Un peu de sémantique

Sémantique, origines

- deux acceptions
  - expérimentale: étude de la signification dans les langues naturelles
  - théorique: étude du sens et du langage
    - philosophie, et, plus particulièrement...
    - ... logique symbolique et mathématique
- application naturelle: les langages artificiels que sont les mathématiques, la logique, puis les langages de l'informatique
- des articulations dont on hérite en informatique
  - *syntaxe* et *sémantique*
  - Gottlob Frege: *dénotation* (Sinn) et *sens* (Bedeutung) dans "Hesperus (1è étoile qui apparaît le soir) = Phosphorus (dernière le matin)", il y a
    - un fait sur le monde (en ce sens, $H = P$ équivaut à $H = H$)
    - un fait sur le langage
      (les "modes de présentation" de la dénotation)

Langues, langages, calculs

- intensionalité: on attache de l'importance à *comment* on décrit les choses, on 'garde le langage'
  - domaines d'application: langages de programmation, méthodes pour l'analyse formelle des systèmes
  - liens avec la théorie de la preuve (correspondance de Curry Howard)
- vision extensionnelle
  - "tout ça c'est des machines de Turing" (c'est Vénus)
  - sens et dénotation
    "I took a speed-reading course and was able to read War and Peace in 20 minutes. It's about Russia." – Woody Allen
  - modèles, complexité: les objectifs et les résultats ne sont pas de même nature

Langages et dénotation

- le λ-calcul est un *calcul de fonctions*
- un programme est une fonction qui calcule un résultat à partir des entrées
- étant donné du 'code', comment savoir quelle fonction il dénote?
  - exprimer le fait qu'à la compilation on préserve le sens
  - dire que certaines fonctions ne sont pas programmables dans tel ou tel langage

Sémantique dénotationnelle

Raisonner sur des programmes impératifs

- on s'intéresse à un petit langage impératif
  - c: commandes (programmes)
    $X{:=}a \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid c_1;c_2 \mid \texttt{skip} \mid \texttt{while } b \texttt{ do } c$
  - $X, Y, \ldots$: variables *entières* (emplacements en mémoire)
  - a: expressions arithmétiques $X, 32, a_1 + a_2, \ldots$
  - b: expressions booléennes $\texttt{true}, \texttt{false}, b_1 \wedge b_2, X > a, \ldots$
- exemples de programmes
  - $X{:=}0;Y{:=}1;Z{:=}1;$
    $\texttt{while } Y \leq n \texttt{ do } (X{:=}X+1;Z{:=}Z+1;Y{:=}Y+Z)$
  - `while true do skip`

Les programmes sont des fonctions

- un programme impératif est une *fonction* qui fait passer d'un état mémoire à un autre état mémoire
- on associe à un programme c (donné par son arbre de syntaxe) une *dénotation*, notée $[\![c]\!]$
  - $[\![c]\!]$ habite dans un espace mathématique, c'est une fonction
  - $[\![c]\!](\sigma) = \sigma'$ si c fait passer de l'état mémoire σ à l'état mémoire σ'
  - qu'est-ce qu'un état mémoire σ?
    - tuple, fonction partielle des variables vers les entiers, ...
- ainsi, à `skip` va correspondre l'identité
  $[\![\texttt{skip}]\!](\sigma) = \sigma$

Les programmes sont des fonctions (2)

- à $\texttt{if } b \texttt{ then } c1 \texttt{ else } c2$ est associée la fonction
  $[\![\texttt{if } b \texttt{ then } c1 \texttt{ else } c2]\!](\sigma) =$
  si b vaut true dans σ, alors $[\![c1]\!](\sigma)$, sinon $[\![c2]\!](\sigma)$
- $[\![c1;c2]\!](\sigma) = [\![c2]\!]([\![c1]\!](\sigma))$
- $[\![X{:=}a]\!](\sigma) = \texttt{let } v = \texttt{eval}(\sigma, a) \texttt{ in } \sigma_{\{X \leftarrow v\}}$
- $[\![\texttt{if } X > 0 \texttt{ then skip else while true do skip}]\!](\sigma) =$
  si $X > 0$, on renvoie σ, sinon on renvoie $\perp$

  $\perp$: *divergence*, la fonction n'est pas définie
  (on manipule des fonctions partielles)

Sémantique dénotationnelle du `while`

- considérons le programme
  w $\stackrel{def}{=}$ `while` $X > 0$ `do` $(Y{:=}X * Y; X{:=}X - 1)$
- ce programme ne manipule que les variables X et Y, on va donc représenter une configuration de la mémoire par un couple (x, y)
- si on passe zéro fois dans le corps de la boucle (i.e., au départ, $X \leq 0$), on peut poser $[\![w]\!] = f_0$, avec

$$f_0(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ \bot & \text{si } x > 0 \end{cases}$$

Une suite de fonctions

$$f_0(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ \bot & \text{si } x \geq 1 \end{cases} \qquad f_1(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ (0,y) & \text{si } x = 1 \\ \bot & \text{si } x \geq 2 \end{cases}$$

$$f_2(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ (0,y) & \text{si } x = 1 \\ (0,2*y) & \text{si } x = 2 \\ \bot & \text{si } x \geq 3 \end{cases} \qquad f_3(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ (0,y) & \text{si } x = 1 \\ (0,2*y) & \text{si } x = 2 \\ (0,6*y) & \text{si } x = 3 \\ \bot & \text{si } x \geq 4 \end{cases}$$

$$f_{n-1}(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ (0,(x!)*y) & \text{si } 0 < x < n \\ \bot & \text{si } x \geq n \end{cases}$$

- f_n est la sémantique de w si on sait que l'exécution fait au plus n tours dans la boucle
  lorsque n croît, on a une fonction 'de plus en plus définie'
- on a envie de dire que la sémantique de w est la *limite* des f_is, définie ici par $f_\infty(x,y) = \begin{cases} (x,y) & \text{si } x \leq 0 \\ (0,(x!)*y) & \text{si } x > 0 \end{cases}$

Des limites de suites de fonctions

- on peut voir les f_is comme des *approximations* de f_∞
  - dans quel espace (mathématique) se promènent les f_is, et en quel sens l'approximation s'améliore-t-elle lorsque i croît?
  - pourquoi la suite des f_i admet f_∞ comme limite?
- Rq: on ne peut pas toujours écrire la définition de f_∞, même si on peut affirmer qu'une telle fonction existe
- c'est quoi la sémantique de `while true do skip` ?

Et le λ-calcul?

- sémantique dénotationnelle de ML
  - mêmes principes, le `let rec` joue le rôle du while
    (sans `let rec`, tous les programmes terminent)
- P. Wadler: "on a *inventé* C, alors qu'on a *découvert* ML"
  (à peu près)
- λ-calcul pur: un espace où tout objet est en même temps une fonction

Logique de Hoare

Assertions

- on va écrire des propriétés 'logiques' sur l'état de la mémoire $X = 52$, $(X > 0) \Rightarrow (Y + Z = 5)$, $\neg(X = 1 \wedge Y = 1)$,...
- on s'intéresse à raisonner sur les programmes en indiquant ce qu'ils font sur la mémoire

Triplets de Hoare

- notation: $\{P\}c\{P'\}$
  - P: précondition
  - c: commande
  - P': postcondition
- interprétation

  partant d'un état mémoire satisfaisant P, si on exécute la commande c **et si cette exécution termine**, *on aboutit à un état mémoire satisfaisant P'*
- un triplet est dit *valide* si son interprétation est toujours vérifiée
- exemples
  - $\{X = 9 \wedge Y = 11\} X := 100 * X + Y \{X = 911 \wedge Y = 11\}$
  - $\{X = 33\}$`while true do skip`$\{$les poules ont des dents$\}$

    avec "les poules ont des dents"=`false`

$\{n \geq 0\}$ `X :=0;Y :=1;Z :=1;` `while` $Y \leq n$ `do` $(X{:=}X+1; Z{:=}Z+1; Y{:=}Y+Z)$ $\{X^2 \leq n < (X+1)^2\}$

Déduction

- on peut adopter la voie pédestre
  - pour montrer $\{X = 9 \wedge Y = 11\} X := 100 * X + Y \{X = 911 \wedge Y = 11\}$, on dit "soit un état mémoire satisfaisant ...; alors en exécutant...; finalement, on aboutit bien à un état mémoire satisfaisant..."
- on peut aussi imaginer des *règles de déduction* qui ne permettent de construire que des triplets valides
  - exemple: on a toujours $\{P\}$`skip`$\{P\}$ *(pour tout P)*
  - pour $c_1;c_2$?

    si on sait que $\{P\}c_1\{P_0\}$ est valide, et que $\{P_0\}c_2\{P'\}$ est valide, on peut déduire que $\{P\}c_1;c_2\{P'\}$ est valide

    ce que l'on note $\dfrac{\{P\}c_1\{P_0\} \qquad \{P_0\}c_2\{P'\}}{\{P\}c_1;c_2\{P'\}}$
  - pour `if` b `then` c_1 `else` c_2 ?

$$\frac{\{P \wedge b\}c_1\{P'\} \qquad \{P \wedge \neg b\}c_2\{P'\}}{\{P\}\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\{P'\}}$$

Règles moins immédiates

- pour $X := a$? $\quad \{P_{\{X \leftarrow a\}}\}\ X := a\ \{P\}$ *(eh oui, ça va à l'envers)*
- pour `while` b `do` c? $\quad \dfrac{\{P \wedge b\}c\{P\}}{\{P\}\texttt{while}\ b\ \texttt{do}\ c\{P \wedge \neg b\}}$
- utilisation des règles de déduction
  - il s'agit d'emboîter les utilisations des règles de déduction de manière à construire des *arbres de dérivation*
  - aux feuilles *(en haut)*, les axiomes *affectation* et `skip`
  - à la racine, le triplet que l'on veut prouver

Règles de déduction, suite

$$\{P\}\texttt{skip}\{P\} \qquad \frac{\{P\}c_1\{P'\} \quad \{P'\}c_2\{P''\}}{\{P\}c_1;c_2\{P''\}} \qquad \{P_{\{X \leftarrow a\}}\}X:=a\{P\}$$

$$\frac{\{P \wedge b\}c_1\{P'\} \quad \{P \wedge \neg b\}c_2\{P'\}}{\{P\}\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\{P'\}} \qquad \frac{\{P \wedge b\}c\{P\}}{\{P\}\texttt{while}\ b\ \texttt{do}\ c\{P \wedge \neg b\}}$$

(comparer avec λ-calcul simplement typé)

- montrons par exemple $\{X \geq 2\}Y:=X*X; Y:=Y+1\{Y \geq 5\}$
- il manque la règle suivante:

$$\frac{\boxed{P_1 \Rightarrow P} \quad \{P\}c\{P'\} \quad \boxed{P' \Rightarrow P_2}}{\{P_1\}c\{P_2\}}$$

$\boxed{P \Rightarrow Q}$: *"tout état mémoire satisfaisant P satisfait Q"*

- ...et on a donné toutes les règles

Propriétés du système de règles d'inférence

- c'est quoi la correction et la complétude du système de règles d'inférence?
  - correction: tout triplet construit à l'aide des règles d'inférence est valide
  - complétude: tout triplet valide peut être construit à l'aide des règles d'inférence
- apprenez la vie:
  - preuve de la correction
    - chaque règle d'inférence préserve la validité des triplets
  - preuve de la complétude: bien plus astucieux *(c'est la vie)* esquisse:
    - si on sait que $\{P\}c\{P'\}$ est valide, on raisonne à l'envers en cherchant une condtion suffisante P_0 pour qu'on puisse garantir $\{P_0\}c\{P'\}$
    - on s'est donc ramené à prouver que l'on a $\boxed{P \Rightarrow P_0}$
- on appelle cela parfois la *sémantique axiomatique*
  - les règles donnent la "signification" des constructions du langage

Terminaison des boucles

- on peut s'intéresser à d'autres formes de triplets

$$[P]\,c\,[Q]\,,$$

que l'on interprète *"lancé sur un état mémoire qui satisfait P,* **le programme c termine** *sur un état qui satisfait Q"*

- les règles sont les mêmes, sauf pour `while`
  - on fait intervenir un *variant* e, ≥ 0 à chaque entrée dans le corps de la boucle, et qui y décroit strictement
  - difficulté: les assertions portent sur *un* état, et pas sur une relation entre des états successifs dans le temps
  - on introduit une variable fantôme X_0 *(ghost variable)* ainsi:

$$\frac{[P \wedge b \wedge e = X_0]\,c\,[P \wedge e < X_0] \qquad \boxed{P \wedge b \Rightarrow e \geq 0}}{[P]\,\texttt{while}\ b\ \texttt{do}\ c\,[P \wedge \neg b]}\ X_0 \text{ fraîche}$$