont des structures d'ensembles munis de relations d'ordre leur donnant des propriétés similaires à des treillis.

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.

Exemple de treillis I

L'ensemble $\mathscr{P}(\{1,2,3\})$ muni de $\subseteq$, l'inclusion ensembliste forme un treillis ici représenté par les relations d'inclusion directe :



- Pour l'ensemble $A = \{\{1,2\},\{2\},\{1,3\}\} \subseteq \mathscr{P}(\{1,2,3\})$, on a $lub(A) = \{1,2,3\}$.

Exemple de treillis II

- On peut vérifier que pour tout $A \subseteq \mathscr{P}(\{1,2,3\})$, il existe une plus petite borne surpérieure et une plus grande borne inférieure dans $\mathscr{P}(\{1,2,3\})$.

Vers une définition des domaines I

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

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 $(\forall x, y, z \in S)\ x \preccurlyeq y \text{ et } y \preccurlyeq z \implies x \preccurlyeq z$.
3. $\preccurlyeq$ est antisymétrique, c'est-à-dire $(\forall x, y \in S)\ x \preccurlyeq y \text{ et } y \preccurlyeq x \implies x = y$.

Vers une définition des domaines II

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, $lub(A) \preccurlyeq m$.

Vers une définition des domaines III

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 $\emptyset$, $\{1\}$, $\{1,3\}$ et $\{1,2,3\}$ forment une chaine ascendante selon l'ordre d'inclusion :

$$\emptyset \subseteq \{1\} \subseteq \{1,3\} \subseteq \{1,2,3\}$$

Vers une définition des domaines IV

Définition (ordre partiel complet)

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

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

Dans le contexte de la sémantique dénotationnelle, $\bot$ est une valeur indéfinie pouvant dénoter l'inconnu ou la non-terminaison.

Nota : les conditions d'un ordre partiel complet (CPO) sont strictement moins fortes que pour un treillis.

Les domaines I

- 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.

Les domaines II

- 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, ...

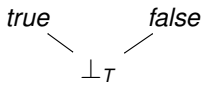
Définition (domaines discrets)

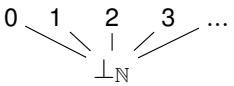
Soit l'ensemble S, on construit le domaine **S** par :

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

$$\forall x, y \in S \quad x \preccurlyeq_S y \quad ssi \; x = y \; ou \; x = \perp_S$$

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





Domaines produits I

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 :

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

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.

Domaines produits II

Définitions (fonctions de projections)

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

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 3th, 4th, ...).

Domaines sommes (unions disjointes) I

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 :

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

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

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

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.

Domaines sommes (unions disjointes) II

Définitions (fonctions d'injection)

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

$$\begin{aligned} in\mathbf{S}_1 &= \lambda a.\langle a,1\rangle \ \in \ \mathbf{A} \to \mathbf{A}\oplus\mathbf{B} \\ in\mathbf{S}_2 &= \lambda b.\langle b,2\rangle \ \in \ \mathbf{B} \to \mathbf{A}\oplus\mathbf{B} \end{aligned}$$

Domaines sommes (unions disjointes) III

Définitions (fonctions de projection)

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

Domaines sommes (unions disjointes) IV

Définitions (fonctions d'inspection)

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

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 I

- 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 (listes) de valeurs du domaine D.

où

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

Domaines séquences II

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

Domaines de fonctions I

Définition (fonction totale et partielle)

Une fonction $f : A \to B$ est *totale* si elle est telle que $f(x) \in B$ est définie pour tout $x \in A$, sinon elle est dite *partielle*.

Définition (domaines de fonctions)

Soient **A** et **B** des domaines munis d'ordres partiels complets, alors on peut définir le domaine $\mathbf{Fun}(\mathbf{A},\mathbf{B})$ des fonctions de **A** dans **B** muni de l'ordre partiel complet $\preccurlyeq_{\mathbf{Fun}(\mathbf{A},\mathbf{B})}$ tel que

$$\forall f, g \in \mathbf{Fun}(\mathbf{A},\mathbf{B}), f \preccurlyeq_{\mathbf{Fun}(\mathbf{A},\mathbf{B})} g \text{ si } f(x) \preccurlyeq_{\mathbf{B}} g(x), \forall x \in \mathbf{A}$$

Domaines de fonctions II

- Des restrictions sont nécessaires pour éviter d'admettre des fonctions dont le comportement n'est pas réalisable, comme par exemple

$$f(x) = \begin{cases} 1 & \text{si } g(x) = \bot \\ 0 & \text{sinon} \end{cases}$$

qui n'est pas calculable puisque $\bot$ représente l'indéfinition de g ou encore sa non-terminaison.

Domaines de fonctions III

Définition (fonction monotone)

Une fonction f est *monotone* si $x \preccurlyeq_{\mathbf{A}} y \implies f(x) \preccurlyeq_{\mathbf{B}} f(y), \forall x, y \in \mathbf{A}$.

Définition (fonction continue)

Une fonction f est *continue* si elle préserve les plus petites bornes supérieures, c'est-à-dire que pour toute chaîne ascendante $x_1 \preccurlyeq_{\mathbf{A}} x_2 \preccurlyeq_{\mathbf{A}} \ldots$, on a $f(lub(\{x_i, i \geq 1\}) = lub(\{f(x_i), i \geq 1\})$.

Théorème

La relation $\preccurlyeq_{\mathbf{Fun}(\mathbf{A},\mathbf{B})}$ sur l'ensemble des fonctions monotones et continues de $\mathbf{Fun}(\mathbf{A},\mathbf{B})$ est un *ordre partiel complet*.

Domaines de fonctions IV

- On peut montrer que la plus petite borne supérieure d'une chaîne ascendante de fonctions monotones et continues est une fonction monotone et continue.
- On peut aussi montrer que la composition de fonctions monotones et continues donne une fonction monotone et continue.
- Enfin, on peut montrer que les fonctions sur les domaines vus précédemment sont monotones et continues.

Retour sur les fonctions récursives

Soit la fonction

```
f(n) = if n = 0 then 5 else
         if n = 1 then f(n+2) else f(n-2)
```

- Quelle fonction peut être le dénotation de cette équation ?
- S'il y en a plus d'une, laquelle devrait-on choisir pour être *la* dénotation de cette équation ?
- Nous allons voir comment la calculer...

Définition de la fonctionnelle

Soit la fonctionnelle F calculant la factorielle telle que

$$F : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$
$$F = \lambda f.\lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } (\times\, n\, (f\, (-\, n\, 1)))$$

- Une fonction f satisfait l'équation originale si et seulement si elle est un point fixe de F, c'est-à-dire $(F\ f) = f$.
- Comment calculer cette fonction f ?

Théorème du point fixe pour les fonctions

Théorème (point fixe)

Soit $\mathbf{D}$ un domaine muni d'un ordre partiel complet $\preccurlyeq_{\mathbf{D}}$ et $g : \mathbf{D} \to \mathbf{D}$ une fonction monotone et continue sur $\mathbf{D}$, alors g a un plus petit point fixe dans $\mathbf{D}$.

Corollaire

Toute fonctionnelle continue $F : (\mathbf{A} \to \mathbf{B}) \to (\mathbf{A} \to \mathbf{B})$, où $\mathbf{A}$ et $\mathbf{B}$ sont des domaines, a un plus petit point fixe $f_{pppf} : \mathbf{A} \to \mathbf{B}$ qui peut être pris comme la dénotation de la définition récursive correspondant à F.

Calcul du plus petit point fixe I

- Le calcul du point fixe va consister à poser une définition initiale de F, puis à itérer sur la fonction obtenue en résultat pour la rendre de mieux en mieux définie...
- Pour la fonctionnelle précédente, on va calculer la chaine ascendante $\bot \preccurlyeq F(\bot) \preccurlyeq F^2(\bot) \preccurlyeq F^3(\bot) \preccurlyeq \ldots$ telle que :

Calcul du plus petit point fixe II

$$
\begin{aligned}
f_0 &= \lambda n.\bot \\
f_1 &= (F\ f_0) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } (\times\ n\ (f_0\ (-\ n\ 1)) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \bot \\
f_2 &= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \\
&\qquad \textbf{else if } n = 1 \textbf{ then } (\times\ 1\ (f_1\ (-\ 1\ 1)) \textbf{ else } (\times\ n\ (f_1\ (-\ n\ 1)) \\
&= \lambda n.\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else if } n = 1 \textbf{ then } 1 \textbf{ else } \bot \\
f_3 &= \ldots \\
\ldots &= \ldots
\end{aligned}
$$

Observation

- Conceptuellement, qu'a-t'on fait ?
- On a d'abord construit la fonction f_0 qui n'est pas récursive et qui est indéfinie partout puisqu'elle a pour résultat $\bot$ pour toute entrée.
- On a ensuite construit la fonction f_1 qui répond 5 pour $n = 0$ puis appelle f_0 les autres cas, donc indéfinie partout ailleurs.
- Et on a poursuivi pour la fonction f_2 qui sait répondre pour $n = 0$ et $n = 2$, mais appelle f_1 pour les autres cas.
- Et ainsi de suite, chaque fonction étant mieux définie que la précédente car capable de répondre pour plus de valeurs de n.
- Le point fixe étant la fonction totale f_∞ qui sait répondre pour tout n en utilisant les fonctions partielles $f_0, f_1, f_2, \ldots$ qui la précèdent dans la chaine.

Vu autrement...

	$f_0 \preccurlyeq$	$f_1 \preccurlyeq$	$f_2 \preccurlyeq$	$f_3 \preccurlyeq$	$f_4 \preccurlyeq$	...	f_∞
0	$\bot$	1	1	1	1	...	1
1	$\bot$	$\bot$	1	1	1	...	1
2	$\bot$	$\bot$	$\bot$	2	2	...	2
3	$\bot$	$\bot$	$\bot$	$\bot$	6	...	6
4	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...	24
...	...	...	...	...	...	...	...

Récapitulons les étapes et reconnectons... I

1. Définition récursive `f(...) = ... f ...`
2. Définition de la fonctionnelle $F\ f = \lambda \ldots f \ldots$
3. Calcul du plus petit point fixe de la fonctionnelle F, ce que l'on définit comme le résultat d'un opérateur

$$\textbf{fix} : (\mathbf{D} \to \mathbf{D}) \to \mathbf{D} \quad \text{où } \textbf{fix} = \lambda F.lub(\{F\ (\bot) | i \geq 0\})$$

avec la propriété

$$(F\ (\textbf{fix}\ F)) = (\textbf{fix}\ F)$$

4. **fix** est bel et bien l'opérateur Y du λ-calcul.

Récapitulons les étapes et reconnectons... II

5. En réalité, il y a plusieurs opérateurs de point fixe :

$$\textbf{fix}_1 \equiv \lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$$
$$\textbf{fix}_2 \equiv \lambda f.((\lambda x.(f\ (\lambda y.((x\ x)\ y))))\ (\lambda x.(f\ (\lambda y.((x\ x)\ y)))))$$

6. $\textbf{fix}_1$ est l'opérateur que nous avons étudié plus tôt, et qui fonctionne bien pour l'ordre normal.
7. $\textbf{fix}_2$ est un autre opérateur qui fonctionne pour l'ordre applicatif.

Revenons sur l'exemple du `while`

- Nous avons défini la dénotation du `while` de notre mini-langage impératif comme étant :
  $execute[\![\texttt{while}\ e\ i]\!]\rho\sigma = (loop\ \sigma)$
  where $loop = \textbf{fix}\ \lambda f.\lambda\sigma.$
  if $out\mathbf{T}(eval[\![e]\!]\rho\sigma)$ **then** $(f\ execute[\![i]\!]\rho\sigma)$ **else** σ
- Que représente *loop* ?

Le point fixe pour *loop* I

Si $loop_0 = \lambda\sigma.\bot$

et
$loop_{n+1} = \lambda\sigma =$ **if** $out\mathbf{T}(eval[\![e]\!]\rho\sigma)$ **then** $(loop_n\ execute[\![i]\!]\rho\sigma)$ **else** σ,

alors on peut voir intuitivement que :

- $loop_0$ représente les cas où la boucle ne s'exécute pas,
- $loop_1$ représente les cas où le test est faux dès le départ et retourne σ inchangée, ou alors n'est pas définie,

Le point fixe pour *loop* II

- $loop_2$ réprésente les cas où le test est vrai, puis où on exécute le corps de la boucle une fois pour constater que le test est maintenant faux, ou alors n'est pas définie,
- et ainsi de suite avec $loop_n$ qui représente le cas où le test est vrai $n-1$ fois puis faux, ou alors n'est pas définie.

Intuitivement, on voit que le point fixe de ce processus va bien donner la fonction $loop_\infty$ qui peut exécuter le corps de la boucle autant de fois que nécessaire pour rendre le test faux et alors s'arrêter.

On y reviendra en TD...