Les types sont des invariants comme les autres

Un peu de sémantique opérationnelle

Nous avons vu des propriétés des types et du typage :

- Structurent la logique du programme.
- Permettent de vérifier *récursivement* que le programme a du sens.

Mais on peut aussi les voir comme un moyen de s'assurer que les opérations définies par notre langage ont du sens.

Pour cela, on ne peut pas se contenter des règles de typage, il faut également décrire comment on veut que le programme se comporte : quel effet ont les instructions, comment s'évaluent les expressions.

On appelle la description de ce comportement la *sémantique opérationnelle du langage*.

Un exemple simple

Décrire le comportement d'un programme peut se faire avec des règles de réduction. On écrit $P \to P'$ si le programme P "calcule" le programme P'.

Par exemple `fst(t,u)` $\to$ `t` est une règle de réduction correspondant au comportement attendu de `fst`.

Remarque

Parfois un programme est bien typé et aucune règle ne s'applique. Par exemple, `fst(x)` est correct si `x` est de type produit, mais la règle de réduction ne s'applique pas (mais d'autres règles peuvent remplacer `x` par un couple...).

Préservation du typage

Propriété

Un ensemble de règles de réduction est correct si, quand $P \to P'$ et P est de type A, alors P' est aussi de type A.

On appelle aussi cette propriété la *réduction du sujet*.

Dans l'exemple précédent, c'est évident, car `fst(t,u)` est du type de `t` par définition de nos règles de typage.

Dans un contexte où une fonction $f : A \to B$ est appliquée à $t : A$, on a $f(t) : B$ [1], et le calcul de $f(t)$ doit en effet produire une valeur de type B.

1. En Ocaml, rappelez-vous que l'on peut écrire indifféremment "`let f:A->B = fun x ...`", ou encore "`let f (x:A) : B = ...`"

Sémantique opérationnelle

Il est incorrect qu'une expression d'un certain type s'évalue en une expression d'un type différent, ou en une instruction. Une instruction se réduit toujours en une instruction.
Mais à nouveau, comment décrire la réduction d'un programme avec des variables, comme `x=(1,true);y=fst(x)` ?

On doit considérer des environnements décrivant non plus le type, mais le contenu des variables[2]

Pour suivre l'évolution des environnements le long de la réduction, on écrira `E|t` $\rightarrow$ `E'|t'`, où `E|t` représente l'élément de programme `t` dans un contexte `E` de la forme `x:=s,y:=t,...`.

2. Ici, il faudra donc avoir vérifié que les variables sont non seulement bien typées, mais ont reçu une valeur.

Réductions avec environnements

À quoi ça pourrait ressembler

`E|x=t;s` $\rightarrow$ `E,x:=t|s`[3] Si `E` contenait déjà une affectation pour `x`, elle est remplacée (on peut penser à une implémentation de `E` par une `Map<String,Expression>` en Java, par exemple).

`E,x:=t|P[x]` $\rightarrow$ `E,x:=t|P[t]`, où `P[x]` représente un élément de programme avec des apparitions de x. C'est une règle de substitution. Pour reprendre l'exemple précédent, on a :

```
   |x=(1,true);y=fst(x)
→ x:=(1,true)|y=fst(x)
→ x:=(1,true)|y=fst(1,true)
→ x:=(1,true)|y=1
```

3. On peut aussi évaluer l'expression `t` avant de la mettre dans l'environnement, c'est dans ce cas un choix de conception du langage en *appel-par-valeur*.

Sémantique opérationnelle des conditions

`E|if(t) s` : il faut réduire `t` à `true` ou à `false`, la préservation du typage nous assure que s'il n'y a pas d'erreur, on obtient soit l'un soit l'autre (l'évaluation d'un booléen ne peut produire que ces deux valeurs).

Ensuite, soit on exécute le corps de la conditionnelle, soit non (et dans ce dernier cas, on se réduit à une instruction vide, mais d'autres pourraient suivre).

$$\texttt{E|if(true) s} \rightarrow \texttt{ E|s}$$
$$\texttt{E|if(false) s} \rightarrow \texttt{ E|}$$

Sémantique opérationnelle des boucles

$\texttt{E|while(b) t} \rightarrow \texttt{E|if(b)\{ t;while(b) t \}}$

La réduction peut ne pas terminer[4], cela dépend de la façon dont la réduction de `t` influera sur l'environnement (si `b` ne dépend pas de son environnement et s'évalue à `true`, c'est mal parti).

4. Comme dans la vraie vie

Pourquoi je vous parle de tout ça ?

Au premier cours, j'ai dit quelque chose comme :

Citation

« Quand on traduit un programme dans un autre langage, c'est pas évident de déterminer si une traduction est correcte ; et ce que veut dire "correct" dans ce contexte n'est pas évident non plus. »
J.C.

Pourquoi je vous parle de tout ça ?

Au premier cours, j'ai dit quelque chose comme :

Citation

« Quand on traduit un programme dans un autre langage, c'est pas évident de déterminer si une traduction est correcte ; et ce que veut dire "correct" dans ce contexte n'est pas évident non plus. »
J.C.

Une première approximation, est de dire que le programme "fait" la même chose. Mais tant qu'on n'est pas arrivé à l'exécutable, il ne *fait* rien... En revanche, il représente ce qu'on **veut qu'il fasse**.

Pourquoi je vous parle de tout ça ?

Au premier cours, j'ai dit quelque chose comme :

Citation

« Quand on traduit un programme dans un autre langage, c'est pas évident de déterminer si une traduction est correcte ; et ce que veut dire "correct" dans ce contexte n'est pas évident non plus. »
J.C.

Une première approximation, est de dire que le programme "fait" la même chose. Mais tant qu'on n'est pas arrivé à l'exécutable, il ne *fait* rien... En revanche, il représente ce qu'on **veut qu'il fasse**.

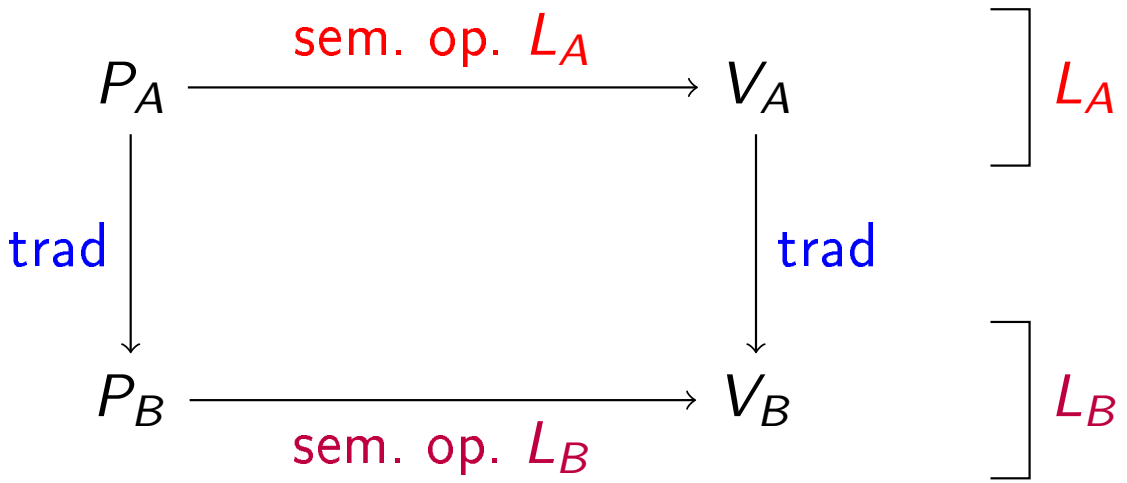
Ce qu'on veut qu'il fasse, ça s'appelle précisément la sémantique opérationnelle.

Alors, c'est quoi une traduction correcte ?

C'est une traduction qui respecte la sémantique opérationnelle. Plus précisément, soit un langage source L_A, un langage cible L_B, et un programme P_A écrit en L_A.

Supposons que P_A soit compilé en P_B. On veut vérifier que si $P_A \to V_A$ (d'après la sémantique op. de L_A), alors il existe V_B tel que $P_B \to V_B$ (d'après la sémantique op. de L_B.), tel que V_B soit une traduction de V_A.

Visuellement :



Un exemple

[au tableau]