INF 544 Luc Maranget

Termes et relations pour la sémantique

Luc.Maranget@inria.fr

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/

Objectif

La sémantique ?

- Comprendre ce que fait un programme.
  - Pour programmer.
  - Et à plus long terme...
    - ★ Très utile pour implémenter un langage (interpréteurs, compilateurs, analyseurs).
    - ★ Indispensable pour prouver quoique ce soit au sujet d'un programme.
- Et au passage,
  - Améliorer sa connaissance de la programmation,
  - (Re-)Découvrir Caml.
  - Aborder les mathématiques utiles pour la sémantique.

Moyens

- Deux enseignants, Luc Maranget et David Baelde.
- Neuf séances, le mardi,
  - Cours[a] de 9h00 à 10h30, ici.
  - TP[b] de 10h30 à 12h30 en salle 35.
- Examen (écrit — 3 heures), le 4 décembre.
- Les cours préalables de Caml (mais ça c'est fait bien sûr).

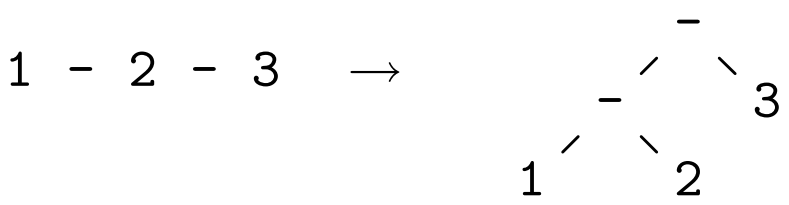
---

[a] `http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/`

[b] `http://www.enseignement.polytechnique.fr/profs/informatique/David.Baelde/INF544/`

Comment définir un langage de programmation ?

- Une syntaxe, c'est du déjà-vu
  - Spécifie les programme bien formés, « **fun** `x -> x * x` » est une expression bien formée (en Caml), mais pas « `+(` ».
  - Outil de base pour définir la syntaxe : grammaires formelles (INF431, poly Chap. 10–12).
  - Un point important : l'analyse syntaxique transforme les fichiers (suites de caractères) en *arbres de syntaxe abstraite*.



- Une sémantique c'est nouveau.

Sémantique ?

Savoir que « *My taylor is rich* » est bien de l'anglais ne suffit pas.

Savoir que « **fun** `x -> x * x` » est bien du Caml ne suffit pas.
On veut en plus savoir ce qui se passe quand on applique ce programme (cette fonction) à une entrée.

$$\texttt{(}\textbf{fun}\ \texttt{x -> x * x)}, \texttt{4} \hookrightarrow \texttt{16}$$

Définir la sémantique c'est définir une relation $P, e \hookrightarrow v$, où P est un programme (ici syntaxe abstraite), e est une entrée (ici une valeur) et v le résultat (ici une valeur).

Si à P et e donnés correspond un unique v, on définit une fonction.

Mais comment définir $\hookrightarrow$? langue naturelle ou mathématiques ?

Connaître la sémantique pour programmer

Une évidence : pour bien programmer il faut savoir ce que font les programmes.

Or ce n'est pas toujours évident,

- Les ordinateurs ont un comportement très précis.
- Les êtres humains sont mal à l'aise devant une telle précision (non-dits, recherche de l'abstraction).

Pourtant on essaie quand même de spécifier le comportement des programmes en langue dit naturelle (français, anglais. . .).

Un exemple de programme

```
class Test {

   static int n = 0 ;

   static int f(int x, int y) { return x ; }

   static int g(int z) { n = n + z ; return n ; }

   public static void main(String [] arg) {
     System.out.println(f(g(2), g(7))) ;
   }
}
```

Quel est l'affichage ? « 2 », car en Java les arguments sont évalués de la gauche vers la droite, comme précisé dans la définition du langage.

Autre exemple

```
let n = ref 0

let f x y = x

let g z = n := !n + z ; !n

let () = Printf.printf "%i\n" (f (g 2) (g 7))
```

Si on essaie (ranger dans `test.ml` et exécuter `ocaml test.ml`), on obtient `9`.

Ici attention, la documentation de OCaml précise que l'ordre d'évaluation des arguments n'est pas spécifié.

Autre exemple pertinent

```
let f x = let g x = x in g

let x = 3

let () = Printf.printf "%i\n" (f 1 2)
```

Le programme affiche : « 2 ».

On comprend un peu mieux en réécrivant :

```
let f y z = z

let x = 3

let () = Printf.printf "%i\n" (f 1 2)
```

Outils dont les entrées sont des programmes

Ce sont :

les interpréteurs, les compilateurs, les analyseurs de programmes...

La sémantique est indispensable pour...

- Mettre les auteurs de compilateurs/interprètes/analyseurs etc. d'accord entre eux et avec les programmeurs.
- Respecter la sémantique des programmes lors des optimisations.
- Prouver des propriétés des programmes rigoureusement, voire automatiquement.
- Prouver les compilateurs/interprètes/analyseurs etc.

Exemple de transformation de programme

Soit la fonction `f` :

```
let f x y = x
```

Peut-on « optimiser » tout appel `f` e_1 e_2 en e_1 ?

Non, car par exemple

```
let z = f 1 (Printf.printf "%i\n" 2 ; 2)
```

Mais on peut transformer :

`f` e_1 e_2 $\Rightarrow$ **let** `x =` e_1 **and** `y =` e_2 **in** `x`

La langue naturelle

- Avantages :
  - Tout le monde connaît (ou croit connaître).
- Insuffisances :
  - C'est dur d'être vraiment précis.
  - Et alors, c'est souvent illisible, il faut une « culture » mal définie.
  - Impossible de faire des démonstrations de théorèmes et encore moins de les automatiser.

Un exemple de sémantique en langue naturelle

The `while` statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is false.

```
WhileStatement:
        while ( Expression ) Statement
```

A `while` statement is executed by first evaluating the *Expression*. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `while` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is true, then the contained *Statement* is executed. Then there is a choice:
  - If execution of the *Statement* completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the *Expression*.
  - If execution of the *Statement* completes abruptly, see §14.12.1 below.
- If the (possibly unboxed) value of the *Expression* is false, no further action is taken and the `while` statement completes normally.

If the (possibly unboxed) value of the *Expression* is false the first time it is evaluated, then the *Statement* is not executed.

La suite dans la documentation de Java.

Sémantique formelle

Dans ce cours...

- Nous présentons les outils qui permettent de définir la sémantique d'un langage de programmation.
- Nous illustrons l'usage de ces outils en spécifiant quelques traits des langages de programmation.
- Nous écrivons des programmes (interpréteur, évaluateur, vérificateur de type) dirrectement inspirés des définitions sémantiques.

Relations

Définition d'un ensemble (d'une relation)

- Ensembles déjà connus ! $\mathbb{N}$, **type** `t = int`.
- Constructions déjà connues : $\mathbb{N}^2$, **type** `t = int * int`.
- Définition explicite :

  $\{x \in \mathbb{N} \mid \exists z \in \mathbb{N}, x = 2 \times x\}$ $\quad$ $\{(x, y) \in \mathbb{N}^2 \mid \exists z \in \mathbb{N}, x = y \times z\}$

  (Note : une relation est un sous-ensembles de paires.)
- Et définition inductives...

  type `intlist = Nil | Cons` **of** `int * intlist`

Premier théorème du point fixe

- E, $\leq$ faiblement complète :
  - $\leq$ reflexive, antisymétrique, transitive.
  - Toute suite croissante u_0, u_1, ..., a une borne supérieure, notée $\lim u_i$.
- E admet un minimum (noté disons $\bot$).
- Soit f fonction croissante et continue : $f(\lim u_i) = \lim f(u_i)$.
- **Théorème :** Il existe $z \in E$, tq. $f(z) = z$ (point fixe).

Preuve : Considérer $\lim f^i(\bot)$, c'est le plus petit point fixe de f.

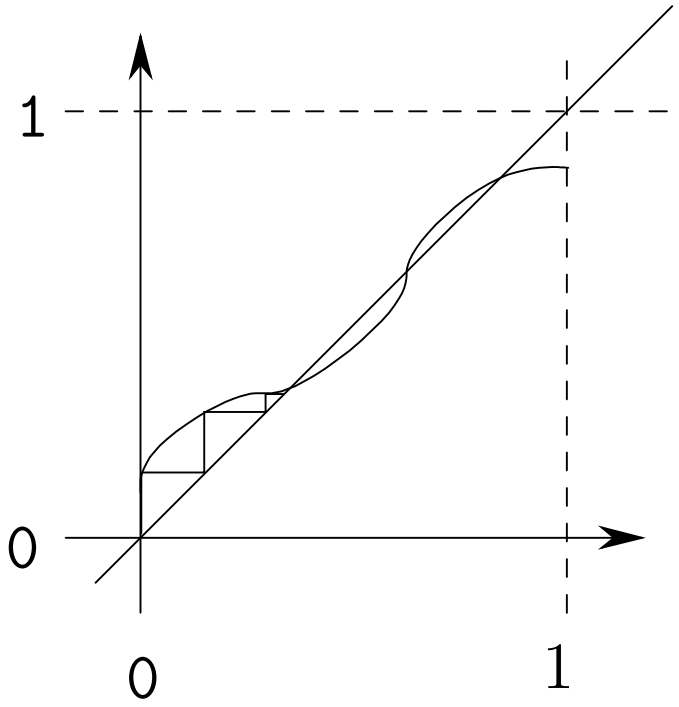
Preuve

Soit $u_i = f^i(\bot)$.

- (u_i) est croissante ($u_0 = \bot \leq f(\bot) = u_1$ et
  $u_i \leq u_{i+1} \Rightarrow u_{i+1} = f(u_i) \leq f(u_{i+1}) = u_{i+2}$).
  Soit p sa limite.
- p est aussi la limite de $(v_i)) = f^{i+1}(\bot)$.
- Donc $p = \lim v_i = \lim f(u_i) = f(\lim u_i) = f(p)$ (continuité).
- Et pour tout point fixe q,
  - $\bot \leq q \Rightarrow u_i \leq f^i(q)$ (croissance).
  - Et donc $p = \lim u_i \leq \lim f^i(q) = q$ (def. de la borne supérieure).

Graphiquement

$[0, 1]$, $\leq$ est faiblement complète.



$\mathbb{R}^+$ faiblement complète ?

Second théoreme du point fixe

- $E, \leq$ fortement complète :
  - Tout sous-ensemble A a une borne sup.
  - Donc : tout sous-ensemble a une borne inf — Considérer la borne sup de l'ensemble des minorants de A.
- Soit f croissante.
- **Théorème :** f a un point fixe.

Preuve : la borne inférieure de $C = \{c \mid f(c) \leq c\}$ est le plus petit point fixe de f.

Preuve

Soit p borne inférieure des « contractés » $C = \{c \in E \mid f(c) \leq c\}$.

- Pour tout $c \in C$,
  - On a $p \leq c$ (p minorant de C).
  - Et donc $f(p) \leq f(c)$ (croissance).
  - Et encore $f(p) \leq c$ (transitivité).
  - Autrement dit, $f(p)$ minorant de C.

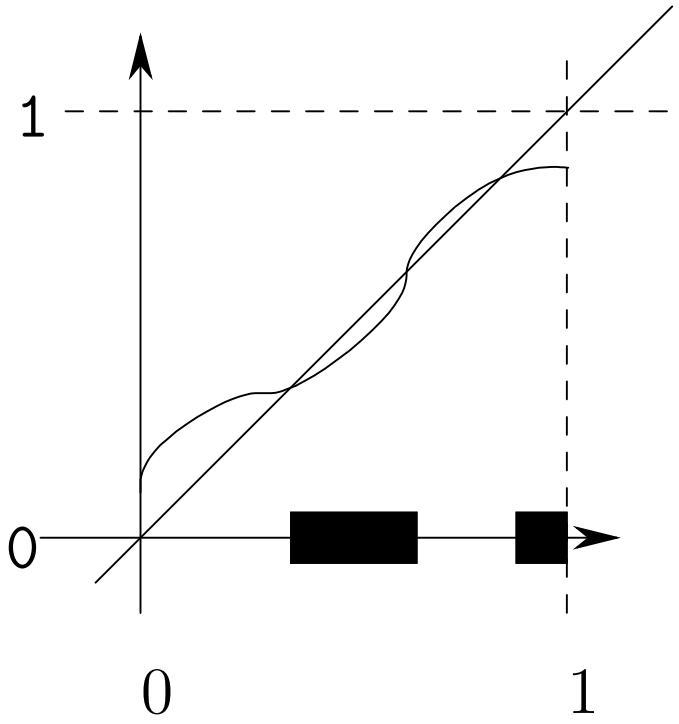
  Soit finalement $f(p) \leq p$ (car p borne inf de C).
- Et dans l'autre sens,
  - $f(f(p)) \leq f(p)$ (croissance).
  - Soit $f(p) \in C$.

  Soit finalement $p \leq f(p)$ (p minorant).
- Par antisymétrie, $p = f(p)$.

  Plus petit point fixe, car tous les points fixes sont dans C.

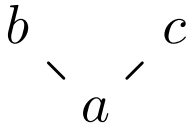
Graphiquement

$[0,1]$, $\leq$ est fortement complète.

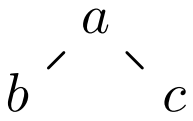


$\mathbb{R}^+$ fortement complète ?

Quelques exemples



- Faiblement complet ? Oui (ensemble fini).
- Fortement complet ? Non ? $\{b, c\}$ n'a pas de borne supérieure.



- Faiblement complet ? Oui.
- Fortement complet ? Non ? $\emptyset$ n'a pas de borne supérieure (et $\{b, c\}$ pas de borne inférieure).

Exemple encore (qui va servir)

- E ensemble quelconque, 2^E ensemble des parties de E.
- 2^E, $\subseteq$ faiblement et fortement complet.
  - Élément minimal : $\emptyset$.
  - Borne sup de la collection $\mathcal{E}$? $\cup_{E \in \mathcal{E}} E$.
  - Borne inf de la collection $\mathcal{E}$? $\cap_{E \in \mathcal{E}} E$.
- $f : 2^E \to 2^E$ croissante a un point fixe.
  - Plus petit point fixe : $\bigcap_{C | f(C) \subseteq C} C$.
  - Et si f continue : $\bigcup_{i \in \mathbb{N}} f^i(\emptyset)$.

Définition inductive par l'exemple

P ensemble des entiers pairs peut être défini par les règles suivantes.

$$0 \in P \qquad\qquad (n \in P \Rightarrow n+2 \in P)$$

Que l'on note plutôt.

$$\frac{}{0 \in P} \qquad \frac{n \in P}{n+2 \in P} \qquad \text{ou même} \qquad \frac{}{0} \qquad \frac{n}{n+2}$$

Autrement dit, P contient 0 et P est clos par la fonction $n \mapsto n+2$.

Il est remarquable que P n'est pas le seul ensemble satisfaisant ces deux propriétés, par ex. $\mathbb{N}$ convient.

Alors ? P est le plus petit de ces ensembles.

Vérifions

Soient les fonctions $f_1 : \mathbb{N}^0 \to \mathbb{N}$, qui à () associe 0 ; et $f_2 : \mathbb{N} \to \mathbb{N}$, qui à x associe $x + 2$.

Soit ensuite $F : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ qui à X associe :

$$\{y \mid y = f_1()\} \cup \{y \mid \exists x \in X, f_2(x) = y\}$$

Soit si on veut :

$$F(X) = \{\, 0 \,\} \cup \{x + 2 \mid x \in X\}$$

F est...

- Croissante, immédiat $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$).
- Continue, à peine plus dur $F(\bigcup_{i \in N} X_i) = \bigcup_{i \in \mathbb{N}} F(X_i)$.
  - 0 présent dans les deux ensembles.
  - Sinon $x = y + 2$ avec $y \in X_i$ pour i donné.

Et selon les théorèmes

Les deux règles définissent donc un ensemble d'entiers P, qui est le plus petit point fixe de F.

- C'est la réunion de $\emptyset, F(\emptyset), F(F(\emptyset))$, soit $\{0, 2, 4, \ldots\}$.
- C'est aussi l'intersection de tous les ensembles d'entiers contenant zéro et clos par $n \mapsto n + 2$.

Et au passage P est l'ensemble des entiers pairs ($2\mathbb{N}$), en effet.

- $2\mathbb{N}$ est un point fixe de F, et donc $P \subseteq 2N$.
- Si $n = 2k$, alors $n \in F^k(\emptyset)$ et donc $n \in P$ (premier th.). Soit finalement $2\mathbb{N} \subseteq P$.

Mais ce qui compte c'est que nous avons pu définir un ensemble (d'entiers) inductivement.

Cas général (poly)

Pour tout ensemble E. On peut définir un sous-ensemble A de E par des fonctions f_k (eventuellement partielles) de E^{n_k} dans E, souvent notées comme les règles :

$$\frac{x_1 \cdots x_{n_k}}{f_k(x_1, \ldots, x_{n_k})}$$

Ce sous-ensemble est le plus petit point fixe de la fonction F de 2^E dans 2^E.

$$F(X) = \bigcup_{k \in K} \{f_k(x_1, \ldots, x_{n_k}) \mid x_1, \ldots, x_{n_k} \in X^{n_k}\}$$

C'est aussi le plus petit ensemble clos par les règles.

Cela permet en particulier de définir des relations, qui sont des ensembles (de paires par ex.)

Exemple de relation définie inductivement

Soit Σ alphabet (ensemble de caractères notés a, b)

Soit $\Sigma^\star$ ensemble des mots (suites finies sur Σ, notés m, n).

Relation de facteur $m \preceq n$

$$\frac{}{\epsilon \preceq m} \qquad \frac{}{m \preceq m} \qquad \frac{m_1 \preceq n_1 \qquad m_2 \preceq n_2}{m_1 m_2 \preceq n_1 n_2}$$

$$\frac{m \preceq n}{m \preceq n n'} \qquad \frac{m \preceq n}{m \preceq n' n}$$

Dérivation

Une *dérivation* ou arbre de preuve est un arbre dont les nœuds sont l'application des règles (les f_k).

L'existence d'une dérivation aboutissant en e prouve que e est dans $F^i(\emptyset)$ (et donc dans le plus petit point fixe).

Par ex, $\texttt{ac} \preceq \texttt{abcde}$.

$$\dfrac{\dfrac{\texttt{a} \preceq \texttt{a} \qquad \dfrac{\dfrac{\texttt{c} \preceq \texttt{c}}{\texttt{c} \preceq \texttt{cde}}}{\texttt{c} \preceq \texttt{bcde}}}{\texttt{ac} \preceq \texttt{abcde}}}$$

Réciproquement, tout élément du plus petit point fixe est l'aboutissement d'une dérivation (récurrence sur i).

Induction structurelle

Pour prouver une propriété $\mathcal{P}$ des éléments du plus petit point fixe.

Il peut suffire de montrer que $\mathcal{P}$ est *héréditaire*, c'est à dire $\mathcal{P}(x_1)$, $\ldots \mathcal{P}(x_{n_k})$ entraîne $\mathcal{P}(f_k(x_1, \ldots, x_{n_k}))$.

Preuve L'ensemble $\{e \in E \mid \mathcal{P}(e)\}$ est clos par F et second théorème.

Par ex. pour montrer que si m est un facteur de n, alors m est plus court que n :

$$\ell(\epsilon) \leq \ell(m) \qquad\qquad \ell(m) \leq \ell(m)$$

$$\ell(m_1) \leq \ell(n_1) \wedge \ell(m_2) \leq \ell(n_2) \Rightarrow \ell(m_1 m_2) \leq \ell(n_1 n_2)$$

$$\ell(m) \leq \ell(n) \Rightarrow \ell(m) \leq \ell(nn')$$

Conséquence directe des propriétés de la longueur $\ell(\epsilon) = 0$ et $\ell(m_1 m_2) = \ell(m_1) + \ell(m_2)$.

Autre exemple d'induction structurelle

Nous avons enfoncé une porte ouverte en prouvant que les règles

$$0 \qquad\qquad \frac{n}{n+2}$$

définissent bien l'ensemble des entiers pairs.

En effet, si $n \in P$ alors n est pair (induction structurelle).
Réciproquement, pour tout $n = 2k$, il existe une dérivation.

On pourrait de même enfoncer une porte (un peu moins) ouverte en démontrant que $m \preceq n$, ssi m est une sous-suite de n au sens usuel.

Un dernier exemple

Soit une relation $\mathcal{R}$, binaire, sur un ensemble E (donc un sous-ensemble du produit cartésien $E \times E$).

On définit $\mathcal{R}^*$, la fermeture reflexive et transitive de $\mathcal{R}$ inductivement :

$$\frac{}{e\ \mathcal{R}^*\ e} \qquad \frac{e\ \mathcal{R}\ e'}{e\ \mathcal{R}^*\ e} \qquad \frac{e_1\ \mathcal{R}^*\ e_2 \qquad e_2\ \mathcal{R}^*\ e_3}{e_1\ \mathcal{R}^*\ e_3}$$

Il est assez immédiat que

$$e\ \mathcal{R}^*\ e'$$

$$\Updownarrow$$

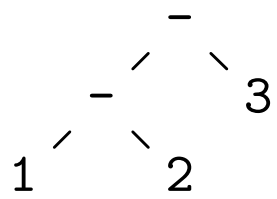
$$\exists\, n \in \mathbb{N},\ \exists\, (e_1, \ldots, e_{n-1}) \in E^{n-1},\ e = e_0\ \mathcal{R}\ e_1\ \mathcal{R} \cdots \mathcal{R}\ e_{n-1}\ \mathcal{R}\ e_n = e'$$

(Induction structurelle.)

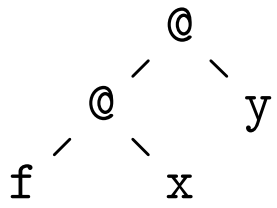
Termes

Insistons

Nous avons déjà dit que l'expression (Caml par ex.) « `1-2-3` » est pour nous le terme :



De même, nous lisons « `f x y` », nous comprenons (application explicitée) :



Mais sommes nous bien sûrs de savoir ce qu'est un terme ?

Termes sans variables

- Soit un ensemble de symboles Σ (dit signature), $\mathtt{f}, \mathtt{g}, \ldots$
- Chaque symbole est muni d'un entier, son arité, noté $n_{\mathtt{f}}$.
- **Vocabulaire :** Si `f` est d'arité zéro, c'est une constante.
- L'ensemble $\mathcal{T}(\Sigma)$ des termes définis sur la signature Σ est l'ensemble d'arbres défini inductivement par les règles :

$$\frac{t_1 \qquad t_{n_{\mathtt{f}}}}{\mathtt{f}(t_1, \ldots, t_{n_{\mathtt{f}}})}$$

  (Une règle par `f` de la signature.)
- Où est l'arnaque ? Qu'est qu'un arbre ?
- Néanmoins il semble admissible (premier théorème) qu'il s'agit de l'ensemble des arbres finis qui respectent l'arité.
- **Notation :** On simplifie systématiquement `f()` en `f`.

Exemple : expressions arithmétiques

- Une infinité de constantes `0`, `1`, `2`, ...
- Quatre symboles binaires `+`, `-`, `*`, `/`.

Et les expressions arithmétiques sont définies.

Une définition à peu près équivalente en Caml.

```
type t =
| Add of t * t | Sub of t * t | Mul of t * t | Div of t * t
| Int of int (* à peu près une infinité *)

let e = Sub (Sub (Int 1, Int 2), Int 3)
```

Quand on veut écrire une expression arithmétique sans dessiner un arbre, il est préférable de donner (presque) toutes les parenthèses.

$t =$ `-(-(1,2),3)` ou encore $t =$ `(1-2)-3`

Sémantique

Définir une sémantique des expressions arithmétiques, c'est définir une relation $e \hookrightarrow n$ entre les expressions et les entiers.

Nous donnons trois techniques.

- Opérationelle à grand pas, ou naturelle : definir une relation $\hookrightarrow$ inductivement sur la structure des expressions.
- Opérationnelle à petit pas :
  - Définir une étape de calcul élémentaire $\rightarrow$.
  - Définir $\hookrightarrow$ comme $\rightarrow^*$.
- Dénotationnelle : ici peu pertinent, nous y reviendrons.

Sémantique naturelle

Au symbole `n` on associe l'entier n, au symbole `+` on associe l'addition (fonction $\mathbb{N}^2$ dans $\mathbb{N}$, notée $+$)

$$\frac{}{\texttt{n} \hookrightarrow n} \qquad \frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{e_1 \texttt{+} e_2 \hookrightarrow n_1 + n_2} \qquad \ldots$$

Une trivialité, mais

```
let rec eval e = match e with
| Int n -> n (* NB: n et n différents *)
| Add (e1,e2) -> (* NB:  + et + différents *)
   let n1 = eval e1 and n2 = eval e2 in
   n1+n2
  ...
```

Certes la présentation par règles d'inférences est plus neutre.

Sémantique à petit pas

On définit donc la relation $\rightarrow$ sur les expressions, par une infinité d'axiomes :

$$\texttt{0+0} \rightarrow \texttt{0} \qquad \texttt{0+1} \rightarrow \texttt{1} \qquad \texttt{1+1} \rightarrow \texttt{2} \qquad \cdots$$

Puis par huit règles d'inférence (qui ici définissent une congruence : ou peut réduire tout sous-terme).

$$\frac{e_1 \rightarrow e_1'}{e_1 \texttt{+} e_2 \rightarrow e_1' \texttt{+} e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1 \texttt{+} e_2 \rightarrow e_1 \texttt{+} e_2'} \qquad \cdots$$

Et on pose $\hookrightarrow = \rightarrow^*$.

Note : Les valeurs sont maintenant nécessairement un sous-ensemble des expressions.

Sémantique dénotationelle

C'est une fonction des expressions dans les entiers, définie explicitement.

$$[\![\texttt{n}]\!] = n \qquad\qquad [\![e_1\texttt{+}e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

Les différences avec la sémantique naturelle n'apparaissent que sur des exemples plus compliqués.

Fun `x -> x+1`

Est la « vraie » fonction sur $\mathbb{N}$, qui à x associe le successeur de x.

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1)
```

Est la factorielle : $0 \mapsto 1$, $1 \mapsto 1 \times 1$, $2 \mapsto 2 \times 1 \times 1$,
$\ldots n \mapsto n \times (n-1) \times \cdots \times 1 \times 1, \ldots$

Les variables

Comment donner une valeur à par exemple « `x+x` » ?

Simple ! Il suffit de remplacer `x` par un terme fixé. C'est la *substitution* $[e\backslash\texttt{x}]$.

Un premier exemple simple : expressions arithmetiques + variables.

$$[e\backslash x]x = e \qquad\qquad [e\backslash x]y = y (y \neq x)$$

$$[e\backslash x](e_1 \ + \ e_2) = ([e\backslash x]e_1) \ + \ ([e\backslash x]e_2) \qquad\qquad \cdots$$

Et donc $[\texttt{1+1}\backslash\texttt{x}](\texttt{x+x}) = \texttt{(1+1)+(1+1)}$.

Noter : « = » et non « », c;est une *meta-opération* (*i.e.* $[e_x\backslash x]e$ ne fait pas partie de la syntaxe des termes).

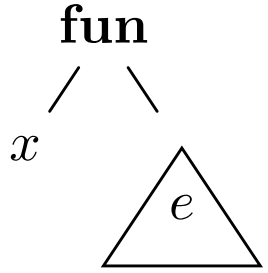
Un langage pour les fonctions

Nous avons déjà écrit par exemple $n \mapsto n + 2$.

En Caml on écrit : « **fun** `n -> n+2` ».

Il est remarquable que $m \mapsto m + 2$ et **fun** `m -> m+2` désignent la même fonction : la variable est *muette*.

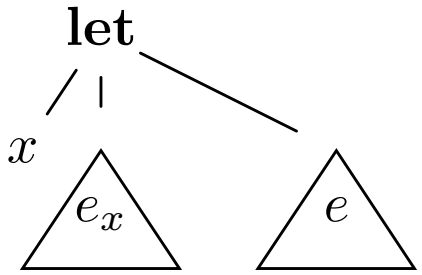
L'expression **fun** x `->` e, où x est une variable (ci-dessus `n` ou `m`) et e est une expression (ci-dessus `n+2` ou `m+2`), est à comprendre comme l'arbre :



Vocabulaire : On dit que la construction **fun** *lie* la variable x dans la sous-expression e. L'occurence de x dans e est ici *libre*.

Autres constructions liantes

Un autre exemple (Caml) est « **let** x **=** e_x **in** e », à lire comme :



où x est liée dans le sous-terme e.

Considérer aussi : « **let** **rec** x **=** e_x **in** e », où x est liée dans e_x et e.

Expressions arithmétiques + variables

Une expression e est :

- Une variable,
- Ou une constante entière $\{\texttt{0}, \texttt{1}, \ldots\}$.
- L'application d'une opération e_1`+`e_2 etc.
- Une liaison **let** $x = e_x$ **in** e.

Dans **let** $x = e_x$ **in** e, x est liée dans e.

```
type var = ...

type t =
| Add of t * t | Sub of t * t | Mul of t * t | Div of t * t
| Int of int
| Var of var
| Let of var * t * t
```

Variables d'une expression

Définies par induction sur la structure des expressions.

$$\mathcal{V}(\mathtt{0}) = \emptyset \qquad \cdots \qquad \mathcal{V}(x) = \{x\} \qquad \mathcal{V}(e_1 \mathtt{+} e_2) = \mathcal{V}(e_1) \cup \mathcal{V}(e_2)$$

$$\mathcal{V}(\textbf{let } x = e_x \textbf{ in } e) = \{x\} \cup \mathcal{V}(e_x) \cup \mathcal{V}(e)$$

```
module VarSet =
  Set.Make
    (struct type t = var let compare = ... end)

let rec vars e = match e with
| Int _ -> VarSet.empty
| Add (e1,e2) | Sub (e1,e2) | Mul (e1,e2) | Div (e1,e2) ->
    VarSet.union (vars e1) (vars e2)
| Var x -> VarSet.singleton x
| Let (x,ex,e) ->
    VarSet.add (VarSet.union (vars ex) (vars e)) x
```

Variables libres

Ce sont les variables non-liées, c'est-à-dire celles dont la valeur doit être fournie.

$$\begin{aligned}
\mathcal{F}(\texttt{x+1}) &= \{\,\texttt{x}\,\} \\
\mathcal{F}(\texttt{x+1+y}) &= \{\,\texttt{x},\texttt{y}\,\} \\
\mathcal{F}(\,\textbf{let}\ \ \texttt{x = 1}\ \ \textbf{in}\ \ \texttt{x+1}) &= \emptyset \\
\mathcal{F}(\,\textbf{let}\ \ \texttt{x = 1}\ \ \textbf{in}\ \ \texttt{x+1+y}) &= \{\,\texttt{y}\,\}
\end{aligned}$$

$$\mathcal{F}(\texttt{0}) = \emptyset \qquad \cdots \qquad \mathcal{F}(x) = \{x\} \qquad \mathcal{F}(e_1 \texttt{ + } e_2) = \mathcal{F}(e_1) \cup \mathcal{F}(e_2)$$

$$\mathcal{F}(\textbf{let}\ x = e_x\ \textbf{in}\ e) = \mathcal{V}(e_x) \cup (\mathcal{V}(e) \setminus \{x\})$$

Et aussi,

$$\mathcal{F}(\textbf{fun}\ x\ \texttt{->}\ e) = \mathcal{F}(e) \setminus \{\,x\,\}$$

Calcul des variables libres

```
let rec free e = match e with
| Int _ -> VarSet.empty
| Add (e1,e2) | Sub (e1,e2) | Mul (e1,e2) | Div (e1,e2) ->
    VarSet.union (free e1) (free e2)
| Var x -> VarSet.singleton x
| Let (x,ex,e) ->
    VarSet.union (free ex) (VarSet.remove (free e) x)
```

Sémantique naturelle des expressions avec variables

Une expression peut maintenant contenir des variables (libres).

- Il faut connaître la valeur d'une variable *libre*.
- Pour la remplacer par cette valeur.

Une solution simple : l'environnement. La relation sémantique est maintenant de la forme

$$\Gamma \vdash e \hookrightarrow v$$

Où ici,

- Γ est un environnement, c'est à dire une fonction (de domaine fini) des variables dans les valeurs.
- e est une expression.
- v est une valeur, ici un entier.

Sémantique des expressions avec variables

$$\frac{}{\Gamma \vdash \mathtt{n} \hookrightarrow n} \qquad \frac{\Gamma(x) = n}{\Gamma \vdash x \hookrightarrow n} \qquad \frac{\Gamma \vdash e_1 \hookrightarrow n_1 \qquad \Gamma \vdash e_2 \hookrightarrow n_2}{\Gamma \vdash e_1 \mathtt{+} e_2 \hookrightarrow n_1 + n_2} \qquad \ldots$$

Programmation

Un environnement est une liste de paires $x \times v$.

```
type env = (var * int) list
```

```
let rec assoc x env = match env with
| [] -> raise Not_found
| (y,vy)::env ->
   if x=y then vy
   else assoc x env
(* Disponible comme List.assoc *)
```

```
let rec eval env e = match e with
| Var x -> assoc x env
| Let (x,ex,e) ->
    let vx = eval env ex in
    eval ((x,vx)::env) e
| ... (* Comme avant ou presque *)
```

Vers la sémantique à petit pas

Comment définir $\rightarrow$ (une étape de calcul) sur un terme qui contient **let** ? L'idée est que on peut effectuer la liason **let** x `=` e_x **in** e en remplaçant x par e_x dans e.

Une substitution est tout simplement une fonction (de domaine fini) des variables dans les... *termes*. On note $[e_x \backslash x]$ la substitution qui à x associe e_x.

Les substitutions sont des *morphismes*, c-à-d respectent la structure de terme.

Les substitutions s'appliquent aux variables libres seulement :

$$
\begin{aligned}
[2\backslash\texttt{x}]\ (\texttt{x+1}) &= \texttt{2+1} \\
[2\backslash\texttt{x}]\ (\textbf{fun}\ \texttt{x}\ \texttt{->}\ \texttt{x+1}) &= \textbf{fun}\ \texttt{x}\ \texttt{->}\ \texttt{x+1} \\
[2\backslash\texttt{x}]\ (\textbf{let}\ \texttt{x}\ \texttt{=}\ \texttt{x+x}\ \textbf{in}\ \texttt{x}) &= \textbf{let}\ \texttt{x}\ \texttt{=}\ \texttt{2+2}\ \textbf{in}\ \texttt{x}
\end{aligned}
$$

Définition de la substitution, premier essai

Remplacer les variables libres par leurs définitions :

$$[e\backslash x]\texttt{0} = \texttt{0} \qquad \cdots \qquad [e\backslash x]x = e \qquad [e\backslash x]y = y$$

$$[e\backslash x]e_1\texttt{+}e_2 = [e\backslash x]e_1\texttt{+}[e\backslash x]e_2$$

$$[e\backslash x](\textbf{let } x \texttt{ = } e_1 \textbf{ in } e_2) = \textbf{let } x \texttt{ = } [e\backslash x]e_1 \textbf{ in } e_2$$

$$[e\backslash x](\textbf{let } y \texttt{ = } e_1 \textbf{ in } e_2) = \textbf{let } y \texttt{ = } [e\backslash x]e_1 \textbf{ in } [e\backslash x]e_2$$

Et de même :

$$[e\backslash x](\textbf{fun } x \texttt{ -> } e_1) = \textbf{fun } x \texttt{ -> } e_1$$

$$[e\backslash x](\textbf{fun } y \texttt{ -> } e_1) = \textbf{fun } y \texttt{ -> } [e\backslash x]e_1$$

Mais attention

Il semble donc que l'on puisse réduire **let** x `=` e_x **in** e en $[e_x \backslash x](e)$.

Soit la fonction qui ajoute `y` libre à son argumemt.

fun `x -> x + y`

Sous-expression par exemple de :

let `y =` e_y **in fun** `x -> x + y`

En substituant...

- Si e_y est `4` on a **fun** `x -> x + 4`.
- Si e_y est `z` on a **fun** `x -> x + z`.
- Si e_y est `x` a-t-on ? **fun** `x -> x + x` ?

  Il ne vaut meux pas ! Considérer :

  let `x = 1` **in let** `y = x` **in fun** `x -> x + y`

Alpha-équivalence

Dans l'exemple [x\y](**fun** x -> x+y), nous sommes victimes de la *capture* de la variable libre x, qui devient liée en traversant la liaison **fun** x -> ...

Or

- Si l'argument de la fonction est nommé z on peut substituer :

$$[\texttt{x}\backslash\texttt{y}](\textbf{fun}\ \texttt{z}\ \texttt{->}\ \texttt{z}\ \texttt{+}\ \texttt{y}) = \textbf{fun}\ \texttt{z}\ \texttt{->}\ \texttt{z}\ \texttt{+}\ \texttt{x}.$$

- Les fonctions **fun** x -> x + y et **fun** z -> z + y sont les mêmes (changement de la variable muette).

L'alpha-équivalence est l'égalité modulo le changement des variables muettes.

Une définition possible de la substitution

En évitant les captures.

$$[e\backslash x](\textbf{let } x \textbf{ = } e_1 \textbf{ in } e_2) = \textbf{let } x \textbf{ = } [e\backslash x]e_1 \textbf{ in } e_2$$

$$[e\backslash x](\textbf{let } y \textbf{ = } e_1 \textbf{ in } e_2) = \textbf{let } z \textbf{ = } [e_x\backslash x]e_1 \textbf{ in } [e_x\backslash x]([z\backslash y]e_2)$$

Où z ci-dessus n'appartient ni à $\mathcal{F}(e_x)$, ni à $\mathcal{F}(e_2)$.

C'est bien compliqué.

Programmer la substitution

En pratique : z est une variable « fraiche », c'est à dire une variable nouvelle.

```
type var = int

let fresh_var =
  let count = ref 0 in
  fun () ->
    let v = !count in
    count := !count+1 ;
    v

let rec subst x ex e = match e with
| Int _ -> e
| Add (e1,e2) -> Add (subst x ex e1, subst x ex e2)
  ... (* Sub, etc. idem *)
| Var y ->  if x = y then ex else e
  ...
```

Programmer la substitution, cas délicat

```
| Let (y, ey, e) ->
    if x=y then
      Let (y, subst x ex ey, e)
    else
      let z = fresh_var () in
      Let (z, subst x ex ey, subst x ex  (subst y (Var z) e))
```

Pas le plus efficace, certainement... Mais suffisant pour définir le calcul sur le **let**.

Pour définir une étape de calcul, il reste à

- Choisir un sous-terme réductible (un *redex*)
- Et à le remplacer par son réduit.

En TP

- Un peu de Caml (ensembles).
- Un peu de termes.
- Et calcul d'un point fixe.