Chapitre 2

Sémantique opérationnelle et dénotationnelle d'un langage fonctionnel simple typé

> Dans ce chapitre, nous allons étudier un langage fonctionnel simple, du point de vue opérationnel et dénotationnel, et on introduira les relations fondamentales entre ce deux approches. On montrera aussi comment on peut vivre dans *Set* en première approximation.

2.1 Un langage fonctionnel simple typé: TFun

On s'interessera maintenant à étudier la sémantique d'un simple langage fonctionnel typé (appelé donc bien TFun[1]), du point de vue dénotationnel et opérationnel. Sur cet exemple simple on introduira aussi les notions fondamentales qui permettent de mettre en relation sémantiques opérationnelles entre elles et avec la sémantique dénotationnelle.

Le langage est donné ici en définissant avant tout

- la syntaxe abstraite
- le système de types

et ensuite on donnera sa sémantique opérationnelle (SOS et naturelle) et dénotationnelle (sur les ensembles).

[1]Pour les experts, c'est bien PCF sans récursion.

2.1.1 Syntaxe abstraite

Notre langage TFun est généré par la BNF suivante, où le catégories syntaxique sont indiquées entre parenthèses:

```
(TYPES)
T : = int | bool | T -> T

(ENTIERS)
I : = 0 | 1 | ...

(BOOLEENS)
B : = true | false

(IDENTIFICATEURS)
Id : = non specifie'

(EXPRESSIONS)
Exp : = I | B | Id | fun Id :  T -> Exp | Exp Exp | if(Exp,Exp,Exp) | ...
```

Voilà un programme légal: `(fun x: int -> fun y: bool -> if(y, x, x+1))`

2.1.2 Typage

On donne ensuite les règles de typage de TFun. Pour faire ça on utilise des *jugements* (ou assertions) de la forme:

$$\Gamma \vdash \texttt{programme :} \quad Exp\texttt{[type]}$$

où Γ (appelé *environnement de typage*) est une liste (même vide, ce que l'on note par $\emptyset$ ou par rien du tout) d'assertions de la forme $\texttt{id :} \quad Id\texttt{[type]}$ qui donnent le type des identificateurs libres du programme.

Remarque 2.1.1 *On demande que chaque identificateur apparaisse une seule fois dans* Γ*.*

Proposition 2.1.2 (Typage décidable, unicité de la dérivation) *Il est décidable si un programme a un type. De plus, pour tout jugement de typage dérivable, il existe une seule dérivation.*

Remarque 2.1.3 *L'unicité de la dérivation de typage est nécessaire pour pouvoir définir de façon cohérent et simple des notions par récurrence sur la dérivation. Si ce n'est pas le cas, on doit prouver des complexes théorèmes de cohérence.*

$$\Gamma \vdash \texttt{id:}\ \ Exp[\texttt{t}] \qquad \text{si } \texttt{id:}\ \ Id[\texttt{t}] \in \Gamma$$

$$\Gamma \vdash \texttt{i:}\ \ Exp[\texttt{int}] \qquad (\text{pour } i \text{ constante entière})$$

$$\Gamma \vdash \texttt{true}: \quad Exp[\texttt{bool}]$$

$$\Gamma \vdash \texttt{false}: \quad Exp[\texttt{bool}]$$

$$\frac{\Gamma, \texttt{id:}\ \ Id[\texttt{t1}] \vdash \texttt{e:}\ \ Exp[\texttt{t2}]}{\Gamma \vdash \texttt{fun id: t1 -> e2} : Exp[\texttt{t1->t2}]}$$

$$\frac{\Gamma \vdash \texttt{e1:}\ \ Exp[\texttt{t1->t2}] \qquad \Gamma \vdash \texttt{e2:}\ \ Exp[\texttt{t1}]}{\Gamma \vdash \texttt{e1 e2} : Exp[\texttt{t2}]}$$

$$\frac{\Gamma \vdash \texttt{e:}\ \ Exp[\texttt{bool}] \qquad \Gamma \vdash \texttt{e1:}\ \ Exp[\texttt{t}] \qquad \Gamma \vdash \texttt{e2:}\ \ Exp[\texttt{t}]}{\Gamma \vdash \texttt{if(e,e1,e2)} : Exp[\texttt{t}]}$$

Remarque 2.1.4 *On laisse a vous de compléter les règles dans les cas des opérations binaires et unaires sur entiers et booléens.*

Exercice 2.1.5 *En vous appuyant sur la notion d'alpha conversion vu dans les autres cours, expliquez pourquoi la restriction sur* Γ *ne nous fait pas perdre en généralité.*

Exercice 2.1.6 *Écrire une dérivation formelle complète en utilisant le règles de typage du fait que*

$$\emptyset \vdash \texttt{(fun x: int -> fun y: bool -> if(y, x, x+1)):}\ \ Exp[\texttt{int -> bool -> int}]$$

2.1.3 Sémantique opérationnelle naturelle de TFun

On donne ici la sémantique opérationnelle *en appel par nom* et *en appel par valeur*:

L'appel par nom: $\Downarrow_n$

constantes:

$$cst \Downarrow_n cst$$

abstraction fonctionnelle et application:

$$\texttt{fun}\ x : A\texttt{->}M \Downarrow_n \texttt{fun}\ x : A\texttt{->}M$$

$$\frac{e_1 \Downarrow_n \texttt{fun}\ x : A\texttt{->}e_0 \qquad e_0[e_2/x] \Downarrow_n v}{e_1(e_2) \Downarrow_n v}$$

conditionnel:

$$\frac{e \Downarrow_n \texttt{true} \qquad e_1 \Downarrow_n v}{if(e, e_1, e_2) \Downarrow_n v} \qquad\qquad \frac{e \Downarrow_n \texttt{false} \qquad e_2 \Downarrow_n v}{if(e, e_1, e_2) \Downarrow_n v}$$

Remarque 2.1.7 (Environnements) *On n'utilise pas les environnements dans l'évaluation.*

Exercice 2.1.8 *Compléter la sémantique en rajoutant les cas pour les opérations sur les entiers et les booléens.*

L'appel par valeur: $\Downarrow_v$

constantes:

$$cst \Downarrow_v cst$$

abstraction fonctionnelle et application:

$$\texttt{fun}\ x : A\texttt{->}M \Downarrow_v \texttt{fun}\ x : A\texttt{->}M$$

$$\frac{e_1 \Downarrow_v \texttt{fun}\ x : A\texttt{->}e_0 \quad e_2 \Downarrow_v v_2 \quad e_0[v_2/x] \Downarrow_v v_0}{e_1(e_2) \Downarrow_v v_0}$$

conditionnel:

$$\frac{e \Downarrow_v \texttt{true} \quad e_1 \Downarrow_v v}{if(e, e_1, e_2) \Downarrow_v v} \qquad \frac{e \Downarrow_v \texttt{false} \quad e_2 \Downarrow_v v}{if(e, e_1, e_2) \Downarrow_v v}$$

2.1.4 Quelques résultats

Définition 2.1.9 (Valeurs) *On définit* valeurs *les constantes ou les abstractions.*

Proposition 2.1.10 (Le résultats sont des valeurs) *Si $M \Downarrow_v N$, alors N est un valeur v. De même pour $\Downarrow_n$.*

Preuve. Par induction sur la dérivation. □

Proposition 2.1.11 (Déterminisme) *Si $M \Downarrow_v v$ et $M \Downarrow_v v'$, alors $v \equiv v'$. De même, si $M \Downarrow_n v$ et $M \Downarrow_n v'$, alors $v \equiv v'$.*

Preuve. Par induction sur les deux dérivations. □

2.1.5 Sémantique opérationnelle structurée (SOS) de TFun

On donne ici la sémantique opérationnelle *en appel par nom* et *en appel par valeur*:

L'appel par nom: $\Rightarrow_n$

application:

$$(\texttt{fun}\ x : A\texttt{->}M)N \Rightarrow_n M[N/x] \qquad\qquad \frac{e_1 \Rightarrow_n e'}{e_1e_2 \Rightarrow_n e'e_2}$$

conditionnel:

$$\frac{e \Rightarrow_n e'}{if(e, e_1, e_2) \Rightarrow_n if(e', e_1, e_2)}$$

$$if(\texttt{true}, e_1, e_2) \Rightarrow_n e_1 \qquad\qquad if(\texttt{false}, e_1, e_2) \Rightarrow_n e_2$$

L'appel par valeur: $\Rightarrow_v$

Remarque 2.1.12 (Valeurs) *On rappelle que les valeurs sont ici les constantes ou les abstractions, et on les marque dans ce qui suit par des noms commençant par v.*

application:

$$(\texttt{fun}\ x : A\texttt{->}M)v \Rightarrow_v M[v/x] \qquad \text{(si } v \text{ est un valeur)}$$

$$\frac{e_1 \Rightarrow_v e'}{e_1e_2 \Rightarrow_v e'e_2} \qquad\qquad \frac{e \Rightarrow_v e'}{ve \Rightarrow_v ve'} \qquad\qquad \text{(si } v \text{ est un valeur)}$$

conditionnel:

$$\frac{e \Rightarrow_v e'}{if(e, e_1, e_2) \Rightarrow_v if(e', e_1, e_2)}$$

$$if(\texttt{true}, e_1, e_2) \Rightarrow_v e_1 \qquad\qquad if(\texttt{false}, e_1, e_2) \Rightarrow_v e_2$$

Remarque 2.1.13 (Environnements) *On n'utilise pas les environnements dans l'évaluation.*

Exercice 2.1.14 *Compléter les sémantique en rajoutant les cas pour les opérations sur les entiers et les booléens, et aussi les couples avec projections.*

Exercice 2.1.15 *Est-ce que l'appel par valeur et par nom donnent des résultats différents dans notre langage TFun? Pourquoi?*

2.1.6 Quelques résultats

Proposition 2.1.16 (Relation entre $\Downarrow$ et $\Rightarrow$) *En appel par nom et par valeur,*

1. *si $M \Downarrow v$, alors $M \Rightarrow_* v$*
2. *si $M \Rightarrow_* v$, et v est un valeur, alors $M \Downarrow v$*

Preuve.

1. par induction sur la dérivation $M \Downarrow v$
2. par induction sur la taille de la dérivation $M \Rightarrow_* v$

□

Exercice 2.1.17 (Déterminisme, équivalence) *Démontrez que (pour v un valeur):*

- *si $M \Rightarrow_v v$ et $M \Rightarrow_v v'$, alors $v \equiv v'$*
- *si $M \Rightarrow_n v$ et $M \Rightarrow_n v'$, alors $v \equiv v'$*
- *si $M \Rightarrow_n v$ et $M \Rightarrow_v v'$, on n'a pas nécessairement $v \equiv v'$*

Théorème 2.1.18 (Normalisation forte) *Toute séquence de réduction $M \Rightarrow \ldots$ termine.*

Preuve. Cela est une conséquence du théorème de normalisation forte pour le lambda calcul typé simple. □

Théorème 2.1.19 (Confluence) *Si $M \Rightarrow N'$ et $M \Rightarrow N''$, il existe N''' t.q. $N' \Rightarrow N'''$ et $N'' \Rightarrow N'''$.*

Preuve. Avec la technique des "réductions parallèles". □

Théorème 2.1.20 (Subject reduction) *Le reduction preserve le typage, i.e. si M est bien typé, et $M \Rightarrow N$, alors N est bien typé.*

Preuve. Par induction sur la dérivation de $M \Rightarrow N$, en utilisant un lemma de substitution approprié (lequel?). $\square$

Corollaire 2.1.21 ($\Rightarrow$ sur programmes clos) *En appel par nom ou par valeur, pour tout programme clos (i.e. sans identificateurs non déclarés) M, il existe un valeur v t.q. $M \Rightarrow v$.*

Preuve. Par induction sur la longueur de la réduction. $\square$

Corollaire 2.1.22 ($\Downarrow$ sur programmes clos) *En appel par nom ou par valeur, pour tout programme clos (i.e. sans identificateurs non déclarés) M, il existe un valeur v t.q. $M \Downarrow v$.*

Preuve. Utiliser le corollaire précédent. $\square$

Exercice 2.1.23 *Montrez que si M a type de base, alors $M \Rightarrow_n v$ et $M \Rightarrow_v v'$ implique $v \equiv v'$*

Exercice 2.1.24 *Montrez que pour tout programme clos e, $e \Rightarrow_v v$ si et seulement si $e \Downarrow_n v$.*